

Artificial intelligence in design '91

edited by

J. S. Gero

University of Sydney



Butterworth-Heinemann Ltd
Linacre House, Jordan Hill, Oxford OX2 8DP



OXFORD LONDON BOSTON
MUNICH NEW DELHI SINGAPORE SYDNEY
TOKYO TORONTO WELLINGTON

First published 1991

© Butterworth-Heinemann Ltd 1991

All rights reserved. No part of this publication may be reproduced in any material form (including photocopying or storing in any medium by electronic means and whether or not transiently or incidentally to some other use of this publication) without the written permission of the copyright holder except in accordance with the provisions of the Copyright, Designs and Patents Act 1988 or under the terms of a licence issued by the Copyright Licensing Agency Ltd, 90 Tottenham Court Road, London, England W1P 9HE. Applications for the copyright holder's written permission to reproduce any part of this publication should be addressed to the publishers.

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library.

Library of Congress Cataloguing in Publication Data

A catalogue record for this book is available from the Library of Congress.

ISBN 0 7506 1188 X

Printed and bound in Great Britain.

Preface

Design is becoming a major research topic in engineering and architecture. It is one of the keys to economic competitiveness and the fundamental precursor to manufacturing. However, our understanding of design as a process and our ability to model it are still very limited. One of the major strands of design research is concerned with developing computational symbolic models of design processes. The foundations of much of this research can be traced to the ideas expounded by Herbert Simon in 1969 in his book *The Sciences of the Artificial*.

There is a twofold interest in design research. The first is concerned with producing a better understanding of design. The second is concerned with producing useful tools to aid human designers and in some areas to automate various aspects of the design process. Just as in linguistics a knowledge of language theory does not necessarily make a writer a Shakespeare but is essential for the production of writing aids, so in design a knowledge of design theory does not necessarily make a designer a better designer but is essential for the production of design aids. The notion that design could be researched dates back to the very early days of the nineteenth century. However, the development of the computational symbolic paradigm over the last thirty years has spurred researchers to produce increasingly useful models and processes which have potential applicability in design.

The articulation of artificial intelligence as a separate area of research and development has opened up new ideas for design researchers. Similarly, design can be treated as a highly complex intelligent activity which provides a rich domain for artificial intelligence researchers. Artificial intelligence in design has been a burgeoning area for a decade now. The papers in this volume are from the First International Conference on Artificial Intelligence in Design held in June 1991 in Edinburgh.

The forty-seven papers, which have been extensively refereed, are grouped under the following headings.

- Artificial intelligence paradigms and design**
- Constraint-based reasoning in design**
- Case-based reasoning in design**
- Interface environments in design**
- Learning in design**
- Design representation**
- Nonmonotonic reasoning in design**
- Cognitive aspects of design**
- Industrial applications of AI in design**
- Conceptual design**
- Design documentation**

Applications of AI in design

Integrated design—systems and tools

Even a cursory glance at this list shows both the depth and breadth of research and application of artificial intelligence in design. This activity has defined a new paradigm in design theory and methodology and a new research field. These papers present the state-of-the-art and the leading edge of that field. They provide the basis for an advanced understanding of artificial in design.

All papers were reviewed by three referees drawn from a large international panel. Many thanks to them for the quality of the accepted papers depends on their efforts. They are listed below. Fay Sudweeks, working extraordinary hours, shaped this volume in her inimitable fashion, from the material submitted by the various authors. As always, special thanks are due to her.

John S. Gero
University of Sydney
April 1991

International Panel of Referees

Alice Agogino	Ernest Edmonds	Lauri Koskela	Michael Rosenman
Shinsuke Akagi	Jose Encarnacao	Sridha Kota	Chris Rowles
Omer Akin	Boi Faltings	F.-L. Krause	George Rzevski
Varol Akman	Paul Fazio	Bryan Lawson	Mark Sapossnek
Roger Allwood	Steve Fenves	Ray Levitt	Warren Seering
Farhad Arbab	Susan Finger	Shaopei Lin	Ron Sharpe
Tomasz Arciszewski	Gerhard Fischer	Xila Liu	Tim Smithers
Bala Balachandran	Ulrich Flemming	Brian Logan	Jaroslaw Sobieski
P. Banerjee	John Frazer	Ken MacCallum	William Spillers
Claude Bedard	Renate Fruchter	Mary Lou Maher	Duv Sriram
Reza Beheshti	James Garrett	Bertil Marksjo	Larry Stauffer
Laszlo Berke	John Gero	Andras Markus	David Steier
Peter Bernus	Hans Grabowski	Kurt Marshek	Louis Steinberg
Aart Bijl	Don Grierson	Cindy Mason	George Stiny
Bill Birmingham	Mark Gross	Krishan Mathur	Peter Struss
Bo-Christer Björk	Barbara Hayes-Roth	Farrokh Mistree	Katia Sycara
Jim Bowen	Jack Hodges	Fumio Mizoguchi	Karl-Heinz Temme
Alan Bridges	David Hoeltzel	Peter Mullarkey	Tetsuo Tomiyama
Andy Brown	Tony Holden	Dundee Navinchandra	Iris Tommelein
David Brown	Se June Hong	Shimon Nof	Chris Tong
Jon Cagan	Craig Howard	Setsuo Ohsuga	Barry Topping
Per Christiansson	Mike Huhns	Tarkko Oksala	Enn Tyugu
Helder Coelho	William Ibbs	Rivka Oxman	David Ullman
Phillippe Coiffet	Kos Ishii	Panos Papalambros	Karl Ulrich
Diane Cook	Yumi Iwasaki	Graham Powell	V. Venkatasubramanian
Richard Coyne	Marwan Jabri	James Powell	Willemien Visser
Robert Coyne	Yehuda Kalay	Kenneth Preiss	Allen Ward
Tom Dietterich	S. M. Kannapan	Terry Purcell	Richard Welch
Jack Dixon	R. L. Kashyap	Tony Radford	Art Westerberg
Alex Duffy	Randy Katz	J. R. Rao	Rob Woodbury
Clive Dym	Fumihiro Kimura	William Rasdorf	Michael Wozny
Chuck Eastman	Ted Kitzmiller	Dan Rehak	Jae Woo Yang
Ernst Eder	Torsten Kjellberg	James Rinderle	Ji Zhou
			Khaldoun Zreik

Representing the engineering design process: two hypotheses

R. Bañares-Alcántara

Department of Chemical Engineering
University of Edinburgh
Edinburgh EH9 3JL Scotland

Abstract. A representation of the design process is proposed based on the exploration model of Smithers *et al.* Although it was developed around the characteristics of the chemical plant design area, we believe that it is applicable to any other area as long as it does not involve spatial reasoning. It supports the exploration of the different alternatives generated by the use of aggregation and refinement operators, the consideration of different operating values, and all the combinations of the above. Aside from being able to explore a network of alternatives and expand any node at any point in time, a designer can propagate changes to the design in any selected direction. The representation depends on two hypotheses: (a) the decomposition of information hypothesis; and (b) The identical object hypothesis. Both have been used previously for different purposes, and the results presented here indicate that they are sound. A CommonLisp object-oriented system is used for the representation of the object to be designed (a chemical plant). An ATMS (Assumption-Based Truth Maintenance System) is used to record the dependencies of the equipment and alternative plants, and to support the exploration of the design space.

1 INTRODUCTION

A useful way to think about the development of AI-based design systems is proposed by Smithers (1989). In that report, he identifies three main activities related to the process of design:

- (a) Development of models for the design process itself.
- (b) Development of techniques to represent and reason about design knowledge.

*On leave from the Facultad de Química, UNAM. México.

(c) Construction of integrated system architectures required by design support systems.

This paper addresses the second item above, taking the design process model proposed by Smithers *et al.* (1989), and proposes a different representation technique for the design process. It is claimed that this representation is an appropriate one for the design of chemical plants in particular, since it was developed around the problems and characteristics described in the chemical engineering design literature. It should be useful, nevertheless, for any type of design where

- there is a very large number of possible alternatives,
- there is a reliable evaluator for the alternatives, and
- the evaluator of alternatives is expensive to use, so we cannot expect to be able to apply it to every alternative.

The next section examines some of the complications associated with modelling the design process in the real world. After reviewing the *state space* representation and how it has been used to represent the process of design, Section 3 compares three alternative views of the design process, and the best one chosen to be the base of the proposed representation (Section 4). In order to implement the proposed representation, an Assumption-Based Truth Maintenance System (ATMS) is used. The last three sections deal with an example of an implementation developed to test our ideas, a discussion of the current state of the system and desired future developments, and the conclusions of this work.

2 THE NATURE OF THE ENGINEERING DESIGN PROCESS

It has been usual to simplify the model of the process of design, and assume away the existence of some of its properties. One of the objectives of the current research is to take into account such complications, in order to have a closer model of a real world design process.

The properties that we intend to take into account are:

(a) Design as an opportunistic activity.

Design is not performed using a fixed set of operators applied in an ordered way, as it is implied in most engineering textbooks. Design is a process where both, the *top-down* and *bottom-up* approaches, are used by the designer in an opportunistic manner. It is only relatively recently that this has been acknowledged (*e.g.* “... *the order of concept generation is not top-down*” (Westerberg *et al.*, 1989)).

(b) Design as an incremental activity.

Design in engineering is evolutionary. Changes are proposed to the current design in order to move to a “better” design (in some metric). These

changes can be seen as improvements or refinements. In general, the changes are small with respect to the total amount of information held in the current design.

(c) Design as an exploration activity.

Smithers *et al.* (1989) propose an exploration-based model of design, describing the design process as a knowledge-based exploration task. Design is classified as exploration, rather than search, because knowledge about the space of possible solutions has to be obtained before goals can be well formulated. Typically, the initial description of the solution is incomplete and/or ambiguous and/or inconsistent.

(d) Complexity of the design process.

Complexity arises from the generation of a large space of possible solutions, that is, from the very large number of alternatives of intermediate and final designs.

(e) Consistency of the design process.

Problems with consistency can arise due to two different and compounding factors:

- Design generally lasts over a long period of time, making it possible to reverse some of the previous decisions made by the designer either because
 - those previous actions have been forgotten,
 - the designer has learned a new fact or procedure, or
 - the designer changed his mind about some previous decision.
- Design can be understood as a social process, a process where the result is more than the sum of its participant's interpretation. The complete design is not in the possession of any individual nor is there a unique description possible (Westerberg *et al.*, 1989).

3 THREE VIEWS OF THE ENGINEERING DESIGN PROCESS

The process of design is a problem-solving activity, and as such, can be represented using a *state space*, where each state corresponds to a possible design. Each of the possible designs can be considered a solution (either intermediate or final), and therefore, we can also refer to it as a *solution space* or a *design space*. Thus, the process of design can be seen as a navigation from an initial state —the specification of the problem, to a final state —the proposed solution, as seen in Figure 1.

There are at least three different ways to use the information generated during the design process. The selection of either one determines how well the system will cope with the complications of design listed in Section 2, and each choice produces design spaces with different topologies. Naming each of the possibilities according to the form of their design spaces we have:

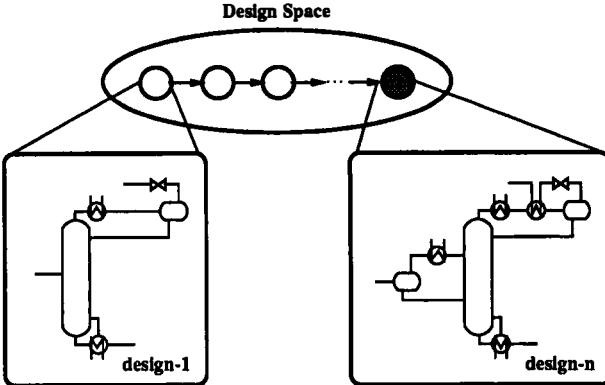


Figure 1: *State space representation of design.*

- (a) Lineal design space topology.
- (b) Tree design space topology.
- (c) Network design space topology.

3.1 Lineal design space topology

In the simplest case, no history of the intermediate stages of design is kept. Every time a modification is proposed to the current design, the modification is made and the previous state is lost. Each intermediate state has therefore only one successor, and the design process is a line of nodes with the initial specification A_1 as the first node and the proposed final solution S_A as the final node (Figure 2). The user has no mechanism to go back and explore an alternative in a



Figure 2: *Lineal design space topology.*

previous node (since it has been lost). While this representation would be the simplest to implement, it is inadequate as it fails to provide a mechanism to explore alternatives.

3.2 Tree design space topology

This is a more general case of the lineal design space topology. If the current design is saved in memory prior to being modified, the designer can at any point examine an intermediate design and propose a different modification. Each intermediate state can, thus, have more than one successor (each representing a

different alternative), originating a tree-like structure as in Figure 3. Actually the structure is a unidirectional graph if we take into account the case of identical states reached by different exploration paths, but a unidirectional graph can be expressed as a tree. Note that (as in the previous case), a decision of the designer

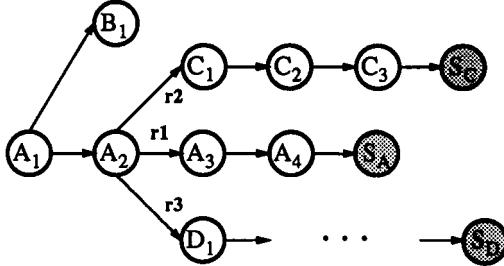


Figure 3: *Tree design space topology.*

has the potential of affecting only design states downstream of a single design path (*i.e.* a change to a current design can only propagate to its successors). For example, a new modification r_2 to A_2 can only affect designs on path $C_1 \rightarrow S_C$ when they are created, but cannot affect the designs on path $A_3 \rightarrow S_A$. If the user wanted to explore the application of both operators on the successors of the current node (*i.e.* propagate r_2 through path $A_3 \rightarrow S_A$), he would have to

- (a) Create a new (third) path where the next successor is the product of applying both operators simultaneously, and apply the rest of the operators of the old path to it, or
- (b) Have a demon-like facility which could visit each of the nodes in the old path and apply the new operator on them. But then, should the designs $A_3 \rightarrow S_A$ be modified by r_2 to $A'_3 \rightarrow S'_A$ (and the originals forgotten)? Or should a new path consisting of designs $A'_3 \rightarrow S'_A$ be created (reducing to the case above)?

This type of strategy is used in most of the current design support systems, and while is clearly better than the first one, it is not flexible enough to fully support exploration.

3.3 Network design space topology

This, in turn, is a more general case than the tree design space topology. In this case, each one of the states is kept in memory and can be dynamically accessed at any point during the design process. Not only can a node have more than one successor, but nodes from different branches can be linked through relations determined by the user. Also, a decision in any particular node has the potential to affect its successors, predecessors or siblings (again, controlled by the designer)

and create new alternatives. The design space takes the form of a bidirectional graph (Figure 4). This strategy is chosen as the basis of the design process

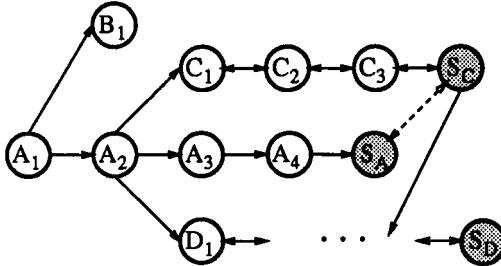


Figure 4: *Network design space topology.*

representation presented in the next section.

4 PROPOSED REPRESENTATION

We need a representation that can cope with the requirements listed in Section 2, and that is also able to represent the generic relations found in engineering design:

(a) Aggregation or Composition.

Each one of the proposed designs is built from components. We relate a design with each one of its components using aggregation relations.

(b) Specialisation or Refinement.

Two proposed designs are related by a specialisation or refinement relation if the second design describes the same object at a more detailed level than the first design.

(c) Alternatives or Versions.

Alternatives of a proposed design may be of two classes:

i. Structural alternatives

Where each alternative differs from the original in the components present or the way they are connected. In general, each one of the alternatives is a specialisation or refinement of the parent node.

ii. Operating alternatives

The alternatives are topologically equivalent, but some of the variables that determine their function have different values.

As noted, there is an overlap between the refinement relations and the alternative relations. At different points during the design process, the user may want to think in terms of one of them.

Each one of the above relations can be used to form hierarchical trees, and the user should be able to interleave them during the design process (since he may want to refine, decompose or explore an alternative of his current design at any point in time). Supporting this process implies that the system

- provides accessible information of the state of the design,
- allows easy movement between activities, and
- avoids imposing restrictions of when and how an activity can be engaged.

4.1 Two Hypotheses

Two key hypotheses have to be made prior to proposing our representation of the design process:

(a) Decomposition of the information hypothesis.

This hypothesis states that the information attached to the components of the object being designed should be decomposed in different classes and treated in a different way. For the specific application of Chemical Engineering design two classes of information are proposed and showed in Figure 5:

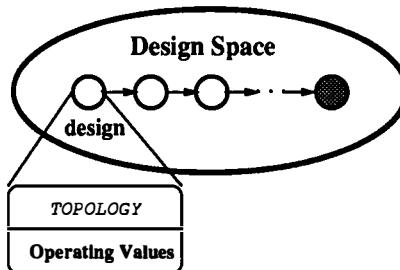


Figure 5: *Decomposition of the information hypothesis.*

- i. Existence and Topology.
- ii. Operating Values.

Although this separation seemed counterintuitive in the first place, it simplifies the final representation, and more importantly, seems to correspond to the way that the designer thinks about the design process (though not necessarily to the way that he thinks about the object being designed).

It has been pointed out (Ponton, 1990) that this decomposition is similar to the one used in the MINLP (Mixed Integer Non Linear Programming) approach to process synthesis, where there are two types of variables: integer and real-valued (see for example (Grossmann, 1989)).

(b) Identical object hypothesis.

Representing the whole current design of a chemical plant in each node of the design space is obviously not a good idea given the amount of repeated information that would have to be stored. Taking advantage of the evolutionary nature of design (see item 2 in Section 2), it is proposed to assume every component of the design to be the same in all the nodes where it is present as long as it has not been modified by an operator. This issue is related to the classical AI *frame problem* (McCarthy and Hayes, 1969).

4.2 Representation of the object to be designed

Although this project shares the same main objectives that other related projects in the Chemical Engineering Design area, it differentiates itself from most of them in its main focus. Instead of developing a syntax for the specification and description of the designed object (as is the main thrust in the ASCEND (Westerberg *et al.*, 1989) and TOOLKIT projects (Stephanopoulos *et al.*, 1987), it attempts to develop a “syntax” for the process design itself, much like the PIP project (Kirkwood *et al.*, 1988) has done for the preliminary design task (in PIP the design is implicitly assumed to be carried out by one designer in one session though).

A language for specification and description of chemical plants (the object to be designed) is a fundamental component of the design process that eases and formalises a very important part of the job. For this experimental implementation, and according to the decomposition of information hypothesis (Section 4.1), each proposed design (here referred to as a *scheme*) is described by a collection of *units* and their connections. A *unit* is a Chemical Engineering unit operation with attached information regarding its inputs and its outputs. In turn, each *unit* has associated one or more items of *equipment*: each item of *equipment* having an alternative set of operating values. Figure 6 shows an *scheme* (*scheme4*) composed of eight *units* (**FEEDFLASH**, **FEEDCOOLER**, **COLUMNW2F**, **CONDENSER**, **EXCHANGER**, **FLASH**, **EXPANSION** and **REBOILER**), each unit with one associated item of *equipment* (e.g. **FLASH** with **fla-1**) or more (e.g. **EXCHANGER** with **exch-1** and **exch-2**).

4.3 Representation of the design process

A proposed design or *scheme* is thus a collection of *units* and their associated *equipment*. Once the designer is satisfied with a *scheme* and decides to evaluate it, he must “save” it. There are two ideas behind saving a *scheme*:

- A *scheme* can only be saved, and therefore evaluated, if it is feasible. Thus the saving mechanism acts as a checkpoint.

Currently, the experimental system checks that all inputs and outputs for all *units* are connected to one stream, but more extensive checks will be

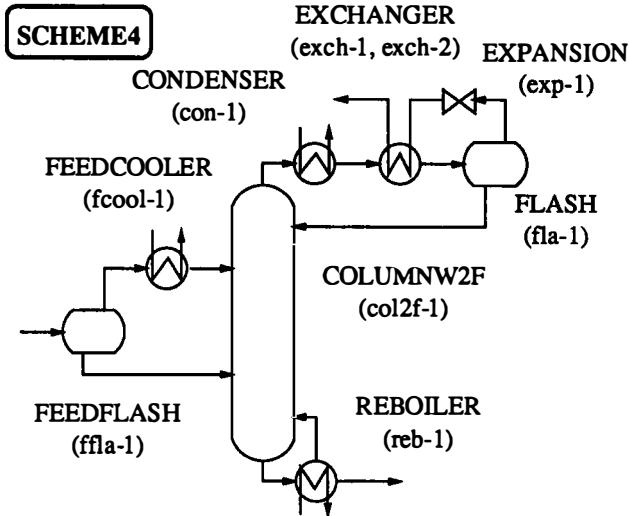


Figure 6: *Example of a scheme with associated units and equipment.*

added. There can be also user-specific checks.

- When a *scheme* is saved, it is not possible to modify it anymore. If the user wishes to explore an alternative topology, he must create another *scheme* with such a modification. This can be done by copying the original *scheme* into another one, and modifying the new *scheme* as desired. Note, though, that it is possible to attach new items of *equipment* to a *unit* at any point in time (as will be seen in the example problem, in Section 5.2).

Saving *schem¹* allows the user to backtrack to them in order to explore alternative designs.

Schema can be grouped in families called *Meta-Schema*. The user can then perform operations not only on a particular *scheme*, but also on a *meta-schema*. One *scheme* may be related to more than one *meta-schema*. Figure 7 shows five *scheme* related to *meta-schema* MSI, *scheme1bis* also related to MSII, and *scheme2*, *scheme3*, and *scheme4* related additionally to MSIII. Several grouping criteria can be used:

- a *meta-schema* for the *scheme* produced by each designer in the group,
- a *meta-schema* for the *scheme* in each line of reasoning followed,
- a *meta-schema* for the *scheme* created in a given time interval,
- a *meta-schema* for a group of *scheme* with common properties (e.g. the ones the user thinks are best),
- arbitrary groupings,

¹In the rest of the paper *schem^a* will be used as the plural of *scheme*.

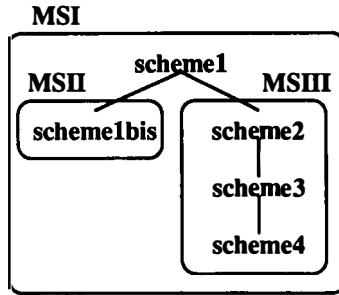


Figure 7: *Schema and Metaschema.*

- combinations of the above.

The global organisation of the objects we have introduced is presented in Figure 8.

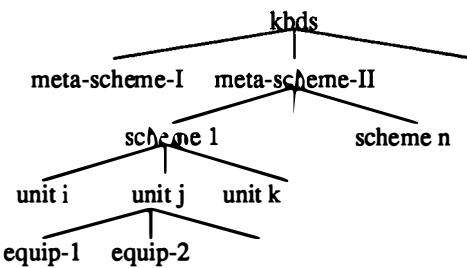


Figure 8: *Hierarchy of objects in the experimental implementation.*

So far, it has been explained how a designer can create and keep track of operating alternatives, structural alternatives and combinations of both. But the user can also control the generation of alternatives by declaring two or more objects incompatible with each other. Declaring an incompatibility amounts to telling the system that it need not consider an alternative where the incompatible objects are present. This property can be used, for example, in a situation like the one in Figure 4. The consideration of an alternative *equipment* in S_C can be propagated through the $S_C \rightarrow C_1$ branch, the $S_D \rightarrow D_1$ branch, and to S_A , but not to the rest of the nodes in the design space. Figure 9 shows the possible ways that the different objects in the system can be declared to be non compatible.

4.4 Assumption-Based Truth Maintenance Systems

The maintenance of alternative lines of reasoning in a system is often referred to as the use of “Contexts” or “Worlds”, and is exemplified by the work of

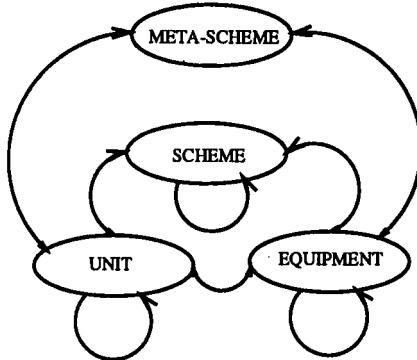


Figure 9: *Possible declarations of non-compatibility.*

Filman (1988) on KEE. Since Contexts can also express problem-solving states, he identified various types of situations that can be represented using them:

- Problem partitioning and recombination (modelling alternatives or changes to a particular state by creating child contexts).
- Checkpointing changes.
- Preserving search state (each time the search makes a choice, it creates a context with that choice).
- Incremental solution construction.
- Hypothetical reasoning.
- Reasoning with incomplete information.
- Reasoning about time and events.

Filman built a Context Mechanism using an ATMS.

Truth Maintenance Systems (TMSs, (Doyle, 1979)) are studied as an area of Artificial Intelligence. An extension to TMSs are the Assumption-Based Truth Maintenance Systems (ATMSs, (deKleer, 1986)), and the possibilities listed above indicate that they may be of great use to meet the requirements mentioned in Section 2. In particular, we are using an ATMS to represent the first four items in the list.

ATMSs characterise appropriately the multiple-context nature of the design process, where each context is inconsistent with the others. An ATMS is a tool for exploration, since it

- preserves deductions across environments (for situations where the same search state would be repeatedly discovered), and
- retains justifications for deductions (to be used during the explanation process, while guiding search, and in evidential reasoning).

5 EXAMPLE PROBLEM AND ITS REPRESENTATION

The attitude of designers describing the design process is similar to the one of mathematicians describing the theorem-proving process. In their published papers and textbooks the painstaking process of trial and error, revision and adjustment are all invisible. Designs and proofs are revamped and polished until all trace of how they were developed is completely hidden (Bundy, 1983).

For this reason, the literature on design does not abound with published examples on how a particular design was reached.

5.1 Description of the example problem

The example is a modified version of the design process presented by King *et al.* (1972). It is outside the scope of this paper to explain in detail the Chemical Engineering concepts involved in the design, but for the purpose of describing the design process, the following description should suffice.

The objective of the design is to remove hydrogen and methane from ethylene (the main product of the plant) and heavier hydrocarbons. The initial mixture is fed as the two phase (vapour and liquid) stream **s1**.

After a detailed evaluation of a proposed initial design (**SCHEME1** in Figure 10), it is found that the cost is too high due to the loss of some ethylene appearing in the hydrogen and methane stream (**s7**). An alternative process **SCHEME1BIS**

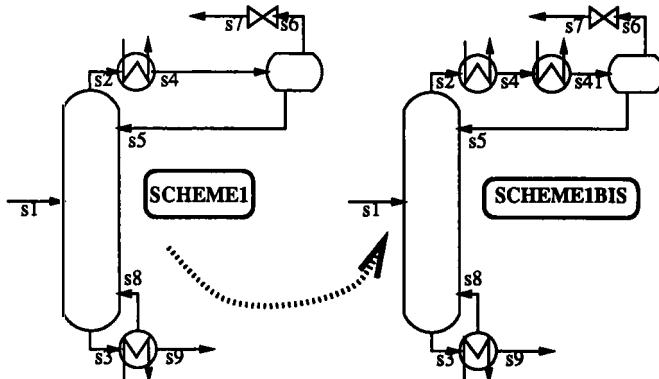


Figure 10: First branch in the exploration of the design space.

is proposed, where a cooler with refrigerant is added at the top of the column, but it is soon abandoned due to the even higher cost involved. This whole line of reasoning is abandoned, and a new set of alternatives explored.

In Figure 11 another alternative to the initial design in **scheme1** is proposed (**scheme2**). **Scheme2** is cheaper since it uses a stream already present in the process to cool **s4**, and less ethylene is lost, but its loss still accounts for most of the total cost. An additional modification is proposed which generates **scheme3**, where the amount of ethylene is reduced at the stream **s2** (and consequently at stream **s71**) by cooling the input to the column. Although this is the cheapest alternative so far, the dominant cost now is the operation of the cooler at **s1**. By cooling only the vapour part of the input stream, big savings can be achieved. This is done in **scheme4** by adding a separator at stream **s03**.

5.2 Results

The system is programmed in an object-oriented extension of Common Lisp. No graphical interface is used yet, so all the operators of definition, access and modification are applied using messages (evaluating lisp functions). The ATMS used in this work was developed in C by Peter Ross (1989).

In the example, the user commands are underlined, and comments about specific features of the system are in italics. The symbol “*kbds*>” is given by the system to prompt the user.

***kbds*> (scheme1)**

{*The description of the first proposed design is loaded from a file. Note that after the units are added and connected, the scheme is saved in memory. Then the units are associated with their corresponding equipment.*}

```

KBDS started
  MSCHEMEI is a meta-schema .
  Changing current meta-schema to (MSCHEMEI) .
  @ SCHEME1: add unit COLUMN of type COLUMN-1F-MLN-FLAVOR .
  @ SCHEME1: add unit CONDENSER of type HEX-SIMPLE-MLN-FLAVOR .
  @ SCHEME1: add unit FLASH of type FLASH-MLN-FLAVOR .
  @ SCHEME1: add unit EXPANSION of type EXPANSION-MLN-FLAVOR .
  @ SCHEME1: add unit REBOILER of type REBOILER-MLN-FLAVOR .
  @ SCHEME1: connect COLUMN (TOP) to CONDENSER (IN) .
  @ SCHEME1: connect COLUMN (BOTTOM) to REBOILER (IN) .
  @ SCHEME1: connect CONDENSER (OUT) to FLASH (FEED) .
  @ SCHEME1: connect FLASH (BOTTOM) to COLUMN (TOP-REFLUX) .
  @ SCHEME1: connect FLASH (TOP) to EXPANSION (IN) .
  @ SCHEME1: connect REBOILER (REFLUX) to COLUMN (BOTTOM-REFLUX)
  @ SCHEME1: save scheme topology .
    unit REBOILER : instantiate to REB-1 .
    unit EXPANSION : instantiate to EXP-1 .
    unit FLASH : instantiate to FLA-1 .
    unit CONDENSER : instantiate to CON-1 .

```

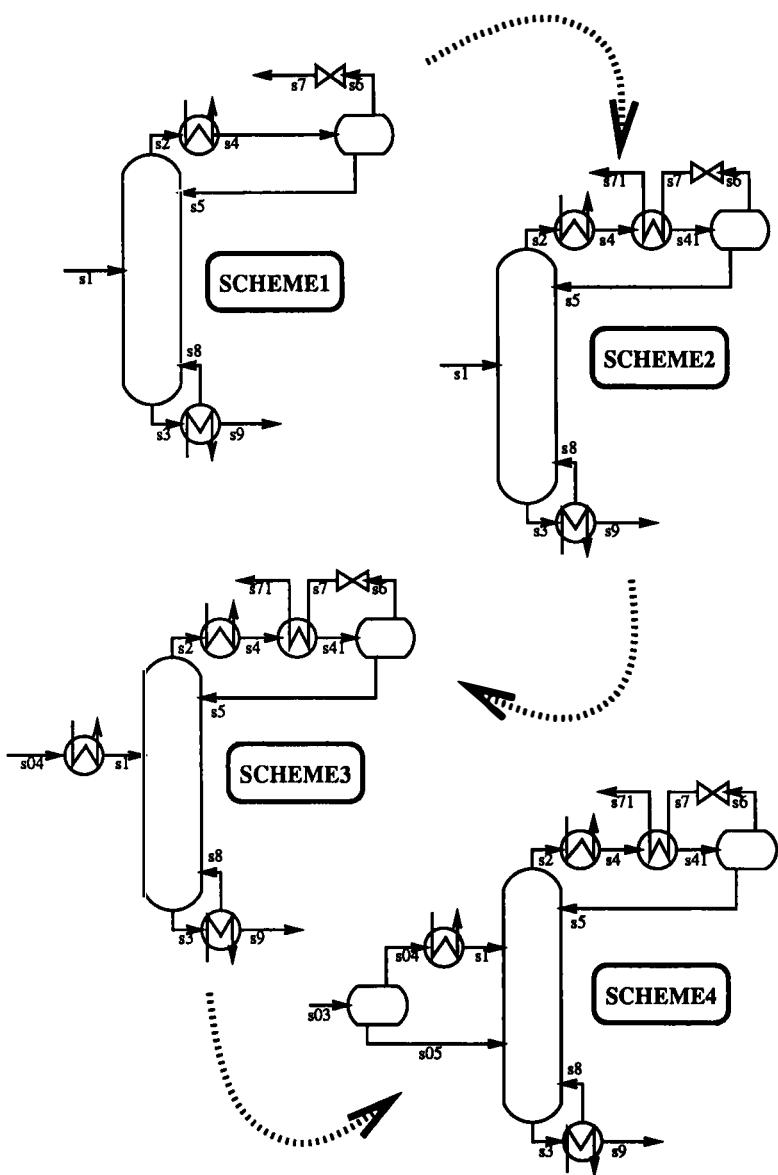


Figure 11: Second branch in the exploration of the design space.

```
unit COLUMN : instantiate to COL-1 .  
unit COLUMN : instantiate to COL-2 .
```

{Note that two different columns (e.g. with different number of stages) are going to be considered.}

***kbds*> (send scheme1 :alternatives)**

{Scheme1 has two alternatives, one for each alternative column.}

Alternatives for scheme SCHEME1:
(COL-1 CON-1 EXP-1 FLA-1 REB-1)
(COL-2 CON-1 EXP-1 FLA-1 REB-1)

***kbds*> (schemelbis)**

{A second scheme is created (scheme1bis) copying the structure of scheme1, and adding to it a new unit with its associated equipment (see Figure 10).}

```
MSCHEMEII is a meta-schema .  
Changing current meta-schema to (MSCHEMELI MSCHEMEII) .  
Copying scheme SCHEME1 into SCHEME1BIS .  
@ SCHEME1BIS: add unit TOPCOOLER of type HEX-SIMPLE-MLN-FLAVOR  
@ SCHEME1BIS: connect TOPCOOLER between CONDENSER and FLASH  
    @ SCHEME1BIS: disconnect CONDENSER from FLASH .  
    @ SCHEME1BIS: connect CONDENSER (OUT) to TOPCOOLER (IN)  
    @ SCHEME1BIS: connect TOPCOOLER (OUT) to FLASH (FEED) .  
    @ SCHEME1BIS: save scheme topology .  
    unit TOPCOOLER : instantiate to TCOOL-1 .
```

{... in the same manner, scheme2, scheme3 and scheme4 are loaded, simulating the decisions of a designer to explore different designs. Also, assume that the designer wishes to explore two versions of a heat exchanger to cool stream s4.}

...

***kbds*> (send kbds :alternatives)**

{List the total number of alternatives at this point. These are the alternatives found in Figures 10 and 11, and organised according to Figure 7.}

```
Alternatives for meta-schema MSCHEMELI:  
Alternatives for scheme SCHEME1:  
(COL-1 CON-1 EXP-1 FLA-1 REB-1)  
(COL-2 CON-1 EXP-1 FLA-1 REB-1)  
Alternatives for scheme SCHEME1BIS:  
(COL-1 CON-1 EXP-1 FLA-1 REB-1 TCOOL-1)  
(COL-2 CON-1 EXP-1 FLA-1 REB-1 TCOOL-1)  
Alternatives for scheme SCHEME2:  
(COL-1 CON-1 EXC-1 EXP-1 FLA-1 REB-1)  
(COL-2 CON-1 EXC-1 EXP-1 FLA-1 REB-1)
```

(COL-1 CON-1 EXC-2 EXP-1 FLA-1 REB-1)
(COL-2 CON-1 EXC-2 EXP-1 FLA-1 REB-1)

Alternatives for scheme SCHEME3:

(COL-2 CON-1 EXC-1 EXP-1 FCOOL-1 FLA-1 REB-1)
(COL-2 CON-1 EXC-2 EXP-1 FCOOL-1 FLA-1 REB-1)
(COL-1 CON-1 EXC-1 EXP-1 FCOOL-1 FLA-1 REB-1)
(COL-1 CON-1 EXC-2 EXP-1 FCOOL-1 FLA-1 REB-1)

Alternatives for scheme SCHEME4:

(COL2F-1 CON-1 EXC-1 EXP-1 FCOOL-1 FFLA-1 FLA-1 REB-1)
(COL2F-1 CON-1 EXC-2 EXP-1 FCOOL-1 FFLA-1 FLA-1 REB-1)

Alternatives for meta-scheme MSCHEMEEII:

Alternatives for scheme SCHEME1BIS:

(COL-1 CON-1 EXP-1 FLA-1 REB-1 TCool-1)
(COL-2 CON-1 EXP-1 FLA-1 REB-1 TCool-1)

Alternatives for meta-scheme MSCHEMEEIII:

Alternatives for scheme SCHEME2:

(COL-1 CON-1 EXC-1 EXP-1 FLA-1 REB-1)
(COL-2 CON-1 EXC-1 EXP-1 FLA-1 REB-1)
(COL-1 CON-1 EXC-2 EXP-1 FLA-1 REB-1)
(COL-2 CON-1 EXC-2 EXP-1 FLA-1 REB-1)

Alternatives for scheme SCHEME3:

(COL-2 CON-1 EXC-1 EXP-1 FCOOL-1 FLA-1 REB-1)
(COL-2 CON-1 EXC-2 EXP-1 FCOOL-1 FLA-1 REB-1)
(COL-1 CON-1 EXC-1 EXP-1 FCOOL-1 FLA-1 REB-1)
(COL-1 CON-1 EXC-2 EXP-1 FCOOL-1 FLA-1 REB-1)

Alternatives for scheme SCHEME4:

(COL2F-1 CON-1 EXC-1 EXP-1 FCOOL-1 FFLA-1 FLA-1 REB-1)
(COL2F-1 CON-1 EXC-2 EXP-1 FCOOL-1 FFLA-1 FLA-1 REB-1)

(MSCHEMEEIII MSCHEMEEI MSCHEMEEII)

kbds> (send reboiler :instantiate-eq 'reb-2**)

{What if a different reboiler were used? Associate the new equipment reb-2** to the reboiler unit.}

unit REBOILER : instantiate to REB-2** .

T

kbds> (send kbds :alternatives)

{The reboiler now has two associated equipments (reb-1 and reb-2**), and since it is present in all the schema, the total number of alternatives doubles. Here only the alternatives for scheme1 are presented.}

...

Alternatives for scheme SCHEME1:

```
(COL-1 CON-1 EXP-1 FLA-1 REB-1)
(COL-2 CON-1 EXP-1 FLA-1 REB-1)
(COL-2 CON-1 EXP-1 FLA-1 REB-2**)
(COL-1 CON-1 EXP-1 FLA-1 REB-2**)
```

...

kbds> (send kbds :eq&mscheme-noncompat 'reb-2** 'mschemeIII)
*{Declare reb-2** incompatible with all the alternatives related to mschemeIII. See Figure 9.}*

T

kbds> (send kbds :alternatives)
*{The existence of reb-2** propagates accordingly.}*

...

Alternatives for meta-scheme MSCHEMEII:

Alternatives for scheme SCHEME1BIS:

```
(COL-1 CON-1 EXP-1 FLA-1 REB-1 TCOOL-1)
(COL-2 CON-1 EXP-1 FLA-1 REB-1 TCOOL-1)
(COL-1 CON-1 EXP-1 FLA-1 REB-2** TCOOL-1)
(COL-2 CON-1 EXP-1 FLA-1 REB-2** TCOOL-1)
```

Alternatives for meta-scheme MSCHEMEIII:

Alternatives for scheme SCHEME2:

```
(COL-1 CON-1 EXC-1 EXP-1 FLA-1 REB-1)
(COL-2 CON-1 EXC-1 EXP-1 FLA-1 REB-1)
(COL-1 CON-1 EXC-2 EXP-1 FLA-1 REB-1)
(COL-2 CON-1 EXC-2 EXP-1 FLA-1 REB-1)
```

...

kbds> (send kbds :equips-not-compatible 'exc-1 'col-2)
{Declare exchanger exc-1 and column col-2 incompatible.}

T

kbds> (send scheme2 :alternatives)
{Scheme2 has now one alternative less.}

Alternatives for scheme SCHEME2:

```
(COL-1 CON-1 EXC-1 EXP-1 FLA-1 REB-1)
(COL-1 CON-1 EXC-2 EXP-1 FLA-1 REB-1)
(COL-2 CON-1 EXC-2 EXP-1 FLA-1 REB-1)
```

kbds> (send kbds :eq&unit-noncompat 'exc-1 'columnnw2f)
{Declare exchanger exc-1 and all columns associated with columnnw2f incompatible.}

T

***kbds*> (send scheme4 :alternatives)**

Alternatives for scheme SCHEME4:
(COL2F-1 CON-1 EXC-2 EXP-1 FCOOL-1 FFLA-1 FLA-1 REB-1)

6 CURRENT AND FUTURE WORK

Two main activities are currently being developed:

(a) Simulation.

Although the system in its current state is capable of performing simple simulation calculations, more general and accurate simulation facilities are needed in order to take full advantage of the tool. Performing simulations is of capital importance in order to have relevant evaluation criteria in the discrimination of the alternatives explored.

It is possible to connect the current system to individual packages written in several other programming languages (*e.g.* PPDS, a physical property prediction package written in FORTRAN (*IChemE*, 1982), calculation routines in C to solve equations numerically), but a more general connection is yet to be provided. The connection should be transparent to the user in the sense that he/she should not worry about which package to use, when and how to use it, and how to input its data and interpret its output.

The solution of this problem should not be difficult from the conceptual point of view, but is very labour intensive.

(b) Representation.

Some issues are not totally understood yet. In particular, whether or not it would be useful to explicitly discriminate alternative relations from refinement relations.

Also some implementation work to extend the system has to be done. For example, to let *schema* be grouped together to build up larger *schema* (aggregation is used now only to group *units* into *schema*).

A system that appropriately represents the object to be designed and the design process can be used as an experimental medium to learn about the process of plant design.

Once the representational issues are solved, we would like to add a solution-agent module to help the designer. In a very generic manner, we can foresee that this will require the development of modules that

- (a) Create and maintain a model of the designer's problem-solving strategy in order to provide him with directed support,**

- (b) Contain and apply knowledge about chemical plants, and
- (c) Contain and apply knowledge about the design process of chemical plants.

Work in the development of a user interface is also planned.

7 CONCLUSIONS

A representation that supports the exploratory nature of design has been proposed. The proposed representation addresses the management of the design process itself, letting the user control the generation, exploration, navigation and propagation of design alternatives. The representation is based on an ATMS. It also

- shows important data explicitly,
- is complete (can represent all possible solutions),
- is concise (eliminates unnecessary details),
- is transparent (easy to understand), and
- is easy to manipulate (create and access objects).

To apply the representation, two hypotheses were made: (a) the decomposition of information hypothesis, that proposes a similar approach to the way information is represented in the MINLP process synthesis technique, and (b) the identical object hypothesis, which is another incarnation of the AI frame problem.

Given the consistency and complexity problems that characterise the process of design, a tool to generate, record, access, and maintain alternatives during the design process will be of great help.

REFERENCES

- Bundy, A. (1983). *The computer modelling of mathematical reasoning*, Series in Computer Science, Academic Press, London, UK.
- de Kleer, J. (1986). An assumption-based truth maintenance system, *Artificial Intelligence*, **28**: 127–162.
- Doyle, J. (1979). A truth maintenance system, *Artificial Intelligence*, **12**: 232–272.
- Filman, R. E. (1988). Reasoning with worlds and truth maintenance in a knowledge-based programming environment, *Communications of the ACM*, **31**(4): 382–401.

- Grossmann, I. E. (1989). MINLP optimization strategies and algorithms for process synthesis, *Foundations of Computer Aided Process Design (FOCAPD)*, Snowmass, CO.
- King, C. J., Gantz, D. W., and Barnés, F. J. (1972). Systematic evolutionary process synthesis, *Industrial and Engineering Chemistry Process Design and Development*, **11**(2): 271–283.
- Kirkwood, R. L., Locke, M. H., and Douglas, J. M. (1988). A prototype expert system for synthesizing chemical process flowsheets, *Computers & Chemical Engineering*, **12**(4): 329–343.
- McCarthy, J. and Hayes, P. J. (1969). Some philosophical problems from the standpoint of artificial intelligence, in, B. Meltzer and D. Michie (eds), *Machine Intelligence*, Edinburgh University Press, UK.
- National Engineering Laboratory (1982). *PPDS. Physical Property Data Service*. The Institution of Chemical Engineers, Rugby, England.
- Ponton, J. W. (1990). Personal communication, November.
- Ross, P. (1989). *A simple ATMS*, Department of Artificial Intelligence, University of Edinburgh, UK, June.
- Smithers, T. (1989). AI-based design versus geometry-based design, or why design cannot be supported by geometry alone, *Computer-Aided Design* **21**(3): 141–150.
- Smithers, T., Conkie A., Doheny J., Logan B., Millington K., and Tang M. X. (1989). Design as intelligent behaviour: an AI in design research programme, *Research Paper DAI 426*, Department of Artificial Intelligence, University of Edinburgh, UK.
- Stephanopoulos, G., Johnston, J., Kriticos, T., Lakshmanan, R., Mavrovouniotis, M. L., and Siletti, C. (1987). Design-kit: an object-oriented environment for process engineering, *Computers & Chemical Engineering*, **11**(6): 655–674.
- Westerberg, A. W., Piela, P., Subrahmanian, E., Podnar, G., and Elm, W. (1989). A future computer environment for preliminary design, *Foundations of Computer Aided Process Design (FOCAPD)*, Snowmass, CO.

Can *planning* be a research paradigm in architectural design?

B. Colajinni,[†] M. de Grassi,[§] M. di Manzo[‡] and B. Naticchia[§]

[†]Dipartimento di Produzione e Costruzione edilizia
Università degli Studi di Palermo—Facoltà di Ingegneria
via delle Scienze 91128 Palermo Italy

[§]Istituto di Edilizia
Università degli Studi di Ancona—Facoltà di Ingegneria
via delle Brecce Bianche 60131 Ancona Italy

[‡]Istituto di Informatica
Università degli Studi di Ancona—Facoltà di Ingegneria
via delle Brecce Bianche 60131 Ancona Italy

Abstract. In the present paper we discuss the possible use of ‘planning’[°] to support architectural design. Architectural design is conceived as a process incorporating non-trivial subprocesses consisting of evolutionary sequences of drawings. The ‘planner’ can be viewed as a tool capable of managing the automatic development of formal descriptions of architectural objects, according to goals and constraints which are interactively assigned or removed by the designer. The main features of a ‘planner’ dedicated to architectural design are then put forward. The system has been implemented at the University of Ancona, Italy.

FOREWORD

The concept of *paradigm* in the field of scientific research was first introduced by Kuhn (Kuhn, 1962); it denotes an implicit frame or set of theoretical assumptions enabling the researcher to identify what is relevant to him and what can point to a fruitful area of research. The paradigm enables the scientist to organize and streamline the complexity of

[°]A distinction should be drawn between the meanings attributed to the words *planning* and *planner* in the field of Artificial Intelligence on the one hand, and in city and land planning on the other. In this paper, the terms *planning* and *planner* are used in their accepted meanings in the field of Artificial Intelligence.

reality and concentrate his research work on "puzzle-solving". Other authors (Boyd R., 1979) introduce the concept of *metaphor* to describe these research guiding principles.

In this paper we investigate to which extent the A.I. results in the field of Planning can be applied to analyse the quality-oriented design processes in Architecture with strictly logical research tools.

First of all, we could wonder why the A.I. planning paradigm should be so appealing. The reasons are the conceptual clarity and the computational effectiveness of both the evolutionary description of the problem domain and the deduction process the first is modelled as a set of action-based state transitions, the second as a goal-oriented decomposition of problems into simpler sub-problems. Thus the solution of a problem consists in the recursive decomposition of an overall goal into "trivial sub-goals" (i.e. goals with known solutions) which can be associated to an ordered set of basic actions whose execution changes the domain from the current state to the desired one.

The question is: can the *planning* be viewed as a *metaphor* of the process of Architectural Design in its totality or, at least, in some of its parts?

Also in the more conservative hypothesis, significant segments of the Architectural Design process can be modelled as a "goal achievement" process, through the development and execution of "design plans", possibly organised on several abstraction levels, within a well defined computational scheme. This would be a significant result from a theoretical viewpoint, with a strong impact both on the teaching strategies and on the critical analysis of architectural objects. Experiments could be carried out to emulate design operations pursuing given architectural qualities. The development of these experiments could lead to the introduction of a new language for the description of steps of the design process; in such a way many underlying aspects of the process could be made explicit.

The general framework and the specific constraints under which this assumption makes sense are discussed in the next section. Section 2 deals with the major characteristics of the planning system developed at the University of Ancona; its interpretations and applications in the field of architectural design are also examined. In the last section an example of automatic manipulation of drawings is shown.

1. Planning and Quality in Architectural Design

Over the last few years several research activities, based primarily on the concept of shape grammar (Stiny G., 1980), have been carried out to identify the specific architectural features or qualities of a set of works, that is, the style of a given author or age (Flemming U., 1987). At the same time, artificial intelligence techniques are being increasingly used in Architectural Design to cope with the combinatorial problems related to layouts, and to develop expert systems which provide assistance and control in building specifications (Rosenman M.A. et al., 1989).

The issue we are addressing is more general than the two approaches outlined above. The question is to which extent artificial intelligence can be a useful tool in emulating, the complex quality-oriented Architectural Design processes.

The production of Architectural Design instruments oriented to measurable quality criteria, has been effectually carried out with Operation Research techniques. We can include into this kind of approach any algorithmic generation as, for instance, optimization based routines (Mitchell W.J., et al. 1976) (Ligget R.S., 1980) (Radford A.D., Gero I.S., 1980) (Ligget R.S., Mitchell W.J., 1981). These techniques allow the manipulation of descriptive, not completely constrained structures in order to produce different instances by specifying the values of the domain variables. In this way the produced solutions are mutually comparable because they share the same descriptive structure. On other hand the generation of structurally different alternatives has been pursued with typically deductive approaches such as the combinatorial algorithms and the shape grammars (Stiny G., 1978) (Stiny G., 1980) (Mitchell W.J., 1989). Anyway the procedural generation processes are typically deductive, this is a serious drawback because, on the contrary, Architectural project can't be modelized as deductive process. Hence the increasing interest in Knowledge Based techniques using Artificial Intelligence concepts (Gero J.S. ed., 1984) (Coyne R.D., Gero J.S. 1985 a,b). In particular in (Flemming U., Coyne R.D., 1990) some A.I. techniques are used to automatically generate and evaluate drawing schemata, in order to implement a forward, heuristics directed search for suitable placements of rectangular objects; this process is organized on several levels of abstraction, where the term "abstraction" is basically intended as "scaling factor".

Over the last few years, a design emulation system, called CASTORP (Colajanni B, De Grassi M., 1989) has been developed at the University of Ancona; it is based on the cooperative activity of three expert systems (*reasoners*) dealing with three Architectural Design operations which can be considered elementary enough:

1. positioning objects in a delimited space;
2. searching an inventory for the closest architectural object, given a set of specifications;
3. reshaping an architectural structure.

The first reasoner handles the qualitative constraints expressing the desired relations among objects and between an object and the contour of the surrounding space. Of course the nature of the problem is independent from scaling factors: problems as placing furniture in a room or buildings in a given piece of land can be approached with the same conceptual model. Since the semantics of the relations handled by this reasoner is often vague (consider, for instance, relations as *near*, *far*, *over*, *under* and so on) it employs fuzzy representations (Ferrari C., Naticchia B., 1989).

The second reasoner has a more complex task to fulfil. The idea underlying the entire system is that, given a design problem, the search for its solution(s) may start looking at an

existing architectural object that is a suitable solution of a "similar" problem. Thus we need for a language describing both design problems and architectural objects which are solutions of problems belonging to the same class. Therefore this reasoner must handle complex descriptions embodying topological and geometric characteristics, as well as performance specifications and natural language connotations (Colajanni B., 1989). These descriptions are points of a space of features, where a minimal distance search can be performed in order to identify the object from which the design process will start. Actually the description of the object is rather coarse, as it takes into considerations only some topologic and geometric characteristics.

The third reasoner operates small geometric deformations, according to a sort of shape grammar, in order to satisfy those requirements that are not fulfilled by the object taken as starting point (De Grassi M., Di Manzo M., 1989).

The interaction among these reasoners results in a rough emulation of some design activities which, even if rather dumb, are nevertheless very frequent, boring, time consuming and cannot be performed without a valuable amount of inferential capability.

In a recent phase of the research, reasoners have been reconceived as planners; so in the following we will remind the main characteristics of planning in the A.I. sense and will expose our views about the possibility of employing planning techniques in the design process, considering the way in which quality is pursued in it.

1.1 Planning

A few words are now appropriate about the aspects of *planning* which are relevant to our problem.

Within a given domain, which must be formally described by means of a suitable language, planning is basically a technique for achieving a desired state of affairs, entrusting an "intelligent" machine with the task of developing the appropriate sequence of actions that can change the current state into the desired one. Thus planning relies on some knowledge about *goals*, *actions* and *domain states*. A goal is typically a (partial) description of the desired domain state, which the planning agent is committed to achieve. Domain states are usually described by symbolic languages (often first order logic), whose syntax and semantics can be unambiguously specified. The set of acceptable goals is related to a classified set of "partial plans"; each partial plan is a tool for expanding a specific goal into a proper sequence of action (the actual plan) or for decomposing it into sub-goals, which are recursively processed until the full actual plan is developed. Actions are "atomic" events, which do not require any further processing by the planner; the artificial agent which will execute the actual plan is assumed to have enough procedural knowledge to perform them correctly (note that the executing agent may or may not coincide with the planning one).

A fully developed plan looks like a tree, having the original goal as root, sub-goals as intermediate nodes, and actions as leaves. A partial plan is a (partial) description of a sub-tree (sub-plan), which singles out the sub-tree root (the partial plan purpose) and its direct sons (the partial-plan structural result).

Of course the Artificial Intelligence approach differs from the Operation Research one, in as much as it resorts to symbolic description, reasoning according to causal theories and hierarchical levels of representation, with no search for any functionally defined "optimality". At present, thanks to the research carried out on planning, rather efficient planners are available (Wilkins D.E., 1988). Their features are briefly outlined below:

- a) Deduction of context-dependent effects by means of causal theories: it allows a great simplification of the structure of partial-plans, which can be focused on "direct" effects of actions.
- b) Non-linear plans: they allow the decomposition of a goal into a set of sub-goals, with no commitment on the ultimate ordering.
- c) Hierarchical planning, i.e. the development of plans on different abstraction levels: a unique level of abstraction may force the development of many trivial details of the plan have to be developed.
- d) Partial definition of variables by means of constraints. The imposition of constraints upon dummy variables restricts the scope of the search for the referenced objects.
- e) Replanning: plan revisions according to heuristic principles.

Thanks to these features, a planner is a versatile research tool which can handle complex situations. It shows us a process evolving through a sequence of actions and eventually achieving a set of given goals. Can the designing process be assimilated to the pattern above?

1.2. Design as search for quality

The sentence that design is a process of searching for quality deserves some clarifying remarks.

1.2.1. What is quality in Architecture.

Quality is always a subjective matter. It can roughly be defined as the amount of praise a subject assigns to something submitted to his judgement. Criteria, of course, can be different from one subject to another, also in appraising the same thing. Objective quality is, really, nothing else than a convention that a set of people agrees on referring to the criteria of appraisal and, if possible, to the ways to apply them. Quality is not necessarily opposed to measure. As an instance, an architectural one, the amount of satisfaction of certain performances can be considered a quality and can be measured. When we speak of the problem of obtaining quality from a procedure or from whatsoever is related with the computer we speak implicitly of non measurable quality. But there is no intuitive meaning of the word quality but in the very general sense we have above said. So if we want to

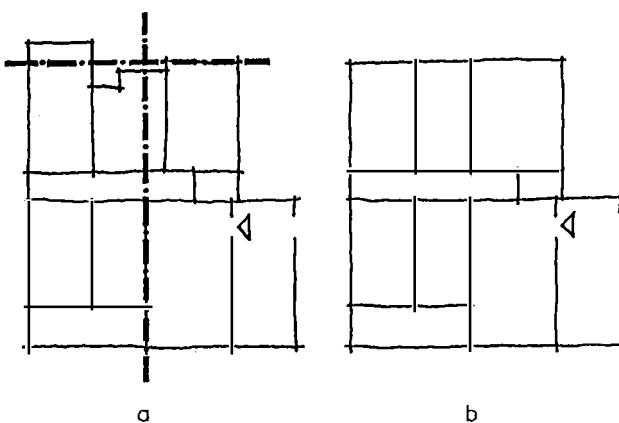


Fig.1

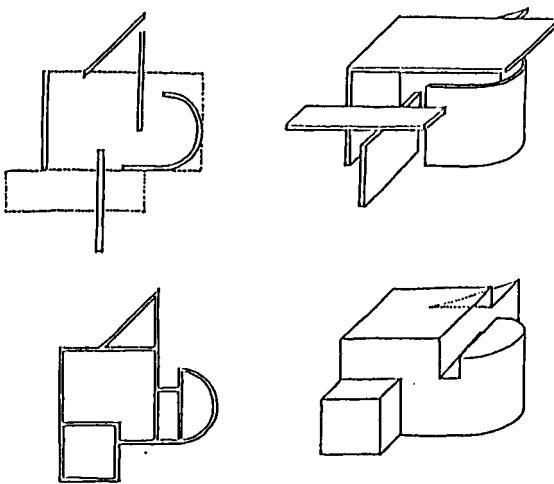


Fig.2

Fig. 1 Reshaping of a design solution resulting from the imposition of new constraints with respect to those which led to the current solution; the perimeter has been straightened and the day and night areas have been separated.

Fig. 2 Planning systems dedicated to architectural design have two classes of elements: spaces and enclosures. These may have not necessarily trivial relations.

match the capability of some procedure to cope with quality we have absolutely to explain which sort of quality we are referring to.

We think that non measurable quality, at least in the field of architecture, has much to do with linguistics and theory of communication. Roughly speaking, we can distinguish two types of quality: the quality of satisfied expectations and the quality of surprise, of realising completely unexpected but equally satisfying solutions to given problems.

The first kind of quality is well represented by the *style*. Quality is the amount of satisfaction of the set of rules that define the style. Quality is a question of coherence. It is, anyhow (clear but not useless to be underlined), the quality of imitators, of mannerists. Uncertainty and, consequently, the amount of information can lie only in the choice of the applied set of rules when different ones are possible. The achieved quality is, in any case, implied in the total set of rules even if non present in the conscience of the designer. The procedure can only bring it to the attention of the designer. If all the conditions of coherence are satisfied and we do not possess quantifying criteria of appraisal, appraisal is again subjective.

At present we don't imagine anything able to reach the second kind of quality. We can imagine only procedures that generate solutions, that can be given sense and hence quality, only *a posteriori*, that is on the base of complex processes of assigning meaning to new signs that, being new, were not present in the pre-existing code.

1.2.2. A metaphor of the design process.

We can imagine the project as a kind of "multi-tasking process" carried out simultaneously on several representation domains. On a creative level this process is carried out through the conception of a set of designer intentions, coming from a number of sources, as the expected performances, the user requirements, the building codes, the stylistic biasing, the desired quality and so on. Often this set of intentions is neither consistent nor complete. The project progress may point out contradictions (i.e. subset of intentions which cannot be concurrently accomplished in any practical architectural object); in such a case some intentions must be abandoned or replaced. On the other hand, it may be incomplete for at least two reasons. The first reason is the possible arising of new intentions, which are suggested by tentative partial solutions and can be consistently added to the active set. The second reason is the intrinsic "vagueness" of most of intentions, so that they cannot uniquely determine the corresponding architectural object; hence the possibility of a stepwise refinement of the set of intentions during the development of the project. Therefore intentions are a dynamic set; they can be considered as a "mental representation" of the architectural object, whose items may involve or not an actual commitment of the designer. Hence intentions may be conceived as abstract concepts, imaging fragments, overall fuzzy mental pictures or anything else. What is worth noting is the incomputable character of these representations. However the design process evolves also on a computable domain of formal representations, which are in a many-to-many relation with the mental ones. From a computational viewpoint, intentions manifest themselves through formal representations that we call drawings (the term "drawing" is intended in a broad sense, including any kind

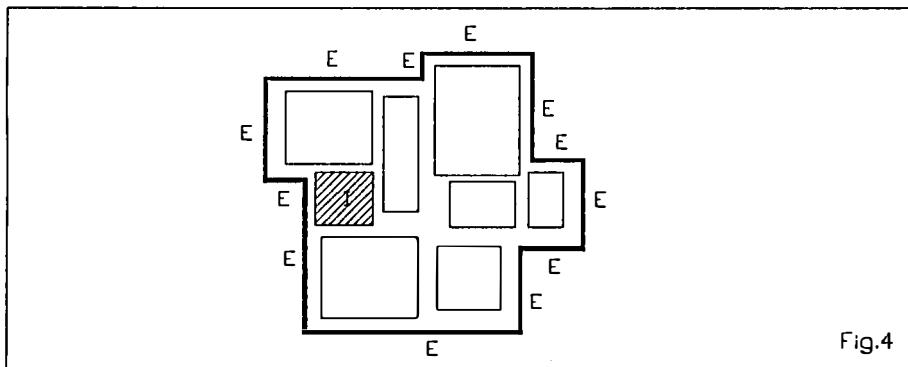
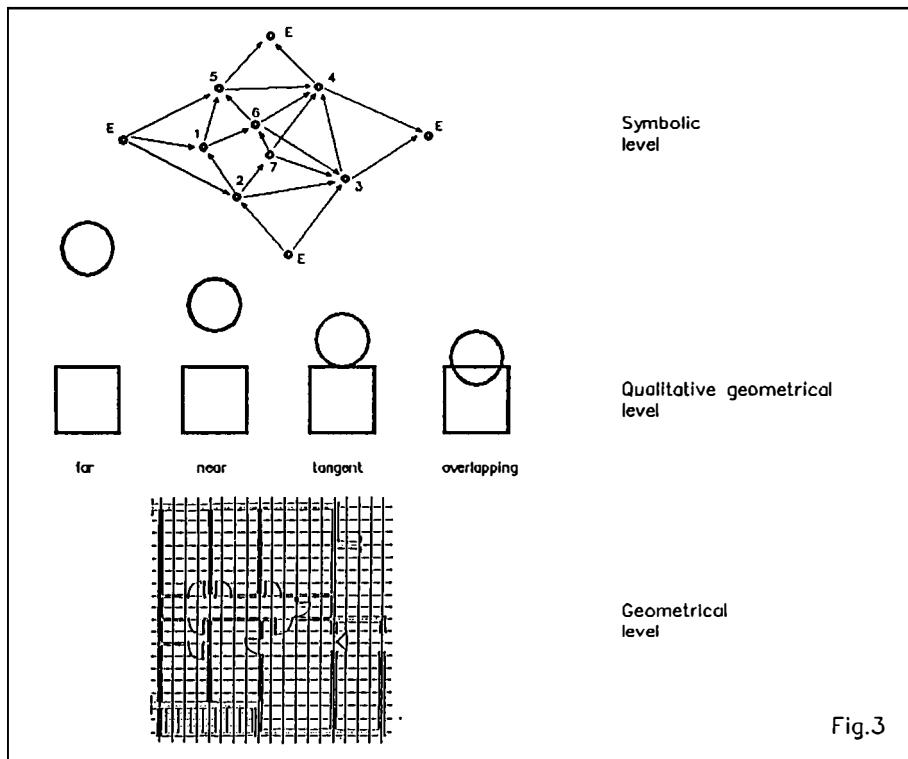


Fig. 3 Domains are described at three levels of abstraction:

- symbolic level, Above and Right relationships graph;
- qualitative geometrical level, relations stand for meaningful dimension ranges;
- geometrical level, metric description of the domain.

Fig. 4 The wire-frame description highlights the constraint of a complex perimeter resulting from the positioning of "token" elements.

of formal description on whatever level of abstraction). Drawings evolve as tentative partial solutions of subsets of intentions, each drawing specifying one among the possibly infinitely many design instances of an intention set.

1.2.3. What can a planner do.

Clearly an artificial machine can work only on formal representations. Hence, the space of mental representations is not accessible to the planner. However, the planner is not conceived as a stand-alone tool; on the contrary, it must be used interactively for generating sets of meaningful solutions to specific problems, coping with the step-by-step transformation of drawings.

Intentions generate goals and constraints for the planner, which tries to modify accordingly the current set of drawings (fig. 1). After each transformation, an appraisal is to be done of the state of the project; if it is not satisfying enough, a consequent reformulation of the set of goals and constraints is necessary in order to trigger the following transformation. In this sense, the mental representation evolves according to the space of drawing, drives the planner and can in turn be refined by possibly unexpected suggestions coming from it.

Then we can say that the planner *emulates* the behaviour of a design assistant, producing the evolution of the drawing triggered by the new set of constraints/goals. In this way the planner *simulates* the role of the assistant in the complex design process.

2. THE STRUCTURE OF A PLANNER FOR ARCHITECTURAL DESIGN

In this section we give a flash of the basic features of the planner implemented at the University of Ancona in conjunction with Mesarteam s.p.a. within the broader framework of the *Progetto Finalizzato Edilizia* (Building Construction Project) sponsored by Italian *Consiglio Nazionale delle Ricerche* (National Research Council).

Our planner looks at the structure of SIPE (Wilkins D.E., 1988) and makes (the best possible) use of Intellicorp's KEE tool. Since it shares with SIPE many fundamental choices, we deal only with the basic structural information needed to make self contained the example discussed in section 3.

The planner provides *descriptive schemata* which allow users to define the following:

1. *objects* and *object properties* ranked within taxonomies when required;
2. *goals* which can be hierarchically organised. Hierarchies typically stem from the decomposition of complex goals into simpler ones; this eventually leads to goals whose requirements are known;
3. *operators* (i.e. partial plans) organising the knowledge-base necessary to decompose goals;

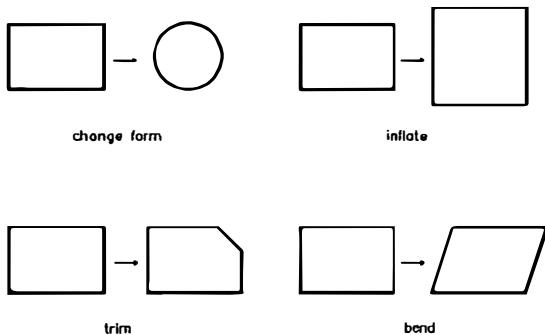


Fig.5

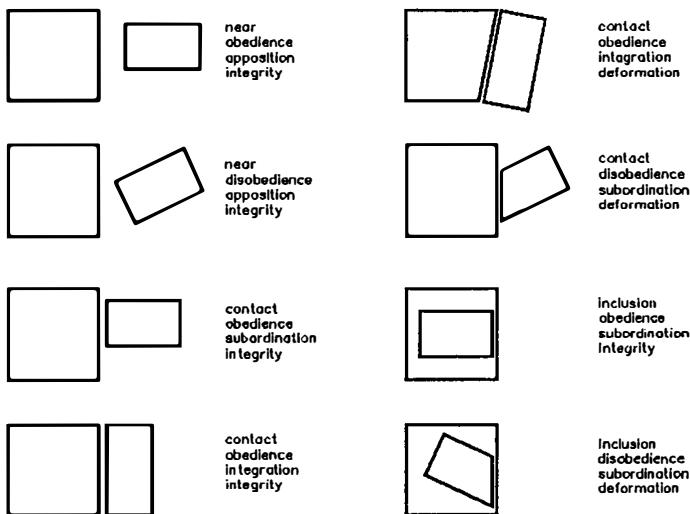


Fig.6

Fig. 5 Basic reshaping can be made by simple graphical editing actions. Four examples of this are shown.

Fig. 6 A dictionary for the description of given relations between objects can be defined. The dictionary can be expanded at any given time and the AS.PI.DE module can manage it through the manipulation of wire-frame descriptions.

4. *causal rules* for the analysis of side effects resulting from the application of operators;
5. *elementary actions* incorporating available procedural knowledge. These make up action sequences which, in turn, make up plans.

The planning process starts with the setting of a goal, then uses operators to decompose goals and ends when goal requirements can be met by elementary actions.

2.1. Operators.

Operators define partial plans; they are the basic tool for processing goals. Operators must incorporate knowledge about:

- their *purpose*, i.e. the goal that the operators can contribute to achieve;
- their *preconditions*, i.e. the constraints on the domain state which must be satisfied in order to let the operator be applied.

An example is given by the following instance of operator, which is meant to position an atrium room into a layout.

OPERATOR:	ATRIUM_INSERTION
ARGUMENTS:	<i>outdoor, indoor, atrium</i>
PURPOSE:	INSERTED(<i>atrium, outdoor, indoor</i>)
PRECONDITIONS:	DAY_CONNECTED(<i>indoor, outdoor</i>)
PLOT:	
GOAL:	COMMUNICATION(<i>indoor, outdoor</i>)
PROCESS:	
ACTION:	ALBA_PUT
ARGUMENTS:	<i>indoor, outdoor, atrium</i>
EFFECTS:	INSERTED(<i>indoor, outdoor, atrium</i>)
END PLOT:	
END OPERATOR:	

Where the fields have the following meaning:

- **PRECONDITIONS:** the precondition for the insertion of an atrium is the existence of a path from outdoors to indoors across the day area.
- **PURPOSE:** the operator is aimed at achieving the goal of modifying the layout in order to have an atrium inserted between an outdoor and an indoor areas.
- **PLOT:** breaking up of the purpose into sequences of sub-goals and elementary actions. In this case we have:

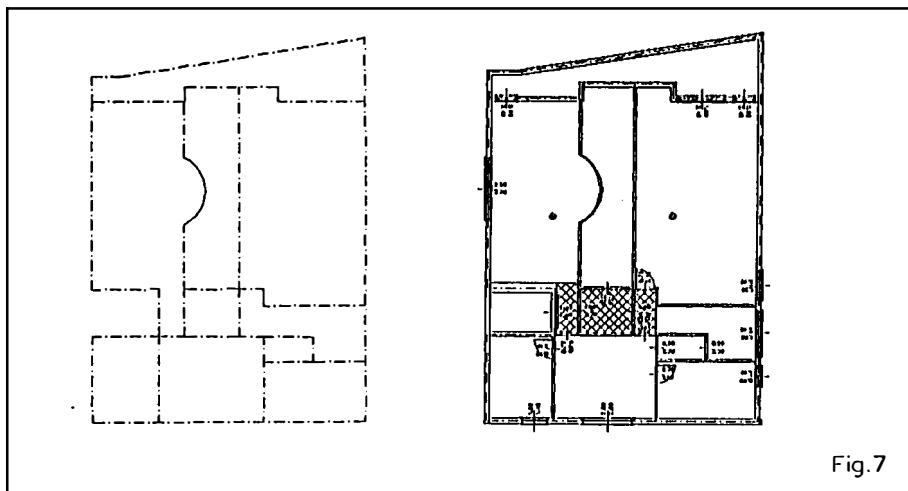


Fig. 7

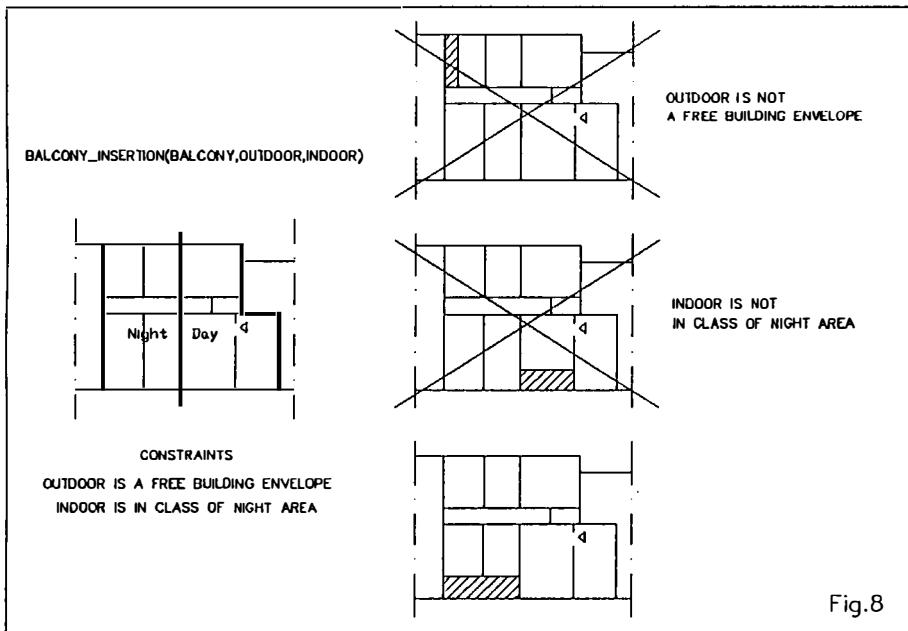


Fig. 8

Fig. 7 Architectural developments of the wire-frame diagram on the left are made using a domain-dependent causal theory. This interprets the schema by assigning architectural meaning to each sign; the description on the right results from the operation.

Fig. 8 Variable constraint management narrows the range of possible instances. Positioning a balcony in the layout may produce several solutions, some of which are not actually implemented since they do not comply with variable constraints.

- **GOAL:** the definition of a sub-goal. In our example, the sub-goal is the existence of a path between the indoor space and the atrium across the day area.
- **PROCESS:** the definition of a primitive action to be taken or an operator to be used. For the explanation of the action ALBA_PUT see the next section.
- **EFFECTS:** the definition of the domain effects of an action. This field is not the unique specification of domain effects. Possible sources of domain effects are:
 - 1) the goals set out in the plot;
 - 2) the effects of processes set out in the plot;
 - 3) consequences deduced from the theory of the domain.

1) and 2) are primary effects, while 3) is a side-effect.

2.2. Primitive actions

Our work in Architectural Design involve two types of elementary actions:

- inserting or removing space units (AL.BA module);
- reshaping space units (AS.PI.DE module).

2.2.1. Positioning objects: The AL.BA module.

The AL.BA module manages a sequence of elementary actions, each performing a specific placement of space units within the overall layout of the building bodies. Within AL.BA, the context is represented by a structural relation graph (Flemming U., 1989) describing the space structure, manipulated by means of relations of the following kind:

ABOVE (v, w): v directly above w

RIGHT (v, w): v directly right of w.

The description is then completed by the dimensional information related to the objects in the pattern.

Briefly, AL.BA represents the context using an orthogonal graph and a list. The input of the module is given by the geometric attributes of the objects to be positioned and a list of binary predicates expressing structural or dimensional relations between any object pair within the context. More complex operations, that can be broken down into sequences of elementary actions, can also be handled, provided an interpreter that can interface AL.BA to the system. AL.BA is capable of positioning new rooms into contexts whose description may be very complex. Positions can be fixed by inserting and then removing simulated elements. The enveloping space can be delimited by very complex boundaries, as shown in fig. 4.

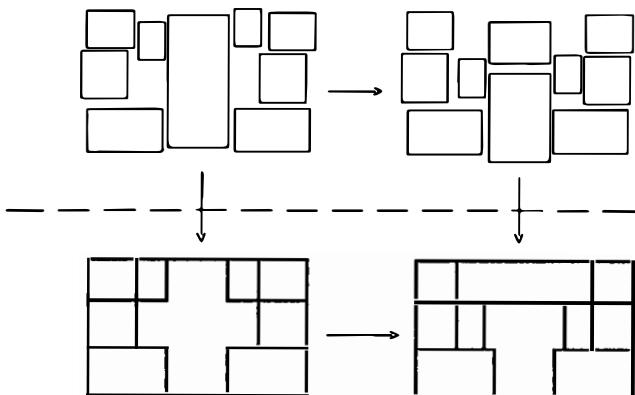


Fig.9

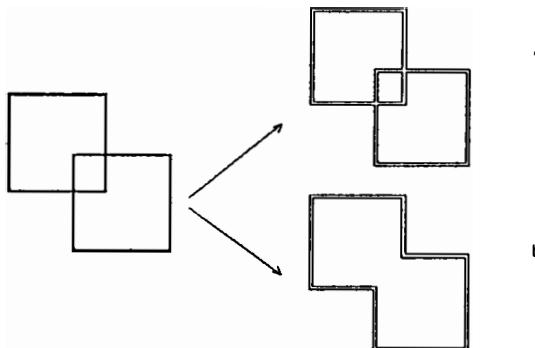


Fig.10

Fig. 9 A plan may contain nodes lying at different levels of abstraction. A complete interaction between symbolic and geometrical levels may constraint plan developments. For instance, planning at a lower (geometrical) level may be possible only when the work on the higher (symbolic) level has been completed.

Fig. 10 Drawings (a) and (b) are both derived from the architectural interpretation of the wire-frame diagram on the left, but they use two separate deductive theories. In (a) the Replanning module displays a conflict caused by the tectonic inconsistency of the description; in (b), instead, the Replanning module does not detect conflictive interactions of this sort.

2.2.2. Elementary geometric shifts: the AS.PI.DE module

AS.PI.DE is a module capable of producing small elementary shifts onto a wire-frame representation of draft designs. This means that the geometric attributes of the patterns are altered while leaving the structure of relations unchanged. If the shifting operations affect the semantics of some relations, a warning will be given. This warning is then processed by the Replanning module (see section 2.5).

AS.PI.DE shifts can fall into two broad categories:

- a) *reshaping* : actions that alter the shape of the pattern elements. Basic reshaping operations can be compared to the graphic editing operations performed by various CAD programs (fig. 5).
- b) *refinement of the relations holding between two elements*: Values defining meaningful intervals of metric attributes are assigned to the relations between objects.

Any two elements in the pattern can be linked by various types of relations: figure 6 shows an example where the distance between elements and the quality of the relation are described by suitable values. AS.PI.DE contains a basic dictionary for the description of attributes by means of a user defined set of geometric features of elements.

2.3. Deductive causal theory

In earlier planners, such as STRIPS (Nilsson N.J., 1982), both direct and indirect domain effects had to be exhaustively described within each operator by means of a list of predicates becoming true and a list of predicates becoming false. This approach forced the operator designer to describe in great detail the effects of the application of each operator. In particular, primary and side-effects were not detached and the operator descriptions were unduly complicated thus undermining the clarity of the whole. We decided to follow the philosophy of SIPE (Wilkins D.E., 1988), where operators list only primary effects and disregard secondary effects. Definitions are thus made far simpler and easier to understand. The domain transformations are made consistent by a set of "causal rules" based on primary effects. Changes made to the state of the world by applying a given operator are thus completed by deducting side-effects from primary ones. Figure 7 shows how deductive rules lead to the interpretation of a wire-frame draft in architectural terms: walls have real widths, there are doors and windows, etc.

At present, the AUTOGRAFICA module can interpret a wire-frame draft, it can automatically tell the outer envelope from partition walls and solve interactions between walls and other problems of the like.

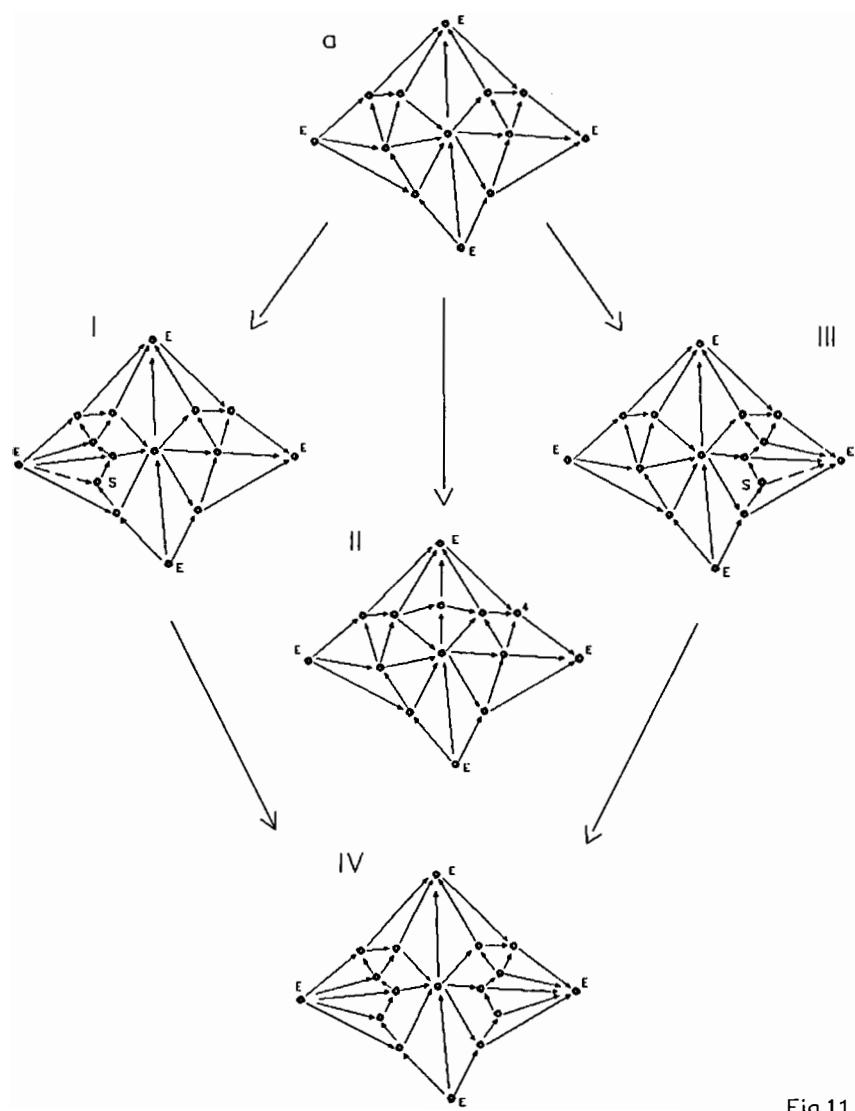


Fig.11

Fig. 11 Positioning an "atrio" in graph "a" - which represents Villa Malcontenta (Palladio 1560) - according to the variable constraints shown in figure, produces three possible solutions (I, II and III).
Plans I and III are not symmetrical, and must be revised. Both cases produce solution IV which - with small alterations - is close to the well-known Rotonda (1550), while solution II is similar to many other Palladian villas.

2.4. Hierarchical planning

Hierarchical planning is a central feature of planners. It leads to a better organisation of the domain knowledge, removing the need of reasoning about facts which are not really relevant to the current task. Hierarchies can be built along two different dimensions, i.e. *abstraction levels* and *planning levels*: The former deals with the levels of detail of the domain models, the latter sets a hierarchy of goals and sub-goals on the same level of detail, thus defining the structure of the plan tree (see, for instance, the decomposition of the goal "make a pattern symmetrical" in section 3).

In our case domains are made of spaces and space enclosures (fig. 2). A domain is described on three levels of representation, namely:

<i>SYMBOLIC</i>	(using graphs of structural relations)
<i>QUALITATIVE-GEOMETRIC</i>	(using qualitative geometry techniques)
<i>GEOMETRIC</i>	(dimensional description)

Figure 3 shows domain descriptions on the three levels.

Some of the relations describing a building structure are as follows:

ADJACENT (a, b) Space "a" is adjacent to space "b"

BOUNDING (x, y) Wall "x" bounds space "y"

while some rules are of the following type:

" a, b $\exists z$ (ADJACENT (a, b) \rightarrow BOUNDING (z, a) \ll BOUNDING (z, b))

which sets the constraint that any two adjacent spaces have a common wall.

The relations required to describe a material building object are set out in (De Grassi M., 1988). Virtual structures such as symmetry axes, modular gridlines, etc. can also be defined (Cochchioni C., Mecca S., 1989).

Theoretically, nodes working on different abstraction levels could be processed within the same planning level. This intermixing may cause some trouble in returning the truth values of predicates and faulty plans may be generated. For instance, on the *symbolic* level a set of relations involving rooms can describe relative space positions within a layout, while at the *geometric* level a space position may be described in terms of spatial coordinates. Whereas on the symbolic level basic layout choices can be made, only on the geometric level we can verify many compelling constraints such as whether the surface area meets the building

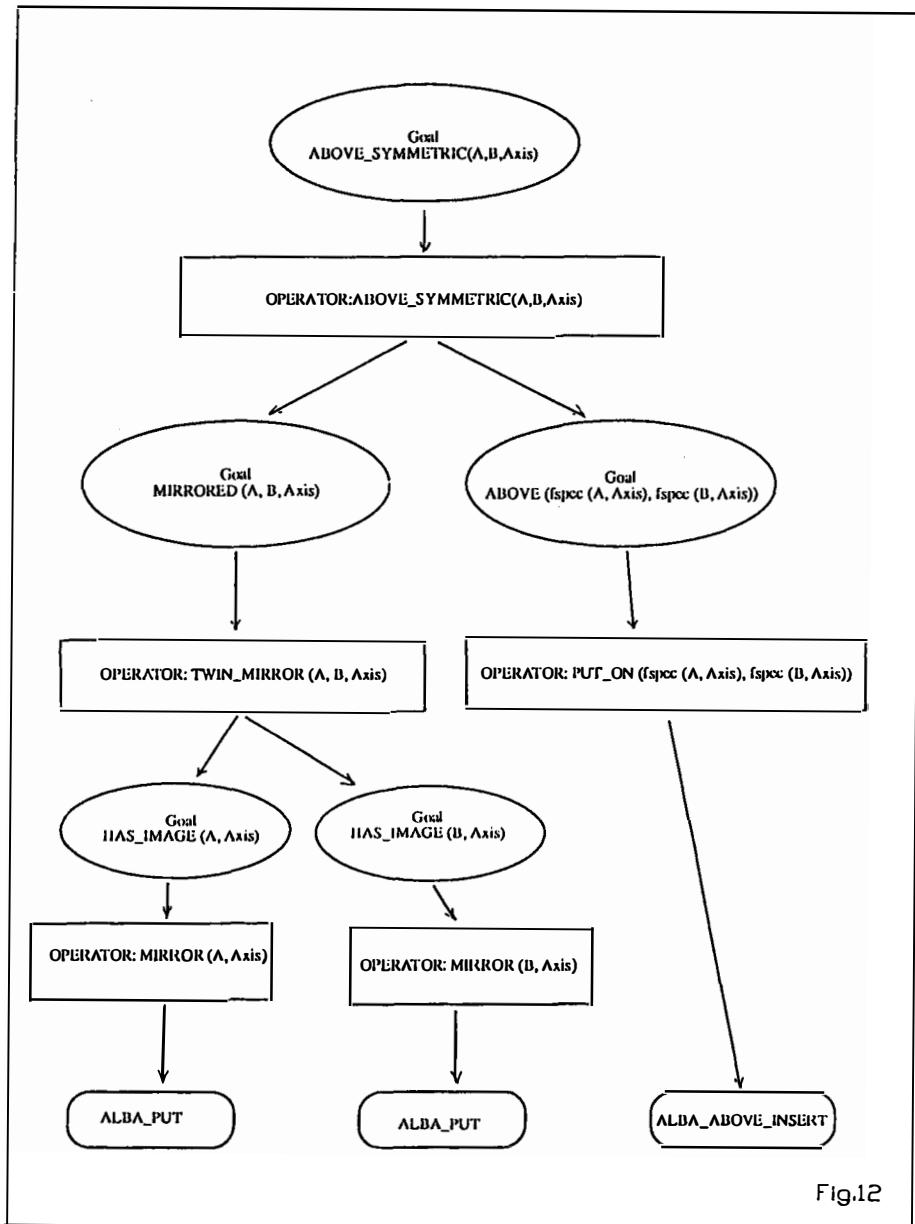


Fig.12

Fig. 12 Representation of part of the plan created by the **MAKE_SYMMETRIC** operator for the symmetrisation of a layout. Several plan levels can be noticed; these are used to break down the Goal until a sequence of elementary actions is reached. These actions can eventually be produced by AL.BA.

standards. The interaction between the symbolic and the geometric levels may put restrictions on the plan development modes (Fig. 9).

2.5. Replanning

The planner work can be analysed by other modules. These point out possible conflicts according to further criteria. If conflicts do occur, the plan may be rejected. This may have two major consequences, which can be interactively handled:

- 1) activation of a replanning level which calls for the planner to work on new goals or applies specified heuristics to correct the plan without having to start afresh;
- 2) activation of a backtracking mechanism exploring the alternatives disregarded in the original formulation of the plan and proposing a new plan.

After using the deductive rules to push interpretation even further, the Replanning module assesses predicates in such a way that architectural conflicting situations of a different kind are found. At present we are capable of formalising a few basic conflict categories:

- *tectonic conflicts*. Their assessment is based upon "consistency" principles which can be directly deduced from the design geometric description. Figure 10 shows an example of a tectonic conflict which can be noticed in the first, wire-frame architectural representation but not in the second one.
- *performance conflicts*. The analysis of quantifiable performance levels is a central aspect of design. This is almost always linked to the meeting of specifications.

3. AN APPLICATION

This section presents a sample application of the planner to a typical design problem: inserting an atrium into a given symmetric architectural layout by holding this property in the resulting pattern.

As we said before, in order to be manageable, any goal has to match an operator that makes explicit its solving structure by giving a description of its first level sub-goals decomposition. In the example, the operator defined in 2.1, called ATRIUM_INSERTION, has the role of describing the atrium placement. Other operators will be introduced in the following to match subgoals produced by the first one.

The plan typically evolves through the different abstraction levels defined in 2.1; however the example, exploits only the symbolic level, in order to simplify the discussion. Initial and final layouts, obtained by the application of ATRIUM_INSERTION, can represent well known Palladian Architectural cases.

3.1. The ATRIUM_INSERTION operator

The ATRIUM_INSERTION operator works on the symbolic representation level because on this level the searching for all the allowable positions of an element on a layout is highly effective. As a matter of course, the operator cannot take into consideration all the geometric implications of its actions; some solutions will therefore be inconsistent. The operator is applied to a start-up state described by a structural relation graph. It produces a new state where the room has been inserted according to the constraints imposed on the variables of the problem.

As we stated in 2.1, the arguments of the operator are:

- a room already placed on the start up layout, denoted *indoor*,
- a building envelope denoted *outdoor*;
- an atrium to be positioned.

At this step the operator positions the atrium, between the room and the building envelope, according to the constraints imposed on the variables of the problem. These are:

- (*indoor* is a LIVING-ROOM)
- (*outdoor* is a FREE BUILDING ENVELOPE).

The graph in Figure 11(a) is a symbolic representation of "Villa Malcontenta" (Palladio 1560) (fig. 13). By applying the operator to this graph, three possible solutions are returned (Fig.11 I, II and III; the drawings I and III are considered disregarding dashed relations and vertices S).

A conflict is generated when the designer introduces a new constraint, as for instance, a local symmetry around the path *indoor-atrium-outdoor*. In this case the drawings I and III require a new room be added around the path in order to resolve the conflict. This operation is carried out by an operator similar to ATRIUM_INSERTION.

The description of the situation resulting from the application of the operator is then submitted to the plan revision module (Replanning) which should spot possible conflicts generated by the insertion. As pointed out in section 2.5, the consistency of the predicates introduced by the Replanning module must be verified; in the case of resulting contradictions, the system must reprocess the plan. In our case, one of the constraints to check is the symmetry of the whole layout (a theoretical principle of paramount importance in Palladian Architecture). If the solution II is selected, this requirement is met: the "symmetry" predicate is true. On the contrary, in the cases I and III the resulting relation graph cannot represent a symmetrical layout and new planning step must then be run: a new operator is applied whose Purpose is to make the layout symmetrical and whose Preconditions are met by the description of the current state.

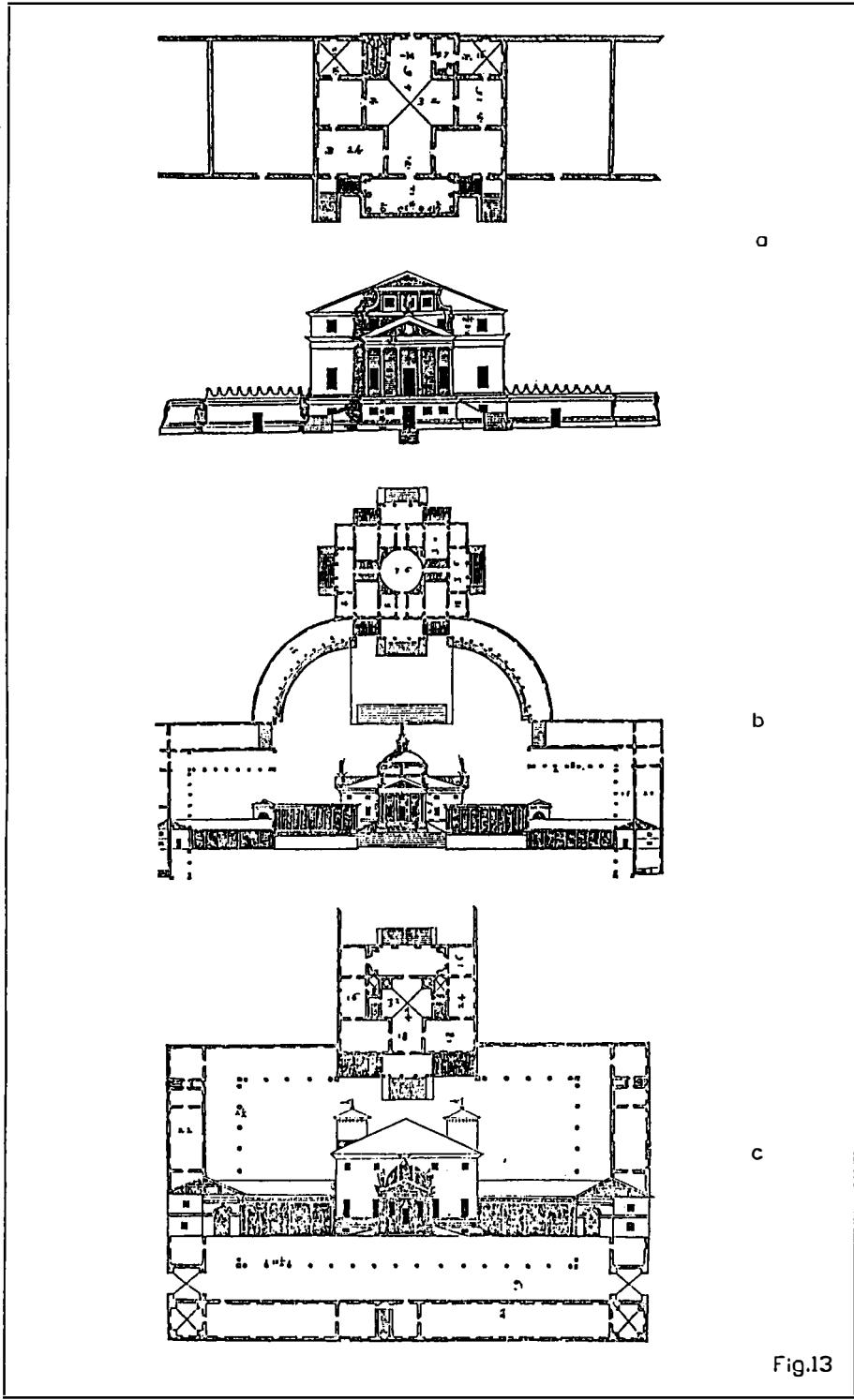


Fig.13

Fig. 13 Three designs of villas by Palladio (from "I Quattro Libri dell'Architettura")

a) Villa Malcontenta (1560), b) La "Rotonda" (1550), c) Villa Pisani (1562).

3.2. The MAKE_SYMMETRIC operator

The MAKE_SYMMETRIC operator breaks down the goal into:

- GRAPH_SYMMETRIC;
- GEOMETRIC REPRESENTATION_SYMMETRIC.

these subgoals determine two planning levels working on different abstraction levels. The first sub-goal is further decomposed by the operator MAKE_GRAPH_SYMMETRIC into two sub-goals:

- ABOVE_SYMMETRIC (a graph which is symmetric with respect to the ABOVE relation);
- RIGHT_SYMMETRIC (a graph which is symmetric with respect to the RIGHT relation);

Again, we will focus on the treatment of the first sub-goal. With reference to Figure 12, the ABOVE SYMMETRIC sub-goal is expressed in terms of a sequence of sub-goals ABOVE SYMMETRIC ($a, b, axis$) where variables a and b are instanced with all room pairs linked by an ABOVE relation. Each individual sub-goal is then matched by the operator ABOVE_MAKE_SYMMETRIC (see Appendix). This operator introduces a further planning level where two new sub-goals are generated:

- MIRRORED ($a, b, axis$); create elements which are identical (same attributes) but lie on the opposite side of the axis;
- ABOVE (fspec ($a, axis$), fspec ($b, axis$)); when it is requested by symmetry constraints, it puts the structural relation ABOVE between elements of the same side; "fspec" is a function returning a mirrored element.

The two sub-goals are captured by two operators sharing the same purpose:

- TWIN_MIRROR;
- PUT_ON.

TWIN_MIRROR must be decomposed into two sub-goals, each applied to one element. This is solved by the operator MIRROR which eventually reaches the elementary operation AL.BA can produce. The PUT_ON operator, on the contrary, directly reaches the elementary operation.

The application of the MAKE_SYMMETRIC operator to drawings I and III produces the drawing IV (fig. 11). This pattern is structurally similar to the "Rotonda" of Vicenza (1550) (Fig. 13).

Drawing II already verifies the symmetry predicate, therefore no conflict arises in the symbolic representation of the resulting state. In this latter case, the pattern is structurally similar to many of Palladian villas of which Villa Pisani Bagnolo is an example (1562) (Fig. 13).

4. CONCLUDING REMARKS

In this paper the hypotheses are discussed under which the tools provided by planning can be employed in Architectural Design assisted by A.I.

From the example shown it seems possible to draw some conclusion. As far as the transformations a designer wants to operate on a given drawing can be unambiguously expressed, they can be rewritten in terms of goals and hence handled by a planner. It is important to emphasize that unambiguity of formulation of the type of desired transformation doesn't necessarily mean unambiguous knowledge of the final result. On the contrary, usefulness of employing planning techniques stems from the possibility of obtaining, as previously stated, transformations that are directed towards the attainment of the general purposes the design desires.

The crucial points seem to stay in the capability of handling concurrently several abstraction levels in the domain representation and in defining effective, user tailored strategies for dealing with related partial solutions on different scaling factors.

At present the experiments have been carried on only at a symbolic level, though the symmetry operator is certainly one of the most difficult to develop.

Work is on in two directions: the implementation of a series of operators allowing a wide management at the symbolic level; the implementation of the operators that have to work at the levels of qualitative and quantitative geometry.

Experiments are currently carried out at the University of Ancona, within an Italian National Project sponsored by the National Research Council.

Acknowledgments

This work has been supported by the CNR-Progetto Finalizzato Edilizia.

REFERENCES

- Boyd, R. (1979). "Metaphor and theory change: What is 'metaphor' a metaphor for?" in *Metaphor and Thought* (Ortony A. ed.) Cambridge University Press, Cambridge.
- Colajanni, B. and De Grassi, M. (1989). "Inferential Mechanism to be employed in CAAD": The Castorp System" Proc. of the *ECAADE Conference 1989*: 7.1 , Aarhus.
- Colajanni, B. (1989). "*Unpossibile codice di rappresentazione degli oggetti edilizi*", RR 6 G.N.P.E., C.N.R. Ancona.
- Cocchioni, C. and Mecca, S. (1989). "Utilization of Rules for Modular Coordination in relational Models to be employed in CAAD" Proc. of the *ECAADE Conference 1989*: 7.5, Aarhus.

- Coyne, R.D. and Gero, J.S. (1985). "Design knowledge and sequential plans", *Environmental and Planning B*, vol. 12: 401, (a).
- Coyne, R.D. and Gero, J.S. (1985). "Design knowledge and context", *Environmental and Planning B*, vol. 12: 419, (b).
- De Grassi, M. (1988). "Una rappresentazione formalizzata dell'oggetto edilizio" *RR 3 G.N.P.E.* CNR Ancona.
- De Grassi, M. and Di Manzo, M. (1989). "The Design of Buildings as Changes of Known Solutions: A Model for 'Reasoner B' in the Castorp System" Proc. of the *ECAADE Conference 1989*: 7.3, Aarhus.
- Ferrari, C. and Naticchia, B. (1989). "Definition of spatial elements of the building system: 'Reasoner A' in the CASTORP system" Proc. of the *ECAADE Conference 1989*: 7.2, Aarhus.
- Flemming, U. (1987). "More than the sum of parts : the grammar of Queen Anne houses" *Environment and Planning B*, vol. 14: 323.
- Flemming, U. (1989). "More on the representation and generation of loosely packed arrangements of rectangles" *Environment and Planning B*, vol. 16: 327.
- Flemming, U. and Coyne, R.F. (1990). "A design system with multiple abstraction capabilities" Proc. of the *Expert system and their application*, General Conference, Avignon, France., vol. 1:107.
- Genesereth, M. and Nilsson, N.J. (1987). *Logical Foundations of Artificial Intelligence* "Morgan Kaufmann Publishers, Los Altos California.
- Gero, J.S.(ed.) (1985). *Knowledge engineering in computer-aided design*" North Holland, Amsterdam.
- Kuhn, T.S. (1962). *The Structure of Scientific Revolution*", University of Chicago Press, Chicago.
- Ligget, R.S. (1980). "The quadratic assignment problem: an analysis of application and solution strategies. *Environment and Planning B*" vol. 7: 141.
- Ligget, R.S. and Mitchell, W.J. (1981). "Optimal space planning in practice." *Computer Aided Design*" vol. 13: 277.
- Mitchell, W.J. and Steadman I.P. and Liggett R.S. (1976). "Synthesis and optimization of small rectangular floor plans". *Environment and Planning B*", vol. 3: 37.
- Mitchell, W.J. (1989) *The logic of Architecture*", Mit Press Cambridge.
- Nilsson, N.J. (1982). *Principles of Artificial Intelligence*" Springer -Verlag Berlin Heidelberg New York.
- Radford, A.D. and Gero, I.S. (1980). "An optimization in Computer Aided Architectural Design"*Building and Environment* vol 15: 73.
- Rosenman, M. A. and Balachandran, B.M. and Gero, J.S. (1989). "The place of expert systems in civil engineering" *Civil Engineering Systems*, E.&F.N. Spon, London, Vol 6 No. 1&2: 11.
- Stiny, G. (1978). "The Palladian Grammar". *Environment and Planning B*, vol. 5: 5.
- Stiny, G. (1980). "Introduction to shape and shape grammars." *Environment and Planning B*, vol. 7: 343.
- Wilkins, D. E. (1988). *Practical Planning: Extending the Classical AI Planning Paradigm*" Morgan Kaufmann Publishers, San Mateo California.

APPENDIX

OPERATOR: ABOVE_MAKE_SYMMETRIC
ARGUMENTS: $a, b, axis$
PRECONDITIONS: LEFT ($a, axis$), LEFT ($b, axis$), ABOVE (a, b)
PURPOSE: ABOVE_SYMMETRIC ($a, b, axis$)

PLOT:

GOAL: MIRRORED ($a, b, axis$)

GOAL: ABOVE (fspec ($a, axis$), fspec ($b, axis$)))

END PLOT

END OPERATOR

OPERATOR: TWIN_MIRROR
ARGUMENTS: $a, b, axis$
PRECONDITIONS: -- HAS_MIRR ($a, axis$) V -- HAS_MIRR ($b, axis$)
PURPOSE: MIRRORED ($a, b, axis$)

PLOT:

GOAL: HAS_MIRR ($a, axis$)

GOAL: HAS_MIRR ($b, axis$)

END PLOT

END OPERATOR

OPERATOR: MIRROR
ARGUMENTS: $a\ axis$
PRECONDITIONS:
PURPOSE: HAS_MIRR ($a\ axis$)

PLOT:

PROCESS: ALBA_PUT

ARGUMENTS: $a, axis, a'$

EFFECTS: HAS_MIRR ($a, axis$) $\wedge a' = \text{fspec}(a, axis)$

END PLOT

END OPERATOR

OPERATOR: PUT_ON
ARGUMENTS: a, b
PRECONDITIONS:
PURPOSE: ABOVE (a, b)

PLOT:
 PROCESS: ALBA_ABOVE_INSERT
 ARGUMENTS: a,b
 EFFECTS: ABOVE (a,b)
END PLOT
END OPERATOR

The impact of connectionist systems on design

S. Newton and R. D. Coyne

Department of Architectural and Design Science
University of Sydney
NSW 2006 Australia

Abstract. In this paper we present a discussion of the application of connectionism to design. The understanding provided by connectionism (as with other AI paradigms) as to how designers actually design (cognitive modelling) is considered to be relatively minor. We therefore present a challenge to the mechanistic metaphors on which cognitive modelling is based. In keeping with post-rationalistic views of cognition, design is well described in terms of non-computable metaphors such as 'play' and 'dialogue.' This view relegates connectionism (and other AI paradigms) to the realm of techniques for the development of tools, and curbs their use as explanatory devices. In this light we present some important features of connectionist systems and explore their use as the basis of design tools.

INTRODUCTION

Connectionist systems promote an interesting perspective on the confluence of artificial intelligence (AI) and design. Connectionism raises immediate questions about the extent to which mechanistic explanations can be brought to bear in our understanding of human behaviour. Can the mechanics of a connectionist model reflect human thought and hence design? The question of a correspondence between connectionism and thought has been canvassed by Rumelhart *et al.*, (1986a), McClelland *et al.*, (1986), Clark, (1989) and Lloyd (1989). In the more specific domain of design, see Coyne *et al.*, (1989).

There are two aims to this paper. The first is to extricate connectionism from the rhetoric of cognitive modelling through which questions such as that proposed above are frequently raised. The second aim is to present a case for the application of connectionist techniques to the development of useful design tools. The investigation of connectionism and its relationship to design provides a strong test case for the impact of AI on design in general.

The paper is presented in two parts. Part One introduces a challenge to the mechanistic view of design. Connectionism may be thought to offer considerable support for the view of 'mind as mechanism.' We present a discussion of the strengths and limitations of connectionism and its application to design. The notion presented is that cognition is better described in terms of the play of metaphors ('mind as metaphor').

In Part Two we introduce the notion of connectionism to those unfamiliar with the

topic. The background is explored from a short review of its history, and guiding principles. There is then a summary introduction to the mechanisms which operate in a simple form of connectionist system. Example applications of the connectionist model are then described in the context of practical, problem-solving techniques. The examples are necessarily simplistic. One of the dilemmas facing workers in this field is the opaque nature of the mechanisms which operate on more complex problem tasks. However, freed from the intractability of pursuing an abstract model, studied independently of designers, we explore the potential of the computer primarily as a tool for designers.

PART ONE: AI AND DESIGN

The confluence of AI and design implies various possibilities and confusions. One possibility is that we see design as an activity that can be modelled and we use the theories of AI to form the basis of this modelling. In doing so we presume that design is something that can be considered in the abstract (somewhat like logic and mathematics) as able to be studied independently of designers (or logicians and mathematicians). So we talk about design as involving such components as state space search, being goal directed and involving non-monotonic reasoning. AI provides the ideas for the models, and the correspondence between models and how designers design is considered relatively unimportant. Some of the claims made for the formal theory of shape grammars (Stiny, 1980; Mitchell, 1990) provides an illustrative example of this. There is no claim that designers use formal rewrite rules while designing, but rather that design as an abstract phenomenon can be so described. This approach to modelling design activity in the abstract is tested on how well the product of the model matches designs produced by other means. Does a 'Palladian' shape grammar really produce a Palladian design? To some extent all modelling operates in the abstract, but the lack of focus given to matching process raises important questions about what it is that is actually being modelled in this approach.

Another possibility of using AI is as a means of modelling designers themselves. This takes us into the realm of cognitive modelling and cognitive science. The motivation for adopting cognitive models is often their biological or psychological plausibility. Except in very restricted cases however, and notwithstanding the efforts of the behavioural sciences, there is little chance that such models can be tested through empirical studies comparing predictions derived from the models with the behaviour of designers. So one of the principal tests of any model is whether or not it furnishes us with understandings that lead to better teaching and better design practice. A further test of a model is whether or not it leads to better computer systems, as we employ cognitive models to gain an understanding of the designers who use the systems. The idea is that, armed with an appropriate 'user model,' we should be better able to design computer systems that interact with designers.

Another possibility is that we look to the tools and techniques of AI (such as symbolic processing, logic, rules and search) to form the basis of computer systems that provide better design support. The emphasis in this case is on computer-based tools to assist designers. Here we may regard shape grammars or other rule-based formalisms as useful devices for generating ideas for designers, or 'AI enhanced' optimisation systems (Balachandran, 1988) as useful tools for evaluation. Of course, the issue of designing and

evaluating design tools is not straightforward. In the same way that the ‘need’ for a hammer cannot be considered independently of the existence of hammers, so it is with design tools. It could be argued that empirical studies—such as that demonstrating the preferences of computer operators for icons versus pull-down menus (Benyon, 1990)—do not capture more than a frozen moment in the development of computer culture. The subject of the usefulness of tools for designers is essentially a projective and speculative exercise.

There has developed a convenient confusion about which of these ways (and there may be others) it is intended that AI be used in design. The confusion is convenient because it means that in our defense of an idea or theory we can traverse freely through these alternative meanings and take whichever is most appropriate to our argument at the time. In this paper we take as our point of access one of the areas of possibility outlined above—modelling cognition.

Mind and Connectionist Machines

As a subfield within AI, the claim of connectionism is that there is some similarity between a connectionist system formulated in a computer and the structure of the brain. Although they are acknowledged as capturing no more than the most rudimentary properties of the nervous system, the patterns of connected units with their activation values, weights and thresholds are thought to carry some biological credence—more so than symbols, rules, frames and search (the paraphernalia of ‘classical cognitivism’). Connectionism very readily suggests that we are getting down to the ‘fundamentals’ of cognition and therefore of design. In the following sections we will pursue the nature of this argument in some detail. However, the motivation for this argument is not primarily an attempt to develop some general commitment to connectionist systems, over and above the various other AI techniques applicable to design. (Although we do present a case for extending the application of connectionist techniques in the development of design tools.) Rather, the commitment we wish to develop is one in which the significance of suggestions such as “we are getting down to the ‘fundamentals’ of cognition and therefore of design” is given less importance. This paper develops the argument that cognitive modelling is not of primary importance.

Connectionist models are sometimes referred to as ‘neural nets.’ The term derives from similarities between a connectionist representation and the physiology of a human brain. Indeed the idea for a connectionist system stems originally from studies of brain physiologies. Units in a connectionist system are intended to equate with the neurons of a brain; weights are intended to equate with properties of the synaptic tissues which link between neurons and which together compose the brain structure. From studies we now know that the human brain comprises in the order of 10^{11} neurons and 10^{15} synapses. Many varieties of neurons have been identified. It would appear that even basic operations of a human brain are orders of magnitude more complex than the simple collections of units which comprise even the most sophisticated of current connectionist systems. Evidence is also strong that, however complex, it is not physically possible for one of the most powerful forms of connectionist model (that based on back propagation) to operate in the human brain. Along with this, Crick and Asanuma, 1986 mention several other properties of connectionist systems that are yet to be observed in any living system. At a fundamental level then, connectionist systems appear to do things in a way that the human mind does

not.

It is also common to refer to connectionist models in terms of parallel distributed processing (PDP) systems. It should become apparent from the descriptions to follow, that connectionist models are intended to use both parallel and distributed processes. Indeed, there is a rapid expansion currently in the number and range of parallel computer architectures being used to implement connectionist models. However, the vast majority of connectionist systems are still, and are likely to remain for some time, implemented within a serial architecture.

Fortunately the problem is not a serious one. Apart from those systems intended to model the physiology of the brain directly, there is no actual requirement for parallel processing, other than in terms of efficiency. Effective approximations of a parallel environment can be produced with relative ease within a serial architecture.

The distributed representation is critical to connectionist models. Many of the distinctive features of connectionist systems are hinged on the notion that units are non-symbolic: that inputs are not pre-processed or mediated, and that categories are emergent rather than 'hard-wired.' Consider the alternative to a distributed representation. At this alternative extreme, all representations would have to be local. Each unit is dedicated to a specific 'thing' (such as a concept, hypothesis, action or entity). The activation of that single unit is a complete display of how the system 'views' that thing. Thus the overall view of the system is manifest in the individual view of the unit. If that unit is removed then the system no longer has a concept of that thing.

If all representations were local, then the reasoning mechanism (such as sifting, chaining, storing and recalling associations between concepts) would have to reside in the connections between units. The human brain would also be entirely symbolic. If the human brain is entirely symbolic, with each neuron representing a single thing, then not only would we have a finite memory (10¹¹ 'things'), but we would also tend to demand a fixed hierarchy of understanding. For a symbol to have meaning it would have to be related to some other symbol, forcing a regress up the hierarchy to some all defining 'grandmother' neuron. (This is the symbol grounding problem of Harnad, 1989.) The alternative is some form of dialectic (Newton, 1987) or narrative (Lloyd, 1989), where meaning is dynamically determined by human behaviour. The more a representation tends to be local, the more the active processes of thought define the symbols and the less of the process is determined by the mechanical structure.

Consider instead the completely distributed representation. No single unit, in isolation, is associated with either a specific input or a specific output. To give any interpretation of the active units, all units (both active and inactive) must be considered together at once. Individual units will participate in many representations. There are problems with a completely distributed representation. Where, for example, are the patterns of activation categorised? Even at the practical levels of mechanisms, the completely distributed representation suffers considerably from cross-talk, communication, invariance and the inability to capture structure. (See, for example, the description of holographic models by Feldman, 1989.)

If all representations were distributed then the interactions between units would never, and could never, be interpreted. This strongly suggests that human cognition is an emergent property, or an epiphenomenon. The connectionist model would not then be modelling cognitive mechanisms, but the primitive (sub-conscious) operations available to cognition.

Clark (1989) develops the hierarchical nature of this model at some length.

In Part Two we present some examples of connectionist systems. None of the examples are completely distributed. More fully distributed examples tend to come from the realm of pattern recognition and systems such as NETtalk (Sejnowski and Rosenberg, 1986, 1987). Even there however the necessary non-symbolic nature of the units is questionable (Steels, 1989). Increasingly the extreme positions of localised and distributed representations are being questioned. The limitations of both positions are being recognised.

The middle ground acknowledges that at some level a connectionist system must be symbolic. The preferred level is variously termed the feature or microfeature level. Individual units or collections of units are representations of features, where multiple interactions between features constitute categories or 'things'. These categories constitute the emergent level of cognition, interpreted at the individual feature (unit) level. The units and weights which compose a connectionist model are therefore considered as models at the level of microcognition (Clark, 1989).

What understandings of design beyond the 'fundamentals' considered thus far are furnished by connectionism? The value of connectionism in explaining certain 'design phenomena' is evident in the case of schema representation. As will be explained in Part Two, the schema is an important concept in models of cognition (Rumelhart *et al*, 1986b) and in understanding the design process (Lansdown, 1987; Gero, 1988). In design, a schema can be characterised as a description that applies to a class of objects along with the knowledge by which instantiations can be made within that class. So schemas can be represented explicitly within hierarchies in a computer, as in frame-based systems (Minsky, 1975). The difficulty with such representation systems is that there are many ways in which they can be set up, and design often seems to involve the instantiation of descriptions that cross the boundaries between the schemas of such systems (Newton and Logan, 1988). In contrast, for schema modelling based on connectionism there are no rigidly defined schema boundaries. If there is any basis to the claim that connectionist systems are firmly grounded in cognition, then schema modelling based on connectionism has implications for understandings of human design processes. Some of the major implications can be summarised as follows (Coyne, 1990):

- (i) Connectionist models enable us to provide a more formal account of the notion that design solutions can appear to emerge in a single step without recourse to formal reasoning.
- (ii) The primacy of a rich memory of specific instances required by connectionist models suggests that there is perhaps less of a need to articulate and externalise sophisticated generalisations in coming to terms with and communicating design knowledge. This suggests that there are aspects of effective design that do not even require rules and reasons—just good examples.
- (iii) The connectionist idea suggests that novelty can arise from the prosaic. There are no special mechanisms, or special training, required by which new ideas (at least new combinations of ideas) can emerge from a memory-based system 'schooled' from a store of competent examples.

Notwithstanding the explanatory force of connectionism, such explanations are essentially

atomistic, and assume that at some level in a connectionist system there are basic units that map onto atomic 'features.' As discussed above, such atomic features are notoriously elusive. More significantly however, connectionism is based on a computational metaphor. Computational metaphors applied to human behaviour at any level are coming under increasingly attack from many quarters (Dreyfus, 1987; Searle, 1987; Winograd and Flores, 1986).

Problems with Mind as Mechanism

As with the projective exercises of Hofstadter and Dennett (1981) it is tempting to entertain ideas that machines based on connectionist architectures may some day design, be creative and generally think. Apart from the difficulties outlined above, many of which may be characterised as 'technical,' there are difficulties posed to understanding design by the reductive tenets of AI and cognitive science generally. The difficulties of any such view are typical of those found within materialism, rationalism, logical positivism and ultimately Cartesian dualism.

An obvious manifestation of the difficulties is found within design, namely the disparity between the mechanistic view of cognition and that which is left out by this view—such as the social-interactive nature of design, design discourse, the cultural context of design decision making and evaluation, and the phenomena of artistry and intuition. The difficulty is not that these considerations are ignored (that AI researchers are unaware of the social context of design or the role of artistry) but that these considerations are something separate and unaccounted for. There are many ways in which these difficult areas are separated off from normal research interests. For example, they may be relegated to parenthetical observations of how things really appear, they may be relegated to the realm of 'future research,' or they can be considered as 'epiphenomena' of large and complex systems that have not yet been worked out. In any event, as they are unaccounted for by the theories they do not enjoy the same privileged status as that which the theories do explain.

In design this split is also seen in terms of entities such as logic and artistry, between that which we can talk about, and that about which we must remain silent—"Whereof one cannot speak, thereof one must be silent" (Wittgenstein, 1922, ¶7). It is also common to separate scholarly activity from unsupported private opinion. Such divisions are particularly annoying and frustrating in research into formal systems, as we would prefer our theories to be explicitly all embracing, or at least to show the possibility of extension to these areas of difficulty. They also pose difficulties in a teaching or practice context as they tend to define the areas of importance. For example, because formalist theories are good for manipulating propositions and shapes we may see these as more important than the consideration of social context.

Many of these difficulties and confusions within the research programmes in which AI and design are entertained are typical of those faced within the 'epistemological tradition' generally. This is a tradition from which the 20th century is being slowly disengaged as the impact of philosophers such as Heidegger, Gadamer, Lyotard, Rorty, Bernstein and Habermas is felt in the areas of science, social science, technology, computer science and design. It is beyond the scope of this paper—neither is it necessary—to embark on a detailed critique of 20th century thought in relation to mind. (Rorty, 1980 presents a powerful and fundamental challenge to the commonly accepted views about mind and

body.) The difficulties of AI and design alluded to above can be circumvented by the simple act of displacing logic and mechanism as primary explanatory devices in dealing with human cognition. This is a difficult step within the realm of AI research as an explanation is not generally considered interesting unless it has its roots in some form of mechanical behaviour.

Mind and Metaphor

A deceptively simple and perceptive argument for extricating us from the imperative of machine and logic-based explanations is developed by Schön, 1963. He describes decision-making (for which we may substitute the word ‘cognition’ without deviating from the object of study) in terms of different metaphors. (Lakoff and Johnson, 1980, provide an extensive catalogue of such metaphors.) The most common metaphors are those of commerce and the market place. We talk of weighing up options, holding ideas in the balance and weighing the evidence. There are also metaphors of tools: sharpening wits, honing skills, and getting to the point. There are metaphors of society and militaristic metaphors. We speak as if the mind is made up of independent personalities engaged in dialogue (“I talked myself into it”), or of reason winning over from emotion. A fundamental metaphor is that in which we construct atomistic cognitive entities such as concepts, memories and experiences. In the light of these examples it seems reasonable to extend the role of metaphor beyond atomistic entities, to view the mechanisms which manipulate entities themselves as a further source of metaphor. Explaining cognition in terms of symbols, rules, procedures and state space search is little more than a sophisticated extension of the metaphors by which we speak of ourselves as getting wound up, experiencing overload, processing data, letting ideas simmer, keeping on hold, cooling down and turning off. Having extended the role of metaphor thus far, it is but one more step to see the application of logic to cognition as the application of yet another metaphor. Of course the great strength of mechanistic and logic-based metaphors is that they suggest methods by which decisions can be made. They involve initial states, relationships between variables, and outcomes. In this way, they also suggest implementation within computer systems. They therefore appeal to our desires to predict, manipulate and control.

The foregoing is but one window into the ‘mind as ...’ debate. The failure of any real consensus to emerge from this debate prompts a question. Are there ways in which we can understand cognition, other than through particular metaphors? The focus on particular instances of metaphor may have been misleading, for it appears that metaphor itself could form the overarching metaphor of human thought. This involves a significant shift. It implies that metaphor is not merely rhetorical embroidery, but it is how language actually operates (Wittgenstein, 1958)—and language is our most apparent manifestation of thought. Every statement we make is metaphorical. Every human thought is similarly metaphorical.

Numerous attempts have been made to explain metaphor in terms of logic and computation (Helman, 1988). Metaphor need neither be subservient to, nor counter to logic, but certainly our understanding of metaphor seems to evade description in terms of such mechanisms. Schön’s (1963) project is to show how both understanding and the exercise of judgement operate within a ‘play of metaphors,’ one against the other. A similar

understanding is presented in the context of design by Snodgrass and Coyne (1990). Metaphor itself is shown to be more amenable to understanding in terms of the metaphors of 'dialogue' and 'play.' This may frustrate AI researchers as these metaphors do not lend themselves to computer implementation. On offer, however, are richer understandings of the designers who use computer systems.

The implications of the metaphorical nature of cognition have immense significance for AI research programmes. Not the least of these is the way in which the rules by which we discuss cognition are changed. Only some of the implications are considered briefly here.

First, one of Schön's (1963) main points is that metaphors change as they are applied to a new situation. This dynamic may be subtle, but it is an essential part of the 'play of metaphors.' An obvious example is the application of the computer metaphor to human behaviour—the 'mind as a computer.' In describing human cognition in terms of computer processes, our view of the computer changes. In quite an obvious way we begin to ascribe human qualities to the computer: such as deciding, learning and designing. As with any use of metaphor such a juxtaposition (Schön's 'displacement of concepts') requires dialogue and negotiation within the situation in which the exchange is being conducted. For example, a logic-based view (as opposed to a metaphorical view) of cognition would require that problematic questions such as "can machines think?" (or "can people break down?") be answerable with a simple yes or no. Here, the extensive application of logic in the exploration of language has tended to juxtapose the concept that logic is a useful means of investigation, to one in which the subject of the investigation (language itself) is considered effective only in so far as it reflects the sharp definitions of logic. It is a propositional interpretation of language. When we displace the propositional nature of language and understanding with one in which metaphor is operating then we are more immediately in a situation of dialogue. Difficult questions can be answered with a question: what do you mean? why are you asking? what will you do with my answer? The displacement of concepts is a matter of negotiation and the appropriateness of a metaphor and its use are dependent on context.

Second, the idea of metaphor displaces the imperative of mechanistic explanations of human behaviour. It demonstrates that there are other sources of explanation that are rich, can be treated rigorously and are useful. Abandoning mechanism need not turn human behaviour into a black box. One approach is simply to orient the focus of design research away from machines (Newton, 1989). In this case, the metaphors we introduce to the research dialogue are less mechanistic, emanating from the fields of management and design activity more directly. The focus is no longer on machines, but on what people actually do and on the models (metaphors) people use and 'live by' (Lakoff and Johnson, 1980). Philosophical hermeneutics also provides a rich source of understanding (Winograd and Flores, 1987; Snodgrass and Coyne, 1990). It argues for the unity of understanding, interpretation and application, and the dissolution of the Cartesian distinction between subject and object.

Third, there are implications in terms of the relationship between people and machines. It is not necessary to embrace machine-based explanations of human behaviour in order to invent machine-based systems to assist designers. Neither is it necessary to have cognitive models of users. Rather, the question is one of relating understandings of designers that are robust and rich, to computers. For example, design considered in terms of the metaphor of 'dialogue' would require computers to establish and enhance the processes of dialogue

between members of the design team, or between a designer and the design situation. There is considerable scope here for computer applications, without the attendant imperative to seek 'artificial intelligence'. Another approach is the use of computer tools as an aid to the exploration of the metaphors themselves (Coyne, 1990).

This does not necessarily imply the abandonment of integrating AI techniques into computer systems. But the role of AI should be subsumed within the role of computers as tools to assist in design. Our concern is that through various displacements (imposition of formal models onto the design process, and projection of mind onto machine), design appears to be being subsumed within AI. The case against connectionism as the basis of a design model can be made against all AI techniques so employed. However, as will be demonstrated below, connectionism (itself, and in conjunction with other AI techniques) does suggest a useful base from which to construct design tools. The utility is irrespective of any cognitive modelling capabilities.

In order to make use of connectionism as a basis for design tools it is first necessary to understand how it operates. We now describe in some detail the connectionist approach to computation.

PART TWO: CONNECTIONIST SYSTEMS

Connectionism in Principle

Connectionist models have a long and checkered history. The effects of many simple units arranged in networks of distributed connections were being explored by McCulloch and Pitts as early as the 1940's. Later, and for nearly a decade, connectionism became one of the two main frameworks for doing AI research. (The other framework being the 'symbolic' one.) By the late 60's and early 70's however, the connectionist approach was loosing institutional support. Various reasons can be associated with this decline, but the single most influential event was undoubtedly the publication of a book by Minsky and Papert, 1969. The book showed that the connectionist models of that time were strictly limited to orthogonal (see later) problem domains. The limitation appeared to consign all connectionist models to a few, largely uninteresting problems.

A book edited by Hinton and Anderson, 1981 signalled the resurgence of interest in connectionism. The major assault on a symbolic framework for cognitive science came in 1986 (Rummelhart *et al*, 1986a). Since then, a growing legion of researchers have aligned themselves to what some commentators regard as a paradigm shift (in the sense of Kuhn, 1970) in cognitive science.

In what respect does the connectionist approach offer a new paradigm of computation? One can certainly construct a compelling argument in favour of connectionism as a new paradigm (Schneider, 1987). There still remains, however, room for doubt (Boden, 1990). Perhaps the more important question is what important breakthroughs and applications might a connectionist approach lead to, which a symbolic approach might miss?

Connectionism in Practice

Connectionist systems are computational devices based on the uniform distribution of

information across networks of relatively simple elements ('weighted arcs' and 'units'). It is not intended that individual units correspond directly with particular variables, pieces of information, concepts, or other external entities. Rather, the correspondence between some external entity and its representation in a connectionist model is intended to be distributed as a pattern throughout all of the units composing a network.

There are two phases in the development and application of a connectionist model. In the first instance, the system is 'taught' a number of example system states. This is generally referred to as the 'training phase.' It is during this phase that the appropriate weights which link units are calculated. In the second instance, the system is 'run.' This is generally referred to as the 'simulation phase.' It is during this phase that some of the most interesting features of the connectionist approach become apparent.

The training phase

We begin with a simple model, based on the pattern associator of Rumelhart and McClelland (1986), which uses the perceptron convergence procedure of Rosenblatt (1962) for training. The model can be understood by considering a simple network of six units, labelled A-F. Each unit is connected to every other unit (including, as a special case, itself) with a directed arc of weight 0 and has a threshold value (q) associated with it, also of 0. If we present this network with a pattern of activations (such as 'A=1, B=0, C=0, D=0, E=1, F=0', referred to as 'Example 1'), then we would expect the weights between units and the threshold values to adjust accordingly. (Note: The algorithm for changing the response of a network is detailed elsewhere. See for example, Coyne *et al.*, 1989, or Coyne and Newton, 1989.) The weights produced by this method are symmetrical, and will reproduce the most appropriate state, or pattern, (that is, either Example 1, or all blank) given any input arrangement (see Figure 1, Phase 1).

Now, consider the case where the previous training example (Example 1) is supplemented by another example (Example 2), where A=0, B=0, C=1, D=0, E=0, F=1. Each example is considered in turn, over several cycles. Once again, the weights produced

Example 1						Example 2						Example 3								
	A	B	C	D	E	F		A	B	C	D	E	F		A	B	C	D	E	F
A	1	0	0	0	1	0	A	1	0	-1	0	1	-1	A	1	0	-1	0	1	-1
B	0	0	0	0	0	0	B	0	0	0	0	0	0	B	0	0	0	0	0	0
C	0	0	0	0	0	0	C	-1	0	1	0	-1	1	C	-1	0	1	-1	0	1
D	0	0	0	0	0	0	D	0	0	0	0	0	0	D	0	0	0	1	-1	0
E	1	0	0	0	1	0	E	1	0	-1	0	1	-1	E	1	0	-1	-1	2	-1
F	0	0	0	0	0	0	F	-1	0	1	0	-1	1	F	-1	0	1	-1	0	1

Phase 1 Phase 2 Phase 3

Figure 1. Weights of connections between six units, produced for different training examples. Positive numbers indicate an excitatory connection, negative numbers indicate an inhibitory connection and zero indicates no connection. Phase 1 results from Example 1, Phase 2 results from Examples 1 & 2, Phase 3 results from Examples 1, 2 & 3. Threshold values in each case are 0.

are symmetrical, and reproduce the most appropriate response given any input (see Figure 1, Phase 2). The symmetry is to be expected where the training examples are completely distinct. Consider, however, the further case where Examples 1 and 2 are supplemented by a third example (Example 3), where $A=1$, $B=0$, $C=0$, $D=1$, $E=0$, $F=0$. In this case, both Example 1 and Example 3 share a common feature, ‘A’. Because of the overlap between Example 1 and Example 3, the resulting set of weights is no longer symmetrical (see Figure 1, Phase 3). A consistent set of weights, able to reproduce all inputs, is still possible however, because the input examples are ‘orthogonal’. That is, there is at least one distinctive feature to each and every example.

It transpires that few interesting problems satisfy the condition of orthogonality, detailed above. In their seminal paper, Minsky and Papert, (1969), identify the strict limitations imposed on connectionist systems as a result. Only more recently has it been demonstrated that the limitations of orthogonality need not apply to more sophisticated connectionist systems (Rummelhart *et al*, 1986a).

The simulation phase

During the training phase weights are adjusted to account for the various examples in the training set. Provided the examples in the training set are orthogonal, the simple algorithm demonstrated above is sufficient to find a set of appropriate weights. The system will respond appropriately to those inputs which match exactly, any of the examples in the training set. However, of considerably more interest, is the response of the system to partial inputs, or inputs outside the original training set.

We describe a simple model here. The model does not, in itself, yield useful results. We use the model only to illustrate the basic mechanics of a simulation phase. The model can be understood by considering the set of weights and threshold values produced after Phase 3 of the previous example (Figure 1). The input to the system would constitute a set of starting conditions for each of the units. If we present the network with a pattern of activations (such as ‘ $A=0$, $B=0$, $C=0$, $D=0$, $E=1$, $F=0$ ’, referred to as ‘Input 1’), then we might expect the system to correctly associate the pattern as a part of the training example, ‘Example 1’.

The algorithm for producing a response from the network is exactly as for the training algorithm. In this case, the ‘predicted’ value becomes the output value. Here the output pattern always corresponds to one of the training examples (except for the special case where no units are active). These output states are referred to as ‘relaxed states’. Not all input configurations arrive at a relaxed state first time around. The system has to take the output from one round (cycle) as the input to the next. This iteration continues until no changes occur from one cycle to the next.

In this form, the system is actually optimizing on a function termed the ‘goodness of fit.’ The goodness of fit is loosely described as a measure, over all units, of how well the activation of each individual unit fits the input pattern, relative to the importance of that unit in terms of its connection weights to all other units. It is defined as the sum of the extent to which each pair of units (i,j) contribute to the ‘goodness’, plus the extent to which each unit satisfies its input value. The extent to which each pair of units (i,j) contribute to the ‘goodness’ is given by the product of their activation values times the weights connecting them:

$$\sum w_{ij} a_i(t) a_j(t) \quad \text{for all } i \text{ and } j \quad (1)$$

The extent to which each unit satisfies its input value is given by the product of its input value and activation value:

$$\sum \text{input}_i(t) a_i(t) \quad \text{for all } i \quad (2)$$

It follows that if the weight (w_{ij}) is positive (excitatory) the 'best fit' would be where both units are as active as possible. Where w_{ij} is negative (inhibitory) at least one unit should be 0 to maximize pairwise goodness. Similarly, if the $\text{input}_i(t)$ is positive then $a_i(t)$ is best the more active it is. If it is negative, $a_i(t)$ should be reduced as close to 0 as possible.

Of course constraints won't always be consistent with these tendencies. Increasing $a_i(t)$ might increase overall goodness in some ways while decreasing it in others. The point here is that it is the overall sum of these individual contributions that the system seeks to maximize. The system moves from state to adjacent state by maximizing the goodness of fit until it settles (relaxes) at a stable point—ie. an optimum.

The goodness of fit forms a surface across the n dimensions of a hypercube, where n equals the number of units in the net. The surface is systematically modified (sculptured) by changes in the starting pattern, so that the clamping (the holding of a unit in a constant state of activation) of different or more units alters the topography and therefore the maximum point towards which the system moves.

The different patterns of units, or interpretations, correspond to the states of the system. The total set of states compose the state space. Peaks in the space represent the best interpretations (maxima). The size and height of the region which surrounds each maxima determines the likelihood of finding that peak. The height of each maxima corresponds to the degree to which the constraints of the problem (weights and activation values) are being met.

CONNECTIONISM AS A DESIGN TOOL

Rules of Classification

Automated classification systems are generally based on the idea of analysing a wide range of examples demonstrating a particular concept, that is, examples that belong to a particular class. The system then detects and generalises on features of the descriptions. The features must capture the concept. The result is some set of rules that facilitates the classification of new examples. They may be in the form of a set of logical rules (Winston, 1975; Michalski, 1983), a decision tree (Quinlan, 1979) or some other representation.

In the case of classification through a connectionist system the outcome is essentially a black box, yet its performance offers certain advantages over conventional rule-based output. One such advantage is the ability of a connectionist system to generalise while retaining information about exceptions (Rumelhart and McClelland, 1986).

The example we use to illustrate the rule 'learning' potential of a connectionist system can be thought of in relation to element libraries and descriptions in a CAD system. (See

also, Coyne and Newton, 1990, and Kan, 1990.). In a CAD system, classes can be established on the basis of many features. However, in the examples which follow, the features used will describe the patterns of linkage of the entire set of elements. That is, specifically, the structure of relationships between elements. Thus, it is the relationships between the elements that is of primary interest here, rather than the contents of each. So information can be organised in a class-instance (*Kind_of*) hierarchy. Elements can also be linked in terms of part/whole (*Part_of*) hierarchies.

Another useful basis for linking elements might be similarity. Lintels and podiums are similar in that they both appear as horizontal elements in elevation. While browsing through an element library it would be useful to be able to tell the system: show me another element that is similar to this one. The system would need to know what is meant by 'similar' in a particular context. One way of telling the system is to point to other elements that are similar, then tell the system to find another one that has common features. Deriving the rules by which objects may be regarded as similar is essentially a classification problem.

There are five kinds of linkages between elements considered here, and we are interested in the numbers of linkages in each case. The connectionist system is required to generalise on the number of each link for each element. In other words the numbers of linkages constitute 'features'. The features are defined as follows:

- (1) *Kind_of* Each basic component (element) is a kind of some more abstract component description. For example, 'SolarGuard' is a *Kind_of* 'Glazing.' This term records the number of *Kinds_of*.
- (2) *Dependent_on* Each element may be dependent upon another, or many others, for its own composition. For example, 'Frame' is *Dependent_on* 'Columns.' This term records the number of dependencies for each element.
- (3) *Part_of* The reciprocal to *Dependent_on*. For example, 'Column' is a *Part_of* 'Frame.' This term records the number elements any given element is a *Part_of*.
- (4) *Siblings* Where there are several *Kinds_of* the same element, each one is said to be a sibling of the other. Siblings share the same basic form. For example, 'Granolithic' and 'Terrazzo' are siblings because they are both *Kinds_of* 'Concrete Floorings.' This term records the number of siblings each element has.
- (5) *Level* The hierarchy is divided into levels. This term records the level at which each element fits.

The possible relationships between a set of 36 hypothetical elements is illustrated in Figure 2. In Figure 2, each node is an element. Each element occurs at a particular level, numbered 10 to 1 down the diagram. Those elements to the right of the level number are *Kinds_of* those elements to the left of the same level number. An element is *Dependent_on* those elements to which it is directly joined below. It is a *Part_of* those elements to which it is directly joined above. Siblings occur where there is more than one element at the same level on the right hand side. Thus, Element 27 is a *Kind_of* Element 25; is *Dependent_on* Elements 16, 20 and 21 (3 elements); is a *Part_of* Elements 34 and 35 (2 elements); and is a *Sibling* of Elements 26, 28, 29 and 30 (4 elements).

The total number of possible feature combinations in this case is 3,500. Of the 36 examples there are 31 unique combinations of structure 'types', or 0.89% of the total possible number of combinations. This is a small sample size, and the performance of the

possible number of combinations. This is a small sample size, and the performance of the connectionist system would be expected to improve with increased sample size.

Training is done by presenting a series of input patterns (the patterns of features for each of the 36 elements) and their corresponding output patterns. In this case the output patterns reflect whether or not an element is 'unique'. For the purposes of this paper, an element is unique if it has no siblings.

The learned weights can be tested exhaustively by presenting the system with the 36 input patterns and checking that the learned output patterns are regenerated. Of considerably more interest, however, is the question of whether the system is able to respond correctly to completely new input patterns.

To test this, the input patterns for further elements can be fed into the system. That is, the system can be asked to generate the output for elements it has not 'seen' before. Analysis of the matrix of weights and threshold values shows that the correct response will be generated for all but a very small percentage (in the order of 1%) of the total number of possible input patterns. This fail rate, from an example set itself representing less than 1% of the possible feature combinations, is an encouraging performance.

The system appears to have learned the rule that an element is unique if it has no siblings. Of course, this rule is never made explicit to the system, and, as the next example will show, adherence to such a rule is not hard and fast. Under certain circumstances the system can accommodate exceptions to the rule.

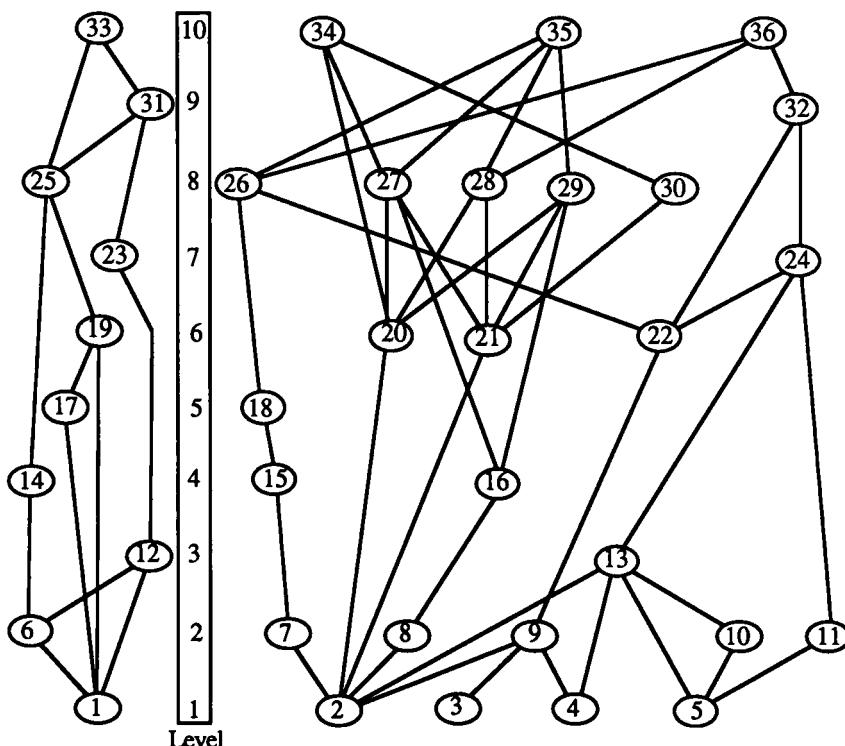


Figure 2. A set of 36 hypothetical elements and their linkages within a Kind_of and a Part_of hierarchy.

Exceptions to the Rule

Here we follow an experiment that closely matches an experiment by Rumelhart and McClelland (1986). In this example we demonstrate how it is possible to 'teach' the connectionist system about exceptions to the 'unique rule.' During the training phase we present an element to the system that has siblings, but we also present an output pattern for that element indicating that it is unique. In other words, we are suggesting that there is some other criterion, other than number of siblings, by which we have decided that the element is unique.

We can demonstrate how the performance of the system changes significantly with the number of training cycles. All 36 elements are presented to the system as unique or not unique in accordance with the uniqueness rule. However, the output of Element 1 is adjusted to indicate that it is an exception to the rule. After a short training run (10 cycles), the system behaves as though Element 1 conforms to the rule. In other words, during the simulation phase, the fact that Element 1 is an exception is overridden by the effect of all the other conforming cards. Rumelhart and McClelland (1986) describe this process as corresponding to the tendency of young children to over-regularise past tenses of English verbs ('comed' instead of 'came'). A point is reached where responses previously correctly learned are incorrectly regulated to conform with the case in general. As progressively more training cycles are introduced the system begins to form a strong pattern of association with the regular rule, and can then also begin to weight the matrix so that exceptions are correctly identified. For our example, after 300 cycles the matrix performs in every respect as previously, with the single exception that the input pattern for Element 1 produces a 'unique' classification in contradiction to the 'rule'.

Implicit Criteria

In the previous examples, while no rule as such is included in the system, an 'explicit' rule has been used and described to the system by means of input and output patterns. The next example is intended to demonstrate that an 'implicit' classification rule can be similarly learned.

This can be demonstrated by keeping the input pattern the same as for the previous examples, but the output pattern is changed to an element numbering system. 36 output units are used so that each element can be individually identified. (There are no units to indicate uniqueness.) There are 'two' training phases. The system is taught the 36 examples, each with its own output pattern (the number of the element). In the second training phase some of the elements are presented to the system again. These are the 14 'unique' elements. The output pattern for each of these elements is the numbers of the 13 elements to which it is linked.

Thus, in this example there is no explicit connection between an element and its class. As far as the system is concerned the 14 elements are linked arbitrarily. We would wish the system to behave as though it had discovered what we know, a rule that determines the 14 unique elements belong to the same class.

During simulation the resultant matrix of weights and threshold values correctly identifies the element number corresponding to each input pattern. However, where the input pattern belongs to an element within the class of 14 linked elements (the 'unique'

elements) every element to which it is linked appears in the output. Neither of these results are particularly startling, or useful. The acid test is to input new patterns and to evaluate the output pattern.

During simulation we can present the system with novel elements. The system effectively identifies whether a new element has siblings, and that this is the rule which governs its similarity to the set of 14 unique elements. Note also that the strength of the association with the various elements varies. The system apparently takes account of rules which provide other sources of similarity between the new element and those that the system already knows about.

Multiple Criteria

The rules by which we decide that certain elements are similar are likely to be more complex than one based on the single feature of uniqueness used in the previous examples. The connectionist system is also able to generalise on the rules governing membership of several overlapping classes. For example, we can also define rules governing classes called 'useful', 'generic' and 'independent':

Class 1—*Unique* Any element with no Siblings.

Class 2—*Useful* Any element which is a Part_of 4 or more others.

Class 3—*Generic* Any element at Levels 1 or 2.

Class 4—*Independent* Any element Dependent_on 1 or less other elements.

It can be determined from Figure 2, that Class 1 has a membership of 14, Class 2 a membership of 4, Class 3 a membership of 11, and Class 4 a membership of 21. 7 elements are not members of any class.

These four class memberships are input to the system as for the previous example. The output patterns generated during the simulation and testing phase demonstrate that the system succeeds in adopting rule-like behaviour. These rules link the majority of elements, with appropriate weights and memberships, but does fail to link with certain elements. Particularly, Element 4 appears to have an input pattern which the system fails to link with others in the same classes. Overall however, considering the complexity of the classes and sub-classes, the system performs remarkably well.

When a new input pattern is presented, and intended to be a member of all classes but different in almost every respect to Element 1 (the only member of all four classes), the system successfully links it to all other class members. This example, in particular, serves to foster confidence in the connectionist approach as a means of establishing the rules on which certain elements may be said to belong to overlapping classes.

These results are entirely in accordance with the results from similar experiments presented by others. The contribution here is the application to 'learning' about association rules between elements of design information. There are some straight forward applications of this idea. The criteria of unique, useful, generic and independent are derived from the explicit relationships within a set of elements. Without putting too much store on the definitions of these terms, they represent the kinds of criteria by which we might decide that certain elements are similar. It is easy to imagine an information system where an informed user tells the system that Element X is similar to Elements A, B, C and so on. In other

words, the user links groups of elements together by some unstated similarity rule. The user does not externalise the rule by which these cards are linked. However, the system has the user's idea of unique, useful, generic and independent, along with other criteria, built into it. The features from each element constitute the training set for a connectionist-based 'information associator'. As described above, the system will certainly make associations. The question is whether the rules which the connectionist system has determined operate to reflect accurately the hidden criteria of the user in establishing the linkages.

Of course, a user of such a system could be invited to specify the rules by which similarities are being established, and this could be used as the input for some rule-based approach, or to customise the connectionist network prior to training. The approach in the examples explained above has been to exploit the Kind_of and Part_of linkages between units of design information. The feeling is that design information (for example, of the kind utilized within a CAD system) is generally heavily structured, and that this structure can assist in reasoning about other associations within the information.

Classification by Schema

Consider a household room schema. This schema may contain information about all (or a subset of all) the things that could be expected in a household room: bath, coffee table, double bed and so on. It would also contain information about different configurations of elements (lounge suite and a small table, blinds and a window, stove and a fridge) and possible sizes (room size, number of tables). One way to classify the different room configurations within such a schema would be to determine in advance the appropriate classification rules. Each example of a room could then be classified according to the features they exhibit, as interpreted by those rules ('classification by rules'). This approach however lacks flexibility, it is prescriptive and forces a classification system onto the data set. Often a more passive approach is required, which would be to let the characteristic features of each classification somehow emerge as a property of the data set itself.

One approach is to establish for each feature a corresponding unit in a connectionist model. The model can then be presented with a number of training examples of feasible room descriptions. A realistic training set may include many features, but the list of features will never be exhaustive for all rooms. Any training set is likely to be idiosyncratic and show a greater familiarity with one form of construction or another. Note also that the examples are likely to fall within many overlapping categories. There is no single feature or group of features by which we could obviously identify a particular room type. The example set is essentially 'non-conforming' and may possibly even contain contradictory information. In all these respects the examples tend to defy a prescriptive, rule-based classification approach.

However, it is a relatively simple matter to extract a description of a room from a connectionist system. From a training set of 50 small scale house plans, using 40 features, it is possible to establish what would be the typical description of a room containing, for example, a bath. We do this by clamping on the 'bath' unit, and watching the system settle on a pattern of mutually supporting activations. By clamping a unit, we hold the activation of that unit at a constant level throughout the simulation (in this case, 'on'). The process is demonstrated in Figure 3, following a system of representation devised by Hinton and

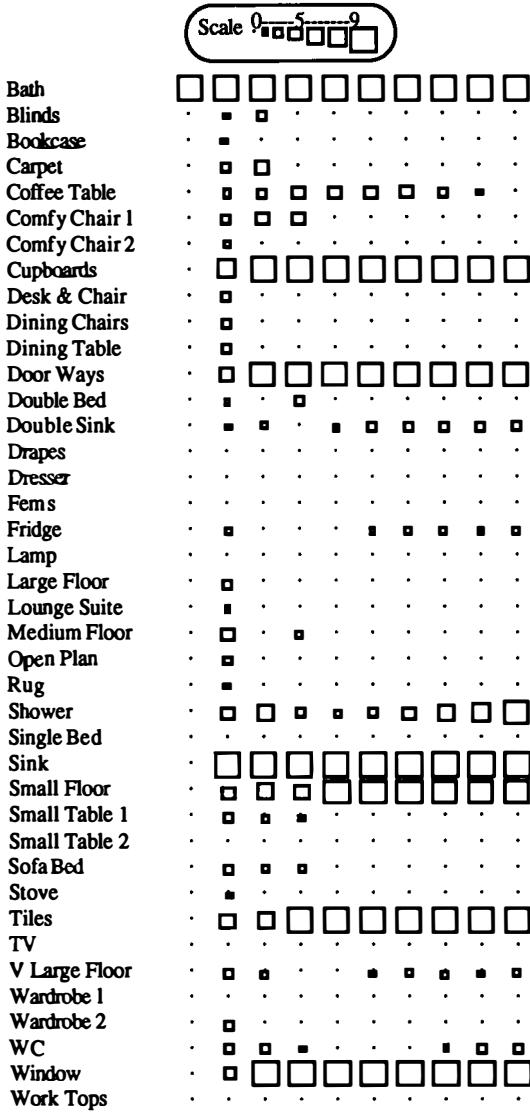


Figure 3. A demonstration of how the system cycles to a settled state, where subsequent cycles produce no further change. Columns, from left to right, represent cycles. Rows represent named units. The unit 'bath' is clamped at the full activation level.

Sejnowski (1986). The activation status of each unit is shown for different cycles of the system during simulation. Each column represents a different cycle. A larger square means that a unit is fully active and a dot means that the unit is inactive. Eventually the system settles into the stable state represented by the rightmost column.

We can observe the various shifts in allegiance as active units excite associated units

and inhibit others. Groups of units then begin to act in cohort to draw in certain units (such as Unit 25, 'shower') and force out others (such as Unit 6, 'coffee table'). Eventually a point of stability is reached—the fully relaxed state. The fuzzy boundary of the schema is indicated by those units which are neither fully active, nor inactive. The final output would be determined by the threshold value which, in this particular system, is set at 0.5. Where the activation of a unit is above 0.5 it would be set to full activation (1), where below it would be set to 0 activation.

Interestingly, by running the system with each unit clamped fully active, in turn, there emerges from the data set a finite number of stable states (room types). In the previous case, for example, there are 17 different potential room types. The room types identified relate to peaks in the goodness of fit surface. The surface is in 40 (the number of units) dimensional space, but can be represented loosely as a 'landscape'. When different combinations of units are clamped, the topography of the landscape is modified: peaks are moved and distorted. The range of potential room types is varied as a consequence. Otherwise competing units can be forced into cohort, creating room descriptions distinctly different from those input in the training set. Here the classification is being forced across schema boundaries. In fact every combination and grouping of input features has the potential to produce room descriptions which emerge as classifications across previous schema boundaries. An interesting effect in the context of creative design. (The emergent properties of connectionist systems are considered in more detail in Coyne, 1991. See also, Coyne *et al*, 1989.)

Data Management

Examples in this section are based on a system first developed by McClelland (1981). The system uses a different network architecture to previous examples. It is referred to as an interactive activation and competition (IAC) network. The network was implemented using the IAC Network software provided in McClelland and Rumelhart (1988).

An IAC network is distinguished from previous architectures, where units work independently, by imposing an external grouping of units into 'pools.' Units in the same pool are usually mutually inhibiting. Individual units can be connected between pools. Inter-pool connections are excitatory, and generally bi-directional. The bi-directional nature of inter-connections means that the processing in each pool both influences and is influenced by processing in the other pools. The previous architectures would model connections in a similar manner, but imposed groupings force a more clearly defined set of activations and competitions between units. The clearer definition (where it is appropriate) improves the performance of the network considerably.

Another key feature of the IAC network is a 'decay' parameter. The decay parameter tends to move the activation of a unit back to a state of rest (commonly an activation value of 0). The larger the value of the decay term, the stronger this tendency. The decay term instigates continual change, and there is no guarantee that activations will ever completely stabilize. Fortunately, in practice the cumulative effects of activation and decay often lead to apparent stability, where effective changes are minimal. The decay parameter also demands that external inputs be continuously replenished.

The example used to illustrate this approach is based on a simple database of a property portfolio. In conventional terminology, the database consists of 30 records. Each record has

portfolio. In conventional terminology, the database consists of 30 records. Each record has the following 7 attributes, and potential attribute values:

- (1) *Attribute 1* Code Number—numerical value from 1 to the number of records in the database
- (2) *Attribute 2* Title—the name of the property
- (3) *Attribute 3* Type—Office, Shop or Industrial
- (4) *Attribute 4* Status—Fully let or Partially let
- (5) *Attribute 5* Location—CBD, City or Suburbs
- (6) *Attribute 6* Length of remaining lease—2 Years, 4 Years or 6 Years
- (7) *Attribute 7* Surveyor in charge—Andy, Brian or Cathy.

The complete data set is shown in Figure 4.

	TYPE			STATUS	LOCATION			LEASE			SURVEYOR				
	Office	Shop	Industrial		Full	Part	CBD	City	Suburbs	2	4	6	Andy	Brian	Cathy
1 City_Rd	1	0	0	1	0	0	0	1		1	0	0	0	0	1
3 Pitt_St	1	0	0	1	0	1	0	0		0	0	1	1	0	0
4 Lamb_St	1	0	0	1	0	0	0	1		0	0	1	0	0	1
5 Grove_St	1	0	0	1	0	0	1	0		0	0	1	0	0	1
9 Plaza	1	0	0	1	0	0	1	0		0	0	1	0	1	0
12 York_St	1	0	0	1	0	1	0	0		0	0	1	0	1	0
13 Evans_St	1	0	0	1	0	1	0	0		1	0	0	0	1	0
14 King_St	1	0	0	1	0	1	0	0		1	0	0	1	0	0
15 Word_Sq	1	0	0	1	0	0	1	0		1	0	0	0	0	1
17 Centre_Pt	1	0	0	0	1	0	1	0		0	0	1	0	0	1
18 Qantas	1	0	0	0	1	0	1	0		1	0	0	0	1	0
22 John_St	1	0	0	0	1	1	0	0		0	1	0	0	1	0
26 Bond_Bldg	1	0	0	0	1	0	1	0		0	0	1	0	1	0
1 Woolworths	0	1	0	1	0	0	1	0		0	1	0	0	0	1
7 Gowings	0	1	0	1	0	1	0	0		1	0	0	0	1	0
8 Grace_Bros	0	1	0	1	0	1	0	0		0	1	0	0	0	1
10 BBC	0	1	0	1	0	1	0	0		0	1	0	0	0	1
16 Dymocks	0	1	0	0	1	0	1	0		1	0	0	1	0	0
19 Russells	0	1	0	0	1	0	0	1		0	1	0	1	0	0
20 Myers	0	1	0	0	1	0	0	1		0	0	1	1	0	0
21 St_Vincent	0	1	0	0	1	0	0	0	1	0	0	1	0	1	0
23 Army_Surplus	0	1	0	0	1	0	0	0	1	0	1	0	0	1	0
25 Strand	0	1	0	0	1	0	1	0		1	0	0	1	0	0
29 Hooker	0	1	0	1	0	1	0	0		0	1	0	1	0	0
6 Beechams	0	0	1	1	0	1	0	0		0	1	0	0	0	1
11 BHP	0	0	1	1	0	1	0	0		0	1	0	0	0	1
24 BP	0	0	1	0	1	0	1	0		0	1	0	0	1	0
27 Telecom	0	0	1	0	1	0	1	0		1	0	0	0	1	0
28 IBM	0	0	1	1	0	0	1	0		1	0	0	0	1	0
30 Holden	0	0	1	0	1	0	0	1		0	0	1	0	0	1

Figure 4. The data set for 30 records in a simple property portfolio data base. Each row represents an individual example (or data record). Examples are numbered (1-30) as per the left-most column. Other columns show the activation level of each unit. Where a value of '1' is shown, the unit is considered to be fully active. Where a value of '0' is shown, the unit is considered to be inactive.

The network consists of 74 units, where each unit represents a single potential attribute value. (Attribute 1 = 30 units, 2 = 30 units, 3 = 3 units, 4 = 2 units, 5 = 3 units, 6 = 3 units, 7 = 3 units, total = 74 units.) The required number of units would vary with changes in the number of records and number of potential attribute values. The units for each attribute are pooled together. Inter-pool connections are all bi-directional and established only through the Attribute 1 ('Code Number') pool.

To retrieve details on a specific data record, an external input is given to the unit representing the appropriate project title, and the network cycles through to a state of apparent stability. For example, when external input is given to the unit for 'City_Rd', the network relaxes to the condition illustrated in Column A, Figure 5. The condition can be confirmed with reference to Figure 4.

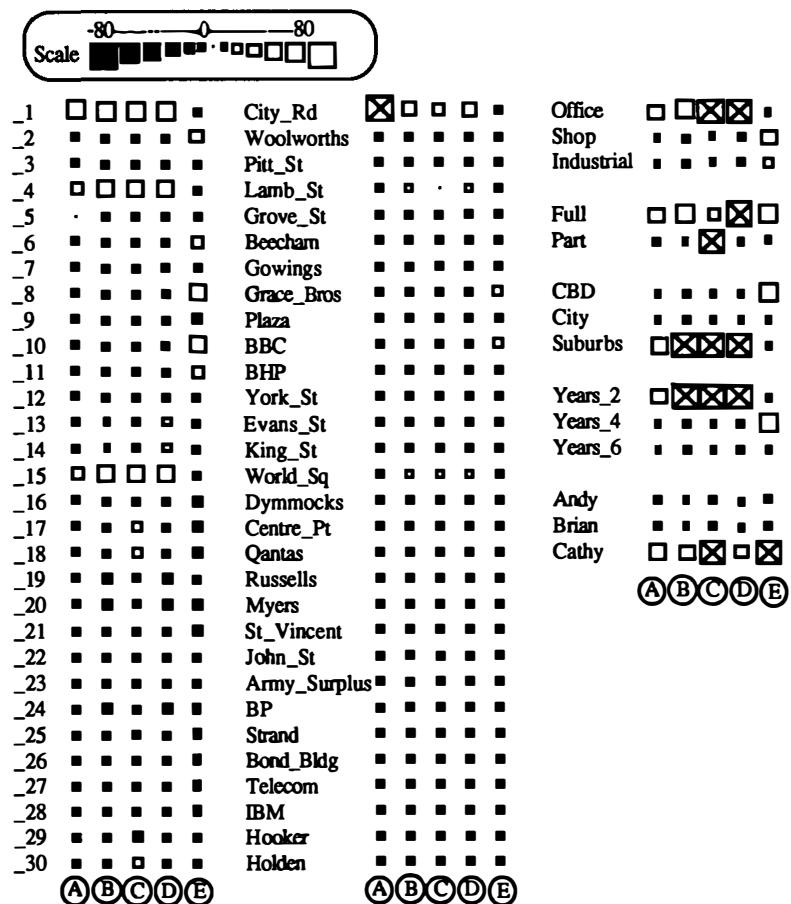


Figure 5. Relaxed states for five different sets of external inputs (labelled 'A'-'E'). External inputs are marked \odot .

The project titles are necessarily unique, and such specific retrievals can be performed by any number of conventional techniques. However, connectionist systems have the added advantage of full content addressability. Individual records can be retrieved based on any combination of attribute values, not just on the unique project titles. For example, Column B, Figure 5 shows how the attribute values 'Suburbs' and 'Years_2' combine to retrieve the same 'City_Rd' record as before, but via an entirely different route.

Graceful Degradation and Default Assignment

Of course, full content addressability is by no means unique to connectionist systems. A 'true' relational database management system has equivalent features. The principal advantage of a connectionist approach is to be found in the distributed way it represents the data. No single unit is used to represent any particular record. The units which activate and decay to provide a set of attribute values equivalent to the 'City_Rd', for example, are the same units which activate and decay to provide every other record. The activations which cause 'City_Rd' attributes to fire are activations across all units. Some activations are excitatory and some are inhibitory, but it is the pattern across all units which causes the network to stabilize on the appropriate attribute values.

The benefit of a distributed representation is that its performance degrades slowly as parts of the network are lost, or inputs become too 'noisy'. For example, the inputs shown in Column C, Figure 5 contain noise. The attribute values for the record of 'City_Rd' have been input externally, with the single mistake that the status of the property is taken as being 'Partly' let rather than (the correct value) 'Fully' let. Though it would not be immune to such noise, Column C, Figure 5 shows how the connectionist system is capable of coping with these types of error. An equivalent, conventional database management system would have considerable problems overcoming the combinatorial explosion associated with errors of this kind. Similarly, where a record is lost in a conventional system the loss is complete. In the connectionist system, loss of a unit may cause problems, but they would be unlikely to be devastating.

Content addressability is illustrated above. If the attribute values 'Office', 'Full', 'Suburbs' and 'Years_2' are input externally, we could expect the attribute values 'Cathy' and 'City_Rd' to be activated. This process simply completes the description of a record, given some partial input, by utilising the direct connections between the various attribute values and the instance unit '_1'. Where there is no direct connection between an instance unit and, say, the attribute values for 'Surveyor', the system will establish a default value.

In the example shown in Column D, Figure 5, the weights between instance unit '_1' and the attribute values for 'Surveyor' have all been severed (set to 0). In effect, there is now no 'Surveyor' associated with 'City_Rd'. The process of default assignment is an interesting one. The external inputs activate the unit 'City_Rd', which inhibits the activation of any other 'Title'. However, the set of external inputs excite a range of the instance units, and these units compete to see which 'Surveyor' should also be activated.

Such an assignment can be considered a generalisation of the case. For example, Column E, Figure 5 shows the generalised case for 'Cathy.' In general, Cathy deals with fully let shops in the CBD with a 4 year lease. There is also a tendency to have industrial properties.

The power of this feature is the capacity to provide generalisations, on demand, on

possibly unanticipated grounds. No formal generalisations are stored, and there is no formal mechanism to generate them. The generalisations emerge from the connectionist form of representation.

FACILITATING DESIGN DIALOGUE

Here we consider one way in which the connectionist models can be of value to a designer, particularly in the context of a CAD system. It is the sense in which connectionist models can be used in a way that is conservative towards the designer's own behaviour. The goal is to establish a form of dialogue and exchange of behavior between that of the system, and that of the user. The approach described above needs to be scaled up in various ways. A useful system should be able to handle many more features and many more examples than in the cases presented so far.

As indicated in the room example, the features are simply statements about objects and design situations, with no a priori ordering or categorization. So some of the features can be interpreted as pertaining to a design context: for example, the situation may be that we are configuring a room layout. The 'small room' or any of the other features may be the starting point for 'dialogue' with a trained connectionist system. The designer registers this interest by 'clamping' the 'small room' feature. In the manner outlined above the system could respond to the clamping of 'small room' with a set of features that constitute a typical small room configurations. The way in which the designer responds to this description could be to accept some of the descriptors as being of interest and to reject others. This is based on the designer's current understanding of the design task, which of course will change in the course of the dialogue with the system. So the designer may clamp certain features on or off, and the system throws back a new description in response to these changes. In response the designer may favour new combinations of features. The original requirement may also be changed. It can also be characterised as a search through a space of feature combinations guided by the designer. The combinations are not random, neither are they determined by rules.

As described here the process does not have a graphical component. We are dealing simply with atomic statements that are interpreted as features of a design. In the context of a CAD system the features could in fact be pointers to geometrically modelled design elements or components, such as those found in a CAD library of elements. The connectionist system could act as a means of bringing components to the ready for placement.

If library elements constitute features in a connectionist system that has been trained from examples then it is possible to establish a dynamic catalogue of components. If the designer producing a room configuration on a CAD system starts by locating a table, the elements pertaining to the prototypical 'room with a table,' such as 'chairs', will be given prominence within a dynamic catalogue of components. The system acts as if in anticipation of a particular configuration and provides a subtle prompting towards that anticipated configuration. Selecting and placing an element is equivalent to clamping the feature in the connectionist system. Of course the suggestions of the system can be overridden, in which case the configuration of the catalogue of components will change accordingly. As elements

are selected and placed the catalogue changes its configuration. Subsequently the composition of a particular design is processed as an example to influence further design episodes. It should also be possible with such a system to begin with a blank slate where there are initially no examples and where the element library is built up by the designer. This is the most interesting application of the connectionist approach as it suggests a system that forms an extension of the designer's particular selection behaviour. The system is conservative towards the designer's own behaviour—within the constraints of the connectionist algorithms. This can speed up the configuration task for the designer. The system also provides a medium that suggests new combinations. But the system is also open to change as directed by the designer.

CONCLUSION

The concern of this paper has been the value of AI in design, considered here in terms of the impact of one AI technique in particular, connectionism. The paper presents a challenge to the widespread use of mechanical metaphors to model designers themselves (cognitive modelling). The mechanistic view of cognition is particularly disabling because in effect it denies the many social interactions (play, dialogue, artistry) manifest in the nature of design. Connectionism claims our attention as a particularly strong paradigm for cognitive modelling. It threatens to disable design even further.

The thrust of the paper has therefore been to present a challenge to the mechanical metaphors of cognitive modelling. It argues that design can be well described by alternative metaphors. Design viewed in these alternative terms curbs the use of AI paradigms as explanatory devices. The principal role of AI is then in the production of design tools. In this light, an investigation of the application of connectionism to design provides a good test case for AI and design in general.

In electing to consider connectionism apart from other techniques, there is no intention that connectionism somehow be raised or promoted 'above' the other AI paradigms. Connectionist models exhibit a range of interesting features, but the critical consideration is our overarching metaphor for AI and design. The immediate need is to extricate AI and design from the mechanistic rhetoric of cognitive modelling.

Through various applications, this paper demonstrates how connectionism provides a means by which designers can engage in a rudimentary kind of dialogue with a computer system. The applications focus on generic processes such as classification and data management. The dialogue considered is of a kind that can enhance the nature of the sense of 'play' evident in the way design is carried out. There is no suggestion here that connectionism provides a means of simulating what the designer does—rather that the computer system can be contrived to act sympathetically to a designer's interests. The computer system is constrained by its inevitable machine-like behaviors while the designer can exploit the capability of the connectionist algorithms to make suggestions and to find new and sensible connections between ideas.

ACKNOWLEDGEMENTS

We are grateful to our colleague Dr Adrian Snodgrass for explaining and rendering obvious the role of metaphor in understanding. Support for this work was provided by the University of Sydney, Key Centre for Design Quality.

REFERENCES

- Balachandran, M.B. (1988). *A Model for Knowledge-Based Design Optimization*, PhD Thesis, Department of Architectural Science, University of Sydney, Sydney.
- Benyon, D. (1990). Adapting systems to individual differences in cognitive style, *Proc. Australasian Society for Cognitive Science: First Annual Conference*, University of New South Wales, Sydney, pp.2-3.
- Boden, M.A. (ed) (1990). *The Philosophy of Artificial Intelligence*, Oxford University Press, Oxford.
- Clark, A. (1989). *Microcognition: Philosophy, Cognitive Science, and Parallel Distributed Processing*, MIT Press, Cambridge, Massachusetts.
- Coyne, R.D. (1990). Design reasoning without explanations, *AI Magazine*, (to appear).
- Coyne, R.D. (1991). Modelling the emergence of design descriptions across schemas, *Environment and Planning B*, to appear.
- Coyne, R.D. and Newton, S. (1989). A tutorial on neural networks and expert systems for design, in *Expert Systems in Architecture, Construction and Engineering Conference*, by J.S. Gero and F. Sudweeks (Eds), University of Sydney, Sydney, pp.321-337.
- Coyne, R.D. and Newton, S. (1990). Design reasoning by association, *Environment and Planning B*, Vol.17, pp.39-56.
- Coyne, R.D., Newton, S. and Sudweeks, F. (1989). Modelling the emergence of schemas in design reasoning, in *Proceedings Modelling Creativity and Knowledge-Baed Creative Design*, J.S. Gero and M.L. Maher (eds), University of Sydney, Sydney, pp.173-205.
- Crick, F. and Asanuma, C. (1986). Certain aspects of the anatomy and physiology of the cerebral cortex, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 2, Psychological and Biological Models*, D.E. Rumelhart, J.L. McClelland and the PDP Research Group, MIT Press, Cambridge, Massachusetts, pp.333-371.
- Dreyfus, H.L. (1987). Misrepresenting human intelligence, in *Artificial Intelligence: the Case Against*, Born, R. (ed.), Croom Helm, Beckenham, Kent, pp.41-54.
- Feldman, J.A. (1989). Connectionist representation of concepts, *Connectionism in Perspective*, R. Pfeifer, Z. Schreter, F. Fogelman-Soulie and L. Steels (eds), Elsevier Science Publishers BV, Amsterdam, pp.25-45.
- Gero, J.S. (1988). Prototypes: A basis for knowledge-based design, in J.S. Gero and T. Oksala (eds), *Symposium on Knowledge-Based Design in Architecture*, Helsinki University of Technology, Helsinki, pp.3-8..
- Harnad, S. (1989). *The Symbol Grounding Problem*, paper presented at CNLS Conference on Emergent Computation, Los Almos, May 1989.

- Helman, D.H. (ed.) (1988). *Analogical Reasoning*. Kluwer Academic Press, Dordrecht, The Netherlands.
- Hinton, G.E. and Anderson, J. (eds) (1981). *Parallel Models of Associative Memory*, Erlbaum, Hillsdale, N.J.
- Hinton, G.E. and Sejnowski, T.J. (1986). Learning and Relearning in Boltzmann Machines, in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1, Foundations*, D.E. Rumelhart, J.L. McClelland and the PDP Research Group, MIT Press, Cambridge, Massachusetts, 282-314.
- Hofstadter, D.R. and Dennett, D.C. (1981). *The Mind's I: Fantasies and Reflections on Self and Soul*, Penguin, Harmondsworth, Middlesex.
- Kan, W.T. (1990). Applications of connectionism in computer-aided design, *Masters Thesis* (Unpublished), Department of Architectural and Design Science, University of Sydney, Sydney.
- Kuhn, T. (1970). *The Structure of Scientific Revolutions*, 2nd. Ed., University of Chicago Press, Chicago.
- Lakoff, G. and Johnson, M. (1980). *Metaphors We Live By*, The University of Chicago Press, Chicago.
- Lansdown, J. (1987). The creative aspects of CAD: a possible approach, *Design Studies*, Vol.8, No.2, pp.76-81.
- Lloyd, D. (1989). *Simple Minds*, MIT Press, Cambridge, Massachusetts.
- McClelland, J.L. (1981). Retrieving general and specific knowledge from stored knowledge of specifics, *Proceedings Third Annual Conference of The Cognitive Science Society*, Berkley, pp.170-172.
- McClelland, J.L. and Rumelhart, D.E. (1988). *Explorations in Parallel Distributed Processing: A Handbook of Models, Programs and Exercises*, MIT Press, Cambridge, Massachusetts.
- McClelland, J.L., Rumelhart, D.E. and the PDP Research Group. (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 2, Psychological and Biological Models*, MIT Press, Cambridge, Massachusetts.
- Michalski, R.S. (1983). A theory and methodology of inductive learning, *Artificial Intelligence*, Vol.20, pp.111-161.
- Minsky, M. (1975). A framework for representing knowledge, in P.H. Winston (ed), *The Psychology of Computer Vision*, McGraw-Hill, New York, pp.211-277.
- Minsky, M. and Papert, S. (1969). *Perceptrons*, MIT Press, Cambridge, Massachusetts.
- Mitchell, W.J. (1990). *The Logic of Architecture: Design, Computation and Cognition*, MIT Press, Cambridge, Massachusetts.
- Newton, S. (1987). Dealing with dialectics: some problems in applying expert systems to building construction, *Proceedings of the Fourth International Symposium on Robotics and Artificial Intelligence in Construction*, Vol.1, Technion, Haifa, pp.53-63.
- Newton, S. (1989). The irrelevant machine, *Design Studies*, Vol.10, No.2, pp.118-123.
- Newton, S. and Logan, B.S. (1988). Causation and its effect: the blackguard in CAD's clothing, *Design Studies*, Vol.9, No.4, pp.196-201.
- Quinlan, J.R. (1979). Discovering rules by induction from large collections of examples, *Expert Systems in the Micro-Electronic Age*, D. Michie (ed), Edinburgh University Press, Edinburgh, pp.168-201.

- Rorty, R. (1980). *Philosophy and the Mirror of Nature*, Basil Blackwell, Oxford.
- Rosenblatt, F. (1962). *Principles of Neurodynamics*, Spartan Books, New York.
- Rumelhart, D.E. and McClelland, J.L. (1986). On learning the past tense of English verbs, in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 2, Psychological and Biological Models*, J.L. McClelland, D.E. Rumelhart and the PDP Research Group, MIT Press, Cambridge, Massachusetts, pp.216-271.
- Rumelhart, D.E., McClelland, J.L. and the PDP Research Group. (1986a). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1, Foundations*, MIT Press, Cambridge, Massachusetts.
- Rumelhart, D.E. Smolensky, P., McClelland, J.L. and Hinton, G.E. (1986b). Schemata and sequential thought processes in PDP Models, in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 2, Psychological and Biological Models*, J.L. McClelland, D.E. Rumelhart and the PDP Research Group, MIT Press, Cambridge, Massachusetts, pp.7-57.
- Schneider, W. (1987). Connectionism: Is it a paradigm shift for psychology?, *Behavior Research Methods, Instruments & Computers*, Vol.19, No.2, pp.73-83.
- Schön, D.A. (1963). *Displacement of Concepts*, Tavistock, London.
- Searle, J.R. (1987). Minds, brains and programs, in *Artificial Intelligence: the Case Against*, Born, R. (ed.), Croom Helm, Beckenham, Kent, pp.18-40.
- Sejnowski, T. and Rosenberg, C. (1986). NETtalk: A parallel network that learns to read aloud, John Hopkins University Electrical Engineering and Computer Science Technical Report JHU/EECS-86/01.
- Sejnowski, T. and Rosenberg, C. (1987). Parallel networks that learn to pronounce english text, *Complex Systems*, Vol.1, pp.145-168.
- Steels, L. (1989). Connectionist problem solving - an AI perspective, *Connectionism in Perspective*, R.Pfeifer, Z. Schreter, F. Fogelman-Soulie and L.Steels (eds), Elsevier Science Publishers BV, Amsterdam, pp.215-228.
- Stiny, G. (1980). Introduction to shape and shape grammars, *Environment and Planning B*, Vol.7, pp.343-351.
- Winograd, T. and Flores, F. (1986). *Understanding Computers and Cognition: a New Foundation for Design*, Addison-Wesley, Reading, Massachusetts.
- Winston, P. (1975). *The Psychology of Computer Vision*, McGraw-Hill, New York.
- Wittgenstein, L. (1922). *Tractatus Logico-Philosophicus*, Routledge and Kegan Paul.
- Wittgenstein, L. (1958). *Philosophical Investigations*, Macmillan, New York.

SPARK: an artificial intelligence constraint network system for concurrent engineering

R. E. Young, A. Greef and P. O'Grady

Group for Intelligent Systems in Design and Manufacturing
Department of Industrial Engineering
North Carolina State University
Raleigh NC 27695-7906 USA

Abstract. This paper presents a new approach to Concurrent Engineering, namely the use of Artificial Intelligence Constraint Networks to advise the designer on improvements that can be made to the design from the perspective of the product's life cycle. The difficulties associated with performing Concurrent Engineering are reviewed, and the various approaches to Concurrent Engineering are discussed. The requirements for a system to support Concurrent Engineering are indicated. An overview of constraint networks is given and this leads into a description of SPARK, an Artificial Intelligence Constraint Networks system for Concurrent Engineering. The operation of SPARK is illustrated by considering an example application of printed wiring board manufacture. The advantages of SPARK include being flexible enough to allow the designer to approach a problem from a variety of viewpoints, allowing the designer to design despite having incomplete information, and being able to handle the wide variety of life cycle information requirements.

INTRODUCTION

Concurrent Engineering (sometimes called Simultaneous Engineering or Life Cycle Engineering) involves the simultaneous consideration of product, function, design, materials, manufacturing processes and cost, taking into account later-stage considerations such as testability, serviceability, quality, reliability and redesign. In such a manner, life cycle factors are brought into consideration as early as possible in the life cycle, that is at the design stage.

Although the evidence is largely anecdotal, it appears that Concurrent Engineering is not generally performed well in western manufacturing industry in that design is carried out without due regard to the various life cycle factors. This can result in designs that are expensive to manufacture, test, service, maintain and redesign.

¹ This work was funded in part by the National Science Foundation, grant number DDM-8914200.

The fact that Concurrent Engineering is not performed well is due primarily to three main sources of difficulty: the characteristics of the design process, the volume and variety of life cycle knowledge, and the separation of life cycle functions [Evans (1988); Ruiz et al. (1970); Harfmann (1987); Baxter (1984); Peck (1973); Runciman, et al. (1985)].

This paper describes the application of Artificial Intelligence Constraint Networks to concurrent engineering to advise the designer on improvements that can be made to the design from the perspective of the product's life cycle. The format of the paper is as follows: the various approaches to Concurrent Engineering are discussed and the requirements for a system to support Concurrent Engineering are indicated. An overview of constraint networks is given and this leads into a description of SPARK, an Artificial Intelligence Constraint Net system for Concurrent Engineering. The operation of SPARK is illustrated by considering an example printed wiring board (PWB) application.

APPROACHES TO CONCURRENT ENGINEERING

There are a number of techniques and systems that support Concurrent Engineering by advising designers on aspects that reduce life cycle problems [Jakiela et. al., 1984]. These include the use of design teams [Heller (1971); Maddux and Jain (1986); Evans (1988); Harfmann (1987)], design handbooks [Bralla (1986); Trucks (1987); Stein and Strasser (1986)], checklists and structured procedures [Eversheim and Muller (1984); Heller (1971); Oakley (1984); Redford et. al. (1981); Redford (1986); Boothroyd et. al. (1981); Boothroyd and Dewhurst (1988); Kroll, et. al. (1988); Dieter (1983); Lu (1986); Gross et. al. (1987); Rosen et. al. (1986)], manufacturing simulation and process planning [Maddux and Jain, 1986], and the use of expert systems [Chao (1985); Sevenier et. al. (1986); Dixon (1986); Sackett and Holbrook (1988); Swift (1987); Chen and Young (1988); Bao (1988)].

The work done thus far on supporting Concurrent Engineering is limited. Design teams that possess the requisite life cycle information can produce good results. However, it is becoming increasingly clear that design teams not only can be difficult to manage but that they can be expensive to operate, especially for the medium/small volume producer. There are also difficulties associated with ensuring the team members have the necessary up-to-date expertise.

As an alternative to design teams, we therefore propose the use of a computerized support tool to bring up-to-date life-cycle information to the designer in a readily usable form. This support tool will, in effect, emulate a good design team by suggesting changes that will improve the design from the life-cycle perspective. The requirements for such a support tool are rigorous and encompass a number of substantial research issues [Bowen and O'Grady (1989)]:

- * it should be flexible enough to allow the design problem to be approached from a variety of viewpoints;
- * it should allow the designer to design despite the absence of complete information;
- * it should handle the large volume, variety, and interdependence of life-cycle information;
- * it should readily interface to database management and CAD systems;
- * it should have a good user interface and be able to explain itself in a manner comprehensible to humans;
- * it should support design audits.

CONSTRAINT NETWORKS

A constraint network is a collection of constraints which are interconnected by virtue of sharing variables. An example is shown in Figure 1. The variables (ovals in Figure 1) link the constraints (rectangles in Figure 1). The network contains the constraints that must be considered in determining a hole diameter in a printed wiring board. This example will be discussed in more detail later in this paper.

There are several constraint based languages including Sketchpad [Sutherland (1963)], ThingLab [Borning (1979)], IDEAL [Van Wyk (1981)], CONSTRAINTS [Sussman and Steele (1980)] and TK!Solver [Konopasek and Jayaraman (1984)]. All these constraint based languages have disadvantages when considered for a design advice system for Concurrent Engineering. The primary disadvantage is that they were mostly developed for fairly narrow application areas and they are therefore not suitable for application to the wide domain of Concurrent Engineering. The development of a viable constraint network language is therefore a crucial step in developing an effective Concurrent Engineering system. The recognition of this has lead to the development of **SPARK** which is a successor to CADEMA [O'Grady et. al. (1988)] and LEO [Bowen, O'Grady and Smith (1990)].

Bowen, O'Grady and Smith (1990), describe an initial constraint network language, LEO, and discuss basic issues of constraint propagation, nonmonotonic reasoning, the use of local propagation, and the computational complexity associated with constraint propagation in constraint networks. In the next section we describe the SPARK language and development environment that includes the capabilities of LEO and adds the constructs of universal quantification and frame-based inheritance structures.

SPARK: A CONSTRAINT NETWORK PROGRAMMING LANGUAGE

SPARK is a constraint network programming language with frame-based inheritance based upon an implementation of first order predicate logic. This allows users to model concurrent engineering systems as constraint networks, e.g., a collection of constraints that are interconnected through shared variables. The constraints, their shared variables, and their interconnections, can be represented graphically as a network. Figure 1 is an example constraint network for hole specification on printed wiring boards. The objective of a SPARK program is to find a set of variable values that doesn't violate any of the constraints. Values are propagated bi-directionally among constraints through the shared variables. The values are determined automatically by the system, when possible, or are input by the user. Consequently, within a SPARK-based constraint network, the user is utilized *a priori* as part of the inference mechanism.

The SPARK System

Figure 2 shows the SPARK system architecture. It consists of a full-screen editor with Wordstar-like editing commands. A program is created either as a text file and loaded into the editor, or created from within the editor. A function key initiates the scanner/parser. The SPARK program is scanned for syntax errors and then parsed into an internal representation. An initial constraint propagation is made by the inference engine and the results are displayed on the program interface.

The inference engine is a theorem prover that tries to find values for variables in the constraints that will make the constraints true. The inference engine uses a three-valued

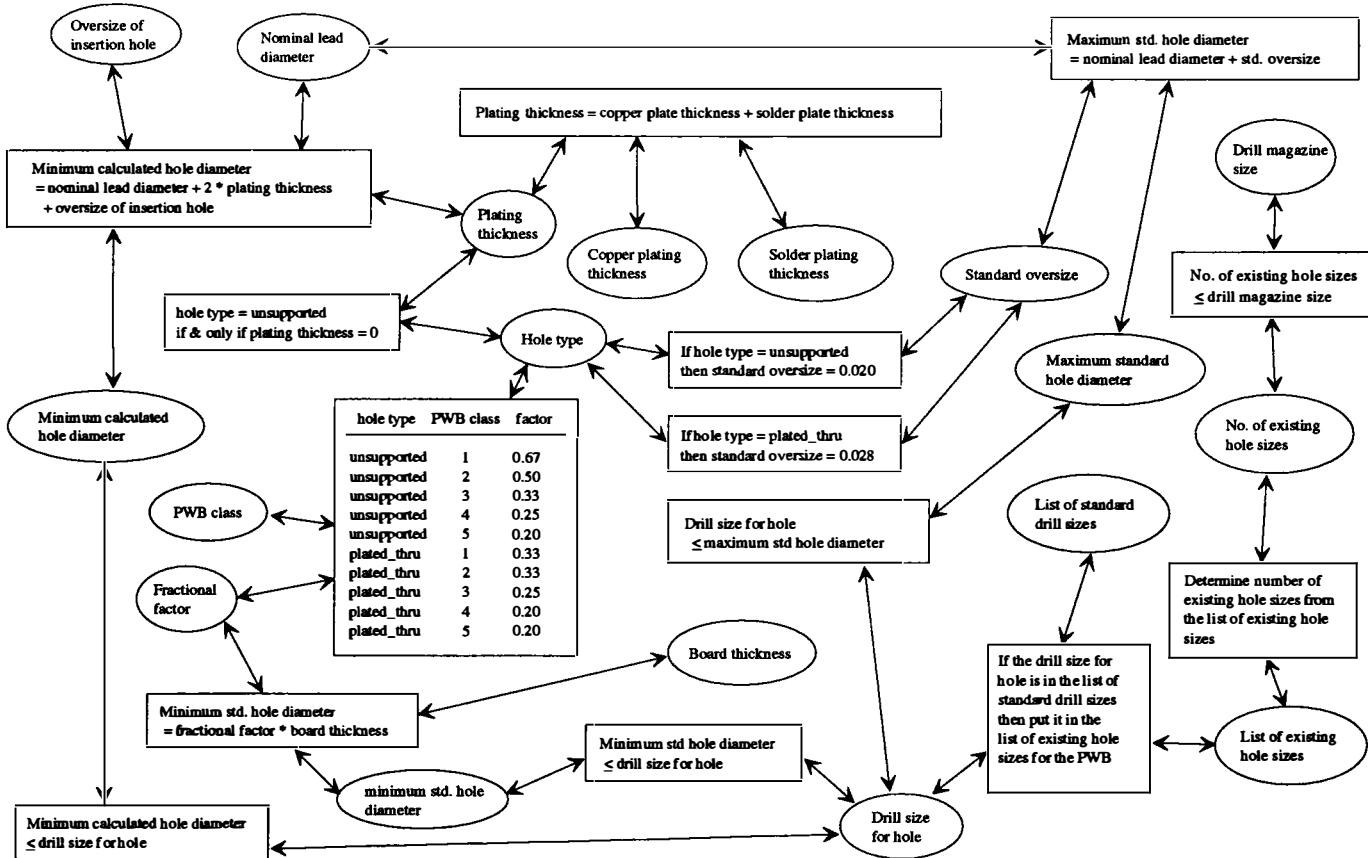


Figure 1. An example constraint network for selecting component lead hole sizes for printed wiring boards.
(modified from Smith, 1989)

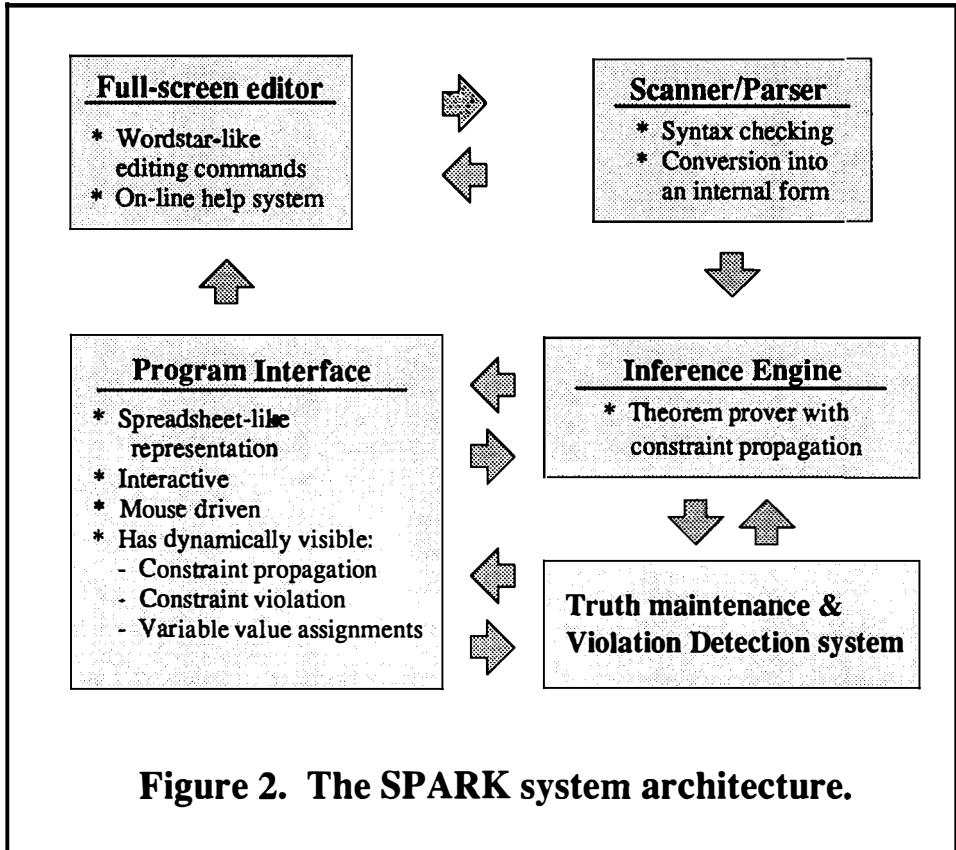


Figure 2. The SPARK system architecture.

logic so that a constraint can be either true, false or undetermined. If values are unknown for any variable in a constraint, then the constraint is logically undetermined and the inference engine will attempt to find values through local propagation. If it is not successful then it returns control to the user and the user must assign values. If it is successful in finding the missing values then the inference engine determines whether the constraint is logically true or false. A logically true constraint is a satisfied constraint and a logically false constraint is a constraint violation. When a constraint is violated the violation detection system traces the possible sources of the violation and displays them to the user. Correcting a violation usually involves changing the value of a variable.

When a variable's value is changed the truth maintenance system retracts the value from the variable and traces through all the dependency links to retract the other variable's values and the constraint's truth values that are dependent upon them. The system then performs a constraint propagation and the variable values are redetermined and constraints re-evaluated. This may result in the constraint being satisfied, its still being violated, or another constraint being violated. When all variables have been assigned values and all constraints are satisfied, then a solution to the problem has been found.

```

1      DOMAINS          /* Domain Declarations */
2      hole_type = {unsupported, plated_thru}.
3      class_type = {1, 2, 3, 4, 5}.
4      factor_type = {0.67, 0.50, 0.33, 0.25, 0.20}.
5      std_ovr_type = {0.020, 0.028}.

6      STRUCTURES        /* Frame-based Inheritance Structure Declarations */
7      plating_spec [
7a      'plating thickness'      (plate),      [0,0],
7b      'solder plating thickness' (solder),    [1,0] (0.0005),
7c      'copper plating thickness' (copper),   [2,0] (0.001) : number].
8      hole_spec [
8a      'nominal lead diameter' (lead_di),    [0,0] (0.02),
8b      'minimum calculated hole diameter' (calc_di), [1,0],
8c      'minimum standard hole diameter' (min_dia), [2,0],
8d      'maximum standard hole diameter' (max_dia), [3,0] : number].
9      unplated_hole [
9a      'hole type'            (holetyp)      [0,0] (plated_thru) : hole_type,
9b      'standard oversize'    (std_ovr)       [1,0] : std_ovr_type,
9c      'drill size for lead hole' (drill),     [2,0],
9d      'oversize of insertion hole' (oversiz)   [3,0] : number,
9e      /* inherited structure */ hole_data     [4,0] : hole_spec].
10     general_hole [
10a    /* inherited structure */ basic_hole   [0,0] : unplated_hole,
10b    /* inherited structure */ plating_data [8,0] : plating_spec].

```

Figure 3a. A SPARK program for hole specification in a printed wiring board: domain and structure declarations.

AN EXAMPLE: THE DESIGN OF PRINTED WIRING BOARDS (PWBs)

In this section we present an example that demonstrates the concurrent engineering objective to simultaneously consider the functional design requirements and the manufacturing requirements. The example is taken from Smith, 1989 and has also been used in Bowen, O'Grady and Smith (1990), and in O'Grady, Young, Greef and Smith (1991). The example illustrates how constraint networks can support correct functional design while helping the designer design a product that manufacturing can produce with existing capability and with reduced machine setups. Figure 3a through 3c is a **SPARK** program for the constraint network shown in Figure 1. It supports the specification of hole

```

11    VARIABLES          /* Variable Declarations */
12    hole1           /* 1st plated hole instance */      [4,4]: general_hole.
13    'size of the drill magazine' (mag_siz) [6,0] (3),
14    'board thickness' (board) [7,0],
15    'number of existing hole sizes' (no_hole) [8,0] : number.

16    'list of standard drill sizes' (std_drl) [9,0],
17    'list of existing hole sizes' (holelst) [10,0] : set of number.

18    'fractional factor' (factor) [11,0] : factor_type.
19    'PWB class' (class) [12,0] : class_type.

/* *** labels for column headings ***/
20a  "(variabl) [0,0],
20b  "(constra) [0,2],
20c  "(holes) [0,6],
20d  "(hole_1) [1,4],
20e  "(hole_2) [1,5],
20f  "(hole_3) [1,6],
20g  "(hole_4) [1,7],
20h  "(hole_5) [1,8],
20i  "(xxxxxxxx) [2,0], "(xxxxxxxx) [2,1]," (xxxxxxxx) [2,2],
20j  "(xxxxxxxx) [2,3], "(xxxxxxxx) [2,4]," (xxxxxxxx) [2,5],
20k  "(xxxxxxxx) [2,6], "(xxxxxxxx) [2,7]," (xxxxxxxx) [2,8] : label.

```

Figure 3b. A SPARK program for hole specification in a printed wiring board: variable declarations.

sizes for component leads on a printed wiring board (PWB). A PWB contains many holes, sized for the leads that will attach components to the board and to connect different conducting planes within a multi-layered circuit board. Hole size is determined by the board thickness, the need to accommodate the size and shape of the component lead, whether copper plating is needed to connect different conducting planes within a multi-layered board, and the need to physically and electrically connect the component lead to the PWB with solder. If the hole is too small, then after plating, a lead may not fit into it. If it does fit and there is insufficient oversize, then solder can't wick into the hole. If the hole is too large, then solder won't wick into it. Poor solder wicking results in poor solder joints and is a source of mechanical and electrical failure.

```

21   RELATIONS      /* Relation Declarations */
22   compatible(hole_type, class_type, factor_type) =
        ( unsupported, 1, 0.67), (unsupported, 2, 0.50),
        (unsupported, 3, 0.33), (unsupported, 4, 0.25),
        (unsupported, 5, 0.20), (plated_thru, 1, 0.33),
        (plated_thru, 2, 0.33), (plated_thru, 3, 0.25),
        (plated_thru, 4, 0.20), (plated_thru, 5, 0.20) ).

23   CONSTRAINTS    /* Constraint statements */
24   1      [3,2]      no_hole = card(holest).
25   2      [4,2]      no_hole <= mag_siz.

26   All (general_hole(X))
26a   [
26b   3      [5,2]      X.min_dia = factor*board.
26c   4      [6,2]      X.calc_di = X.lead_di + 2*X.plate + X.oversiz.
26d   5      [7,2]      X.plate = X.copper + X.solder.
26e   6      [8,2]      X.max_dia = X.lead_di + X.std_ovr.
26f   7      [9,2]      X.plate = 0 <=> X.holetyp = unsupported.
26g   8      [10,2]     X.holetyp = unsupported ==> X.std_ovr = 0.020.
26h   9      [11,2]     X.holetyp = plated_thru ==> X.std_ovr = 0.028.
26i   10     [12,2]    compatible(X.holetyp, class, factor).
26j   11     [13,2]    X.drill >= X.min_dia.
26k   12     [14,2]    X.drill <= X.max_dia.
26l   13     [15,2]    X.drill >= X.calc_di.
26m   14     [16,2]    X.drill element_of std_drl ==> X.drill -> holest. ]
27   15     [17,2]    std_drl = {0.024, 0.031, 0.043, 0.045, 0.052, 0.055, 0.063}.

```

Figure 3c. A SPARK program for hole specification in a printed wiring board: relation declarations and constraint statements.

From a manufacturing perspective, hole specification should be constrained by manufacturing capability and cost. Although there is potentially an infinite number of hole sizes, all production facilities have a finite set of drill sizes. The drills are held in the tool inventory. To avoid adding tool sizes to the tool inventory, a hole size should be selected that can be drilled by an existing tool. If the hole size variation on a board exceeds the magazine size for the drilling machine, then additional machine setups are required. If we can keep the hole size variation within the maximum drill magazine size, then we can manufacture the PWB with a single setup and avoid increased production time and cost. Consequently, the example successfully demonstrates the concurrent engineering objective

to simultaneously meet the functional design requirements and the manufacturing requirements.

The SPARK Program

A SPARK program consists of definitions and declarations. Each program must contain variable and constraint declarations. A program may also contain domain definitions, structure definitions, and relation definitions. Figures 3a, 3b and 3c contain the SPARK program for the network shown in Figure 1. The bold numbers on the left in the figure are not part of the program and are included to assist in discussing the example. The declarations and definitions are grouped together under identifying headings as shown in lines 1, 6, 11, 21 and 23.

Domain Declarations

Domain declarations allow the possible values for variables to be restricted to a programmer-defined set. When a variable has been restricted to a specific domain, the SPARK system will not allow the user to assign it a value outside the Programmer-defined domain. Definitions defining domains for some of the SPARK variables are shown in lines 2 through 5. Diverse domains are possible in SPARK. The syntax for a programmer defined domain declaration is:

domain name = {*list of possible values*}.

Variable Declarations

The variable declarations identify variables that are used in constraints. A variable's properties are defined within its declaration. These include its domain, whether it is a structure or not, whether it belongs to a structure or not, its description and name, and where it will be located within the spreadsheet-like interface. SPARK uses a two dimensional spreadsheet-like user interface to display the variables, any structures, and the constraints. The syntax for variable declarations is as follows:

'*variable name*' (*pseudonym*) [*location*] (*default value*) : *domain declaration*.

Default values are optional.

As an example, in Figure 3b line 15 contains the declaration for the variable **no_hole**. **no_hole** has a descriptive name of "number of existing hole sizes", a pseudonym of **no_hole** located in position [6,0] on the spreadsheet-like interface, no default value, and uses the domain of real numbers. Since the variables in lines 13 through 15 all have the domain of real numbers, they are grouped together and use a common domain declaration, ":number" (in line 15), specifying the real number domain. Other variables have explicitly defined domains grouped together under domain declarations and are a means of constraining variables to a finite set of possibilities.

Frame-based Inheritance Structures

Frame-based inheritance allows us to represent the set of all holes in the PWB by capturing the notion of a hole. Using inheritance structures we group together the general properties that define a hole. We then create *instances* of this general definition and tailor each

instance to define a specific hole. The "tailoring" is accomplished by satisfying the constraints. The use of inheritance structures in constraint statements is discussed in the section on universal quantification. The syntax for an inheritance structure declaration is:

inheritance structure name [x_1, \dots, x_r].

where $x_k, k=1, \dots, r$ are either variables, v_i , or are other inheritance structures, is_j , that are incorporated into the definition. $v_i, i=1, \dots, n$ are variable declarations whose syntax is defined in a previous section. $is_j, j=1, \dots, m$ define other structures from which properties are inherited and have the syntax:

structure instance name [location] : referenced inheritance structure name.

Figure 3a contains an example of a complex inheritance structure. In the figure there are four inter-related structures, **plating_spec** (line 7), **hole_spec** (line 8), **unplated_hole** (line 9), and **general_hole** (line 10).

When we need to describe a specific object, such as a specific hole in a circuit board, we create an *instance* of the general object and assign it a unique name. The name acts as a unique identifier for each instance. To denote a specific object's properties we assign values to the variables for each instance. In the example, we would denote a specific hole's properties by assigning values to the variables for each hole we instantiated. In Figure 3b, line 12 creates the first instance of a specific hole. Other instances of holes are created dynamically at the user interface by the user as they work through all the holes on a PWB. Consequently, the example in Figure 3 is a generic program that can be used to determine hole specifications for a PWB with any number of holes. Figure 4 is an example of a user introduced instantiation. It is a screen image from the spreadsheet-like interface. It shows variables (column 0), the constraints C1 through C15 (column 2), and the first instantiation of a hole (column 4).

Relation Declarations

Relations are a means of representing and indexing among tabular information. A relation is defined in Figure 3c, line 22 for the tabular constraint shown in Figure 1. A relation has the following syntax:

relation name (domain 1, ..., domain n) =

{ (v₁₁, ..., v_{1n}), ..., (v_{i1}, ..., v_{in}), ..., (v_{m1}, ..., v_{mn}) }

where there are $i = 1, \dots, m$ rows and $j=1, \dots, n$ columns of values, v_{ij} , in the table.

If $n-1$ values are assigned to n variables in a relation, the missing variable value will be determined by the inference engine from searching the table.

Universal Quantification of Inheritance Structures

Although frame-based inheritance allows us to differentiate among different objects which have the same general properties, it says nothing about how we should apply constraints to the instances that define specific objects. To specify how constraints are to be applied to instantiated objects we have included universal quantification for structures. This is similar to universal quantification for predicate calculus except that it is restricted to structures.

```

----- Spark User Interface -----
0      1      2      3      4      5      6      7      8
0 variabl    constra          holes
1      2      3      4      5      6      7      8
2 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
3      4      5      6      7      8
3     ►C1◄      hole_1      hole_2      hole_3      hole_4      hole_5
4     ►C2◄      holetyp
5     ►C3◄      std_ovr
6 mag_siz   ►C4◄      drill
7 board     ►C5◄      oversiz
8 no_hole    ►C6◄      lead_dia
9 std_drl    ►C7◄      calc_dia
10 holelst   ►C8◄      min_dia
11 factor    ►C9◄      max_dia
12 class     ►C10◄     plate
13          ►C11◄     solder
14          ►C12◄     copper
15          ►C13◄
16          ►C14◄
17          ►C15◄
18
=>inst hole2 [4,5] : general_hole.
SPARK Ver 1.0 Copyright (C) 1990 NCSU Industrial Engineering Department =
Enter command. <CR> to process. <ESC> to cancel.

```

Figure 4. Screen printout of the SPARK spreadsheet-like interface for the example with instantiation command for hole 2 entered on the command line.

The syntax for universally quantifying inheritance structures is:

All (is₁(X), ..., is_n(X)) {cs₁, ..., cs_m}

where *is_i*, *i=1, ..., n* are the inheritance structures to be quantified, and

cs_j, *j=1, ..., m* are the constraints within the quantified structure. The syntax for constraints is defined in the next section. A constraint, *cs_j*, can contain variables applicable to all objects and variables applicable to instances of an object. Variables applicable to instances of an object are preceded by a capital letter, such as X, and a period.

In the example, to differentiate between values applicable to all holes and values uniquely applicable to a specific instance of a hole we quantified the structure **general_hole** as shown in line 26 of Figure 3c in which All (**general_hole(X)**) appears. Line 26 specifies that the set of values for each instance of a hole is propagated through the constraints enclosed within the left and right square brackets. The variables preceded by an X. appear within the super-structure **general_hole**. As a new value is assigned to a variable within an instance of a hole all the variables within the instance's structure are propagated through the constraints. In this way, the constraint network of Figure 1 is applied to each instance of a hole for the PWB.

Constraint Declarations

Constraints are declared in lines 24 through 27. The syntax for a constraint is:

constraint number [location] constraint.

The numbers in square brackets define the position on the spreadsheet-like interface where the constraint will be located. The constraint number identifies the constraint in the interface display. For example, in Figure 3c, line 25 shows constraint number 2 restricting the number of selected hole sizes to be less than or equal to the size of the drill magazine. The constraint is located in row 4 of column 2 in Figures 4, and is identified by C2. All constraints in the spreadsheet-like interface are identified by their constraint number preceded by a "C". Thus, in Figures 4 all fifteen constraints are located in column 2.

Constraints are defined using logical operators. Constraints in Figure 3c contains both atomic and logic constraints. Examples of atomic constraints with equality (=) and with inequality ($=<$) are shown in lines 24 through 25. Examples of atomic constraints with the set operators, inclusion testing (**element_of**) and element insertion ($->$), are shown in lines 26m. Another set operator, cardinality (**card**), is shown in line 24. Logic constraints are composed from atomic constraints using negation (the "not" operator), conjunction (the "and" operator, &), disjunction (the "or" operator, #), logical implication (the "implies" operator, ==>), and logical equivalence (the "equivalence" operator, <=>). Conjunction, disjunction and negation are not included in this example; however, an example of logical equivalence (<=>) is shown in line 26f. Conditional constraints using implication (==>) are shown in lines 26g, 26h and in line 26m. In a conditional constraint, the antecedent (i.e., X.holetyp = plated_thru in line 26h) acts as a guard expression for the consequent (X.stovr = 0.028). The consequent is asserted only when the antecedent is satisfied.

OPERATION OF SPARK

To illustrate the operation of **SPARK**, let us assume the designer has instantiated a new hole, say hole four, and now proceeds to determine a drill for it. Let us also assume that the following values for variables in hole4's structure have been set to:

hole type (holetyp)	plated_thru
solder plating thickness (solder)	0.0005
copper plating thickness (copper)	0.001
maximum standard hole diameter (max_dia)	0.048
minimum standard hole diameter (min_dia)	0.025

In addition, let us assume that as a result of specifying variable values for previous holes, the list of existing holes is:

list of existing hole sizes (holelst) 0.031, 0.045, 0.063

The designer now proceeds to select a drill size (drill) for hole4 which has a minimum calculated hole diameter (calc_di) of 0.043. Since the minimum calculated hole diameter (calc_di) now exceeds the minimum standard hole diameter (min_dia) for this hole, constraint 11 in line 26j will never be violated; thus, for this hole, the minimum calculated hole diameter defines the minimum range value. If the designer decides to use a value (such as 0.052 as shown in Figure 5) outside the range defined by the calculated hole

```

Spark User Interface
0: SOBJ->drill size for lead hole  D: number
    0   1   2   3   4   5   6   7   8
0  variabl  constra  holes
1                               VARIABLE
2
3  Long Variable Name: SOBJ->drill size for lead hole
4
5  Short Variable Name: general_hole:hole1+
6          unplated_hole:basic_hole.drill
7
8  Variable Domain: number
9
10 Variable Value: 0.052
11
12
13
14
15
16
17
18
=>
= SPARK Ver 1.0 Copyright (C) 1990 NCSU Industrial Engineering Department =
Please enter the new variable value. (It must be a number)

```

Figure 5. Variable assignment window shown with 0.052 assigned to the variable for the selected drill size.

```

Spark User Interface
0: SOBJ->drill size for lead hole  D: number
    0   1   2   3   4   5   6   7   8
0  variabl  constra  holes
1                               VIOLATIONS
2
3  Constraint 12.001 is violated due to the following user assignments :
4          general_hole:hole1+unplated_hole:basic_hole.drill = 0
5          .052
6          general_hole:hole1+unplated_hole:basic_hole:hole_spec:hole_da-
7          ta.lead_di = 0.02
8          general_hole:hole1+unplated_hole:basic_hole.holetyp = plated_
9          thru
10
11
12  Do you want to save this in the
13      VIOLATE.TXT file ? (y/n):
14
15
16
17
18
=>
= SPARK Ver 1.0 Copyright (C) 1990 NCSU Industrial Engineering Department =
Please enter the new variable value. (It must be a number)

```

Figure 6. Constraint violation window showing that constraint 12 has been violated.

diameter (`calc_di`) or the maximum standard hole diameter (`max_dia`), 0.043 through 0.048, **SPARK** detects a violation of either constraint 12 or constraint 13 and will notify the designer. Figure 6 shows a violation of constraint 12. If the designer decides to use a non-standard drill size, **SPARK** also detects a violation of constraint 14.

Suppose, however, that the designer decides to use a standard drill size of 0.043, which satisfies the constraints of being a standard drill size and of being within the minimum or maximum range for this hole (constraints 12, 13 and 14). **SPARK** then detects a violation of constraint 2 and will then point out to the designer that the number of bits would exceed the capacity of the drill magazine.

At this stage, the designer could decide to use a different drill machine with a larger drill magazine size, or the designer could choose a different drill size. Of these options, the more sensible approach would be to use a drill size of 0.045 inches. If this is done, **SPARK** will check the network for constraint violations and finding none will report this to the designer. In this way a designer would work through all the hole sizes in a printed wiring board by instantiating a hole and then assigning values to the variables in the hole's frame-based inheritance structure until all variables in all structures for all holes have been assigned values that do not violate any constraints.

This example illustrates how the constraint networks in **SPARK** provide life-cycle information to a designer that results in hole specifications for a printed wiring board using standard drill sizes while remaining within drill magazine capacity, thereby reducing both potential manufacturing problems and cost.

SUMMARY AND CONCLUSIONS

Concurrent Engineering is important in that it is at the design stage that much of a products costs are specified. This paper has reviewed the difficulties associated with performing Concurrent Engineering, discussed the various approaches to Concurrent Engineering, and described the requirements for a system to support Concurrent Engineering. A new approach to Concurrent Engineering that uses AI-based constraint networks to meet these requirements has been presented. The **SPARK** constraint network programming language has been described. An example Concurrent Engineering problem of PWB design is presented. This example demonstrates a system that combines design guidelines with manufacturing constraints to help a designer produce a PWB that simultaneously meets its functional specifications and could be manufactured on existing resources with minimum machine setups.

The use of AI constraint networks in **SPARK** provides a powerful system to support Concurrent Engineering. The advantages of **SPARK** include being flexible enough to allow the designer to approach a problem from a variety of viewpoints, allowing the designer to design despite having incomplete information, and being able to handle the wide variety of life cycle information requirements. This approach also allows the integration of knowledge from all aspects of the product's life-cycle into a common format that is accessible to the designer.

REFERENCES

- Bao, H. P. (1988). An Expert System for SMT Printed Circuit Board Design for Assembly, *Manufacturing Review* 1(4).

- Baxter, R. (1984). Companies must Integrate—or Stagnate! *Production Engineering*, Institute of Production Engineers, London (April).
- Boothroyd, G., Swift, K. and Redford, A. (1981). *Design for Assembly Handbook*, University of Salford Industrial Centre Ltd, UK.
- Boothroyd, G., Dewhurst, P. (1988). Product Design for Manufacture and Assembly, *Manufacturing Engineering*, Society of Manufacturing Engineers, USA (April).
- Borning, A. (1979). ThingLab—A Constraint-Oriented Simulation Laboratory, *Stanford Technical Report STAN-CS-79-746*, Stanford University, USA.
- Bowen, J. and O'Grady, P. (1989). Characteristics of a Support Tool for Life Cycle Engineering, *LISDEM Technical Report*, North Carolina State University, USA.
- Bowen, J., O'Grady, P. and Smith, L. (1990). A Constraint Programming Language for Life-Cycle Engineering, *Artificial Intelligence in Engineering*, 5(4).
- Bralla, J. G. (ed.) (1986). *Handbook of Product Design for Manufacturing*, McGraw Hill, USA.
- Chao, Nien-Hua (1985). The Application of a Knowledge Based System to Design for Manufacture, *IEEE Document number CH2125-7/85/0000/0182*.
- Chen, Y. and Young, R. E. (1988). PACIES (A Part Code Identification Expert System), *IIE Transactions*, 20(2): 132–136.
- Dieter, G. E. (1983). *Engineering Design: A Materials and Processing Approach*, McGraw Hill, USA.
- Dixon, J. R. (1986). Artificial Intelligence and Design: A Mechanical Engineering View, *Proceedings of Fifth National Conference on Artificial Intelligence (AAAI-86) Volume 2*, American Association of Artificial Intelligence, Philadelphia, PA.
- Evans, B. (1988). Simultaneous Engineering, *Mechanical Engineering*, 110(2), American Society of Mechanical Engineers, New York.
- Eversheim, W., and Muller, W. (1984). Assembly Oriented Design, in W. B. Heginbotham (ed.), *Programmable Assembly*, IFS Ltd, UK.
- Gross, M., Ervin, S., Anderson, J. and Fleisher, A. (1987). Designing with Constraints, *Computability of Design*, John Wiley, USA.
- Harfmann, A. C. (1987). The Rationalizing of Design, *Computability of Design*, John Wiley, USA.
- Heller, E. D. (1971). *Value Management: Value Engineering and Cost Reduction*, Addison-Wesley, USA.
- Jakela, M., Papalambros, P., Ulsoy, A. G. (1984). Programming Optimal Suggestions in the Design Concept Phase: Application to the Boothroyd Assembly Charts, *ASME Technical Paper No. 84-DET-77*, American Society of Mechanical Engineers, New York.
- Konopasek M. and Jayaraman, S. (1984). *The TK!Solver Book*, Osborne/McGraw-Hill, Berkeley, CA.
- Kroll, E., Lenz, E., Wolfberg, J. R. (1988). A Knowledge-Based Solution to the Design for Assembly Problem, *Manufacturing Review*, 1(2), American Society of Mechanical Engineers, USA.
- Lu, C-Y. S. (1986). Knowledge Based Expert Systems: A New Horizon for Manufacturing Automation. *Knowledge Based Expert Systems for Manufacturing*, PED-Vol. 24, American Society of Mechanical Engineers, New York.

- Maddux, K. C. and Jain, S. C. (1986). CAE for the Manufacturing Engineer: The Role of Process Simulation in Concurrent Engineering, in A. A. Tseng, D. R. Durham, and R. Komanduri (eds), *Manufacturing Simulation and Processes*, ASME, New York.
- O'Grady, P., Young R., Greef A. and Smith L. (1991). An Advice System for Concurrent Engineering, *International Journal of Computer-Integrated Manufacturing* (April).
- O'Grady, P. J., Ramers D. and Bowen J. (1988). Artificial Intelligence Constraint Nets Applied to Design for Economic Manufacture and Assembly, *Computer Integrated Manufacturing Systems*, 1(4).
- Oakley, M. (1984). *Managing Product Design*, John Wiley, USA.
- Peck, H. (1973). *Designing for Manufacturing*, Pitman, London.
- Redford, A. H., Swift, K. G., Howie, R. (1981). Product Design for Automatic Assembly, *Proceedings 2nd International Conference on Assembly Automation*, IFS (Conferences) Ltd, UK.
- Redford, A. (1986). Software Aid to Design for Assembly , *Assembly Automation*, IFS Publications, UK.
- Rosen, D., Riley, D., Erdman, A. (1986). Development of an Inference Engine, with Applications to Mechanism Synthesis, Part 1: Introduction of the Inference Engine, *Technical Report 86-DET-156*, New York.
- Ruiz, C., Koenigsberger, F. (1970). *Design for Strength and Production*, Gordon Breach Science Publishers, New York.
- Runciman, C. and Swift, K. (1985). Expert System Guided CAD for Automatic Assembly, *Assembly Automation*, 5(3) (August).
- Sackett, P. J. and Holbrook, A. E. (1988). DFA as a Primary Process Decreases Design Deficiencies, *Assembly Automation*, (August), pp. 137-140.
- Sevenler, K., Reghupathi, P. S., Altan, T., and Miller, R. A. (1986). Knowledge-Based System Approach to Forming Sequence Design for Cold Forging. Knowledge Based Expert Systems for Manufacturing, *PED-Vol. 24*, American Society of Mechanical Engineers, New York.
- Smith, L. (1989). Life Cycle Engineering and Constraint Nets Applied to Printed Circuit Board Design, *IMSEI Project Report*, Integrated Manufacturing Systems Engineering Institute, North Carolina State University, USA.
- Stein, J., Strasser, F. (1986). Metal Stamping, in J. G. Bralla, J.G. (ed.), *Handbook of Product Design for Manufacturing*, McGraw Hill, USA.
- Sussman G. and Steele, G. (1980). CONSTRAINTS—A Language for Expressing Almost-Hierarchical Descriptions, *Artificial Intelligence*, 14: 1-39.
- Sutherland, I. (1963). SKETCHPAD: A Man-Machine Graphical Communication System, *IFIPS Proceedings of the Spring Joint Computer Conference*.
- Swift, K. (1987). *Knowledge Based Design for Manufacture*, Kogan Page.
- Trucks, H. E. (1987). *Designing for Economical Production*, Society of Manufacturing Engineers, Dearborn, MI.
- Van Wyk, C. (1981). IDEAL User's Manual, *Bell Labs Computer Science Technical Report 103*.

A constraint-driven approach to object-oriented design representation

C. M. Coupal,[†] P. G. Sorenson[‡] and J-P. Tremblay[†]

[†]Department of Computational Science
University of Saskatchewan
Saskatoon Saskatchewan S7N 0W0
Canada

[‡]Department of Computing Science
University of Alberta
Edmonton Alberta T6G 2H1
Canada

ABSTRACT. This paper summarizes on going research into constraint specification in an object-oriented environment with possible application to computer-assisted architectural design of residential buildings. We present the paradigm in practical terms and demonstrate with examples how the paradigm is applied to architectural design problems. The paradigm contributes to an understanding of the relationships between constraint requirements of a design problem and the design objects manipulated by the designer, working towards a design solution.

1. INTRODUCTION

In this paper we propose a paradigm, not yet implemented, to create class descriptions for an object-oriented environment that capture the characteristics and behaviour of real world objects. This is accomplished through the definition and incorporation of design constraints on those real world objects. The main notions of this paradigm are two-fold.

First, constraints are used to drive the definition and development of classes used to populate the object-oriented environment in which the designer will later work. These constraints are transformed into class features which, during object instantiation, are implicitly satisfied by the object-oriented system. Second, constraints on objects are used to specify the attribute values during the actual design of an assembly structure, such as a residential building. Using constraint satisfaction implicit to the objects, the system can guide the designer through the many decisions to be made. Since the constraints are integrated with

the class definitions of the objects, the designer need not worry about overlooking some crucial aspect of the design.

We begin by introducing architectural design of residential structures and describe briefly the types of problems faced by the designer. In Section 3, we present constraints as a means of capturing the necessary knowledge about the design objects as class definitions. Section 4 introduces the paradigm, in practical terms, and presents a brief example of the paradigm to demonstrate how it can be applied. Section 5 presents the conclusions and future directions for this work.

2. ARCHITECTURAL DESIGN OF SMALL BUILDINGS

The design and construction of residential buildings tend to be relatively low-cost, often one-of-a-kind projects. This places unique restrictions on the tools and methods one may use to design and build such structures. The design and production of blueprints for a structure must be kept inexpensive (rarely more than 1% of the final structure cost) while providing accuracy and detail sufficient for construction work. This section briefly discusses the architectural design problem and characteristic approaches to arriving at a solution. Various existing computer-aided design methods are discussed.

2.1 Architectural Design Characteristics

The initial sketches and documentation provided by the client for a structure can vary widely in detail, from the revision of the designs for a completed project, to a set of rough sketches and concepts. The designer must integrate this information to produce a design that meets the needs and desires of the client, and the regulations of existing building codes at local and national levels. Building materials and local construction methods must also be considered. Much of the detail found in the final design solution is derived by integrating these various styles and building requirements.

As the solution develops, newly added requirements may conflict with existing ones; rarely does one have all requirements specified before beginning a design. If a conflict exists between a desired requirement and an absolute requirement, the desired one must be rejected. If a conflict is between two desired requirements, either can be selected. Requirements have varying degrees of importance from absolute requirements, such as the regulations in governing building codes, to optional ones which can be rejected without consequence if they are not met. There can be varying degrees of optional constraints from strongly desired to mildly desired. The typical design process is to find a set of specifications which meet all absolute requirements and as many optional requirements as possible.

In an effort to reduce the complexity of producing an acceptable solution, and keep costs down, existing designs are often modified. Existing designs have the virtue of already satisfying a set of requirements. If an existing design can be found with requirements very much like those of the new design, it is more effective and efficient to modify the existing design. In the authors' experience, a residential design rarely begins without reference to existing structures.

As modifications are made, the designer must ensure that the changes do not invalidate the design; a local alteration to some portion of the structure may result in global changes that are invalid. For example, moving a wall by some dimension may cause a room at the opposite end of the structure to become too small, although there is no other relationship between the room and the modified wall (see Figure 1).

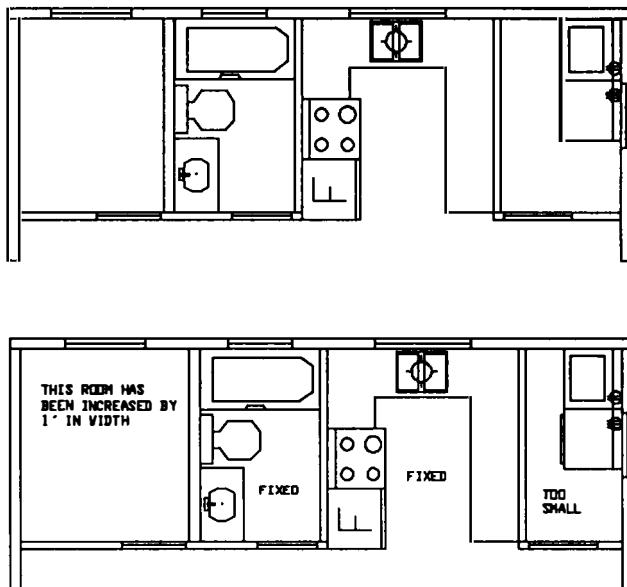


Figure 1: Original (upper) and modified (lower) floor plans.

An object-oriented approach to design representation enhances the reuse of existing designs since the reuse of class definitions is a integral part of the object-oriented philosophy to design management. By exploiting the object-oriented notions of reuse and extensibility in architectural design, similar benefits may be obtained.

2.2 Existing Methods of Computer-Aided Architectural Design

Developments in computer technology are being applied to the development of design solutions to architectural problems. A simple application is the microcomputer-based drafting system which does little more than replace the drafting table and pencil with a mouse and video screen. In large-scale buildings where there is much repetition of design modules (each floor of a high-rise office building will be modified only slightly from the others) the ability of the computer to repeatedly copy, scale and draw design components with little modification, becomes an advantage. In small-scale construction where there is little repetition and a greater degree of uniqueness, the use of a computer can be more time consuming than traditional design methods (Stewart, 1990). Much of the potential benefit of applying a computer to the design problem is lost due to the characteristics unique to residential architecture.

Many computer-aided design and drafting (CADD) systems have extended the basic capabilities to include the definition of commonly occurring drawing units known as *components*. A *component* is a series of drawing commands formed into a unit and given a name. The *component* can then be referenced by name, scaled, moved, rotated, and manipulated in many other ways, as a unit. This is an advantage when the same group of drawing elements are often related in a common way. It is less beneficial in residential drawing where the repetitive elements (electrical symbols, plumbing fixtures, etc.) are quite simple to begin with and tend to be few for any single design. In addition, a large library

of components is required to store the many variations present in residential structures creating the problem of locating the appropriate component among many with the correct specifications. In most systems, the components are stored as image files that can be referenced by name only (Logitech, 1986).

The use of libraries of components, and the reuse of typical detail drawing segments can increase productivity in a manufacturing environment. It has been shown that where many buildings are constructed from the same basic design with minimal variations, design productivity can increase by up to 75% (Stewart, 1990). CADD systems provide the designer with a greater degree of control over the design process by extending their drawing production capabilities.

Additions can be made to CADD software to make it more functional. The development of specialized program modules which can interface either with the CADD software, or are compatible with the drawing file formats, can assist in the design of specialized building features. Automatic generation of elements such as double-wall construction, hip-roof systems and special roof trusses exist as add-on or stand-alone programs (Smith, 1990). These programs recognize the need for assistance to design beyond simple drawing capabilities. They contain specific knowledge about particular design problems and a particular method of solution. This is beneficial in that a designer can quickly produce a working design for a particular problem if the problem fits the model inherent in the assistant program. But it is a drawback because the designer can not apply any creativity or alternate model to the problem since that model is not contained within the program.

The most sophisticated design applications are knowledge-based design generation systems. Such systems have evolved from early research in VLSI design (Gosling, 1983, Adiba, 1984, Horstmann, 1984, Barbuceanu, 1984, Adiba, 1986). These systems, many in research stages of study and development, tend to focus on the design activity and how a designer develops a solution (Bijl, 1985). Knowledge is often captured as propositions or statements of fact as they relate to the design problem domain. The design activity becomes one of specifying the constraints on a design and allowing the design system to search the solution space for one or more acceptable solutions (Coupal, 1985, Kalay, 1985, Akin, 1986, Balachandran, 1987). Varying degrees of success have been achieved in producing specialized design configurations (Fawcett, 1986), room layouts as systems of loosely packed rectangles (Flemming, 1986), kitchen layouts based on rules in a knowledge base and a dialogue with the designer (Rosenman, 1986), architectural detailing based on design grammars and a generative expert system (Radford, 1985), and application to design problems unique to high-rise buildings (Rehak, 1985, Cooper, 1987).

The development of knowledge-based systems tend toward automation of a part of the design process, capturing knowledge in rule form and providing the designer with an assistant-like tool. It is our belief the concept of an assistant, as opposed to fully automated design, seems the most plausible for residential architectural design, since it permits the creativity of the designer and integrates well with the generally held notions of the design process (Jakiela, 1989).

2.3 The Need for a Constraint-Driven Object-Oriented Design Representation

The residential design problem is not a likely candidate for current sophisticated knowledge-based design systems because of the cost and complexity associated with such systems, and the required level of expertise of the system user. Our paradigm attempts to incorporate design knowledge, as constraints, directly into an object-oriented environment suitable for residential design systems of smaller scale. We chose this approach due to a number of

factors.

- 1) The use of components in simple CAAD systems can be exploited further by the reuse of class definitions in extensive class libraries. Class definitions are more flexible than CAAD components found in simple CAAD systems.
- 2) Architectural systems applied to residential housing must be low in cost and simple to use. By incorporating limited knowledge directly into class definitions compiled into an executable system forming a design environment, we should be able to obtain efficiency and maintain a lower cost.
- 3) The process of design is computation-intensive requiring the designer to be cognizant of many details at a time. The computer is well suited to such tasks in the role as intelligent assistant, with the designer assuming the role of design supervisor and conflict arbitrator.
- 4) Partial or incomplete requirements at the outset of design development requires the design system to be flexible in its representation model. The object-oriented paradigm accommodates such flexibility through its use of *inheritance* and *redefinition* mechanisms.

These considerations have prompted the study into methods of capturing constraints directly in class definitions.

3. DESIGN REQUIREMENTS (CONSTRAINTS) AS ABSTRACT SOLUTIONS

We can consider the initial text descriptions and sketches as a constraint-based abstraction of a set of final solutions. As mentioned, the descriptions and sketches consist of required elements (strong constraints) and desired elements (weak constraints) forming a hierarchy of constraints (Borning, 1987). This abstraction becomes more detailed when additional constraints from the use of particular building materials, methods and building codes are added to the original descriptions and sketches. At some point, there is enough detail in the set of constraints to begin design of the actual structure object and its components. Strong constraints can be implemented in the design environment, while weak constraints may be implemented through design manipulation at the discretion of the designer.

The constraints accumulated thus far deal with the observable characteristics of the single design object (house) and its many component objects. These observable characteristics include the classification of the object (in the context of the design activity), the quantifiable properties of interest (attributes) (Takagaki, 1990), and the role to be played by the object within the structure (methods).

The object-oriented paradigm captures all these notions well and is a logical choice for a representation model of architectural design incorporating constraints in the design activity. It is widely recognized that the object-oriented design paradigm is well suited to engineering and CAD applications (Lorie, 1983, Barbuceanu, 1984, Lorie, 1985, Kemper, 1987, Kim, 1987, Cmelik, 1988, Katabchi, 1988).

The structure of the object is reflected through the existence of non-primitive objects (i.e. objects not inherent to the object-oriented system but rather added to the system by the designer for use in the design environment). The *is-part-of hierarchy* captures the structural characteristics present in the abstract definition.

Object behaviour is captured through the methods associated with the object as well as

any pre- and post-conditions that might exist. For example, a pre-condition to the insertion of a structural beam may be that support posts are in place.

The notion of a *class* allows the designer to define templates for objects of a given type, to be used in future designs. Specific properties can be manipulated when an object is created through instantiation. Any constraints inherent in the class definition automatically apply to the instantiated object and are, therefore, implicitly satisfied. Classes can be closed by compiling them into the design environment library, and extended through *inheritance*, by defining descendent classes based on closed parent classes (Meyer, 1988, 1989).

The design activity becomes a two-stage process of defining classes as needed for the design problem from the constraint system, and then manipulating object attributes as required by a specific design solution. Actual attribute values can be specified as constants of primitive object types, or algebraic and constraint expressions consisting of primitive and complex object references. These references may be to other object attributes, or to functional methods of various objects. Through this mechanism, the final relationships governed by the problem constraints can be implemented and a particular solution realized. In some instances, with a few changes to constant specifications an alternate solution can be quickly obtained. (Figure 1 was generated in this manner: the changes evident in the lower floor plan were accomplished by adjusting one identifier in the location expression of one wall. All other expressions remained the same producing new values when re-evaluated.)

4. A PARADIGM FOR DESIGN DEVELOPMENT

Figure 2 depicts the notion of design as a two-phase approach: class definition through constraint transformation and environment update, and object instantiation and attribute manipulation through constraint specification. In this model, constraints form the first (abstract) specification of the set of design solutions. Through transformation, these constraints are captured in the various mechanisms of class definition in an object-oriented design environment, and implicitly satisfied when objects are instantiated.

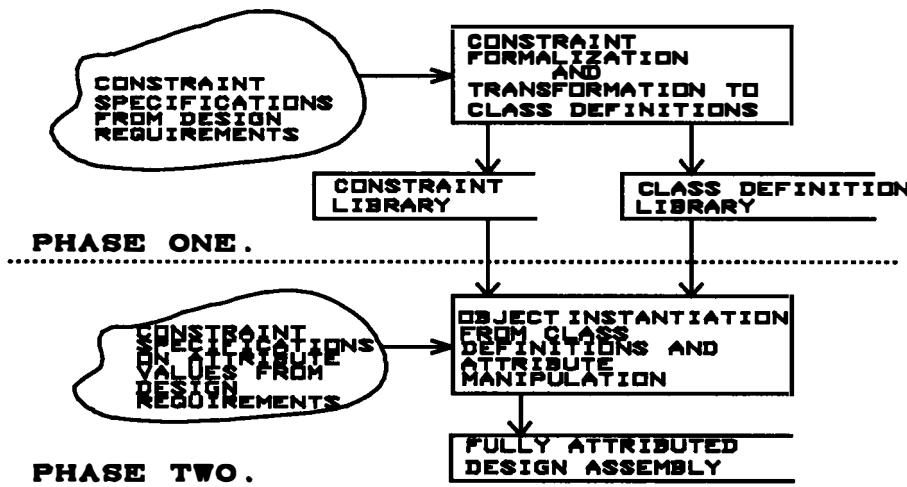


Figure 2: Two-Phase constraint driven design approach.

The set of classes in the environment are utilized by the designer to instantiate objects as required by the implicit constraints now contained in the class definitions, and as required by manipulating attributes of these instantiated objects. In effect, manipulating attributes is a selection process that reduces the set of potential solutions to a single solution. Each solution in the solution set is an architectural structure object with a unique state (set of attribute values). The final outcome is an object with all attributes assigned values: the residential structure.

4.1 Capturing Constraints in Class Definitions

From the constraint specifications we can capture class definitions for structure (is-part-of), attribute type (is-a), behaviour (methods), and restrictions on attribute values (invariants, pre- and post-conditions). To do this, we must identify those components of a constraint specification that represent each of these aspects of a class.

The *is-a*, or type characteristic comes from abstracting references to general objects in the constraints, to classes in the object-oriented environment. The constraint phrase:

all windows must have a header if....,

makes reference to two specific objects, *window* and *header*. The quantifier *all* serves to indicate there can be more than one. In each case, the specific object is a representative of a class of objects and thus serves to indicate the need for the creation of a class (class definition).

Other constraints may serve to extend the type definition by further reference to the object in a different context. For example, *a window is a construction component that may be placed into exterior walls*. This reference adds to the type requirement of *window* by saying that the class of objects, represented by *window*, is also a class of construction objects. If a class of construction objects already exist, the *window* class will inherit some of its characteristics from the more general construction object class. If it doesn't, this constraint indicates a need for a construction object class.

To identify class definition requirements, one looks at the constraint specification for *is-a* references and defines, or specifies inheritance for, the necessary class accordingly. Whether to define or inherit depends on the relative importance of the constraint and its context within the rest of the design environment.

The component objects or primitive-value attributes of interest for a given class are obtained from references to relationships with other objects. The phrase ... *must have a header* ... serves to indicate an attribute *header* in the class *window*. What the attribute is remains undefined unless further specification is found in the constraint.

The phrase ... *that may be placed into exterior walls*, is ambiguous without further clarification since it is not apparent from the statement whether the constraint is on the *window* object (*windows* may only be placed into exterior walls), or the *wall* object (*exterior walls* may contain *windows*). Both are possible constraints on the design environment.

Consider the interpretation that exterior walls may contain *window* components. This constraint requires the *wall* object to have an attribute consisting of a set of *window* objects. The set may be empty due to the term "*may*." In structured English, we can phrase the constraint as:

if the wall is-a exterior-wall there exists an attribute (of exterior-wall) labelled window_set, of the type: {window}.

If window objects may only be placed into exterior walls, the constraint becomes more difficult to implement. First, the exterior-wall class requires the window-set attribute, as mentioned above, by interpreting the *...may be placed into...* phrase as specifying a part-of assembly on wall. Second, the window object must ensure that the wall into which it is being placed is an exterior wall, and not some other component. An implementation of this window class would contain an invariant such as:

classtype(assembly_obj) = exterior_wall

where *classtype()* is a system function that returns the class identifier of the parameter object, and the parameter *assembly_obj* is the identity of the *wall* causing the *window* to be instantiated. During the design process, the designer may establish a window in the exterior wall by a command, for example

ADD wdw.create(W1) TO W1.window_set.

The following simple constraint statement,

All doors to bedrooms must have antique-brass passage-sets.

indicates:

- there is a class representing *bedroom* objects,
- the class of *bedroom* objects has a component of the class representing *door* objects,
- the class of *door* objects has a component of the class representing *passage-set* objects,
- the class of *passage-set* objects has a *style* component with the value *antique-brass*.

In the absence of other constraints, this single statement indicates the need for three class definitions, and a *style* attribute with the value *antique-brass*.

Similarly, behaviour and functionality can be captured as *methods* in class definitions. A method may produce a computed attribute, such as the floor area of a room object, or control behavioral requirements associated with using the object. For example, the instantiation method of a *beam* object may include a constraint to identify the minimum number of support posts, if they exist, or invoke a routine to instantiate and quantify the attributes of the necessary posts. The instantiation of the beam object automatically guarantees the necessary support structures through its constraint behaviour (method). The process of instantiating a post may invoke a routine to instantiate a foundational support pad located below the post. The designer would either agree to create a pad, or optionally, identify an existing foundation object that can serve as a pad. Figure 3 demonstrates the potential activities initiated by the constrained behaviour of the *beam* class. The designer instantiates a beam object as part of the current design activity. The requirement that the beam be supported by a post, temporarily suspends the definition of the beam and requires the designer to instantiate or identify a post object. Instantiation of the post object, in turn, requires the designer to instantiate or identify an existing support for the post. Once the necessary objects have been instantiated and attribute values have been defined, the designer is returned to the initial design operation and may proceed to complete the beam definition. The numbered arrows indicate the order of operations followed by the designer.

This approach to constraint satisfaction tends to tie the sequence of constraint satisfaction to object instantiation. As such, the designer will be presented with design considerations at the most appropriate time: when the problem is first encountered. By following the sequence

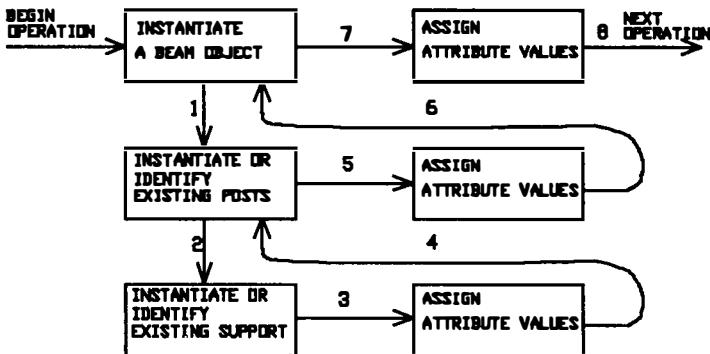


Figure 3: Constraint-guided design of a beam and related objects.

generated by the system, all related problems, discovered by the constraints implicit to the instantiated objects and their relationships with other objects, can be solved. However, a mechanism must exist to temporarily suspend a constraint violation and hold it pending until the designer can come back to it. Due to design dependencies, an object attribute may not be completely defined first. The designer may wish to instantiate and manipulate other object attributes first, or there may be mutually dependent objects with simultaneous constraint requirements. Leler (1988) refers to such instances as cyclical constraint dependencies (see Figure 4).

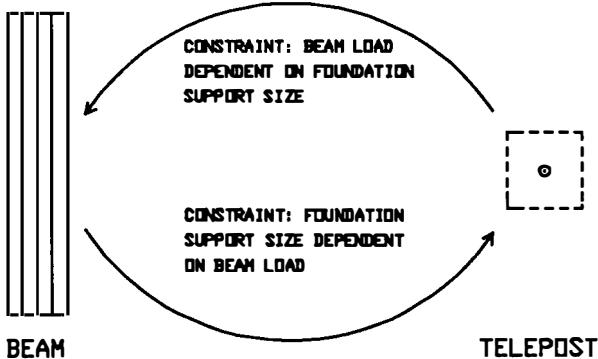


Figure 4: Cyclical constraint dependencies.

Since the final solution can be considered a set of values which simultaneously satisfy all constraints on the design, the particular order in which the designer works is irrelevant. There may be a personal choice, or a design dependency which can not be easily captured in constraint form, i.e., a floor system may depend on room configuration, sunken floor requirements, etc.

The class methods serve a dual purpose in that they not only capture the required behaviour of the object, but assist the designer in navigating the many design decisions which must be made by the designer to arrive at correct behaviour.

The final components to be obtained from the constraint specifications are the *class invariants* and *pre- and post-conditions*. These specifications are found in constraint phrases that place restrictions on attribute values or the existence of an object (i.e., a non-VOID

attribute value). The constraint *a bedroom window area must be at least 5% of room floor area*, is such a statement. Placed into the bedroom class, this constraint indicates that a component of the room class is *floor area* which may be either a stored primitive value, or a functional computation. It also states that a window component *must exist* in bedroom objects, and that the window class has an area attribute with an assigned, or computed value greater than or equal to 5% of the room floor area. During instantiation of a *bedroom* object, the object-oriented system would require that a window object be instantiated with an area not less than 5% of the current specified floor area so that the class invariant is met upon completion of the *bedroom* instantiation process. The whole process ensures that the original design constraint, as specified above, is implicitly satisfied. Since the requirement is a class invariant, any invocation of the *bedroom* methods for alteration purposes, would ensure the window remains consistent. The designer may change the window specification at will so long as the window area never goes below 5% of room floor area.

4.2 Attribute Values as Algebraic Constraint Expressions

During the second phase of design development, corresponding to execution of the object-oriented design environment, objects are instantiated and attributes are manipulated.

The design process begins by instantiating an architectural assembly object. The implicit constraints then guide the designer through the instantiation of required objects and attribute assignment as illustrated in Figure 3. It is during this process of establishing actual attribute values that the designer, in effect, narrows the set of acceptable solutions defined abstractly by the constraints.

As the designer interacts with the system, he or she is prompted for different object instantiations and attribute assignments. This would typically involve the specification of constant values (primitives), object identifiers, algebraic expressions (Gosling, 1983), and constraint expressions (Morgenstern, 1984, 1986, 1988), representing values yet to be computed (Coupal, 1985).

The evaluation sub-system can evaluate these expressions automatically as expression identifiers become available (partial evaluation) or when the designer instructs the system to process them. In either case, the intent is to reduce the expression to an attribute value, in turn, transforming the object from an abstract state to a particular actual state. The authors have experimented successfully with design through expressions in previous work (Coupal, 1982, 1985).

4.3 Example (Phase 1): Defining Classes from a Constraint

The transformation of a constraint specification to a class specification requires a formal description of the class implementation of the constraint. We define the equivalent object-oriented implementation specification S_{oo} as a set of class definitions. Straube (1989) suggests a class can be defined as a 5-tuple

$$c = (n, cd, p, MT, AM)$$

where

- | | |
|-----------|--|
| <i>n</i> | a class name, |
| <i>cd</i> | a class description which can be either basic, set or tuple, |
| <i>p</i> | a sequence of parent classes (to be inherited), |

<i>MT</i>	a consistent set of methods (to be implemented), and
<i>AM</i>	a set of applicable methods (i.e., all methods in <i>MT</i> plus all inherited methods).

A *set of invariants* are class restrictions imposed by the constraints. We will extend the above class definition by adding one more component, *I*, the set of invariants:

$$c = (n, cd, p, MT, I, AM)$$

As a constraint statement is analyzed, it is transformed into class descriptions based on this tuple. Since the constraints specify only visible, or *exported* characteristics, the implementation-specific components will remain invisible when the class is implemented.

In general, a constraint will contain references to objects, object classes, or inter-object relationships, and can be represented in short-form by the following conventions:

(x)	x is an object which exists,
A(x)	x is an object instance of the class A,
B(A,Y)	A has part B that references an object of class type Y which may be a basic, set, or tuple class,
C((D))	C is a set class; elements of the class are objects of class D, and
F(x) → E(x)	F is a specialization of E.

Each term must be a non-functional reference. That is, the term must reflect a relationship, not a value: *A(x)* represents a classification of *x* as an *A*. The distinction is purely notational but consistent with object-oriented conventions where functions are simply implementation specific methods for computing an attribute value. The constraints identify attribute requirements but do not specify whether the implementation of an attribute is to be explicit (assigned value) or functional (computed value) (Meyer, 1988).

The following examples serve to demonstrate the major notions of the design paradigm. The set of class definitions S_{∞}

$$S_{\infty} = \{c_1, c_2, \dots\}$$

will be given in the following form, where | separates alternative descriptions, and <> surround items to be replaced in an actual definition:

$c_i = ($	$n = <\text{class name}>$
	$cd = <\text{tuple structure specification}> \{ \text{set of some class type} \} $
	$<\text{primitive type}>$
	$p = <\text{list of parent classes, ex., } c_1, c_2, \dots>$
	$MT = \{ \text{set of methods, including for each method the type reference, parameters, pre- and post-conditions, and structured English or algorithmic description of the behaviour.} \}$
	$I = \{ \text{set of invariant conditions} \}$
$)$	

Example 1. A structural component is a special component that supports some other component. All structural support components in a small building must support some

other component in the building. If the structural component is a floor-beam, it must be supported by structural components located at points 1/4 of its length apart but not more than ten feet apart.

By analyzing the statement, we can make the following observations:

- 1) There is a *small-building* class represented by the general reference to *small building*.
- 2) A *small-building* has a set of components, each of which is a *component*.
- 3) There is a specialized class of component called *structural-component*.
- 4) The attribute reference of *structural-component* that identifies the *component* it supports, can not be *VOID*.
- 5) There is a specialized class of structural component called *floor-beam*.
- 6) Floor-beam has an attribute *supported-by* consisting of a set of structural components with constrained locations along the beam.
- 7) Floor-beam has an attribute *length*.
- 8) Structural components have an attribute *location* identifying where in the small building the component can be found.

Observation (8) does not imply that only structural components can have a *location* attribute but simply that structural components do have one. It is likely every design component in a building will require a location as defined by other constraints on the system. For this example, we can assume the location attribute is defined and inherited from a parent class of component and does not appear on the extensions of attributes and behaviour listed in the class definitions below. It will become important when the attribute values are specified in phase 2. We can summarize these requirements as follows:

set of *component* is-part-of *small-building*,
structural-component is-a *component*,
supports of type *component* is-part-of *structural-component*,
structural-component.supports ≠ *VOID*, (invariant)
beam is-a *structural-component*,
supported-by of type set of *structural-component* is-part-of *beam*,
length of type *numeric* is-part-of *beam*, (a primitive attribute)
the distance between consecutive *supported-by* members must be less than or equal to ten feet.

and write them in short-form (capital letters represent class names, small letters represent objects and attributes mentioned in the constraint statement):

```
SMALL_BUILDING(x),
component_set(SMALL_BUILDING,(COMPONENT)),
STRUCTURAL_COMPONENT(y) → COMPONENT(y),
supports(STRUCTURAL_COMPONENT, z) ∧ z ≠ VOID,
FLOORBEAM(a) → STRUCTURAL_COMPONENT(a),
supported_by(FLOORBEAM, {STRUCTURAL_COMPONENT}),
length(FLOORBEAM,length_value), NUMERIC(length_value),
for all yi ∈ supported_by, 1 ≤ i ≤ lsupported_by|
    location(yi-1, loci-1), location(yi, loci),
    |loci - loci-1| ≤ 120.0
```

When a structural component is created, its *supports* reference will be initially VOID. This would violate the invariant *structural.supports ≠ VOID*, if left ignored. Creation of a structural support component requires that we define a component to satisfy the invariant when this object is instantiated. The design environment must prompt the designer for a component reference to satisfy the constraint by making the *structural.supports* non-VOID. The creation of an object is a system requirement and the implicit method *create* will require an extension to prompt the designer and obtain the correct reference. This is an implementation detail, however, since the constraint specification does not cover how the non-VOID reference is obtained, simply that it can not be VOID. The decision of what object is to be supported and to actually create it must be made by the designer.

The object-oriented class specification, $S_{\infty} = \{c1, c2, c3, c4\}$ can be obtained as the set of definitions shown in Figure 5. In this set of descriptions we do not show the *applicable methods* since they include all inherited methods and do not need to be listed.

```

 $S_{\infty} = \{c1, c2, c3, c4\} :$ 

c1 = (n = component
      cd =  $\emptyset$ 
      p = system_design_object      -- defined by system
      MT =  $\emptyset$ 
      I =  $\emptyset$ )

c2 = (n = structural
      cd = <supports:c1>
      p = c1
      MT = {create: root --> c1
             [Establish an object instance of component class supported by this
              object and set the supports attribute equal to identity of component
              object. NOTE: This definition forms an extension of the system-defined
              create method that establishes an initial non-VOID reference to satisfy
              the class invariant.]}
      I = {supports <> VOID})

c3 = (n = smallbuilding
      cd = <component_set : {component}>           -- the attribute component set consists of a set
                                                       of components
      p =  $\emptyset$ 
      MT =  $\emptyset$ 
      I =  $\emptyset$ )

c4 = (n = floorbeam
      cd = <supported_by: {structural}; length: real> -- real is a basic system data type
      p = structural
      MT =  $\emptyset$ 
      I = {for all yi ∈ supported_by, 1 ≤ i ≤ |supported_by|
            location(yi-1, loci-1), location(yi, loci), |loci - loci-1| ≤ 120.0}

```

Figure 5: Class requirement specification tuples for an object-oriented implementation of the constraint.

Finally, our original constraint can be implemented as definitions derived from the S_{∞} set in Figure 5. Figure 6 lists these descriptions presented in a form similar to the object-oriented language Eiffel (Meyer, 1989), a language that contains many of the features necessary to implement constraints in an object-oriented environment.

4.4 Example (Phase 2): Specification of Attributes through Constraint Expressions

The following example demonstrates how the designer might set attribute values during system execution. It follows from the example above.

Example: Phase 2. *Instantiation of a beam requires the instantiation of a set of structural objects for beam support, placed as required below the beam.*

```

class component
export Ø
inherit Ø
feature Ø
invariant Ø
end -- class component

class structural
export supports
inherit component
feature
    supports : component;
    create is
        do      "If an object of type component, to be supported, does not exist
               then instantiate the component object.
               Set the supports attribute value to the identity of the object
               (instantiated if necessary) supported by this object"
    end
invariant
    supports <> VOID;
end -- class structural

class smallbuilding
export component_set
inherit Ø
feature
    component_set : universal_set(component); -- a utility class implementation of set
invariant Ø
end -- class smallbuilding

class floorbeam
export supported_by, length
inherit structural
feature
    supported_by : universal_set(structural_component);
    length : real;
    supports_ok : boolean is
        do      "supports_ok := true;
               for all  $y_i \in supported\_by$ ,  $1 < i < |supported\_by|$ 
               if  $location(y_{i-1}, ly_{i-1}), location(y_i, ly_i), |ly_i - ly_{i-1}| > 120.0$ 
               then supports_ok := false;"
    end
invariant supports_ok
end -- class beam

```

Figure 6: Class definitions in implementation form.

The specifications shown in Figure 7 are representative of the type of constraint specifications the designer might make. During the instantiation of the *Beam1* object, the system will require the instantiation, or identification, of the support objects *P1*, *P2*, *P3*, and *P4*. The *location* attribute of each structural support component can be specified through an algebraic constraint, for example,

$$P1.location + (Beam1.length / 4)$$

Such a specification is considered to satisfy the positional requirements of the beam on its support posts, until such time as it is evaluated. The actual dimensional value of the *location* attribute will be used in the final determination of the validity of the design.

5. OTHER ASPECTS OF THE PARADIGM

The main notions behind the model are the use of constraints to drive the development of object-oriented class definitions, the use of constraints to establish attribute values for a particular design, and the automatic satisfaction, by the design environment, of design constraints during design development. The paradigm is currently in very early stages of development, and many aspects remain to be studied and evaluated. In addition, problem

```

SPECIFICATIONS FOR:

Object: Beam1 [Class: FLOOR_BEAM]
Attributes:
  supported_by: {P1,P2,P3,P4}
  location: (x,y,z) [inherited attribute]
  length: 30 * 12 [length specified in inches]

Object: P1 [Class: STRUCTURAL_COMPONENT]
Attributes:
  location: Beam1.location

Object: P2 [Class: STRUCTURAL_COMPONENT]
Attributes:
  location: (P1.location(x) + (Beam1.length /4),P1.location(y,z))

Object: P3 [Class: STRUCTURAL_COMPONENT]
Attributes:
  location: (P2.location(x) + (Beam1.length /4),P1.location(y,z))

Object: P4 [Class: STRUCTURAL_COMPONENT]
Attributes:
  location: (Beam1.location(x) + Beam1.length,Beam1.location(y,z))

NOTE: Reference to (x,y,z) is simply to indicate possible algebraic references to various parts
of the object's structure. It does not imply any particular representation formalism.

```

Figure 7: Specification of object attributes using constraint formula.

areas exist which may be difficult to incorporate into the design paradigm. Some of these aspects are briefly presented.

5.1 Meta-Constraints and Design Calculus

In addition to the requirements placed on the system classes by architectural design constraints, there are the requirements placed on the classes by the system itself. These constraints are *meta-constraints*, not part of the architectural design problem, but related to it.

For example, if the designer increases the length of the beam used in the example, the system should respond by requiring additional supports. If the designer shortens the beam length, or changes a support definition (from a post to an existing bearing wall) the necessary changes should be made to the current set of design objects. The latter change may necessitate the removal of a design object thereby necessitating the modification or creation of other related objects.

Such behaviour must be encapsulated in the class definitions, but comes from the requirements and design philosophy of the environment rather than the design problem. The requirements of the environment, or *design calculus requirements*, are the characteristics which must be present in the class definitions to implement such designer commands as *create* an object, *change* an object, *move* an object, *rotate* an object, and other similar design actions. Research into these aspects of the environment has begun and will be the subject of a future paper.

5.2 Class Definition, Identification, and Storage

As classes become implemented in the system, there is the potential for redundant definitions because an appropriate class is not recognised, or a candidate class not properly extended. The generation of classes from constraints may provide a mechanism for *browsing* through classes to find a particular class for some application. Since the need for a class is due to a constraint, this constraint can be matched against existing constraints. Any exact or close

matches can be used to identify appropriate classes. All classes which contain some component due to a constraint can be listed along with the constraint providing a self-documenting mechanism.

Each component of a class definition can be tagged with the identifier of the constraint to which it relates. Thus, any visible class component can be traced back to the constraint upon which it depends. This is useful for updates on existing classes. We are examining this notion as part of our current research.

5.3 Contributions of the Paradigm

The model appears promising in advancing our understanding of the design process and the relationships between objects in the real world, the constraints on those objects, and the representation of those objects in a computer system. Some of the contributions of the model are:

1) *Constraint design requirements remain intact in object-oriented form as class definitions.* The paradigm flows seamlessly, from constraint specification, through class definition, to attribute manipulation. From a design point of view, this is important since the original specifications remain intact in object-oriented form.

2) *The class requirements are minimal in that only those requirements specified in the constraints become class definition components.* A class definition can be traced back to the constraint component requiring it. This is important in helping to reduce the number of classes in the environment. Since classes are derived from constraints, no additional visible class characteristics are added. Implementation details are hidden and not relevant to the designer.

3) *The generation of classes from constraints, and the specification of attribute values with constraint expressions, makes it easier to adjust an existing design through selective modification of its attribute expressions, while maintaining the correctness of the solution.* This is maintained through the constraints, as design knowledge, implicit in the classes themselves. The use of constraint expressions for attribute values allows the designer more freedom to modify specific characteristics of certain design objects, than does traditional CADD component utilization, by allowing the system to ensure constraint satisfaction in the role of design assistant.

4) *The model exploits important aspects of the object-oriented model of design.* The paradigm allows the designer greater creativity to solve problems by allowing closed classes to be extended through *inheritance*. Since the classes are derived from a constraint system, additional constraints can be incorporated if they do not conflict with existing constraints, thereby enhancing the class library. The constraint requirements can be implemented in object-oriented programming languages used to build the design environment.

5) *The formalization of constraints prior to transformation into classes focuses attention on the different aspects of the constraints.* Certain constraint characteristics must be maintained across all designs (e.g., building code, engineering principles) while others are variable conditions meaningful only in the context of a particular design. By focusing attention onto the constraints, the system designer must evaluate each constraint and determine its nature and importance to the system. Irrelevant constraints, conflicting constraints, and redundant constraints, can be better identified prior to system construction.

6) *Although not yet proven, it appears the transformation of constraints to classes is complete, i.e., the set of dependent class components capture all the pertinent information contained in the constraints, since the class definitions are a direct consequence of the constraints.* This is an important feature in determining if the constraint requirements are

sufficient for producing a design, and ensuring that the class definitions are complete and can fully represent the design. In addition, no class characteristics will appear that can not be traced back to a constraint requirement.

5.4 Problems and Limitations

Various problems remain to be studied in the context of the model.

1) Many constraints may not be representable in a formal framework suitable for class representation. For example, a subjective constraint, such as *the exterior finish must be durable*, is not representable in class form since we have no definition of what constitutes *durable*. If the exterior finish is to be selected from a set of finishes, that are considered *durable*, then the constraint can be implemented.

2) The significant number of constraints on even a simple structure require careful management of both the formal constraint statements, and the classes derived from them. There is significant potential for constraint conflict as new constraints are added to the system. The system may require complete regeneration after any constraint change. This would suggest the need for a constraint-to-class compiler, similar to a programming language compiler, which can automatically generate the class definitions from the constraint statements. This regeneration must not interfere with existing hidden implementation details, yet provide a correct class implementation when finished. Little research has been undertaken in this area to date.

3) A special constraint language may need to be developed for the model. The model has been presented using various forms of constraint specification, from structured English to predicate calculus. If a constraint compiler is to be developed, a constraint language with appropriate characteristics, may be required. In addition, during design generation, the designer requires appropriate manipulation operations, using a command language or graphical interface. This concern has been considered but not formalized in the model as yet.

4) To provide the functions of the design calculus mentioned above, system support functions may be quite extensive. Such features will require formalized definition, to identify the exact behaviour of the operation during design manipulation. The characteristics of the environment have been studied in some detail but are not yet formalized in the model.

Future research into this model will involve the development of a prototype design environment to test the efficiency of capturing constraints and, through transformation, the compilation of class definitions ready for implementation. Through prototype development and analysis, the model may be expanded and enhanced to accommodate current restrictions and problem areas.

REFERENCES

- Adiba, M. and Nguyen, G. (1984). Information processing for CAD/VLSI on a generalized data management system, *Proceedings of the 10th International Conference on Very Large Databases*, pp. 371–374.
- Adiba, M. and Nguyen, G. (1986). Handling constraints and meta-data on a generalized data management system, in L. Kerschberg (ed.), *Expert Database Systems, Proceedings of the 1st International Workshop*, Benjaminn-Cummings, pp. 487–504.

- Akin, O. and Dave, B. (1986). Formal representation of design knowledge and process, *CIB.86: Advancing Building Technology*, Proceedings of the International Council for Building Research, Studies and Documentation, pp.251–259.
- Balachandran, M. and Gero, J. S. (1987). A knowledge-based approach to mathematical design modelling and optimization, *Engineering Optimization* 12: 91–115.
- Barbuceanu, M. (1984). An object-centred framework for expert systems in computer-aided design, in J. S. Gero (ed.), *IFIP WG 5.2 Working Conference on Knowledge Engineering in CAD*, North-Holland, Amsterdam, pp.223–253.
- Bijl, A. (1985). A CAD logic modelling environment, *Architectural Science Review* 28(14): 104–114.
- Borning, A., Duisberg, R. Freeman-Benson, B., Kramer, A. and Woolf, M. (1987). Constraint hierarchies, *OOPSLA'87 Proceedings*, October 4–8, ACM, pp.48–60.
- Cmelik, R. F. and Gehani, N. H. (1988). Dimensional Analysis with C++, *IEEE Software* 5(3): 21–27.
- Cooper, C. N. (1987). CRANES—a rule-based assistant with graphics for construction planning engineers, *Third International Conference on Civil and Structural Engineering Computing*, London, pp.1–14.
- Coupal, C. (1982). A low cost CAD system for manufactured housing, *Proceedings Graphics Interface '82*, pp.161–168.
- Coupal, C. (1985). *An Approach to Interactive CAD for Building Construction*, Masters Thesis, Department of Computational Science, University of Saskatchewan.
- Fawcett, W. H. (1986). Design knowledge in architectural CAD, *CAD* 18(2): 83–90.
- Flemming, U. (1986). A generative expert system for the design of building layouts, Version 1, *Progress Report*, Department of Architecture, Design Research Center and Center for Art and Technology, Carnegie Mellon University, Pittsburgh, PA.
- Gosling, J. (1983). *Algebraic Constraints*, PhD Thesis, Carnegie Mellon University, *Technical Report No. CMU-CS-83-132*.
- Horstmann, P. and Stabler, E. (1984). Computer aided design (CAD) using logic programming, *21st Design Automation Conference*, IEEE, pp. 144–151.
- Jakiela, M. J. and Papalambros, P. Y. (1989). Design and implementation of a prototype ‘intelligent’ CAD system, *Transactions of the ASME Journal of Mechanisms, Transmissions and Automation in Design* 111(June): 252–258.
- Kalay, Y. E. (1985). Refining the role of computers in architecture: from drafting/modelling tools to knowledge-based design assistants, *Computer Aided Design* 17(7): 319–328.
- Ketabchi, M. A. and Berzins, V. (1988). Mathematical model of composite objects and its application for organizing engineering databases, *IEEE Transactions on Software Engineering* 14(1): 71–84.
- Kemper, A., Lockemann, P. and Wallrath, M. (1987). An object-oriented database system for engineering applications, *Proceedings of ACM-SIGMOD'87*, San Francisco, pp.299–310.
- Kim, W., Banerjee, J., Chou, H-T., Garza, J. F. and Woelk, D. (1987). Composite object support in an object-oriented database system, *OOPSLA'87 Proceedings*, pp.118–125.
- Leler, W. (1988). *Constraint Programming Languages: Their Specification and Generation*, Addison-Wesley, New York.
- Logitech (1986). *LOGICADD User's Manual*, Generic Cadd Version 2.0, Logitech Inc., Redwood City, California.

- Lorie, R. and Plouffe, W. (1983). Complex objects and their use in design transactions, *Proceedings Databases for Engineering Applications*, Database Week, pp. 115–121.
- Lorie, R., Kim, W., McNabb, D., Plouffe, W. and Meier, A. (1985). Supporting complex objects in a relational system for engineering databases, in Won Kim, David Reiner and Don Batory (eds), *Query Processing in Database Systems*, Springer-Verlag, NY, pp. 145–155.
- Meyer, B. (1988). *Object-Oriented Software Construction*, Prentice-Hall International Series in Computer Science.
- Morgenstern, M. (1984). Constraint equations: declarative expression of constraints with automatic enforcement, *Proceedings of the 10th International Conference on VLDB*, Singapore, pp. 291–300.
- Morgenstern, M. (1986). The role of constraints in databases, expert systems and knowledge representation, in L. Kerschberg, (ed.), *Expert Database Systems*, Benjamin-Cummings, pp. 351–368.
- Morgenstern, M. (1988). Constraint-based systems: knowledge about data, in L. Kerschberg (ed.), *Expert Database Systems*, Proceedings of the 2nd International Conference, pp. 87–88.
- Radford, A. D. and Gero, J. S. (1985). Towards generative expert systems for architectural detailing, *Computer-Aided Design* 17(9): 428–435.
- Rehak, D., Howard, H. and Sriram, D. (1985). Architecture of an integrated knowledge based environment for structural engineering applications, in J. S. Gero (ed.), *IFIP WG 5.2 Working Conference on Knowledge Engineering in CAD*, North-Holland, pp. 89–117.
- Rosenman, M. A., Gero, J. S., Hutchinson, P. J. and Oxman, R. (1986). Expert systems applications in computer-aided design, *Computer-Aided Design* 18(10): 546–551.
- Smith, A. D. (1990). Software tool enhances AutoCAD, *Architectural Engineering Construction (Canada)* 2(1): 40–41.
- Stewart, W. D. (1990). CADD simplifies repetitive tasks in building construction projects, *Architectural Engineering Construction (Canada)* 2(1): 26–30.
- Straube, D. D. and Ozsu, M. T. (1989). Queries in object-oriented databases: an equivalent calculus and algebra, *Technical Report TR-89-16*, Department of Computer Science, University of Alberta, Edmonton, Canada.
- Takagaki, K. (1990). *A Formalism for Object-Based Information Systems Development*, PhD Thesis, University of British Columbia, Canada.

ArchObjects: design codes as constraints in an object-oriented KBMS

B. K. MacKellar[†] and F. Ozel[‡]

[†]Department of Computer and Information Science
New Jersey Institute of Technology
Newark NJ 07102 USA

[‡]School of Architecture
New Jersey Institute of Technology
Newark NJ 07102 USA

Abstract. This paper describes an intelligent architectural design assistant that integrates an object-oriented model of design objects and design codes modeled as constraints. The features of ArchObjects include an object-oriented knowledge representation that models geometric knowledge, tight integration of design constraints within the KBMS, and design evaluation at any level of granularity. The knowledge representation contains object types, instances, and associated methods organized in a specialization network. The initial prototype implements the portions of fire code concerned with means of egress and fire protection contained in the *Life Safety Code Handbook* published by the National Fire Protection Association(Lathrop, 1988). Fire code provisions are modeled as semantic constraints, permitting integration of rules into the data model. Integrating design requirements into the model means that the requirements can be enforced across design sessions and groups of designers. The system uses a trigger-based approach to run-time constraint management which allows specification of actions to be taken upon failure.

INTRODUCTION

There is much interest among both artificial intelligence and database researchers in systems that automate the design task. Researchers in AI have investigated the fundamental nature of the design task(Williams, 1990) and also have produced expert systems that aid the designer in various ways(Sriram 1986, Rosenman et al 1986). Research in design databases has been geared towards the development of data models that accurately model the design object(Cammarata and Melkanoff 1986, Wilkes, Klahold, and

Schlageter, 1989). There is a need, however, to integrate both views : that of the design task and of the design object. That is one of the primary goals of ArchObjects.

ArchObjects functions as an intelligent assistant to the designer. Examples of such systems may be found in other areas such as software engineering(Rich and Shrobe 1978). Such systems ease the human designer's task by performing the more routine tasks, for example, checking the design for compliance with various design codes, or retrieving past successful designs. Therefore, our system is not concerned with complete automation of the design task.

ArchObjects is based on an object-oriented model of the design object that permits geometric reasoning, with integration of design requirements into the data model. The data model supports geometric queries, meaning that intelligent applications that require geometric knowledge may be built. This has been termed *graphical deep knowledge* by Geller(Geller and Shapiro 1987). Integrating design requirements into the model means that the requirements can be enforced across design sessions and groups of designers. In building our initial prototype, we are focusing on building design and fire code provisions as a set of design requirements that must be enforced in this manner. The approach is easily extendible to other types of design codes, however, and the object-oriented data model permits other types of design applications to be easily built on top of the system. For example, retrieval of similar design objects is facilitated by such a representation(MacKellar and Maryanski 1989).

BACKGROUND

Building codes are laws that set forth minimum requirements for design and construction of buildings and structures. The requirements contained in building codes are generally based on the properties of materials, the hazards presented by various occupancies and the height and area limitations imposed to prevent possible fire conflagrations.

There are several widely used model building codes, such as the BOCA building code and the Uniform Building code. These constitute the main body of knowledge transferred into a building design process. Since rules and definitions in these codes can vary from one to another, any system developed for code compliance needs to be able to address this variation. Furthermore, the codes change on a regular basis. Due to the large number of rules, it is difficult to automate verification of design codes.

Architectural designers usually must evaluate an incomplete design for code compliance and later go through several evaluation phases as the building design gets more and more complete and detailed. Therefore, an automated design assistant that can evaluate an incomplete design for compliance is very useful during this process. The current system is intended to address these problems. We are initially concentrating on portions of the fire code contained in the Life Safety Handbook published by the National Fire Protection Association(Lathrop 1988). The sections that are being implemented are those that deal with *means of egress* and *fire protection*. These sections deal with such topics as exits, travel distances, fire and smoke barriers, and compartmentation. These sections were selected since they form the core of the fire code provisions contained in the handbook and are heavily referenced in subsequent chapters. The features of our architectural

design assistant include

- a tightly integrated KBMS
- an object-oriented knowledge representation that models geometric knowledge
- design evaluation at any level of granularity, from individual elements such as doorways to the entire building.
- actions associated with each design constraint to be taken upon failure

Tight integration in this case means that the constraints are an integral part of the data model. Constraints express real world semantics of the objects being represented. This approach can be used in any application area in which constraints can be identified.

Expert System Approaches

Researchers have investigated the use of codes and standards as the interpretive knowledge base in architectural expert systems. Gero and Rosenman describe a system in which parts of the Australian Model Uniform Building Code (1982) are translated into a set of if-then rules as the basis of a simple deductive expert system(Coyne et al. 1990). Among other such systems is a rule-based system developed by Kim(1986). This system accesses the CAEADS CAD system data base, which provides parallel data structures for representing building parts such as walls, windows, doors, floors and staircases. A third software system that addresses the issue of code compliance is ARCH:Firesafety(Ozel 1985). The system addresses the procedural basis of fire code compliance. Travel distances in a building, the distance between smoke and fire walls, areas of fire zones, smoke areas, the capacity of exits and egress routes are automatically calculated from the graphical representation and checked. ARCH:Firesafety also uses the CAEADS data base, but does not model building codes as rules.

A system that is based on a design object model is FIRECODE(Buis et al.,1987). The system helps the designer check whether a building meets the requirements of a draft standard for fire safety. The system uses a frame-based classification hierarchy to represent objects and inference rules to represent the fire code provisions. Most of the qualitative information such as the floor area and the length of the longest travel path in the building are asked from the user as an input, rather than being calculated from the graphic data base of the building.

In general, the expert systems developed so far perform forward or backward chaining on symbolic representations. FIRECODE uses a classification hierarchy but does not infer facts from a geometric representation. ARCH:Firesafey uses a geometric representation, but does not encode its fire code provisions as rules.

Semantic integrity constraints

An approach, which produces a tightly coupled intelligent design database, is that of specifying fire code provisions as semantic integrity constraints. In database theory, an integrity constraint expressed as a logical formula (Gallaire, Minker, and Nicolas

1984), governs the possible values that can be stored in a database. The semantics of a database are described by the database's conceptual schema, which provides a typing scheme for classifying the entities of the database and a set of rules which hold in the application area. These rules are the consistency constraints; they are an integral part of the conceptual schema and describe some aspect of the real world meaning of objects in the database(Eick, Kumar, and Liu 1989). An example is "All employee salaries must be less than their supervisor's salaries". Constraints are checked only when the database is updated. In order to prevent exhaustive testing of all constraints for each update, various methods have been proposed (Eick, Kumar, and Liu 1989) to identify the constraints that might be violated by a given update.

According to Nicolas (1982), a database state is an interpretation of the set of integrity constraints, and valid database states are models of the constraints. A model is an interpretation which satisfies all of a set of formulas. When the DB is modified, a new interpretation is generated which must be checked to verify if it is still a model of the constraints. Most approaches to constraint verification analyze the updates before they are performed, allowing the system to reject updates that would result in an invalid state. The major difference between integrity constraints and deductive rules is that application of a deductive rule produces new knowledge. In the rule-based paradigm which is at the heart of many expert systems, actual data changes cause rules to fire which cause more data changes and further firing of rules, until a goal state is reached. In a constraint verification system, however, potential updates cause constraint rules to be checked in order to prevent an inconsistent database state.

Several researchers have proposed the use of semantic integrity constraints in design databases to specify restrictions on possible designs (Cammarata and Melkanoff, 1986, Dittrich et al,1986, Dayal and Smith, 1986). This is particularly suitable for systems that function as intelligent advisors. In this paradigm, the intelligent assistant does not create a design, but instead, advises the human assistant by pointing out design errors. Adding design constraints to a constraint manager in a KBMS has the advantage of integrating the constraints into the overall conceptual schema. This permits close integration between the designer's task and the intelligent assistant's task. In current practice, design constraints, if automated at all, are scattered among various application programs, such as analysis programs and expert systems. Inconsistencies among the applications may occur as constraints change over time. In a situation involving several design groups, each group may have its own set of constraints that must not be violated by other groups. Managing the constraints within the KBMS prevents such inconsistencies and ensures that the constraints will be enforced for all design sessions.

There are several difficulties to be overcome, however, if design constraints are to become part of a design data model. Design objects and the constraints on them are far more complex and numerous than traditional database objects and constraints. Also, during the design process, the system may have to tolerate inconsistency within the objects for long periods of time. Furthermore, there is a distinction between global consistency, which may not occur until the very end of the process, and local consistency, which is achieved in stages throughout the design process (Dittrich, Kotz, and Mulle 1986). Therefore, violations of design constraints must not force entire transactions to be undone. This would cause the loss of entire design sessions, and is clearly not acceptable

to designers. Instead, constraints should have different types of actions associated with them to be taken upon violation. For many design constraints, the preferable action is merely to warn the designer. Even this facility is not flexible enough, however. Many designers would prefer not to work in an environment where warnings are constantly being generated. Therefore, checks for violations of many kinds of design constraints should be triggered by the user.

In ArchObjects, constraints are used to model a notion of correctness with regard to both the internal data model and the context-dependent design specifications. To differentiate between these purposes, constraints are divided into three types :

- Specification constraints : These are part of the design requirements.
- Domain constraints : These are either legally required or generally accepted constraints that apply to all designs in a given domain.
- Data model constraints : Constraints generated by the object type specifications. These are not explicitly specified, but are implicit in the data model and type specifications.

The specification constraints apply only to a particular design case, and consists of the required parameter ranges and desired behaviors of the design object. An example might be "The building must fit on a 50 by 50 foot lot". Domain constraints, on the other hand, apply to all design objects. These constraints represent domain-specific knowledge. There are two basic types: commonsense constraints and code constraints. Commonsense constraints represent knowledge about commonsense properties of the physical world in which the design object must operate. An example might be "A door cannot be wider than its corresponding doorway". Code constraints are more specialized, and may represent either legally required or generally accepted design knowledge, such as that found in design handbooks(Allen and Iano, 1989) or model codes(Lathrop 1988). These govern aspects of a building design from the height to the types of permitted interior finishes. The current prototype focuses on sections of fire code as an example of design codes.

Data model constraints result from the representation of the domain types in terms of the data model. There are four types of data model constraints :

- Definitional constraints : All instances of a type T must have component instances that correspond to the component object types of T and attributes that correspond to the attributes of T.
- Range constraints : An object type T may have a value range associated with one or more of its attributes. All instances of type T then must have values for that attribute within the specified range.
- Derivational constraints : An attribute's value may depend on the values of other attributes of the object type.
- View definition constraints : Constraints that govern the generation of views from objects.

This classification is based on that used by (Peckham 1990). Like the domain constraints, these apply to all object instances. The first three types are specified as part of the process of creating object type definitions. The last type is specified by defining derived views on the object types. These govern, for instance, the mapping of the base representation of a building to 2-D floor plans, or to its structural system representation.

The constraint types may be viewed as forming a hierarchy, from most general to most case-specific, applying only to a particular design. This is true, however, only for the more case-specific constraints. Therefore, in the transaction model of ArchObjects, all of the data model constraints must be completely satisfied in order for a design session to complete normally. This is true as well for the commonsense constraints. This means that a designer will not be able to commit a design session if, for instance, a door is wider than its corresponding doorway, or a staircase runs horizontally rather than vertically. Such constraints are termed invariant constraints. One of the characteristics of the design process, however, is that design-specific constraints may be incrementally satisfied. Therefore, code constraints and specification constraints need not be completely satisfied until the very end of the design process. This is controlled by the division of the design process into stages, which are represented by transactions after which a certain subset of the constraints are guaranteed to be satisfied. At a given point in time, the set of constraints that are satisfied by a particular design object O is termed O's satisfied constraint set. A further description of ArchObject's transaction model may be found in (MacKellar 1991).

PROJECT OVERVIEW

ArchObjects consists of the following components:

- Graphical interface
- Knowledge base
- Constraint base
- DBMS

shown in Figure 1. The knowledge base contains the object types, instances, and associated methods organized in a specialization network. The constraint base contains the rules that specify correctness conditions for buildings. We are currently using Postgres(Stonebraker, Rowe, and Hirohama 1990) to provide persistent storage, although in the future, an object-oriented DBMS such as ONTOS(Ontologic 1990) may be used. Taken together, the knowledge representation, constraints and DBMS form a knowledge base management system (KBMS) that is based upon a object-oriented data model, a reasoning component and persistent storage. The interface is based on AutoCAD(Autodesk 1990), a CAD package that is widely used among architects. It was chosen in order to provide a familiar interface for architects, and because it has powerful graphical capabilities.

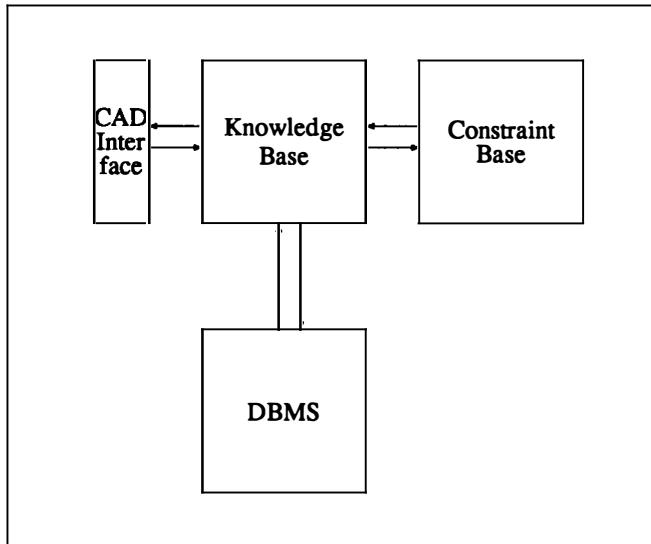


Figure 1: System architecture

Interaction with designer

A *design session* is a (usually) long transaction during which a version of the design object is retrieved from the DBMS and manipulated in various ways. During a design session, an object exists as an instance in the object type hierarchy, which is described in more detail in Section . This is a semantic representation of the object, rather than an unstructured graphical file. A set of methods that translate between the internal, semantic representation of an object and its graphical representation are associated with the object types. The graphical representation is necessary both for geometric reasoning and for the user interface. The interface allows the designer to interact with the system by means of the graphical representation or the internal representation. The first method is the most important, since practicing architects are accustomed to working with drawings, graphical representations and CAD packages. In this mode of interaction, the designer may select iconic representations of architectural elements from a menu and position them on the screen. The icons are standard graphical representations and are associated with object types in the knowledge representation. Methods that translate between the graphical representation and the internal representation are associated with the object types. Therefore, selection of a component causes invocation of the appropriate translation method, meaning that changes made to the graphical representation are translated immediately into changes in the internal representation. Updates to persistent storage only occur at the end of the session, in order to permit verification with respect to the constraints of the changes made during the session. For example, if the designer is working with a door design, its internal, symbolic representation would consist of `door has_instance d1`, where `d1` refers to the particular door design, and its graphical representation would

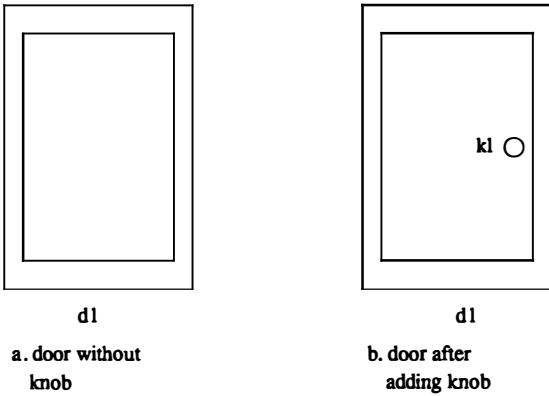


Figure 2: Graphical representation of a door object

appear as in Figure 2a.

If the designer adds a graphical representation of a doorknob, as in Figure 2b, the system generates this series of update actions:

```
insert (doorknob has_instance k1)
insert (d1 has_part k1)
```

which are used both for later constraint checking and to update the internal representation to

```
door has_instance d1
doorknob has_instance k1
d1 has_part k1
```

Currently, the designer must label graphical objects with their type or select from a standard menu of objects; in the future, better type inference capabilities are planned.

Checking the design object for compliance with fire code can happen at two points. Verification automatically occurs when the design session is about to be closed and the database updated with the new version of the design object. The designer may also request verification at any point in the session, and may request verification of any portion or component of the design object. As the designer works, a record of update actions is maintained. At verification time, these records are used to trigger the proper fire code rules based on these updates. Constraint triggers are described in greater detail in Section . Upon failure of a rule, a warning is generated for the user. This warning informs the designer of the change that generated the violation and may include some further explanatory material. Satisfied constraints are added to the object's satisfied constraint set; this set is used by the transaction and version control managers(MacKellar 1991).

When checking an unfinished design, many of the rules will be only partially instantiated because of missing components in the design object. There are several strategies for handling this :

- Only evaluate rules that can be completely instantiated
- Use the uninstantiated portions of the rules to generate messages to the designer on additionally required components and properties.
- Use prototype components to fill in the missing information.

As an example of the second strategy, consider a rule that states “The floor levels on either side of a door in the means of egress must have the same level”. If a door object exists, and a floor object exists on only one side of the door in the design, the designer may be warned that the floor object on the other side of the door must have a value for its *level* property that equals the value for the existing floor object. Currently, a strategy for dealing with partially instantiated rules has not been chosen. At the end of a design session, the design is stored as a version in the DBMS. Any uncorrected constraint violations are stored with the version, allowing designers to keep track of problems.

KNOWLEDGE REPRESENTATION

Representing Architectural Knowledge

The role of the knowledge base is to create a semantic representation of a building. Traditional CAD systems such as AutoCAD(Autodesk 1990) store all design objects in the same way, in terms of geometric components such as lines, points and surfaces. Since no higher-level semantics is involved, all design objects from a single floor to a car engine component are stored in exactly the same way, using the same primitives. In other words, object representation is solely in terms of the drawing of the object, rather than the object. This precludes applications that require knowledge about the actual object.

There are many difficulties in designing a semantic representation for design objects such as buildings. The first difficulty is that a building is very large and complex, requiring many component objects and many relationships between the components. Secondly, in order to fully represent a complex design object, several types of data must be stored. A building representation must include graphical information, numerical information, symbolic information such as relationships, and procedural knowledge. Another problem is that representations of two objects of the same type may vary quite a bit from each other, even though the objects are actually quite similar. This means that representation schemes that depend heavily on standardized sets of attributes for objects of the same type will not be suitable for representing this type of data. For example, an object type *room* with slots for only 4 walls would preclude a *room* instance with 5 walls.

These considerations mean that many traditional knowledge representation methods cannot be used. Particularly, the relational model is inappropriate for this type of object. The relational model is based on the idea of large amounts of data which is all represented in *exactly* the same way. Even AI knowledge representation methods such as frames, have difficulty with design objects, since in a classic frame system, the slots are predetermined. Furthermore, AI knowledge representation methods have difficulty with numerous, large objects. On the other hand, AI knowledge representation methods are closer in many respects to what is needed.

A fundamental characteristic of design objects is that they are hierarchical in nature. For example, a single floor may be composed of a set of rooms. Each room has a set of walls as components, and at least one entrance. An entrance may include a door assembly, which is composed of a frame and a door. The many leveled hierarchy seen in this example is quite typical. Therefore, a fundamental relationship that must be modeled is that of the *has-part* link. Another requirement in this domain is that geometric relationships must be modeled. For instance, in a building, the relationship of spaces to their boundaries is very important. Other relations, such as connectivity and adjacency are also needed.

Another important requirement is the ability to view an object in multiple ways. For example, a designer may require a standard floor plan, or a representation of a design in terms of elevations. In a similar vein, it must be possible to slice out certain parts of an object based on some functionality. For example, it should be possible to isolate the portion of a floor plan that serves as an escape route or a set of fire barriers. Therefore, functional views of the object must be stored or generated.

Object type hierarchy

The knowledge representation described here is based largely on previous work on WharfRat(MacKellar 1989, MacKellar and Maryanski 1989). Each object type has a name, a definition, and associated methods and constraints. The definition of an object type is made up of relationships between the object type and components. Atomic object types contain no links or subobjects. More formally,

Definition 0.1 *An object type O can be represented by*

$$O = (Name, Def, M, C)$$

where

- *Name is the name of the object type*
- *Def is a graph representing the definition of the object type*
- *M is a set of methods on the object type*
- *C is a set of constraints defined on the object type*

The system defined link types are `<has_spec>`, `<has_attr>`, `<has_member>`, `<has_part>`, `<has_role>` and `<has_instance>`.

The `<has_attr>` link relates an object type to a property associated with objects of that type. For example,

```
room has_attr max_height,  
      has_attr max_length,  
      has_attr max_width
```

```

obtype(room, [room has_attr max_height,
              room has_attr max_width,
              room has_attr max_length,
              room has_part walls,
              room has_part floor,
              room has_part ceiling,
              room has_part openings]

```

Figure 3: An object type

states that all objects of the type `room` have attributes `max_height`, `max_length`, and `max_width`. The `<has_attr>` link is abbreviated as `ObjType.Attr` for an object type or `Obj:objType.Val:Attr` for an object instance. The symbol ':' represents the relationship between an object instance and its type, which will be discussed shortly.

Aggregation is represented by the `<has_part>` link. An aggregate type is an object type that consists of one or more explicitly enumerated subobject types. For example,

```

room has_part walls
      has_part floor
      has_part ceiling
      has_part openings

```

defines `room` as an object type that consists of types `walls`, `floor`, `ceiling`, and `openings`.

The third link type is the `<has_member>` link, which represents association. An association type is a set; it is composed of one or more subobjects linked to it by a `<has_member>` link. The subobjects must all be of the same type. In this example,

```
walls has_member wall
```

`walls` is a set of one or more `wall` objects. There is no restriction on how many `wall` objects may be in this set. This link is very useful in situations where an object may have an indeterminate number of component objects of one type, particularly if the set of objects needs to be modeled as a group for some reasoning tasks.

The `<has_part>` and `<has_member>` links define a part hierarchy of objects. Besides participating in a part hierarchy, all object types and instances participate in a type hierarchy, based on the `<has_instance>` and `<has_spec>` links. The `<has_instance>` link associates an object instance with its most specific type. The `<has_spec>` link represents specialization. Specialization allows differentiated types to be formed from higher level types, forming a type hierarchy with *graph* as the root.

An example of an object type may be seen in Figure 3. Only the name and definition are shown.

The `<has_spec>` link represents specialization. For example, if

$$A <\text{has_spec}> B$$

then B is a specialization of A . In order for B to be a specialization of A , every component in A must either be inherited or inherited and specialized in B . Thus, object types describe the necessary properties rather than the typical properties of some class of objects.

A distinction is made between object types and object instances. Object instances are the actual “things” of the domain. Each instance is a member of at least one of the type classes in the set of object types. The link label that is used to denote this relationship between an object instance and an object type is $\langle \text{has_instance} \rangle$. This link is represented as

$A \langle \text{has_instance} \rangle B$

and is allowed between A and B if and only if A is an object type and B is an object instance and for every component object type c_i in A there is a c'_i which is a component object in B and $c_i \langle \text{has_instance} \rangle c'_i$. The instance may include component instances as well that are not in type A ’s definition. This is very important in a design database, where each design object, or instance, will typically have many more components and greater detail than the object types.

The Building Component Hierarchy

During a design session, a building is represented in terms of the building component hierarchy. At the top of the hierarchy is the object type `building`. This object type has various specializations, such as `educational_building` and `residential_building`. Part of its definition consists of

```
building has_member build_component
```

Therefore, a building is viewed as being made up of a set of components. Specializations of this object type include most standard architectural components such as walls, windows, doors, and so forth. A portion of the building component hierarchy may be seen in Figure 4. With each object type is associated a set of required attributes and component object types, and a set of constraints and methods. For example, the `hinged-door` object type includes attributes such as `length`, `width`, `height`, and `angle-of-swing`. An associated method calculates the projection of the door’s swing. Since each object instance is represented by a part hierarchy, graphical information needed to display the object and reason about its geometry is stored at each level in the hierarchy.

The Functional Hierarchy

One of the characteristics of design objects is that various groupings of components can be viewed as a functional module. For example, in Figure 5, corridor C and doorway E may be viewed individually as objects or as a functional group providing a means of egress. Within this functional group, the corridor functions as the `exit_access` and the doorway functions as the `protected_exit`. Such functional components cannot be easily incorporated into the primary object type hierarchy since the functionality may cut across several object types. For example, not all doorways function as protected

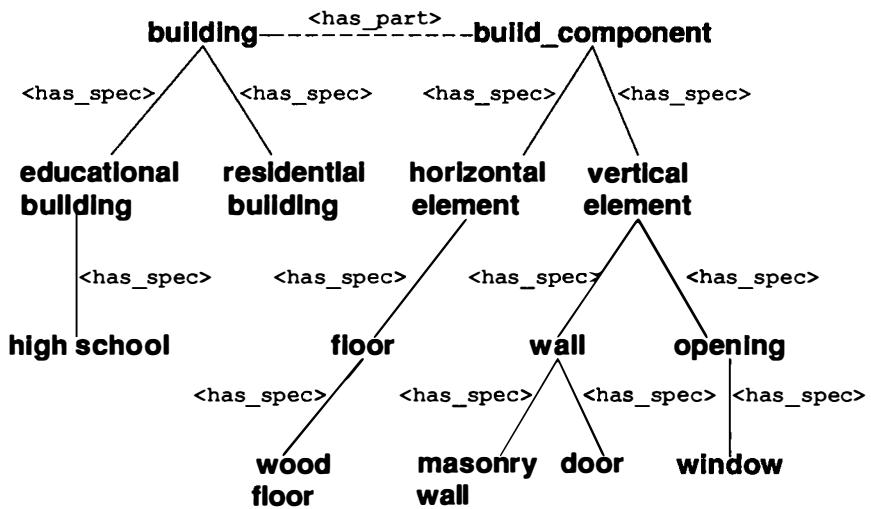


Figure 4: The building component hierarchy

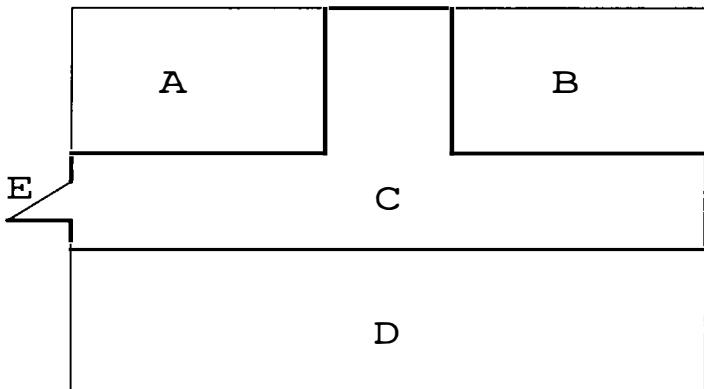


Figure 5: A floor plan

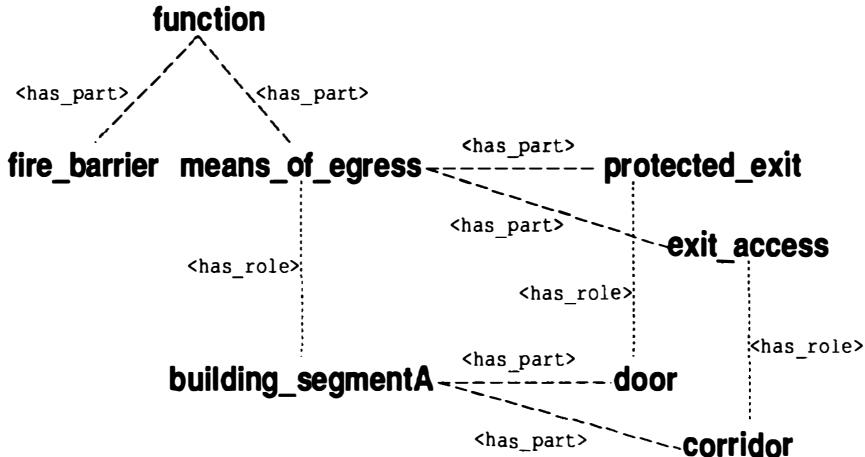


Figure 6: An example of the role link

exits (that is, fire exits), and in addition, other objects such as windows may in certain circumstances, function as exits.

Therefore, a separate type hierarchy based on functionality is part of the knowledge representation. This hierarchy is generated by the $\langle \text{has_spec} \rangle$ and $\langle \text{has_part} \rangle$ links in a fashion similar to the building component hierarchy. Object instances, however, are linked to their functional type by the $\langle \text{has_role} \rangle$ link. The link

$$A \langle \text{has_role} \rangle B$$

is allowed between A and B if and only if A is a functional object type and B is an object instance and for every component object type c_i in A there is a c'_i which is a component object in B and $c_i \langle \text{has_instance} \rangle c'_i$. An example of this link is shown in Figure 6. This figure also contains part of a functional hierarchy. In this diagram, *building_segment_A* represents the piece of the building containing the corridor and exterior door. It has the function *means_of_egress* and its two components have the functions *exit* and *exit access*. Figure 7 shows a portion of the functional hierarchy. The type *function* is a metaclass; that is, it consists of all possible classes or types of functions.

Various specialized methods may be associated with a functional type. For instance, *get_travel_dist* is associated with *means_of_egress*, and generates the travel distance to an exit. The functional type *firebarrier* includes the attribute *fire resistance rating*. An object instance of type T that is linked to functional type F by the $\langle \text{has_role} \rangle$ link will inherit the attributes, methods, and constraints of both T and F . For example, if

```
wall has_instance w1
w1 has_role fire_barrier
```

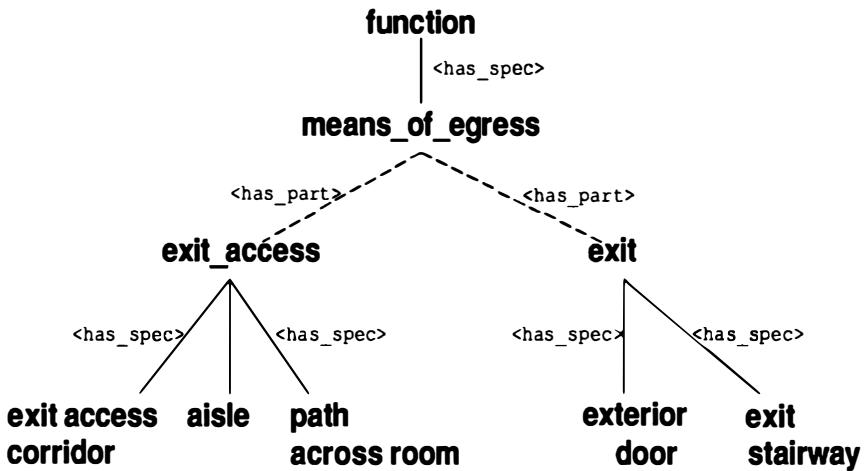


Figure 7: The functional hierarchy

then w1 will inherit length and color from wall and fire resistance rating from fire_barrier. Thus, objects and groups of objects may be viewed according to their function in the building design.

FIRE CODE CONSTRAINTS

An approach to run-time constraint management which has gained quite a bit of interest is that of triggers(Dittrich, Kotz, and Mulle 1986, Shepard and Kerschberg 1984, Dayal and Smith 1986, Eick, Kumar and Liu 1989). A trigger is composed of a condition-action pair where the condition is some logical statement about the database, and the action that is specified is taken automatically if the trigger condition occurs. The approach closest to that taken in this work is the Event/Trigger model proposed by Dittrich(Dittrich, Kotz, and Mulle 1986). This model permits user-defined actions to be triggered by events that may be raised at arbitrary times. Our approach is to model the relationship between a constraint rule and the objects it constrains as part of the data model, and then use this knowledge to determine which constraints should be checked upon update.

Syntax of constraint rules

Currently, design code constraints are implemented as IF-THEN rules. This representation was chosen because as stated earlier, a constraint is permitted to be any formula of predicate calculus, and fire code is most naturally modeled as conditional rules, since it already exists in that form. The rules consist of object instances, object type names, relations, method calls and logical connectors. The permissible operators are AND, OR and NOT. Within a rule, each term is one of the following:

1. a predicate that can be immediately evaluated from the current knowledge representation, that is, a $R b$ where R is explicitly represented in the knowledge representation.
2. a predicate that is inferrable from the knowledge representation.
3. a method call.

Methods are needed to represent procedural knowledge and may return values other than true or false. Many of the methods appear in rule antecedents and are slow and expensive. Also, there are potentially many rules in this system. If in a design session, only a doorway length attribute is modified, the system should not evaluate the entire building against all of the rules. Therefore, an important feature is selective evaluation of the rules; only rules that constrain objects, attributes, and relationships that have actually changed will be evaluated. In ArchObjects, checking is initiated at the end of a design session or whenever the user initiates it. In either case, only the objects and attributes that were actually inserted or modified in some way should be evaluated. Typically, in a session where a large segment of the building is developed, there will be many objects inserted and hence a large amount of checking activity will be generated at the end. It is also possible to work incrementally, modifying a building, checking the modification, and then making any corrections.

These constraints are not enforced by transaction rollback. Rather, violations merely generate warnings that are returned to the user. A record of violations must also be maintained, since the user may not correct the problem. If the designer then returns to the design later and rechecks it, the violation should again be reported, without having to reevaluate the rule *unless* the designer modified the affected objects.

In our system model, there are three basic operations : insert, delete, update. These operations are defined according to these rules :

- An object is inserted if the fact $object : objecttype$ is added to the KB and all required attributes and components of the type $objecttype$ are added.
- An object is deleted if $object : objecttype$ is deleted and all required attributes and components are deleted and all external relationships are deleted.
- An object is updated if any attribute is updated or any component object is updated, inserted, or deleted.
- A relationship R is inserted if a $R b$ is added to the KB
- A relationship R is deleted if a $R b$ is deleted from the KB.
- An attribute is updated if $O : obtype.a : attr_type$ is updated - that is, the attribute value is changed.

Note that ensuring the consistency of object insertions and deletions may require the evaluation of definitional constraints.

The next step is to analyze the relationship of the constraint rules to the possible system actions. For any rule expressed as a conditional:

IF $p_1 \dots p_n$ THEN $d_1 \dots d_j$

if any p_i is inserted, the antecedent of the rule may evaluate to true. If so, the consequent must then be evaluated. If any d_i is deleted, on the other hand, the consequent may evaluate to false. If it does, the antecedent must be checked. If it evaluates to true, the constraint fails. For example, consider the rule

```
R3 (IF D:door_assembly AND
    D:means_of_egress_component AND
    derivecomp(D, lock, L) AND
    L.keyrequired = true
THEN
    D has_part S:sign AND
    S.msg = "Warning...")
```

In other words, if D is a door assembly in a means of egress and its lock requires a key, then a warning sign is required. This rule should be evaluated upon insertion of door1:door_assembly or deletion of door1 has_part sign3:sign, for instance.

For each rule, a set of triggers is maintained. Each trigger consists of an activation pattern and a rule to fire if the pattern is matched by an update action. Triggers are generated for a rule when it is added to the system, based on the relations that appear in it. Currently, specification of the triggers is done by the rule designer, although methods exist for analyzing such rules in order to automatically generate triggers(Urban 1989). For example, if a rule S is added to the constraint base, and X:Y appears in the antecedent of S, then

```
trigger(X:Y, insert, S)
trigger(X:Y, update, S)
```

is generated. If X:Y appears in the consequent,

```
trigger(X:Y, delete, S)
trigger(X:Y, update, S)
```

is generated. Triggers are maintained as part of the constraint base. Therefore, if the action insert (X:Y) is produced by some change to the graphical representation, constraint rule S is checked. Some of the clauses in these rules are method calls rather than explicitly represented in the KB. These must be triggered when values specified in the derivational constraints associated with methods are updated.

The system functions in the following manner. As the architect works with a design, a series of insert, delete, and update actions are recorded, based on changes the architect makes to the graphical representation of the design object. At the point at which checking is to be done, these actions are analyzed against the triggers. For example, during a design session, the architect might erase a warning sign component from a door object, causing the system to generate this update action :

```
delete(door1:doorassembly has_part s1:sign)
```

If there is a trigger

```
trigger(D:door_assembly has_part S:sign, delete, R3)
```

the rule R3, shown above, will be evaluated with its variables bound to the constants recorded in the delete action. In this case, the initial bindings will be D/door1 and S/s1. The system then searches the internal representation of the design object being verified for a consistent set of bindings. Then, if the rule antecedent evaluates to true, and the delete action would cause the consequent to evaluate to false, a constraint violation is produced. Remember that this checking may take place quite some time after the architect actually deleted the relationship. If the rule fails,

```
violate(door1:door_assembly has_part s1:sign, Rule3)
```

will be stored. These violations will be stored with the design object in the DBMS.

CONCLUSION

This paper describes a system that integrates an object-oriented model of design objects and design codes modeled as constraints. Fire code as specified in the *Life Safety Code Handbook* was chosen to demonstrate the capabilities of the system. Modeling fire code provisions as constraints on the possible building instances that may be generated has the following advantages:

- Fire code is standardized : therefore, centralization is useful since one consistent version should be maintained.
- Fire code is legally required : therefore, verification should be enforced for all designs and designers.

This system is currently being prototyped using Prolog, C, and Postgres on a Sun workstation. In order to accomplish this, a version of the WharfRat knowledge representation system(MacKellar 1989) is being modified to handle constraints and graphical knowledge. Future work on this system includes the addition of version management capabilities, automatic generation of triggers by rule analysis, and integration with an object-oriented DBMS that includes persistent storage.

References

- Allen, E. and Iano, J. (1989). *The Architect's Studio Companion*. John Wiley and Sons.
- Autodesk, Inc., Sausalito, CA. *AutoCAD User's Manual, Release 10*, 1989.
- Buis, M., Hamer, J. Hosking, J.G. and Mugridge, W.B. (1987) An expert advisory system for fire safety code. In J.R. Quinlan, editor, *Applications of Expert Systems*, pp. 85–101. Addison-Wesley.

- Cammarata, S.J. and Melkanoff, M.A. (1986). An interactive data dictionary facility for CAD/CAM databases. In *First Workshop on Expert Database Systems*, pp. 423–440.
- Coyne, R.D., Radford, M.A., Rosenman, A.D., Balachandran, M., and Gero, J.S.(1990) *Knowledge Based Design Systems*. Addison-Wesley, New York.
- Dayal, U. and Smith, J.M. (1986). PROBE : A knowledge-oriented database management system. In M.L. Brodie and J. Mylopoulos, editors, *On Knowledge Base Management Systems*, pp. 227–257. Springer-Verlag.
- Dittrich, K.R., Kotz, M. and Mulle, J.A. (1986). An Event/Trigger mechanism to enforce complex consistency constraints in design databases. *SIGMOD Record*, 15(3):22 – 36, September 1986.
- Eick, C.F., Kumar, S. and Liu, J.L. (1989). Tolerant consistency enforcement for knowledge based systems. *Methodologies for Intelligent Systems*, 4:94–101.
- Gallaire, H., Minker, J. and Nicolas, J.(1984). Logic and databases : a deductive approach. *ACM Computing Surveys*, 16(2):153–185.
- Geller, J. and Shapiro, S.C. (1987). Graphical deep knowledge for intelligent machine drafting. In *Tenth International Joint Conference on Artificial Intelligence*, pp. 545–551, Los Altos, CA, 1987. Morgan Kaufmann Publishers.
- Kim, U. (1986). Model for an integrated design evaluation system using knowledge bases. In *ACADIA Workshop*, pp. 203–215.
- Lathrop, J.K. (1988). *Life Safety Code Handbook*. National Fire Protection Association, Inc.
- MacKellar, B.K. and Maryanski, F.J. (1989). A knowledge base for code reuse by similarity. In *Proceedings of COMPSAC*, pp. 634–641.
- MacKellar, B.K. (1989). *Retrieval by Similarity in a Knowledge Base of Reusable Code*. PhD thesis, University of Connecticut.
- MacKellar, B.K. (1991). A constraint-based model of design object versions. Paper submitted for publication.
- Nicolas, J.M. (1982). Logic for improving integrity checking in relational databases. *Acta Informatica*, 18(3):227–253.
- Ontologic, Inc. (1990), Three Burlington Woods, Burlington, MA. *ONTOS Object Database Documentation, Release 1.5*.
- Ozel, F. (1985). Fire safety code evaluation and computer-aided design. In *Proceedings of AIA Research and Design Conference*, pp. 197–202.
- Peckham, J.M. (1990). *Constraint Based Analysis of Database Update Propagation*. PhD thesis, University of Connecticut.

- Rich, C.H. and Shrobe, E. (1978). Initial report on a LISP programmer's apprentice. *IEEE Transactions on Software Engineering*, 4(6):456–467, Nov. 1978.
- Rosenman, M.A., Gero, J.S., Hutchinson, P.J., and Oxman, R. (1986). Expert systems applications in computer-aided design. In A. Smith, editor, *Knowledge Engineering and Computer Modelling in CAD*, pages 218–225. Butterworth and Co.
- Shepard, A. and Kerschberg, L. (1984). Prism : A knowledge based system for semantic integrity specification and enforcement in database systems. In *Proceedings of SIGMOD*, pp. 307–315.
- Sriram, D. (1986). Destiny : A model for integrated structural design. In A. Smith, editor, *Knowledge Engineering and Computer Modelling in CAD*, pp. 226–235. Butterworth and Co.
- Stonebraker, M., Rowe, L.A., and Hirohama, M. (1990). The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, March 1990.
- Urban, S.D. (1989). Alice : An assertion language for integrity constraint expression. In *Proceedings of COMPSAC*, pp. 292–299.
- Wilkes, W., Klahold, P., and Schlageter, G. (1989). Complex and composite objects in CAD/CAM databases. In *Fifth Conference on Data Engineering*, pp. 443–450.
- Williams, B.C. (1990). Interaction-based invention : Designing novel devices from first principles. In *Proceedings of AAAI*, pp. 349–356.

CADSYN: using case and decomposition knowledge for design synthesis

M. L. Maher and D. M. Zhang

Department of Architectural and Design Science
University of Sydney NSW 2006
Australia

Abstract. Design synthesis is a part of the design process during which alternative design solutions are generated. Two process models of design synthesis are case-based reasoning and decomposition. Combining the two approaches allows previous experience to be used directly when available and relevant, and generalized knowledge to be used when direct experience is not appropriate. This hybrid approach to knowledge-based design has been implemented as an extension to an existing knowledge-based design environment, EDESYN. EDESYN provides a domain independent inference mechanism for design by decomposition using a constraint-directed depth-first search through a dynamically defined hierarchy of systems and components. The knowledge base in EDESYN is represented by systems/subsystems, planning rules for decomposing in a specific context, constraints, and mathematical expressions for assigning numerical values to attributes. EDESYN has been extended to accommodate case-based reasoning by adding a case base of design examples and providing a pattern matching algorithm to locate relevant cases. The advantage of using the two approaches in the same system is that the generalized knowledge in EDESYN's knowledge base can be used to transform a previous design situation to the new design context.

INTRODUCTION

Design synthesis is defined to be the generation of alternative design solutions, usually a synthesis of components and systems available in the design domain that is appropriate for a specific design context. This aspect of design requires a search for appropriate components such that a set of design specifications is satisfied. Two computational approaches to design synthesis are case-based reasoning and decomposition (Maher, 1990). The use of these computational approaches requires the development and representation of design knowledge. The knowledge required to apply these two approaches as knowledge-based design systems is very different. The case-based approach requires specific cases or examples of design situations and generalized knowledge about how to transform a previous design situation to work in a new design context. The decomposition approach requires generalized knowledge about how a particular domain is decomposed into various systems and components and how decomposed solutions can be recomposed into a coherent whole solution. The advantage of combining the two approaches in one knowledge-based design system lies in

the access to previous experience in the form of specific design situations directly when available and relevant and the use of generalized knowledge when direct experience is not appropriate.

This paper briefly reviews the decomposition and case-base reasoning approaches to design, then gives an overview of EDESYN as an implementation of the decomposition approach. The combined approach of decomposition and case-based reasoning for design has been implemented as a system called CADSYN. After a description of CADSYN, conclusions and directions are presented.

DECOMPOSITION

Decomposition, as illustrated in Figure 1, is perhaps the most ubiquitous model of design. It follows directly from the development of design methodology and has been shown to be useful in the development of knowledge-based design systems.

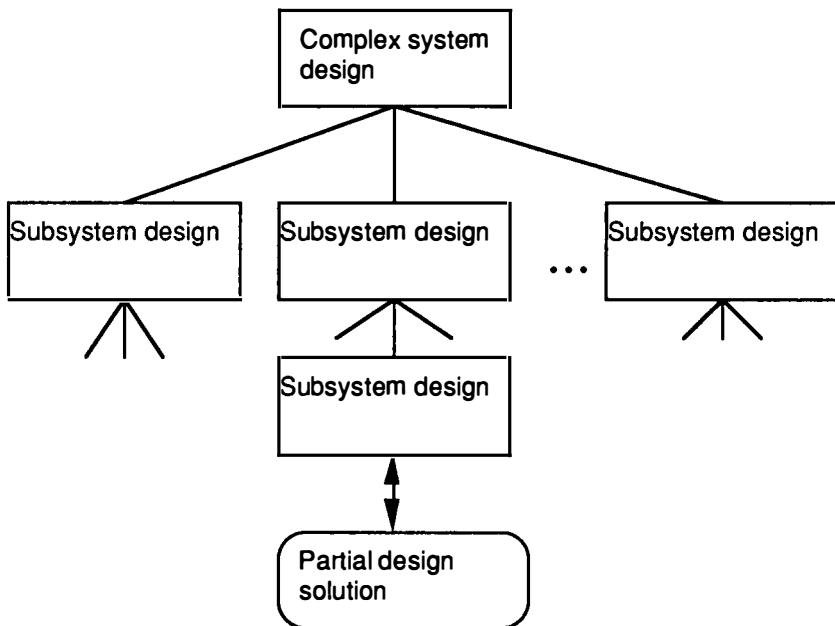


Figure 1. Decomposition approach to design

Decomposition by definition means that something is decomposed, it also implies a recomposition. The early expert systems for design synthesis implicitly used the decomposition model. HI-RISE (Maher and Fenves, 1984; Maher, 1988), an expert system for the preliminary design of high rise buildings, used both a rule-based and frame-based representation of design knowledge. R1 (XCON), an expert system for the configuration of computer systems, used a rule-based approach. In both cases, the representation of the design knowledge was dictated by the language in which the system was implemented; PSRL (Rychener, 1984) in the case of HI-RISE and OPS5 (Forgney, 1981) for R1. These

expert systems led to the generalization of this approach to design to define decomposition languages for developing a knowledge base. For example, HI-RISE led to EDESYN (Maher, 1988) and R1 to SALT (Marcus et. al., 1988). These two systems were not unique in this sense. Many efforts in developing design expert systems led to the development of languages for decomposition in the anticipation that this would facilitate the development of additional design expert systems and provide a formalism for representing design knowledge.

Knowledge-based systems for design by decomposition have been developed that identify specific languages for describing design knowledge. Examples of such a languages include DSPL (Brown, 1985), EDESYN, VEXED (Steinberg, 1987). The identification of various languages for describing design knowledge has raised a number of issues in the application of the decomposition approach to complex design problems. Guidelines for implementing a decomposition approach to design synthesis are: representation of decomposition and methods for recombination.

Decomposition. When developing a knowledge base for design by decomposition, the design knowledge is generalized into various classes of elements. Such classes may be organized according to different categories of design knowledge. For example, Gero describes prototypes as generalized schemes of design knowledge organized around function, behavior, and structure (Gero, 1990). In DSPL, the design decomposition knowledge is organized around agents, specialists, etc. In VEXED, the design knowledge is organized around refinement rules. A fixed decomposition implies that it is not altered for a specific problem, but is used without modification for all problems presented. At a general level of abstraction this may be possible. For example it is useful to decompose a building design problem into architectural space planning, structural design, and services design at a general level, but a more detailed level of structural design may not have a fixed decomposition.

Recomposition. The assumption behind decomposition is that each problem can be solved independently of the other subproblems. However, this is an idealisation rather than a reality for any problem definition. The rule of thumb in using a decomposition approach is to decompose into nearly independent subproblems, or loosely coupled subproblems. A well understood problem is easier to decompose than one that has not been explored before. Once a decomposition approach is used, recombination becomes an issue. Recombination can occur implicitly, in which the solutions to the subproblems are considered to be the solution to the entire problem. Recombination usually introduces complications through the interaction of the subproblem solutions. One way of representing the interactions is as constraints; then the issue of recombination becomes one of constraint satisfaction.

The various computer environments that have been developed for design by decomposition address each of these issues through the identification of a structured language or syntax for describing a knowledge base. As an example, EDESYN provides two major representation structures for the representation of design knowledge: systems and constraints. The representation of various systems allows the representation of the decomposition knowledge as discrete systems, although the decomposition of any one system can vary as design

proceeds. EDESYN provides a uniform representation for function, structure and behavior, leaving the distinction to the knowledge base developer. The representation of various constraints allows the recomposition knowledge to be stated explicitly. What we gain from the implementation of the decomposition approach as domain independent computer environments for design is a better understanding of decomposition and its associated issues, but more importantly, the realization that such approaches to design are not domain specific but are augmented by a declarative representation of domain specific knowledge.

CASE-BASED REASONING

Case-based reasoning is a model of design that makes direct use of design experience in the form of episodes, rather than compiling and generalizing the experience. The model, as illustrated in Figure 2, employs analogical reasoning to select and transform specific solutions to previous design problems to be appropriate as solutions for a new design problem. This model is attractive because the knowledge acquisition for developing generalized representations of design knowledge in a particular domain can be difficult and time consuming. The issues associated with using this model for design include the identification of the necessary information about a design episode in order to reason about its applicability in a different context, the meaning of a similar design, and the transformation of the solution from the original context to the new context.

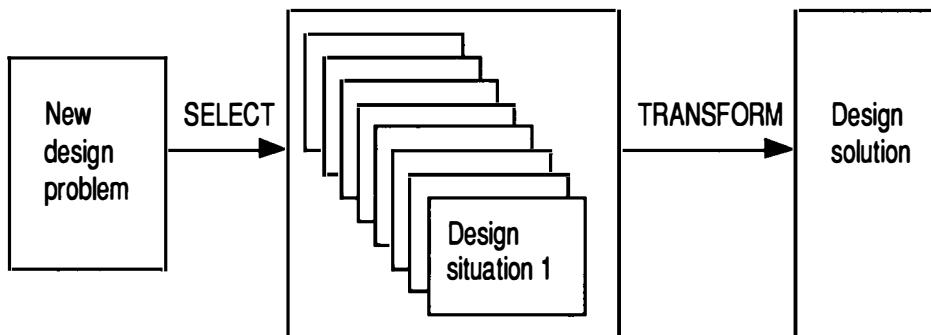


Figure 2. Case-based reasoning approach to design

Case-based reasoning in design involves the generation of a design solution using the solution or solution process from a previous design problem as a basis. This model of design synthesis requires specific design situations rather than generalizations about a design domain. Examples of a case-based reasoning approach to design include STRUPLE (Zhao and Maher, 1988) and BOGART (Mostow, et.al., 1989). STRUPLE uses a database of building design solutions, finds partial matches to a new design problem, and provides a set of relevant design components for potential solutions of the new design problem. BOGART uses a library of circuit design plans, allows the user to select a relevant previous design, and replays the previous design plan for a new design problem. These two systems are based on previous efforts in developing decomposition languages for design (EDESYN in the case of STRUPLE and VEXED in the case of BOGART) and the need to use experience more

directly as the knowledge acquisition of generalized decompositions was proving to be difficult.

In design synthesis, the operations of most interest are centered around case retrieval, selection, and modification. Assuming that case memory is very large, that is, many design cases are stored, retrieval becomes a search problem in a very large space. The selection of a case among potential cases requires recognition of the relevance of each case and how close the case is to providing a solution to the design problem. The modification of the case for the new design problem introduces the issues of: what should be changed and what stays the same. It can be assumed that the changes are based on the results of a partial match, where the features of the case that did not match should be changed, but this may not be sufficient.

There are guidelines to the development of a case-based reasoning system, regardless of whether the problem is design, planning, or diagnosis. For example: case memory organization, retrieval algorithms, and selecting the best case.

Case memory organization. The extremes in representing cases in memory are to represent each case in its entirety (e.g., Riesbeck, 1988; Stanfill and Waltz, 1986; Koton, 1988; Hammond 1989) or to break each case into pieces (e.g. Carbonell, 1983; Kolodner, 1988; Hinrichs, 1988). When breaking a case into pieces there are various approaches, such as conceptual clustering (Fischer, 1988) and discrimination networks (Feigenbaum, 1963).

Retrieval algorithms. There are basically two guidelines in the development of a retrieval algorithm: concept refinement and parallel search. Concept refinement assumes that the structure of case memory is hierarchical in nature. The more general features, or shared features of various cases are searched first, eliminating large amounts of cases according to these shared features. There are examples in which cases were stored in entirety using redundant discriminate networks (Kolodner 1983) and in which cases were stored in pieces (Kolodner, 1988). Parallel implementations use multiple processors so that every case in memory can be checked simultaneously (Stanfill and Waltz, 1986).

Selecting the best case. A weighted count of matching features provides one way to select the best case, however, this does not take into account that the case itself may determine the importance of a feature. Some approaches to finding the best case are: preference heuristics (Kolodner, 1988), dimensional analysis (Rissland and Ashley, 1988), and dynamically-changing weighted evaluation functions (Stanfill, 1987).

These guidelines provide a basic understanding of case-based reasoning, but do not directly address the issues of case-based reasoning as a model for design synthesis. When storing design situations as cases, the content as well as the organization of a case must be considered. Most design solutions are stored as descriptions of the physical object or system, most commonly this description is in the form of drawings or geometric models. The synthesis of a design solution for a new problem starts with a set of specifications and requirements. These specifications do not consist solely of geometric information. So, unless the cases include information about the intended function, behavior, or performances of the design solution, retrieval of specific design cases may be distorted towards similar

geometries. This is partly addressed by the identification of a representation of design cases that includes function, behavior, performance, as well as geometric descriptions.

These guidelines do not address the transformation problem. Once a case has been selected to be a candidate starting point for a new design context, decisions have to be made about what stays the same and what changes. The two alternatives for reusing a previous design situation are to replay the operators that produced the solution and to reuse the design solution itself. These are called derivational analogy and transformational analogy, respectively (Carbonell, 1986). In order to implement this as a knowledge-based design system, generalized transformation knowledge must be developed and represented. Since one of the attractive features of case-based reasoning is the ability to represent design experience directly rather than as generalizations, the required generalized knowledge for the transformation process is an issue that needs to be resolved.

EDESYN

EDESYN provides an environment for developing knowledge-based design synthesis programs. The architecture of EDESYN is illustrated in Figure 3. A more detailed description of EDESYN is given in (Maher, 1988). Only the knowledge representation details relevant to CADSYN are described here.

The *knowledge base* is made up of decomposition knowledge in the form of generalized systems, subsystems and their attributes. The planning knowledge is in the form of generalized rules for revising the default decomposition for a particular design context.

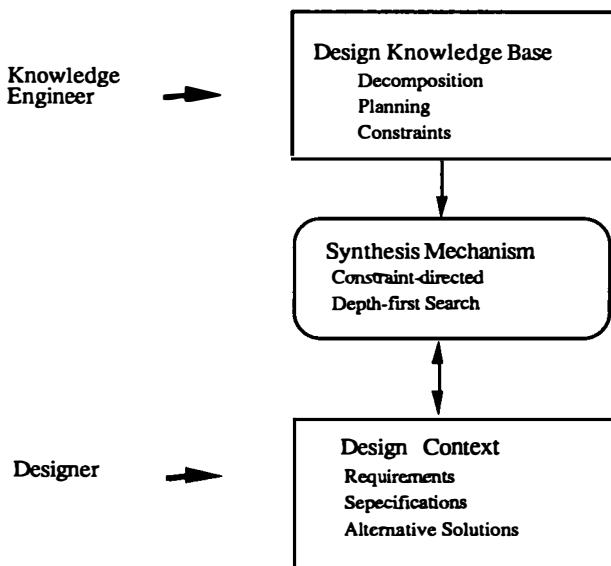


Figure 3. Architecture of EDESYN

The constraints knowledge is in the form of rules that represent elimination constraints on recombination. The *synthesis mechanism* in EDESYN is a constraint-directed depth-first search through the knowledge base, using the planning knowledge to guide the decomposition and the constraint knowledge to guide the generation of feasible design solutions. The synthesis mechanism is implemented as an algorithm which operates on the declarative representation of the knowledge base. The *design context* initially contains the design specifications and requirements. As the design process proceeds, the design context is expanded to include the alternative design solutions.

Figure 4 illustrates the transformation from a generalized description of the design knowledge as decomposed systems into a set of specific alternative solutions. Each subsystem in the knowledge base is considered as a template for producing alternative design solutions. The decomposition tree is an AND tree in which each relevant system is considered as a part of the design solution space. The tree of alternative solutions is an OR tree in which each alternative instance of an attribute of a system is represented by a node in the tree. A design solution is a path through the tree. In this way, EDESYN produces the "feasible" alternative solutions, where "feasible" is defined by the generalized constraints.

Using EDESYN to produce a knowledge-based design system involves defining:

- decomposition knowledge; generalized systems, subsystems, their attributes, and the potential values for the attributes,
- planning knowledge; rules for revising the default decomposition as the design context expands, and
- constraint knowledge; generalized constraints that define the boundaries of feasible solutions.

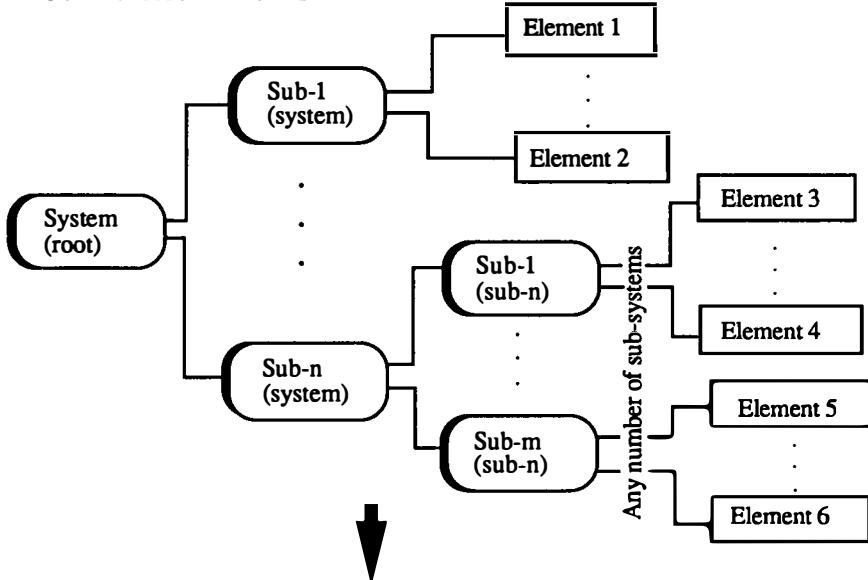
CADSYN

CADSYN is an extension of EDESYN developed to explore the use of case-based reasoning in conjunction with decomposition knowledge for the synthesis of design solutions. The purpose of adding a case memory to the EDESYN knowledge base is to take advantage of the two kinds of knowledge available to a designer: generalized knowledge of classes of design solutions and specific examples of design situations.

CADSYN generates a design solution by considering relevant previous design situations when appropriate, otherwise EDESYN's decomposition approach is used. The two approaches are tightly integrated. When a system is to be designed, an anticipator is invoked before decomposition is used. The anticipator checks for relevant cases in case memory. If a relevant case is found, it is transformed into a solution for the new design problem. If a relevant case is not found, the system is decomposed. This process is applied recursively to each system encountered in the decomposition process until a final solution is generated.

CADSYN is illustrated in Figure 5. The system comprises four major modules: case memory, case-based reasoning mechanism, generalized design knowledge, and the EDESYN synthesis mechanism. The case memory contains previous design situations to

DECOMPOSITION KNOWLEDGE



ALTERNATIVE SOLUTIONS

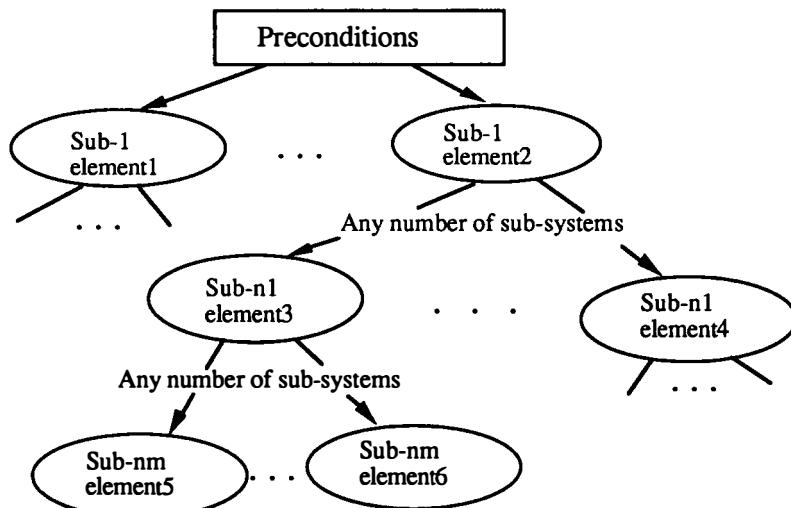


Figure 4. Generating alternative designs in EDESYN

serve as a starting point for a new design problem. The case-based reasoning mechanism uses a retrieve-select-transform algorithm for finding and using the relevant previous design situations. The generalised knowledge comprises the knowledge-base of EDESYN and some generalised adaptation rules for the case-based reasoning mechanism. The EDESYN synthesis mechanism is a constraint-directed depth-first search described in the previous section. Case memory organization and the case-based reasoner are described further below.

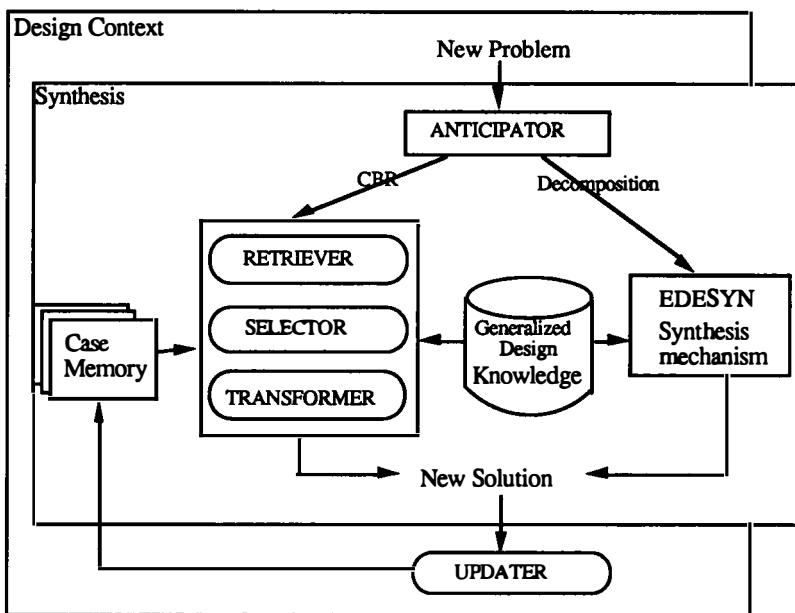


Figure 5. Architecture of CADSYN

CASE MEMORY ORGANISATION

The organization of case memory in CADSYN can be viewed as two components: internal case hierarchies and case indexing representation. The internal case hierarchies are the description of what will be stored in a case and how a case is organized. The case indexing representation provides an efficient search of cases through indexing cases.

A design case is stored declaratively in the form of attribute-value pairs in an abstraction hierarchy structure. Each case includes a supercase and a number of subcases, as illustrated in Figure 6. An entire case is represented as a path through the tree where each node corresponds to a subsystem in the generalized decomposition knowledge. The supercase of a case provides the initial design requirements and each subcase corresponds to a subsystem description.

The case indexing representation contains a problem hierarchy and a solution list. The problem hierarchy is a set of generalization nodes based on previous sets of design requirements and is developed using a conceptual clustering approach. In the problem hierarchy, design problem requirements are used as indices of cases to retrieve relevant design solution descriptions.

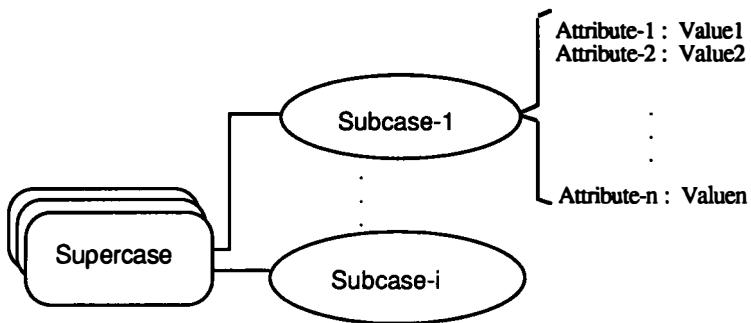


Figure 6. Structure of a case in CADSYN.

The solution list consists of all systems and subsystems in the decomposition knowledge in EDESYN's knowledge base, and serves to index subsystems of cases. Each system or subsystem index indicates relevant cases that contain a specific subcase. The solution list is used to find relevant cases which have a given subsystem solution whenever the case-based reasoner is invoked at any point of decomposition. This representation is illustrated in Figure 7.

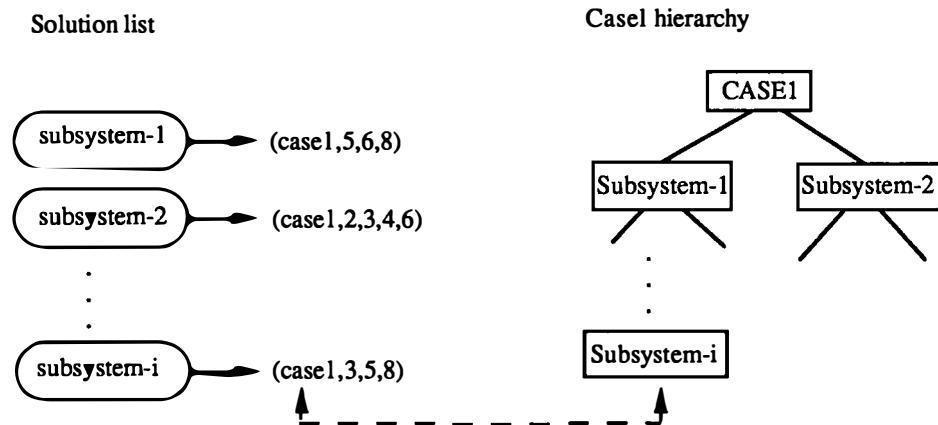


Figure 7. The solution list.

CASE-BASED REASONER

The case-based reasoner of CADSYN uses a retrieve-select-transform algorithm for finding and adapting the most relevant previous design situation to create a solution for a new design problem. In CADSYN, a new design problem is defined by the specifications of the design problem plus any system attributes that may have been assigned by the EDESYN synthesis mechanism. Since the case-based reasoner can be invoked at any point in the design process, rather than only at the beginning, the design problem may include a partial design solution.

These specifications serve as the features to be matched when retrieving relevant cases. The case-based reasoner comprises three processes: retriever, selector, and transformer.

The retriever uses the concept refinement technique to find relevant cases for a new design problem. According to the problem hierarchy in the case memory, the retriever searches previous relevant design examples from the more general nodes to the more specific nodes, then presents those retrieved relevant cases to the selector. After the most relevant case is identified, the design solution of this case is retrieved from its internal case hierarchy.

The case-based reasoner applies a weighted count approach as a similarity metric to select the most relevant case from retrieved relevant cases. The measure of an individual matching case is determined by feature evaluation and global evaluation. The process of selection is described below in detail.

- (a) A similarity metric is set up for judging the similarity between cases and the new problem. The metric is comprised of problem requirements and any attribute-value pairs of a partial solution to the new problem. A feature evaluation is made on each attribute in the metric by using matching criteria. The matching criteria provides weights of matching for a case according to matches of attributes and values between relevant cases and the new problem.
- (b) The global evaluation is used to measure how a relevant case matches the new problem under consideration of all feature evaluations, which is the summation of all feature evaluations.
- (c) The designer is queried to identify the most relevant case for the new problem on the results of the global evaluation of all relevant cases.

When the most relevant case is selected, the transformer is invoked. The transformation process is illustrated in Figure 8. In CADSYN, the transformation of a case is an iterative process by integration of transformational and derivational analogies.

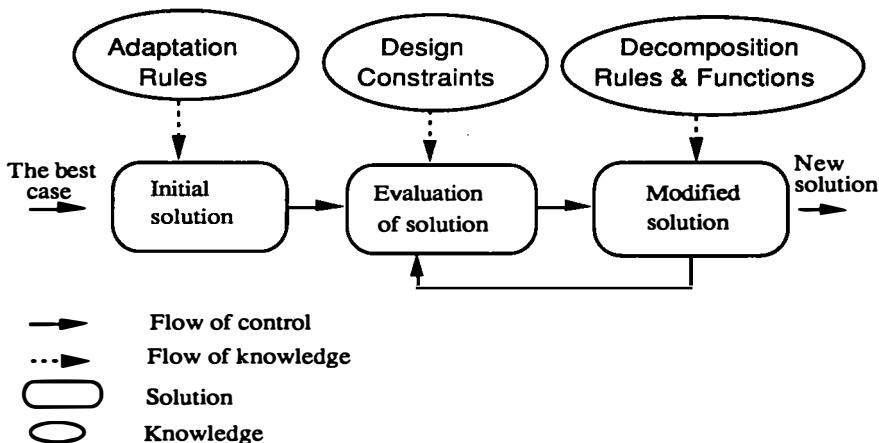


Figure 8. Transformation of a case.

Through adaptation of the most relevant case, an initial potential solution for new problem is created by using adaptation rules in the generalized knowledge base. Once a solution has been adapted, the transformer checks whether the solution is feasible using the design constraints in the generalized design knowledge. If the solution produced violates design constraints, the transformer uses decomposition rules or replays the reasoning process for modifying the current solution, then checks the solution again. This process proceeds until all constraints are satisfied. The transformation process is further described below.

- (a) An initial solution is constructed based on transformational analogy by transforming the most relevant case, through a set of adaptation rules. This is done by adding missing attributes to the case description that are part of the new requirements and changing the values of attributes that have different values in the case to those in the new design.
- (b) The potential solution is checked for feasibility. The constraints in the generalized design knowledge base are used to identify attribute-values that violate one or more constraints.
- (c) The solution is modified by decomposition rules or the procedural domain knowledge in the generalized design knowledge. That means altering values of attributes variables or replaying the subgoal process procedures according to derivational analogy to assign new values.

CONCLUSIONS AND DIRECTIONS

CADSYN is currently being applied in the domain of structural design of buildings. The generalised decomposition knowledge includes systems such as rigid frames, braced frames, trusses, floor slabs, etc. The cases in the initial case memory are produced by running EDESYN for various design problems. In such a tightly coupled system, the initial results are improved performance of the design system due to the reuse of similar designs.

The implementation of CADSYN has directly addressed some of the issues in using either a decomposition or case-based reasoning approach to design synthesis. The issues of representation of decomposition and recombination knowledge are addressed specifically by the EDESYN knowledge base. Decomposition knowledge is represented by generalized systems where the decomposition of function, behavior and structure is left to the knowledge engineer defining the knowledge base. The decomposition is flexible since the selection and use of relevant systems is done during the design process. The recombination issue is addressed by the use of elimination rules that serve as constraints bounding the feasible design space.

The issues of case memory organisation, retrieval, selection, and transformation are addressed by CADSYN. Case memory is organised in a hierarchy, forming a tree in which the nodes are defined by EDESYN's decomposition knowledge. Retrieval and selection are based on simple pattern matching. A more sophisticated pattern matching is envisioned

where the importance of various attributes and their corresponding values is taken into consideration. The issue of transformation is where CADSYN's contribution lies. In combining case-based reasoning with decomposition, the generalized decomposition knowledge is available to assist in the transformation. The use of EDESYN's constraints and decomposition knowledge allowed case-based reasoning to be implemented without additional generalized design knowledge. This simplifies the use of and facilitates access to previous design solutions.

Directions for further work in this project are: to test the system with a larger knowledge base and case base and compare the performance with the use of decomposition alone, consider more domain specific retrieval and selection processes where the domain knowledge comes from the generalized decomposition knowledge rather than requiring additional generalized knowledge, and test the transformation process more extensively.

REFERENCES

- Brown, D. and Chandrasekaran, B. (1985). Expert systems for a class of mechanical design activity, in J. Gero (ed.), *Knowledge Engineering in Computer-Aided Design*, North-Holland, Amsterdam, pp.259-283.
- Carbonell, J.G. (1983). Derivational analogy and its role in problem solving, *Proceedings of the Third National Conference on Artificial Intelligence (AAAI83)*. Minneapolis, Minnesota.
- Feigenbaum, E.A. (1963). The simulation of verbal learning behavior, in E.A. Feigenbaum and J. Feldman (eds), *Computers and Thought*, McGraw Hill, New York, pp.297-309.
- Fischer, D.H. (1988). A computational account of basic level and typicality effects, *Proceedings of the Seventh International Conference on Artificial Intelligence*, Minneapolis, Minnesota.
- Forgey, C.L. (1981). OPSS User's Manual, *Technical Report, CMU-CS-81-135*, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA.
- Gero, J.S. (1990). Design prototypes: A knowledge representation scheme for design, *AI Magazine*, 11(4): 26-36.
- Hammond, K. (1989). *Case-Based Planning*, Academic Press, New York.
- Hinrichs, T.R. (1988). Towards an architecture for open world problem solving, *Proceedings of the DARPA Workshop on Case-Based Reasoning*, Morgan Kaufmann, Clearwater, Florida.
- Kolodner, J.L. (1983). Towards an understanding of the role of experience in the evolution from novice to expert. *International Journal of Man-Machine Studies*, 19: 497-518.
- Kolodner, J.L. (1988). Retrieving events from a case memory: a parallel implementation, *Proceedings of the DARPA Workshop on Case-Based Reasoning*, Morgan Kaufmann, Clearwater, Florida.
- Koton, P. (1988). Reasoning about evidence in causal explanations, *Proceedings of the DARPA Workshop on Case-Based Reasoning*, Morgan Kaufmann, Clearwater, Florida.

- Maher, M.L. and Fenves. S.J. (1984). HI-RISE: A knowledge-based expert system for the preliminary structural design of high rise buildings, *Technical Report , R-85-146*, Department of Civil Engineering, Carnegie Mellon University, Pittsburgh.
- Maher, M.L. (1988). HI-RISE: An expert system for preliminary structural design, in Michael Rychener (ed.), *Expert Systems for Engineering Design*, MacMillan, New York.
- Maher, M.L. (1988). Engineering design synthesis: a domain independent representation, *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 1(3).
- Maher, M.L. (1990). Process models for design synthesis, *AI Magazine*, 11(4): 49-58..
- Marcus, S., Stout, J., and McDermott, J. (1988). VT: an expert elevator designer that uses knowledge-based backtracking. *AI Magazine* 9(1): 95-114.
- Mostow, J., Barley, M. and Weinrich, T. (1989). Automated reuse of design plans. *Artificial Intelligence in Engineering* 4(4): 181-196.
- Reisbeck, C.K.(1988). An interface for case-based knowledge acquisition, *Proceedings of the DARPA Workshop on Case-Based Reasoning*, Morgan Kaufmann, Clearwater, Florida.
- Rissland, E.L. and Ashley, K.D. (1988). Credit assignment and the problem of competing factors in case-based reasoning, *Proceedings of the DARPA Workshop on Case-Based Reasoning*, Morgan Kaufmann, Clearwater, Florida.
- Rychener, M.D. (1984). PSRL: An SRL-based production-rule system, *Reference Manual*, Department of Computer Science, Carnegie Mellon University, Pittsburgh.
- Stanfill, C. and Waltz, D. (1986). Toward memory-based reasoning, *Communications of the ACM* 29(12): 1213-28.
- Stanfill, C. (1987). Memory-based reasoning applied to English pronunciation, *Proceedings of the Sixth National Conference on Artificial Intelligence*, Seattle, Washington.
- Steinberg, L. (1987). Design as refinement plus constraint propagation: the VEXED experience, *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI87)*, Seattle, Washington.
- Zhao, F. and Maher, M.L. (1988). Using analogical reasoning to design buildings. *Engineering with Computers* 4.

A design-dependent approach to integrated structural design

J. Wang and H. C. Howard

Department of Civil Engineering
Stanford University
Stanford CA 94305-4020 USA

Abstract. Design-dependent knowledge can be characterized as a memory of good (and bad) designs and design strategies together with the rationale that supports them. Expert designers have traditionally used this type of experience-based knowledge to produce design productivity and quality. This paper presents the architecture of DDIS, a prototype that combines case-based reasoning with design-independent knowledge in a blackboard framework. DDIS has been implemented in KEE using an object-oriented approach to solve structural engineering design problems.

INTRODUCTION

The design of engineering structures is a complex process requiring knowledge of structural material properties, mechanics of materials, structural analysis and design specifications, in combination with experience built up over years of practice. The experiential knowledge ranges from detailed positive and negative experiences associated with specific design cases to abstract heuristics and general rules of thumb generalized from many projects. The former is what we call *design-dependent knowledge*, and the latter is *design-independent knowledge* (Howard, 1991 and 1989; Wang, 1988). Design-dependent knowledge is the knowledge about specific previous designs and their supporting reasoning, including previous design solutions, plan, assumptions, history, decisions and the rationale behind them. Design-independent knowledge is independent of specific design cases and can be applied generally to the problem domain.

Our goal is to capture the structural engineer's design-dependent knowledge and apply it to new design problems using case-based reasoning techniques adapted from current research in the field of artificial intelligence. Our approach combines design-dependent and design-independent knowledge in an integrated, knowledge-based structural design system (see Figure 1). The design-independent components use rule-based and frame-based methods to represent abstract knowledge about the problem domain and problem solving strategies. The

design-dependent components use case-based reasoning techniques to transfer knowledge from previous designs to current design tasks. Furthermore, the model provides for the extraction of design-independent knowledge from design-dependent knowledge (a process otherwise known as *learning*). The long-term objective is to produce integrated design systems that interact with designers during design tasks, functioning both as intelligent design assistants and as knowledge acquisition systems that record the designers' steps and rationale. In this way, the design systems become true apprentices to the experienced designers, progressively learning to solve more and more difficult design problems. In this paper, we describe DDIS (*D*esign *D*ependent and *I*ndependent *S*ystem), a prototype system that implements part of the integrated model and is the initial effort of our long-term design-dependent knowledge project.

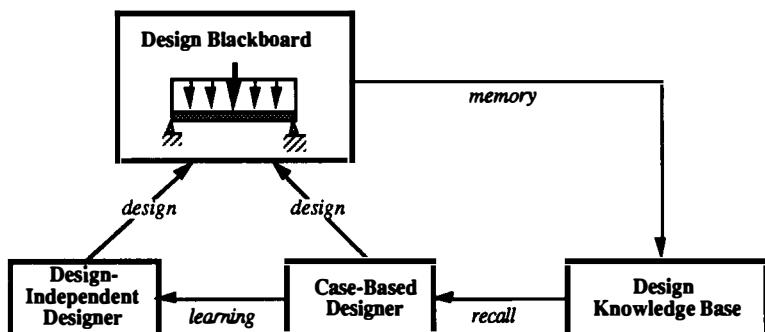


Figure 1. Overview of Integrated Knowledge-based Design System

RELATED CASED-BASED REASONING EFFORTS

Analogy has been an active research topic in cognitive psychology for many years (Winston, 1980; Gentner, 1983; Kedar-Cabelli, 1985; Prieditis, 1988, etc). Recently, increasing attention is given to case-based reasoning in the artificial intelligence community. A variety of recent efforts can be found in (Kolodner, 1988; Slade, 1988; AAAI, 1988; DARPA, 1989; AAAI, 1990). However, using previous experiences to solve a new design problem analogously is a relatively new research topic in applying knowledge-based system in the design process.

Some of the applicable analogy techniques have been implemented recently in design systems, including STRUPLE (Zhao, 1988), CYCLOPS (Navinchandra, 1988), ARGO (Huhns, 1987) and BOGART (Mostow, 1987).¹ STRUPLE is a prototype system that uses previous design solutions to identify the relevant design elements for a structural design synthesizer. The system applies transformational analogies to transfer a set of appropriate design elements to a new design as the design vocabulary so that the search space of the new

¹ This list is not intended to be comprehensive, but rather a sampling of the recent work that we find relevant to our projects.

design is better confined. CYCLOPS is a problem solver that uses case-based reasoning as one of its strategies for solving landscape design and planning problems. ARGO is an analogical reasoning system that has been applied to the design of VLSI digital circuits. BOGART reuses design plans for designing VLSI circuits, which is built on top of the interactive circuit design system VEXED to reduce the amount of required user interaction.

In our own research program, we have experimented with two small prototypes: FIRST (Daube, 1988; Howard, 1989), a system that redesigns structural beams by transformational analogy using a case memory of solution plans and RESTCOM (REdesign of STructural COMponent) (Rafiq, 1989; Howard, 1989), a Prolog-based research prototype for experimenting with similarity and matching in a small case base of reinforced concrete beams.

The ideas from case-based reasoning ideas from the artificial intelligence community combined with our initial experiments have led us to a model for combining design-dependent reasoning and design-independent reasoning in an integrated, cooperative design system.

THE DDIS ENVIRONMENT

DDIS (Design Dependent and Independent System) is a knowledge-based environment under development for the cooperative use of design-dependent and design-independent knowledge in solving design problems. In DDIS, designs are divided into subtasks, and the knowledge for solving individual subtasks is stored in the knowledge base. The problem solver uses that knowledge to develop design solutions through a generate-test-modify paradigm. Top-down decomposition plans are also stored in the knowledge base. Each step in a plan is called a goal; goals guide the problem solver toward the desired solution. The problem solver checks design constraints through the design as appropriate.

DDIS integrates design-dependent and design-independent knowledge in its knowledge base and uses it opportunistically. Design processes are divided into subtasks. Knowledge that specifies how to carry out a subtask, give a redesign suggestion, or decompose a design goal can be either case-based or heuristic-based. With a case memory, DDIS can perform each design subtask based on either design-dependent and design-independent knowledge. The system architecture of DDIS (presented later) allows case-based reasoning to compete with design-independent reasoning throughout the design process, which makes DDIS an integrated opportunistic design system.

Whenever a constraint violation is found, DDIS has to overcome the design failure by redesign (i.e., modifying the partial design). Redesign in DDIS is based on dependency-directed backtracking with knowledge-based advice. The problem solver's truth maintenance system (TMS) provides a dependency explanation for a design failure. The problem solver analyzes the dependency information and the violated constraints to suggest what part of the partial design to modify and how to fix it. The TMS updates the current design according to the modification, and the problem solver proceeds from there to its generate-test-modify cycle.

System Architecture

DDIS is based on a blackboard architecture (Nii, 1986) because that basis facilitates the integration of diverse reasoning methods. In the blackboard model, the knowledge needed to solve a problem is partitioned into independent knowledge sources that are grouped into several knowledge modules in the knowledge base. The knowledge sources modify only a global knowledge structure (the blackboard) and respond opportunistically to the changes on the blackboard. Using a blackboard architecture, DDIS can apply both design-dependent and design-independent knowledge to perform collaborative and opportunistic design. Figure 2 shows the conceptual architecture of the integrated knowledge-based design system, emphasizing the data and knowledge flow between the components.

The DDIS blackboard is divided into a control blackboard and a design information blackboard (details of the blackboard objects are presented later):

- The **Control Blackboard** has three levels: design plan, design goal and retrieved design, reflecting different levels of abstraction of control knowledge. The information on this blackboard contains the control decisions that the system uses to schedule the blackboard activities.
- The **Design Information Blackboard** stores information generated by various design knowledge sources including the evolving solutions and the design history.

The knowledge base of DDIS consists of four modules: a design-dependent (case-based) reasoner, a case recorder, a case memory knowledge base, and a design-independent reasoner. The four knowledge modules can be further divided into subsidiary modules with specific tasks:

- The **Design-independent Reasoner** contains the knowledge required for design-independent reasoning. It is comprised of abstract design rules corresponding to the general design heuristic knowledge. This module also contains some deep knowledge about domain-specific aspects such as design codes, constraints and analysis procedures. This deep knowledge is problem-dependent but task-independent knowledge, so it is not exclusive to this module.
- The **Case Memory Knowledge Base** is used to save previous design cases so that the system is able to refer to and learn from its own experiences. The organization of the case memory knowledge base is divided into a memory of plans, a memory of goals and a memory of solutions according to the contents that the case recorder stores. Note that there are no generalized cases in the memory. Cases come from the case recorder after a typical system run, or they can be manually generated and stored in the case memory.
 - The **Design-dependent Reasoner** assesses previous design cases and transfers knowledge from them to the current design task. This knowledge module consists of two subsidiary knowledge modules:

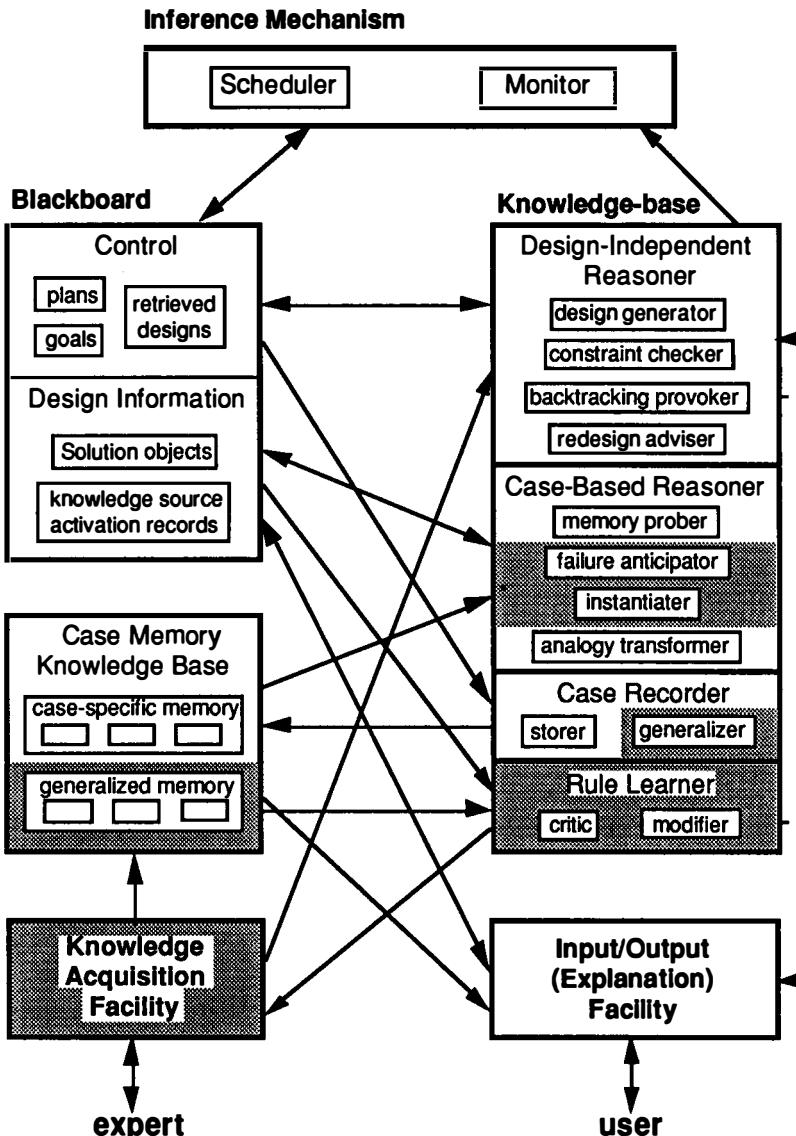


Figure 2. Data and Knowledge Flow in the Integrated Design System

- ° The **Memory Prober** retrieves potentially applicable cases from the case memory knowledge base. These cases supply the knowledge used in the other design-dependent modules. The ability of the memory prober in this version of DDIS is very limited. It retrieves all applicable cases according to the given criteria and leaves the ranking and final selection to the user.

- The *Analogy Transformer* transfers the design solution or solution strategy of a retrieved design to the new design and modifies it to satisfy the requirements of the new design. This submodule is capable of performing case-based reasoning in several aspects of the design process. It can adapt a previous design solution as the starting point of a new design session, reuse a previous design solution as part of a new design, adapt an old design plan to the new design, etc.
- The **Case Recorder** processes the design case after a design session is finished to capture the associated design solutions and plans, index them by the features that enable the memory prober to find them efficiently, and store them in the case memory knowledge base.

Knowledge Representation

Knowledge representation in DDIS is based on an object-oriented (or frame-based) approach. Due to the hierarchical nature of physical elements of structures, the frame-based data structure with value inheritance is convenient to represent the design information at both component and system level. The abstract data type and class-instance inheritance supported by an object-oriented paradigm largely facilitates the capture of the interrelatedness of design information and storage of different levels of generalization of design knowledge.

Almost all knowledge in DDIS is stored as objects at the lowest level (e.g., physical objects, design actions, plans, constraints, and design cases). Figure 3 shows some objects contained in the knowledge base of DDIS.

Each object type defines special values and procedures for a certain class of design objects or knowledge. The design objects, variables, constraints, methods, plans, and heuristics that make up the knowledge base are instances of those predefined object types. The object types are described in more detail below:

- **Control Objects** — The blackboard control objects are used to define control knowledge in DDIS. When they are posted on the control blackboard, DDIS uses the information contained in them to rate and schedule available actions. Control objects include:
 - **Plans** — A plan is the global problem solving strategy. It specifies a sequence of design goals to be followed and the intention of the plan as well as several other attributes. Plan objects can be classified into design-independent plans and design-dependent plans (design plans associated to a particular case in the case memory knowledge base). Design-dependent plans are treated exactly like design-independent plans except that they have an additional attribute to indicate their origin. Examples of a design-independent plan and a design-dependent plan are given below:

Beam-Column.System.Design.Plan

Goal.List: (Choose.Designation Choose.Material Check.Constraint)

Intention: satisfy all the design constraints

Weight: 5

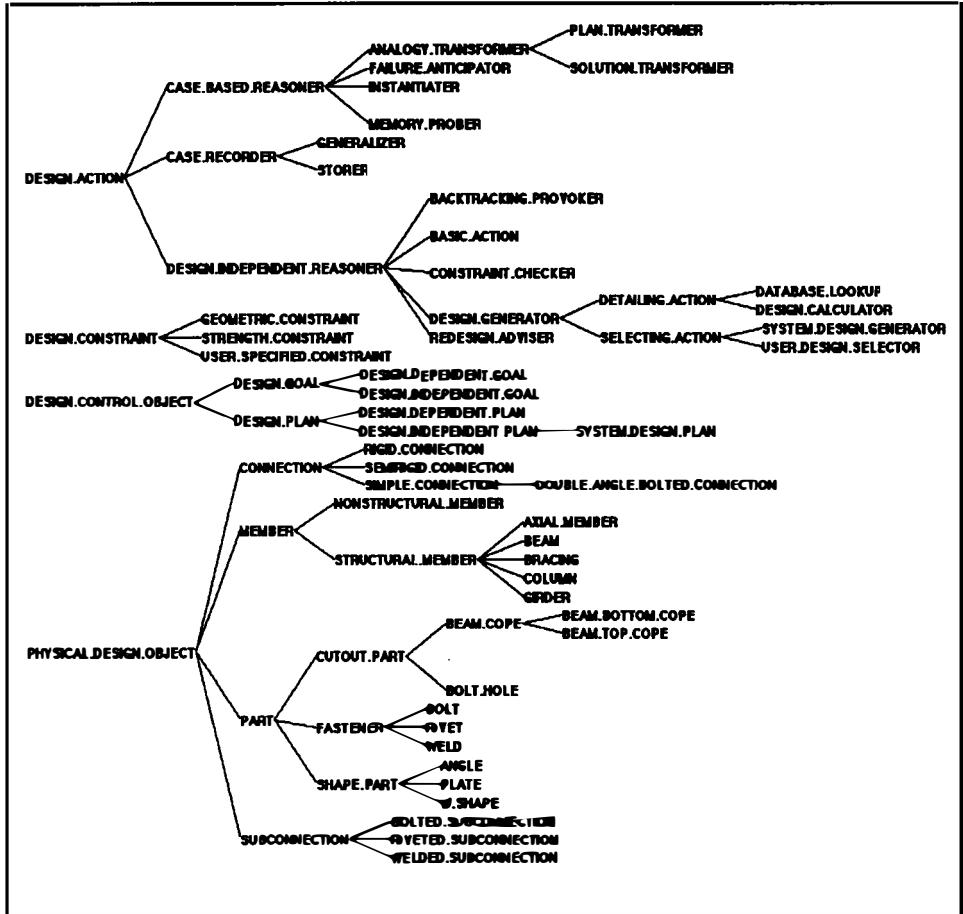


Figure 3. A Sample of the DDIS Object Hierarchy

Case.2.Design.Plan

Goal.List: (System.Select.Material System.Select.Designation
 Check.Constraint.Optimization.1
 Check.Remaining.Constraints)

Intention: satisfy all the design constraints

Originated.Case: Case.2

Weight: 5

- **Goals** — A goal is the primary rating object for the blackboard system. It can be a part of a design plan or a stand-alone design consideration. DDIS use all the activated goals on the blackboard to rate available actions. Goal objects can be classified into design-independent goals and design-dependent goals (design goals associated to a particular design-dependent plan). Design-dependent goals

have exactly the same structure as design-independent goals and are treated the same as design-independent goals on the control blackboard. Examples of a design-independent goal and a design-dependent goal are given below:

Choose.Designation

Function: favor actions that design the designation of the beam column
Included.In: Beam-Column.System.Design.Plan
Intention: design the designation of the beam column
Weight: 5

System.Select.Designation

Function: favor the SYSTEM.SELECT.DESIGNATION action
Included.In: Case.2.Design.Plan
Intention: design the designation of the beam column
Weight: 7

- *Retrieved Designs* — A retrieved design is the source of design-dependent reasoning in DDIS. It provides previous design solutions and plans that may be reused and specifies their similarity ratings, which can influence the blackboard's rating and scheduling decisions. Knowledge sources in the memory prober module are responsible for creating retrieved design objects and posting them on the blackboard.
- **Solution Objects** — Solution objects are used to store design data generated during the design process. They include the following:
 - *Physical Objects* are instantiated from design object classes in the knowledge base. Design values are stored in their attributes.
 - *Variable Objects* are used for design calculation and constraint checking. Their behaviors are inherited from their parent data item.
 - *Constraint Objects* represent design requirements that need to be satisfied, including strength constraints, geometric constraints and user specified constraints.
- **Action Objects** — Action objects include:
 - *Knowledge Source* — A knowledge source is an action object that contains information about when it is applicable, how its bindings are set, and what action to perform. Knowledge sources can be divided into domain and control knowledge sources. Domain knowledge sources contain knowledge about performing a particular design task in the problem domain. They modify only the design information blackboard. Control knowledge sources (which modify the control blackboard) contain knowledge about planning the blackboard activities. Knowledge sources may also be classified according to the knowledge they apply—both domain and control knowledge sources can be further categorized as design-independent and design-dependent. Design-independent knowledge sources are based on generalized domain knowledge, and design-dependent knowledge sources contain cased-based knowledge for transfer information from previous design cases to new designs. All the knowledge sources are treated the same in DDIS. Knowledge sources compete for execution based on how well

they fit into the current design plans on the blackboard. In DDIS, knowledge sources reside in the knowledge base.

- *Knowledge Source Activation Record (KSAR)* — A KSAR is created at runtime for each possible combination of a triggered knowledge source and a set of problem variables (a context). DDIS allows the creation of multiple KSARs from a single knowledge source with multiple contexts when triggered. KSARs instantiated from the same KS with different context bindings have different names. The attributes of a KSAR and their values are largely inherited from its parent knowledge source.

Execution Cycle

DDIS uses the blackboard model of problem solving. The design solution is generated on the blackboard incrementally by applying knowledge sources one at a time. The system is able to reason opportunistically because the sequence of executable knowledge source is determined dynamically base on the applicability of knowledge sources and the latest blackboard state (including information on both control and design information blackboard).

The problem solving cycle of DDIS involves three stages: blackboard updating, knowledge source interpreting, and scheduling:

- **Blackboard Updating** — At the beginning of every cycle, control objects on the control blackboard are updated according to the new blackboard modifications resulting from the completion of the previous cycle. Inapplicable plans, plans with no active goals, and orphaned design-dependent plans (those for which the corresponding case is no longer applicable) are removed from the blackboard. Inapplicable goals and orphaned goals (those for which the plans are no longer applicable) are removed from the blackboard. New goals are posted from plans whose previous goals have been satisfied.
- **Knowledge Source Interpretation** — The applicability of every knowledge source under the current blackboard situation is determined by a three-step process.
 - *Triggering* — A knowledge source is triggered when its primary requirements (the trigger condition) are satisfied. Triggered knowledge source is placed on the triggered agenda of DDIS.
 - *Instantiating* — A knowledge source activation record (KSAR) is created for each possible combination of a triggered knowledge source and a set of problem variables (a context). Several KSARs can be created from a single knowledge source when multiple triggering contexts exist on the blackboard.
 - *Verifying* — After the triggering and instantiating process, every KSAR's context specific requirements (the precondition) are checked. A KSAR is added to the executable agenda when its precondition is met.
- **Scheduling** — Every KSAR on the executable agenda is rated according to the control objects on the blackboard (the plans and goals), and the highest-rated KSAR is recommended to the user for execution. The result of executing the KSAR is posted back to the blackboard, and the process repeats itself until the problem is solved or no executable KSARs are found.

Note that the problem solving cycle does not differentiate between design-dependent and design-independent knowledge sources. All the knowledge sources are treated the same in the integrated system, except that design-dependent knowledge sources are triggered by the presence of the retrieved similar designs on the blackboard and have special context variables to associate them with the retrieved designs. The ratings of design-dependent KSARs, however, have to be adjusted according to the similarity of the case from which it transfers design-dependent knowledge.

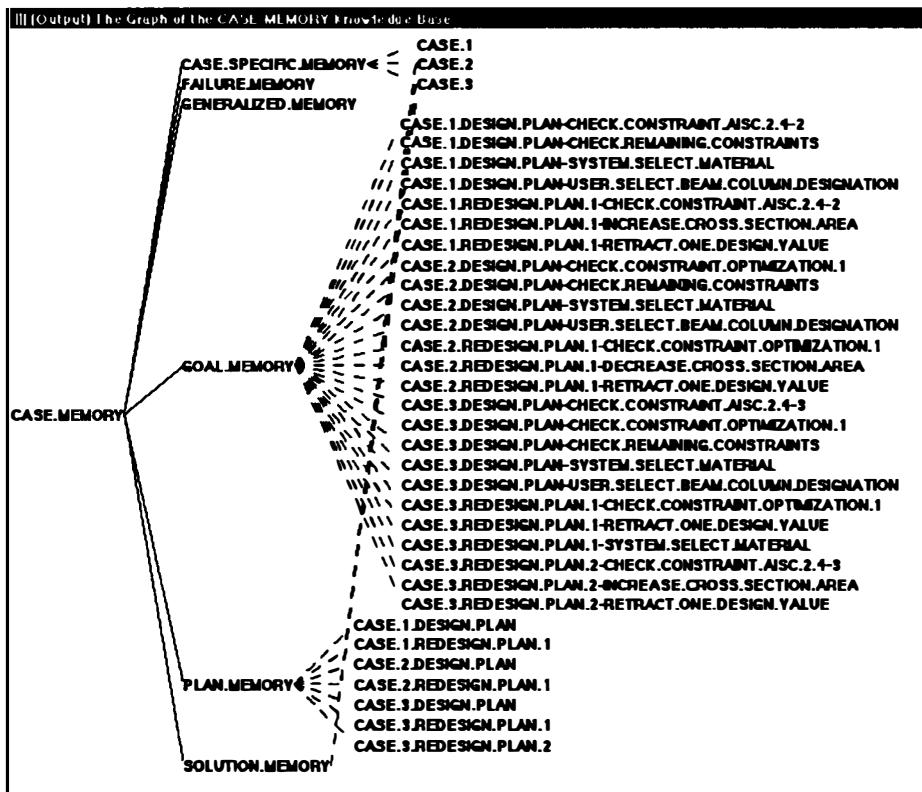


Figure 4. The Hierarchy of Case Memory Knowledge Base in DDIS

THE CASE MEMORY

The case memory knowledge base is the resource of case-based design in DDIS. Newly solved designs can be stored in the memory for reuse by the design-dependent reasoner. The case recorder captures design-dependent knowledge in several forms. The case memory needs to be able to accommodate all of them and still let the memory prober find them effectively. Therefore, the memory is divided into three parts: a memory of solutions, a memory of plans and a memory of goals, with interrelated objects (all knowledge bases in

DDIS consist of objects). Three kinds of objects are used in the case memory: design cases, design-dependent plans and design-dependent goals. Figure 4 shows the structure of the case memory with three cases.

Design Cases

A design case is the primary storage element of design-dependent knowledge in DDIS. It stores information about a particular previous design, including the problem specification, final solution, intermediate propositions, design history, design plan, redesign plans, etc. The case recorder is responsible for creating design case objects and saving them in the memory.

Design-Dependent Plans

A design-dependent plan stores the problem solving or backtracking strategy of a particular previous design. It specifies the sequence of design goals achieved by the previous design steps. The design-dependent goals are discussed in the next section.

The case recorder is responsible for creating design-dependent plan objects and saving them in the memory. The control knowledge of a previous design session can be abstracted to one global design plan and several redesign plans—design modification strategies. The plan extraction process that the case recorder uses to capture design plans is stated below:

- **Identify major design actions.** The design history is analyzed, and all KSs that modified the solution blackboard are gathered. This step is to filter out unnecessary design steps that do not directly contribute to the solution process (e.g., control KSs that only modify the control blackboard).
- **Create design-dependent goals.** A design-dependent goal is created for each identified major action (KS) in order to prefer the same KS in the future. This process is discussed in detail in the next section.
- **Differentiate design and redesign goals.** The major design actions can be classified into design and redesign actions. Therefore, the design-dependent goals are assigned to one design plan that represents the major design path and several redesign plans that represent the various backtracking processes. The goals then make up the GOAL.LIST of their plan.
- **State the intention of the plans.** The intention of a global plan is to satisfy all the applicable constraints of the design. The intention of a design-dependent redesign plan is to satisfy all unsatisfied constraints that triggered the redesign process.

Design-Dependent Goals

A design-dependent goal represents one step of the design-dependent plan that it belongs to. It contains a rating function to evaluate the usefulness of future KSARs for reproducing the effect that resulted from the past action taken at that step.

Three different rating conditions are used by the case recorder to create each design-dependent goal. The highest rating is given to the KSARs that are the same as previously used to accomplish the goal (i.e. KSARs from same KS with same bindings). KSARs that modify the same design objects and attributes attain a moderate rating. Finally, the same type of action (i.e., KSARs under the same classification as the previously used KSAR) scores a limited rating. When a design-dependent plan is reused by the plan transformers, each step of the past plan can be followed precisely, closely or loosely by DDIS's scheduling mechanism.

The intention of a design-dependent goal is the negative of the trigger condition of the KS previously executed. A goal is not applicable when its intention is true. The action is not appropriate or is already executed when its trigger condition is not true. Therefore, when the trigger condition of a previous action is not true, the design-dependent goal denoting that action is not applicable (either the goal is already accomplished or the intended purpose of the goal is not desirable). The formulation works well with the plan and goal updating mechanism of DDIS. Inapplicable goals from past plan are delayed or skipped under this setup.

THE SCHEDULING PROCESS

DDIS uses the control knowledge in plan and goal objects on the control blackboard to schedule the problem solving actions. When design-dependent actions are involved, knowledge about design similarity in retrieved designs also plays a part in the rating.

Each KSAR in the executable agenda is rated against every goal on the control blackboard by calling the function stored as the value of the FUNCTION attribute of the goal. Then the rating of a KSAR with respect to the whole control blackboard situation is the sum of its individual rating against each goal multiplied by the weight of the goal and the weight of the parent plan. After all the executable KSARs are rated, DDIS recommends the highest rated action for execution. If there are no posted plans and goals on the control blackboard, the ratings for all KSARs are zero and the most recently invoked KSAR is recommended.

Control actions compete with domain actions, and design-dependent actions compete with design-independent actions for scheduling. The rating method described above applies to every KSAR except that the individual rating of a design-dependent KSAR has to be adjusted for the similarity of its source case.

The general scheduling approach of DDIS is to prefer control KSARs to domain KSARs and to prefer design-independent KSARs to design-dependent KSARs. The individual rating of a KSAR can change with the blackboard state and case similarities. When a retrieved case is very similar to the new design, the rating of the design-dependent actions associated with the case increases. Therefore, design-independent and control actions have to compete with design-dependent actions for execution.

IMPLEMENTATION

DDIS is implemented in IntelliCorp's KEE running on a Texas Instruments MicroExplorer installed in an Apple Macintosh II. KEE is a flexible and general-purpose commercial AI development system. It is a hybrid environment, which integrates frame-based knowledge representation, rule-based reasoning, data-driven inference and object-oriented programming. It also has interactive graphics tools and a truth maintenance system (TMS).

All the elements in DDIS, as described earlier, are all implemented as classes of objects using KEE's frame-based representation. Figure 5 shows a domain knowledge base consisting of instances instantiated from the system's object classes. The rule system in KEE is not used (except for a few TMS justifications). DDIS is mainly implemented using an object-oriented programming approach.

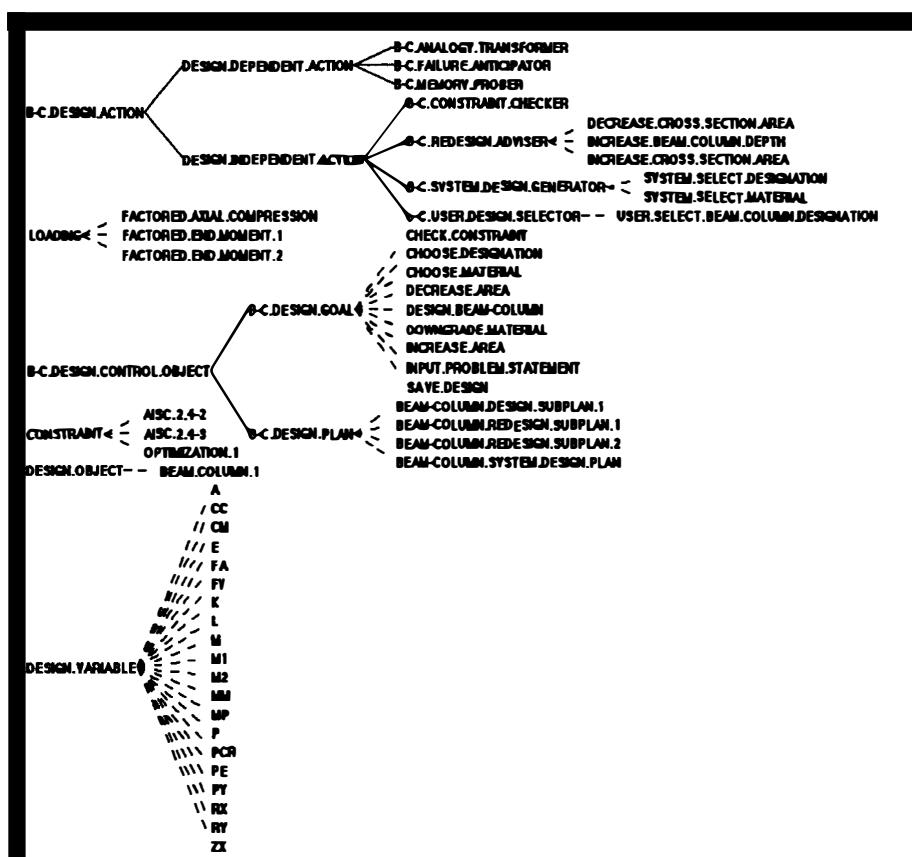


Figure 5. The Beam-Column Knowledge Base

The blackboard reasoning paradigm is built in the KEE frame system utilizing its data-driven capabilities (implemented as active values) and its object-oriented programming facility. Each major blackboard component is also represented as an object with attached methods that define its procedural characteristics. The blackboard activities are sequences of data-driven message passing actions.

KEE's Truth Maintenance System (TMS) is a set of facilities for setting up and maintaining dependencies between facts in the knowledge bases. Figure 6 shows the dependence graph of a variable during design. The dependencies play an important role in supporting backtracking, maintaining design consistency and propagating redesign modifications in DDIS. Although KEEworlds and TMS are used, DDIS does not support parallel design. Multiple partial designs cannot exist simultaneously; DDIS follows only one line of reasoning.

The user interface of DDIS uses the graphic facility provided by both MicroExplorer and KEE. The input, output, and blackboard interfaces are implemented using the windowing system of MicroExplorer and the ActiveImage package of KEE.

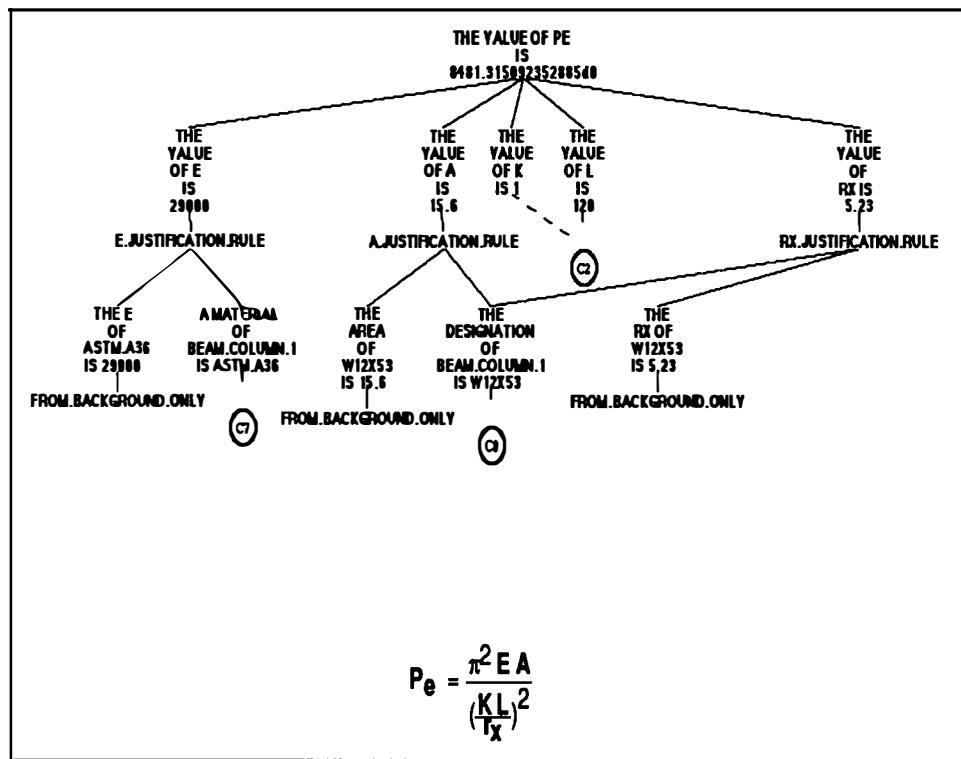


Figure 6. Variable Dependence Graph

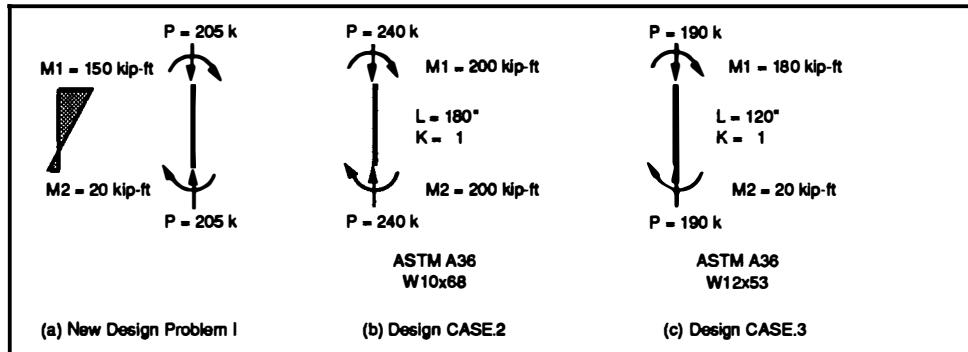


Figure 7. Beam-Column Design Examples

AN EXAMPLE

In order to demonstrate the integrated design approach of DDIS, consider the beam-column design problem shown in Figure 7 (a). The column is only restricted at ends and is subjected to an axial compression load and two end moments. The design task is to find an efficient wide-flange structural section and the steel material for the given load using the AISC plastic design method (AISC, 1980). Figure 5 shows the beam-column design knowledge base in DDIS.

The design object of the problem is BEAM.COLUMN.1. The problem inputs are P (factored axial loading in kips), K (effective length factor), L (length of the beam column in inches), M_1 and M_2 (factored bending moment at two ends in kip-ft). The design constraints are the two interaction equations from AISC and an optimization constraint:

$$\frac{P_y}{1.18 M_p} + \frac{M}{1.18 M_p} \leq 1 \quad \text{AISC Formula (2.4-3)}$$

$$\frac{P_y}{P_{cr}} + \frac{C_m M}{(1 - P/P_e) M_m} \leq 1 \quad \text{AISC Formula (2.4-2)}$$

$$\text{Max} \left[\left(\frac{P_y}{P_{cr}} + \frac{C_m M}{(1 - P/P_e) M_m} \right), \left(\frac{P_y}{1.18 M_p} + \frac{M}{1.18 M_p} \right) \right] \geq 0.9$$

in which

- P factored axial compression load.
- M factored primary bending moment.
- P_{cr} ultimate strength of an axially loaded compression member.
- M_p plastic moment.
- M_m maximum resisting moment in the absence of axial load.
- P_e Euler buckling load.
- C_m column curvature factor.

At the beginning of the run, the case memory has three design cases as shown in Figures 4 and 7 (b),(c). Figure 7 (a) illustrates the new problem and Table 1 summarizes the design session. The rest of the section describes the DDIS problem solving session step by step.

Cycle	Action	Cycle	Action
1	state problem	9	check critical constraint OPT.1
2	input	10	import redesign plan from CASE.2
3	retrieve similar designs	11	redesign beam-column designation
4	rank CASE.2 and CASE.3	12	decrease cross section area
5	reuse case-based plan from CASE.2	13	recheck OPTIMIZATION.1 const.
6	expand goal to system heuristic plan	14 - 15	check remaining constraints
7	select beam-column material	16	end design session
8	reuse designation from CASE.3	17	save design

Table 1. Action Overview of Sample Design Session

Cycle 1—state problem. The design plan BEAM-COLUMN.SYSTEM.DESIGN.PLAN is posted on the blackboard as the global control strategy of the design session. The plan has three sequential goals: INPUT.PROBLEM.STATEMENT, DESIGN.BEAM-COLUMN and SAVE.DESIGN.

Cycle 2—input. The beam-column problem input action is executed. All the values of the variables that need to be provided by the user are entered through a pop-up window.

Cycle 3—retrieve similar designs. Once the new problem inputs have been entered, the global plan moves to its second goal—DESIGN.BEAM.COLUMN, and the RETRIEVE.SIMILAR.CASES action is recommended. The user decides to retrieve similar design CASE.3 and CASE.2 (see Figure 7) because the loading condition of CASE.3 is closer to the new requirement and the design plan of CASE.2 has fewer redesign cycles comparing to CASE.3.

Cycle 4—rank cases. Because of the better design plan in CASE.2, the user assigns 50 and 80 respectively to the solution similarity and plan similarity of CASE.2. On the other hand, the solution similarity and plan similarity of CASE.3 are ranked 78 and 60, respectively because of the very similar loading condition.

Cycle 5—reuse design-dependent plan. The retrieval of CASE.2 and CASE.3 triggers several design-dependent actions at cycle 5. However, the two retrieved designs are not similar enough that their solution can be reused as a whole. Therefore, DDIS recommends the reuse of the design plan of CASE.2.

Cycle 6—expand goal to design-independent plan. The DESIGN.BEAM.COLUMN goal was expanded into CASE.2.DESIGN.PLAN at cycle 5. However, the design-dependent plan is not considered good enough to lead the design session along because of the moderate plan similarity ratings of CASE.2. Therefore, the design-

independent BEAM-COLUMN.DESIGN.SUBPLAN.1 becomes the second subplan of DESIGN.BEAM.COLUMN.

Cycle 7—select material. Using the two plans, two conflicting goals are on the blackboard: CHOOSE.DESIGNATION and CASE.2.DESIGN.PLAN-SELECT.MATERIAL. Since the design-dependent plan has a higher weight (i.e., 20/3 versus 5), DDIS follows the design-dependent plan from CASE.2 and executes the design-independent USE.ASTM.A36.STEEL action. Figure 8 shows the blackboard interface at this cycle. All the executable actions and their rating are presented on the right, and three plans (two design-independent and one design-dependent) and their currently active goals are shown on the left.

Cycle 8—reuse partial solution. The execution of the action satisfies the design-dependent goal and invokes the next goal in CASE.2.DESIGN.PLAN, which is CASE.2.DESIGN.PLAN-SELECT.DESIGNATION. Now the two plans agree on selecting the designation for the beam-column. There are three executable actions on the blackboard for this task, one design-independent action which estimates the required section based on heuristic rules and two design-dependent actions which want to reuse the previously used section in CASE.2 and CASE.3. In this case, DDIS chooses the highest rated action, which is REUSE.W12X53.DESIGNATION.FROM.CASE.3.

DDIS has a very flexible architecture and representation that can use any subset of a past design. Past design solutions can be applied to very similar new designs while previous design plans can be applied to guide designs with less surface similarities. The flexibility of solution reuse is achieved by the solution transformers, which apply part of a previous design solution to satisfy a new goal. The flexibility of plan reuse comes from several places. Previous design steps of a recorded design are decomposed into a main design plan and several redesign plans in the case memory, so that they can be reused separately. The different levels of generalization built into design-dependent plans enable them to be followed very flexibly according to the new situations they encounter.

Furthermore, DDIS can work with multiple design cases simultaneously. For instance, DDIS can follow a design-dependent plan from one case by satisfying one of its goals with a design solution of another case or DDIS can reuse a partial solution from one case to satisfy a goal and reuse a solution from another case to satisfy another goal.

Another level of flexibility comes from the integrated reasoning paradigm of DDIS. Design-dependent actions compete with design-independent actions. Design-dependent and design-independent plans can work together competitively or in a complementary fashion. Case-based design actions are not always available or competent. DDIS has the flexibility to use its design-independent actions instead. It should be noted that the selection between heuristic-based and case-based design approach can be altered from one design step to another since the design is divided into generic actions and subgoals, and the blackboard architecture provides a framework for integrating multiple sources of knowledge to solve a shared problem.

We did not address the similarity problem in this paper. DDIS relies on users to retrieve relevant designs from the case memory and to decide how similar they are to the new design. Our intent is to rely heavily on human interaction during design retrieval at this stage of the research since the focus of the research is now on the integration of design-dependent and design independent reasoning.

The DDIS prototype has been tested in the design of steel beam-columns with a small case memory. A knowledge base for steel base plate design is being implemented to provide a basis for further testing and refinement of the prototype. Also, DDIS will be extended for more case-based design ability.

Control Blackboard				Design Dependent Executable Actions	Rating	
		Click to Start a New Run				
		CONTINUE				
Restart		Continue		STEP		
Cycle	Control Status					
7	Design plan posted					
BEAM-COLUMN.DESIGN.SUBPLAN.1 CASE.2.DESIGN.PLAN BEAM-COLUMN.SYSTEM.DESIGN.PLAN		REUSE.CASE.3 REUSE.W12X53.DESIGNATION.FROM.CASE.3 REUSE.ASTM.A36.MATERIAL.FROM.CASE.3 REUSE.SOLUTION.FROM.CASE.3 REUSE.CASE.2 REUSE.W10X68.DESIGNATION.FROM.CASE.2 REUSE.ASTM.A36.MATERIAL.FROM.CASE.2 REUSE.SOLUTION.FROM.CASE.2		REUSE.CASE.3 REUSE.W12X53.DESIGNATION.FROM.CASE.3 REUSE.ASTM.A36.MATERIAL.FROM.CASE.3 REUSE.SOLUTION.FROM.CASE.3 REUSE.CASE.2 REUSE.W10X68.DESIGNATION.FROM.CASE.2 REUSE.ASTM.A36.MATERIAL.FROM.CASE.2 REUSE.SOLUTION.FROM.CASE.2	16.71 16.71 13.93 11.14 10.71 10.71 8.93 7.14	
CASE.2.DESIGN.PLAN-SYSTEM.SELECT.MATERIAL CHOOSE.DESIGNATION DESIGN.BEAM-COLUMN		USE.ASTM.A36.STEEL.FOR.BEAM.COLUMN USE.ASTM.A588.STEEL.FOR.BEAM.COLUMN USE.ASTM.A572.GR50.STEEL.FOR.BEAM.COLUMN USER.SELECT.BEAM.COLUMN.DESIGNATION.ACTION		USE.ASTM.A36.STEEL.FOR.BEAM.COLUMN USE.ASTM.A588.STEEL.FOR.BEAM.COLUMN USE.ASTM.A572.GR50.STEEL.FOR.BEAM.COLUMN USER.SELECT.BEAM.COLUMN.DESIGNATION.ACTION	17.45 15.51 15.51 5.51	
CASE.2 CASE.3		6) EXPAND DESIGN.BEAM-COLUMN INTO BEAM-COLUMN.DESIGN.SUBPLAN.1 5) REUSE.CASE.2.DESIGN.PLAN.FOR DESIGN.BEAM-COLUMN 4) RANK CASES 3) RETRIEVE SIMILAR CASES 2) PROBLEM INPUT 1) USER SELECT SYSTEM.PLAN		6) EXPAND DESIGN.BEAM-COLUMN INTO BEAM-COLUMN.DESIGN.SUBPLAN.1 5) REUSE.CASE.2.DESIGN.PLAN.FOR DESIGN.BEAM-COLUMN 4) RANK CASES 3) RETRIEVE SIMILAR CASES 2) PROBLEM INPUT 1) USER SELECT SYSTEM.PLAN		
		Executing action: EXPAND.DESIGN.BEAM-COLUMN..INTO.BEAM-COLUMN.DESIGN.SUBPLA N.1		I recommend action: USE.ASTM.A36.STEEL.FOR.BEAM.COLUMN.■		

Figure 8. The Blackboard Interface of DDIS

Acknowledgements. This work has been supported by grants from the National Science Foundation (MSM-8958316) and the Stanford Center for Integrated Facility Engineering.

REFERENCES

- AAAI (1990). *Working Notes of the AAAI-90 Spring Symposium Series—Case-Based Reasoning*, Stanford University, American Association for Artificial Intelligence.
- AAAI (1988). *Proceedings of the AAAI-88 Case-Based Reasoning Workshop*, Minneapolis - St. Paul, Minnesota, American Association for Artificial Intelligence.
- AISC. (1980). *Manual of Steel Construction* (Eighth ed.). American Institute of Steel Construction.
- DARPA (1989). *Proceedings of the DARPA Workshop on Case-Based Reasoning*, Pensacola Beach, Florida, Kluwer Academic.
- Daube, Francois, and Hayes-Roth, Barbara (1988). FIRST: A Case-Based Redesign System in the BB1 Blackboard Architecture, *Proceedings of the AAAI-88 Case-Based Reasoning Workshop*.
- Gentner, D. (1983). Structure-Mapping: A Theoretical Framework for Analogy, *Cognitive Science*, 7(2): 155-170.
- Howard, H. C. (1991). Project-Specific Knowledge Bases in AEC Industry, *Journal of Computing in Civil Engineering*, 5(1): 25-41.
- Howard, H. C., Wang, J., Daube, F., and Rafiq, T. (1989). Applying Design-Dependent Knowledge in Structural Engineering Design, *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 3(2): 111-123.
- Huhns, M. N. and Acosta, R. D. (1987). *Argo: An Analogical Reasoning System for Solving Design Problems*, MCC Technical Report AI/CAD-092-87.
- Kedar-Cabelli, S. (1985). *Purpose-Directed Analogy*, Technical Report ML-TR-1, Department of Computer Science, Rutgers University, New Brunswick, New Jersey .
- Kolodner, J. L (eds) (1988). *Proceedings, DARPA Workshop on Case-Based Reasoning*, Clearwater Beach, Florida, Morgan Kaufmann.
- Mostow, J. and Barley, M. (1987). *Automated Reuse of Design Plans*, Technical Report ML-TR-14, Department of Computer Science, Rutgers University.
- Navinchandra, D. (1988). Case Based Reasoning in CYCLOPS, a Design Problem Solver, in Kolodner, J. (eds), *DARPA Case-Based Reasoning Workshop*.
- Nii, H. P. (1986). Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures (Part One), *AI Magazine*: 38-53.
- Prieditis, A. (eds) (1988). *Analogica*, Morgan Kaufmann, Los Altos, CA.
- Rafiq, T. (1989). *Similarity in Structural Component Case Bases*, unpublished Engineer's degree thesis, Department of Civil Engineering, Stanford University.
- Slade, S. (1988). *Case-Based Reasoning: A Research Paradigm*, Technical Report YALEU/CSD/RR#644, Department of Computer Science, Yale University.
- Wang, J., and Howard, H. C. (1988). Design-Dependent Knowledge for Structural Engineering Design, *Artificial Intelligence in Engineering: Design* (Proceedings of the

- Third International Conference on Applications of Artificial Intelligence in Engineering,
Palo Alto, CA, August 8-11), Computational Mechanics, Southampton.
- Winston, P. H. (1980). Learning and Reasoning by Analogy: The Details, *Technical Report AIM 520*, Artificial Intelligence Laboratory, Massachusetts Institute of Technology.
- Zhao, F. and Maher M. L. (1988). Using Analogical Reasoning to Design Buildings, *Engineering with Computers*, 4(3): 107-119.

Assembly sequence planning using case-based reasoning techniques

P. Pu and M. Reschberger

Department of Computer Science and Engineering
University of Connecticut
Storrs CT 06269-3155 USA

Abstract Future computer-aided-design (CAD) systems will be not only smart, but also efficient in terms of solution strategies as well as time costs. In this paper we present a problem-solving paradigm, namely case-based reasoning (CBR), and show how it solves some constraint-oriented design problems efficiently. CBR solves a new problem by retrieving from its case library a solution which has solved a similar problem in the past and then adapting the solution to the new problem. The efficiency of such a system relies on the completeness and compactness (small size) of such a case library. These two characteristics of a case library in turn rely on an indexing scheme for the set of cases in the library. We illustrate here how these issues can be solved and how to design a case-based reasoning system for such design problems. We present our theory as it is applied to the area of assembly sequence planning.

INTRODUCTION

Case-based reasoning (Kolodner, 1988, Hammond, 1989a, and Riesbeck and Schank, 1989) provides an efficient strategy for solving problems, since systems designed by such techniques remember old solutions and apply them to new problems whenever circumstances permit. This paradigm also permits us to apply expertise in a domain where rules are hard to define. Problem solving using CBR techniques usually involves retrieving relevant previous solutions, adapting and combining previous solutions to solve the new problem, and recording failures so that they can be avoided in the future. The efficiency of such a system relies on the completeness and compactness of such a case library. These two characteristics of a case library in turn rely on an indexing scheme for the set of cases in the library.

Many design problems are computationally expensive, thus provide a good domain

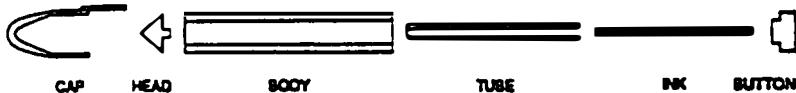


Figure 1: The ball-point pen as an example of the ASP problems

to apply CBR techniques. In this paper, we examine the domain of assembly sequence planning (ASP) (Sanderson and Homem de Mello, 1987, De Fazio and Whitney, 1987, Homem de Mello, 1989, and Sanderson *et al.*, 1990) and analyze how CBR techniques can be applied to such a domain. Although it is often called planning, ASP problems are a type of constraint-oriented design problems in a sense that a goal is to be achieved while a set of constraints are to be satisfied. Illustrated in Figure 1, the ball-point pen, is an example of ASP problems. The goal is to provide a sequence to assemble two parts at a time without backtracking so that the final configuration (a set of constraints) of the product is satisfied. One solution to this problem is to place the ink inside the tube, place the ink-tube inside the body, put on the head, put on the button, and finally put on the cap. A common mistake is to overlook the spatial relationship among the ink, tube, body, head, and button and suggest to put on the head and button on to the body before the tube and ink are placed inside the body.

An initial examination of the nature of ASP problems made us realize that the goal-oriented planning paradigm (as in CHEF, Hammond, 1989b, for example) found in the current CBR literature does not satisfactorily provide a model for solving design problems. CHEF takes a set of goals (e.g., "give me a stir-fry with chicken and broccoli") as input and generates recipes that satisfy these goals. ASP requires a set of constraints to be given as input. While the goals in CHEF can be readily used to index the case library to facilitate the search for candidate cases, it is much less clear how the given constraints (being the only input to the ASP problems) can be used to guide the search. In a later section we discuss this observation in more detail. Only when we examine the constraints more closely, we realize that they give rise to many implicit goals. So somehow each problem must be decomposed into subproblems so that a CBR system can select from the case library the appropriate cases for each of the subproblems.

This paper will examine four key issues related to the design of such a CBR system which solves constraint-oriented design problems: (a) how to successively decompose a given problem, (b) how to represent the cases in the case library, (c) how to control the search space, and (d) how to select the best case. These discussions are proceeded by a section where we define what is an ASP problem. Our CBR system, CAB-Assembler, is fully implemented. Thus we describe the other components of the system after we would have presented the theory. Several example devices which CAB-Assembler successfully solved are discussed in the performance section. Some run-time measures are also shown there to illustrate the performance of our system. Finally we conclude the paper by recapitulating the main points of this work and pointing out some important issues which are subject to further investigation.

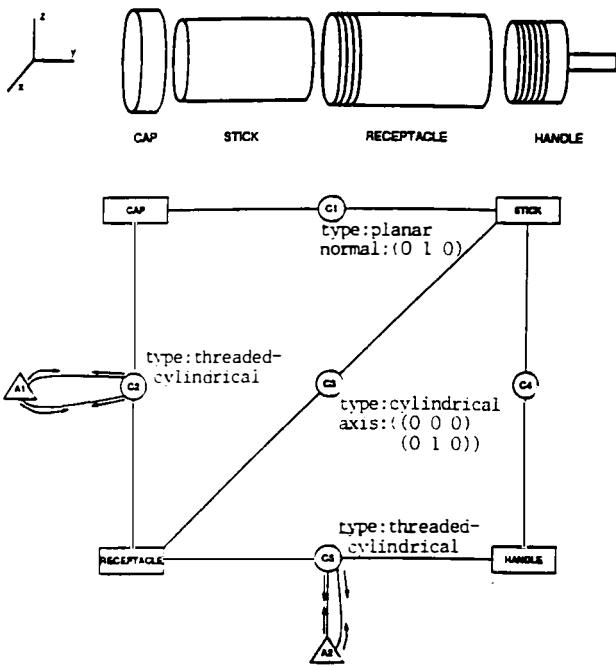


Figure 2: a) The receptacle device b) Its relational model

PROBLEM DESCRIPTION

The problem of finding a sequence with which the parts of a product can be assembled is called the assembly sequence planning problem. One of the valid sequences for the receptacle device shown in Figure 2a is as follows (Sanderson *et al*, 1990):

```
((cap) (handle) (receptacle) (stick)}    --->
{ (cap receptacle) (handle) (stick)}    --->
{ (cap receptacle stick) (handle)}    --->
{ (cap receptacle stick handle)}
```

Each union of two pairs of parentheses defines an assembly operation. The input to an ASP system is always the set of objects or parts of a product in a totally disassembled state. Since the product has been successfully designed, there is a unique configuration with which the parts are arranged. This configuration, which is also part of the input, is called the relational model by Bourjault (1984) and defines a set of constraints on the set of parts. The relational model of the receptacle device is shown in Figure 2b. The labels on each arch of the relational model (or graph) are specifications of constraints among the parts. For example, it is specified that the cap is to be connected with the

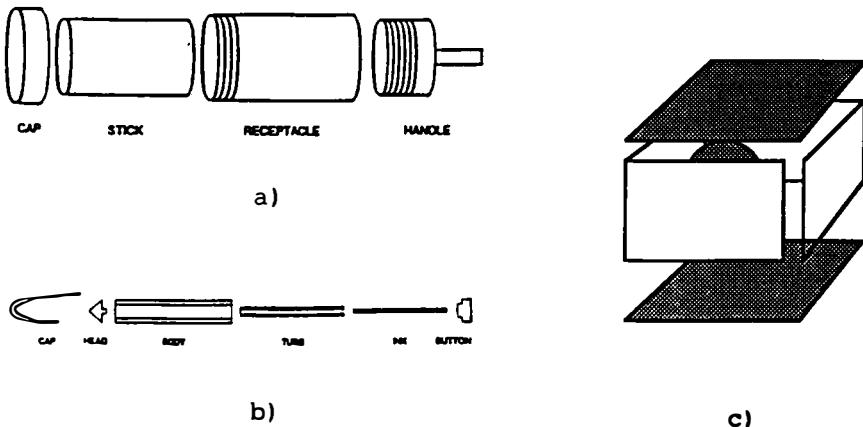


Figure 3: a) receptacle b) ball-point pen c) box

receptacle via a threaded-cylindrical contact (screwing on), the stick to be connected with the receptacle via a cylindrical contact (putting it inside), etc. But the relation model does not define the assembly order: it does not specify that the stick must be put inside the receptacle before both cap and handle are screwed into the receptacle.

In a product, a *part* is the smallest non-decomposable entity. When two parts are joined, they form a *subassembly*. A part can also be joined with a subassembly to form another subassembly. We use the word *piece* to refer to either a part or a subassembly. An *assembly plan*, the output, is an ordered sequence of (assemble A B) actions where A and B are both pieces. Notice that one does not have to pay attention to how A and B are put together, for instance whether they are joined by a planar, or threaded-cylindrical contact. These relations are called contact and attachment relations by Homem de Mello (1989). This information is provided by the designer.

A CBR SYSTEM FOR ASP

To design an efficient CBR system, we have identified two important issues: a complete and compact case library and an effective indexing scheme. By a complete case library, we refer to the fact that after a small training period the CBR system should be able to solve most of the problems in the domain without having to add more cases in the library. By compactness, we mean a small and manageable case library. By an effective indexing scheme, we mean the ability to retrieve similar cases and the successfulness of applying these cases to new problems. The intuition of our solution to these issues is that we keep a library of cases that correspond to *subolutions* so that a whole problem is solved by combining subsolutions, and we use constraints, which are input to the ASP process, as indices to organize cases. The reason why the

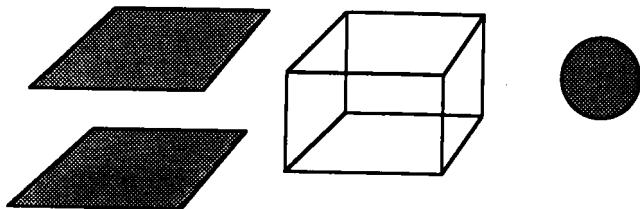


Figure 4: A partially assembled box

library designed this way is small and complete is because in many engineering domains, problems are composed by a set of primitives, or called building blocks. For instance, in the electrical circuit domain, systems consist of AND, OR, and NAND gates at the gate level. In certain mechanical domains, building blocks are gears, ratchets, levers, and linkages. In the ASP domain, there is on the order of 10 basic ways of combining parts into subassemblies.

In the next few sections, we explain the solution to these issues in detail. Several examples are used throughout these sections for illustrational purposes. We describe these examples first as follows:

The three devices shown in Figure 3 are different devices at first glance. Under closer examination, we realize that all three devices have a common enclosure characteristic: some parts in these devices eventually form a spatial enclosure; some parts of these devices must be placed inside the enclosure before it is completely formed. But how does one recognize that the enclosure characteristic is common in all three devices so that the solution for the receptacle can be applied to the ball-point pen and box for example? For assembling the box, somehow the system has to first put the wall pieces together before it can use the receptacle solution to solve the enclosure characteristic. How can that be done?

Successive reminding

Suppose we know how to solve the receptacle device and want to use this experience to solve the box example. Initially it is hard to see the enclosure with the given seven pieces of the box. But if four walls were assembled together, we would begin to get reminded of the receptacle. Thus such an enclosure characteristic may only be apparent to us sometimes in a later stage of our planning process. The successive reminding idea is to make the system apply those solutions that remind it of the current stage of the assembly process. For this box example, the system is to be reminded of a case which just puts two pieces together in the beginning. Hopefully the system applies this solution until the wall pieces and the bottom of the box have been connected. Then the system is to be reminded of the receptacle case in order to solve the enclosure. Figure 4 shows the stage right before the receptacle case is applied.

In the next section, we will discuss our case representation. We illustrate how a case (called the fundamental case) which just assembles two pieces together and the

<i>feature</i>	<i>weight</i>	<i>number of successes</i>	<i>number of failures</i>
receptacle	6	1	0
cap	6	1	0
stick	6	1	0
handle	6	1	0
Solution:		$\{(receptacle\ handle)(stick)(cover)\}$ $\{(receptacle\ handle\ stick)(cover)\}$ $\{(receptacle\ handle\ stick\ cover)\}$	

TABLE 1: The case representation of the receptacle device.

receptacle are kept in the case library. To solve the box example, our system uses the fundamental case four times to put the walls together and the receptacle case once to place the ball inside the box.

Case library

Almost no two devices have exactly the same compositions. Thus our case library captures a set of subsolutions (building blocks), such as the enclosure case and the fundamental case, rather than complete device assemblies. The case library accommodates all ASP problems we studied with a small set of cases. Each case corresponds to one step in assembling a subassembly. It has the following representation:

- the abstract problem description (= APD, generalization) which is a set of features, where each feature defines a particular characteristic of a problem
- feature weights
- success and failure history
- the plan

Table 1 illustrates the case representation of the receptacle device (Figure 2a). In the beginning, all #succ and #fail fields are initialized to 1 and 0 respectively. Each piece involved in the case is a mandatory feature and has the highest weight, 6. Later on we discuss how other optional features can be included with lower weights. Although Table 1 presents the ASP solution of an entire device, it can be used to solve the ball-point pen (Figure 1) partially. More precisely, it solves the enclosure problem in the ball-point pen. In fact, most cases represent solutions to subproblems. Shown in Table 2 is a solution used to partially assemble many devices. For our own convenience, we call such a case the pan-with-cover case, but it can be used to assemble any two pieces if they are connected via the surface-surface contact.

To facilitate matching, a set of features (or called a feature vector) is associated with each case as an index and serves as the generalization of a problem. In other words, this set of features represent the characteristics of not only this particular case, but

<i>feature</i>	<i>weight</i>	<i>number of successes</i>	<i>number of failures</i>
pan	6	1	0
cover	6	1	0
surf-surf pan cover	6	1	0
Solution:	{(pan cover)}		

TABLE 2: The case representation of a pan-cover device

also a class of similar cases. We will elaborate on the indexing issue further in the next section. Each feature in the feature vector has the following additional information: feature weight and success and failure histories. The weight assignment gives preference to the more important features. Success and failure histories help the system select the best case in the future by recording experimental data. For each case, a plan or solution is attached so that once this case is chosen for a new problem, the solution is applied.

Since the library is kept small, it is ordered *sequentially* instead of in a more sophisticated way, such as MOPs (Schank, 1982 and Riesbeck and Schank, 1989).

Controlling the search

The problem introduced by the successive reminding approach is a severe search problem which is illustrated with the following scenario.

The input to the system is a device in a totally disassembled state. The challenge is to identify in this totally disassembled device which characteristic problem in the case library to apply and when. This defines a large search space since there are usually a lot more parts involved in a disassembled device than those involved in a characteristic case. Doing the problem mathematically, the space is exponential. When the box example is initially matched with the receptacle case, there are 840 different ways of matching these two problems. The box initially has seven pieces and the receptacle case has four. Taking a permutation of four out of seven each time, there are $7 \times 6 \times 5 \times 4 = 840$ distinctive ways. This is computationally too expensive!

With closer examination of the problem, we came up with a much more efficient way for this task. Each case in the library and the ASP problem at hand are defined with a set of features. If we just include the number of pieces as mandatory elements in the feature field, the search space is exponential. However, once we include geometrical constraints as mandatory elements in the feature field, we came down to 187 possible combinations for this example. With the inclusion of spatial constraints, we came further down to 10 possible combinations. By including geometrical and spatial constraints to the feature field, we have avoided comparing incomparable cases.

One observation is necessary here. The geometrical and spatial constraints are merely relationships between the parts of a device, which are presumed to be available in a data file. Thus these constraints do not require any overhead on the part of the users.

Selecting the best case

There are still 10 ways to fit the box into the receptacle case. If the case library contains more cases, the search space is still quite large. Determining the best case(s) is important to any CBR system. Many systems use heuristic metrics (Ashley and Rissland, 1988a, Ashley and Rissland, 1988b, and Kolodner, 1989) to determine the similarity between two cases. Often such a metric is domain-dependent and requires a set of desiderata to be defined first. Our approach is quantitative and calculates a priority in terms of usefulness of each feature using information from past experiences.

For each feature, we calculate its priority as follows:

$$\text{priority}(F) = \begin{cases} 0 & \text{if mandatory features do not match or} \\ & \# \text{successes} < \# \text{failures} \\ \frac{\# \text{successes}}{\# \text{successes} + \# \text{failures}} * (\# \text{successes} - \# \text{failures}) & \text{otherwise} \end{cases}$$

The priority of a case is the sum of the product of each feature priority and its weight.

The formula for each feature is justified by the following considerations:

1. **Match** If the feature of a case does not match with the feature of the problem at hand, the priority is neutral (=zero);
2. **Success Ratio** Measure the number of successes over the total number of times this feature is selected;
3. **Exaggeration** If a case is very successful, the chance of selecting it should be increased (encouragement); if a case has had many failures, the chance of selecting it should be decreased (punishment).
4. **Weight** A feature which is supposed to be more important (higher weight) should have a higher chance of influencing the case priority than one that is of less importance. This is called "salient-feature preference" (Kolodner, 1989).

The case base is initialized to 1 for all #successes fields and 0 for all #failure fields. By considering the success- and failure-history of a feature and therefore of a case, the reasoner selects not only the most similar, but also the most useful case for a problem (Kolodner, 1989).

HYPO (Ashley and Rissland, 1988a and Ashley and Rissland, 1988b) uses previous experience in the form of weights (called "factors" or "dimensions") in order to assess similarity. However that method is combined with information about the reasoner's task. Also the reasoner uses the weights only partially (notion of "least commitment approach"). That method is therefore not suitable for the ASP domain.

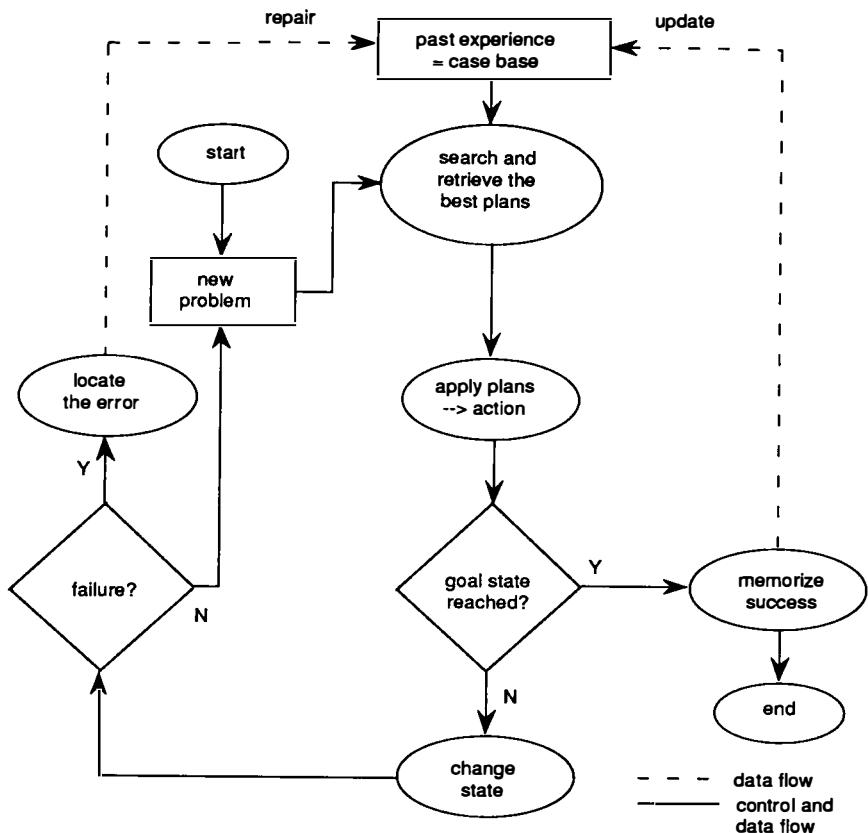


Figure 5: System architecture for CAB-Assembler

AN OVERVIEW OF CAB-ASSEMBLER

CAB-Assembler is an implementation of the ideas presented here and has various other components besides issues discussed earlier. A flow chart of how the system works is shown in Figure 5. We briefly describe the other components of the system as follows:

Input The input to the search and case selection mechanism of the case-based system are

- the current state of a device (totally or partially disassembled), described in terms of predicate relations;
- the case base

Feature Weights The features should not only be divided into important (mandatory) and unimportant (optional) but they should also be further subdivided into degrees of importance, also called weights. The overall range of the weights should depend on how exact the differences among the features can be determined. This is very domain specific. For the ASP domain we use a range between 1 and 6.

Plan In the ASP domain an action and also a group of actions can be represented by using “assemble” and “not assemble” as the only two operators. Some examples for the internal representation of plans:

- (1) (assemble 1 2)
- (2) (not assemble 2 3)
- (3) (not assemble 1 3)

where 1, 2, and 3 are piece variables. “Not assemble” is important in avoiding failures. This operation is often included in a plan after a failure has occurred.

Success- and Failure-History The number of successes tells how often this case has contributed to successfully reaching a goal state when this specific feature matched with the new problem. However, if the goal state is not reached and this case with its “bad” plan is declared as the reason for failure, all features that matched are increased by one for the #failure field.

Combining Plans The matching algorithm returns the results of the case examinations. These results are sorted in a descending order according to case priorities. The best plan is combined with the plan of the second closest case, the plan of the third closest case, and so on - until a specific action is found.

Failure Detection and Error-Locating Failure-driven learning depends on the detection of the failure and on finding the reason for this failure. It is admitted as a very difficult task (Kolodner, 1988, Hammond, 1989b, and Riesbeck and Schank, 1989). Using a heuristic approach is a common way of locating the error. The implemented system solves this problem by simply asking the user whether the proposed action is acceptable.

Similar to failure detection, locating the reason for failure is only important in the training stage. Therefore it is also solved through user interaction.

Once the reason for failure is located, we know which action has caused the failure. Two methods are used in order to avoid repeating the same failure again: memorizing the failure and creating a new case for solving this problem. Whenever failure appears, both methods are applied.

Learning by Success When people solve a problem successfully, they tend to be encouraged by such an experience. Sometimes later the same or a similar problem appears they tend to use a plan that was more successful than one that has not been.

In order to prefer the selection of a case, and with the case automatically its plan which was very successful in the past, success is memorized in a case. We discussed this issue already in the paragraph on Success- and Failure-History.

CAB-ASSEMBLER'S PERFORMANCES

Spatial reasoning has been particularly difficult in the ASP domain (Homem de Mello, 1989). However many spatial problems, although differ on the surface, share many similar characteristics. Thus our observation is that if spatial problems are difficult, not every one of them has to be solved from scratch. We go through two spatial examples in this section, step by step, to show how our system works and discuss the performance in terms of time costs. The first example, the receptacle device, is meant to show how a case base is initially established. It encounters failures and is corrected by users via an interactive mode. Once a case base is established, we show that it offers flexibility in terms of solving similar problems. Furthermore, our research results show that it is best to train a case base with a more complicated example in order to accumulate more experiences. The case development for the ball-point pen is quite long, thus is omitted here. See Reschberger (1990) for detailed transcriptions. But we will show how the case base trained by the ball-point pen can be used to solve the receptacle case without failure (example 2).

The receptacle example represents a typical enclosure problem: the stick cannot be put into its place before the other three parts already form a subassembly.

The scenario:

Step 1: input: {cap handle receptacle stick}, "cb.fund". The device is completely disassembled. The reasoner decides to assemble the parts cap and receptacle.

Step 2: input: {((cap receptacle) handle stick)}, "cb.fund". The reasoner decides to connect the subassembly "(cap receptacle)" with the part "handle".

Step 3: input: {((handle cap receptacle) stick)}, "cb.fund". The action "connect stick (handle cap receptacle)" cannot be performed. Failure occurs.

The action which connected the part "handle" with the subassembly "(cap receptacle)" is blamed for the failure. The reasoner creates a new case which contains three pieces which refer to "handle", "(cap receptacle)", and "stick", respectively. The plan of the new case should in the future prevent the reasoner from connecting the pieces "handle" and "(cap receptacle)" again. This prevention should not only work for the same problem, but also for any other situations similar to this one.

The second part of the following transcript shows that the improved case base avoids doing the same mistake, and solves this example successfully:

*** start of transcript ***

```
<c1> (solve-problem "prob.rec.vert.si" "cb.fund")
```

```
i am in training-mode
```

Do you accept to connect
piece (CAP) with
piece (RECEPTACLE) ?
(y/n) --> y

Do you accept to connect
piece (HANDLE) with
piece (CAP RECEPTACLE) ?
(y/n) --> y

Do you accept to connect
piece (STICK) with
piece (HANDLE CAP RECEPTACLE) ?
(y/n) --> n

Was connecting (CAP) with (RECEPTACLE) the error?
(y/n) ---> n

Was connecting (HANDLE) with (CAP RECEPTACLE) the error?
(y/n) ---> y

remembering SUCCESS for case FUNDAMENTAL-CASE
bad case: FUNDAMENTAL-CASE
Should (STICK) be added to the new case?
(y/n) ---> y

Should (HANDLE) be kept in the case?
(y/n) ---> y

Should (CAP RECEPTACLE) be kept in the case?
(y/n) ---> y

How would you like to name the new case?
name ---> rec-1

remembering FAILURE for case FUNDAMENTAL-CASE
Name for new case base ---> cb.rec.1

storing improved case base in file CB.REC.1
end of problem solving

NIL

<cl> (solve-problem "prob.rec.vert.s1" "cb.rec.1")

i am in training-mode
Do you accept to connect
piece (RECEPTACLE) with
piece (CAP) ?

(y/n) --> y

Do you accept to connect
piece (RECEPTACLE CAP) with
piece (STICK) ?
(y/n) --> y

Do you accept to connect
piece (HANDLE) with
piece (RECEPTACLE CAP STICK) ?
(y/n) --> y

remembering SUCCESS for case REC-1
remembering SUCCESS for case REC-1
remembering SUCCESS for case FUNDAMENTAL-CASE
Name for new case base ---> cb.rec.1.2

storing improved case base in file CB.REC.1.2
end of problem solving

NIL

<c1>

*** end of transcript ***

In the following transcript, prob.rec.vert.s1 is the receptacle problem given to the system. The case base is cb.pen.4 which is obtained by training the system with the ball-point pen example.

*** start of transcript ***

<c1> (solve-problem "prob.rec.vert.s1" "cb.pen.4")

i am in training-mode
Do you accept to connect
piece (RECEPTACLE) with
piece (HANDLE) ?
(y/n) --> y

Do you accept to connect
piece (CAP) with
piece (STICK) ?
(y/n) --> y

Do you accept to connect
piece (RECEPTACLE HANDLE) with
piece (CAP STICK) ?
(y/n) --> y

# pieces left to assemble	cb.fund	cb.rec.1	cb.pen.4
4	8	19	54
3	5	10	17
2	3 f	4	10
1		≈2	≈2
Σ	-	35	83

TABLE 3: Run-time behavior: receptacle example

# pieces left to assemble	cb.fund	cb.pen.1	cb.pen.2	cb.pen.3	cb.pen.4	cb.aft.5
6	17	24	40	56	108	91
5	12	18	28	37	47	48
4	9	13	18	22	27	25
3	6	8 f	9	9	13	13
2	4 f		4 f	5 f	5	5
1					≈2	≈2
Σ	-	-	-	-	202	184

TABLE 4: Run-time behavior: ball-point pen example

```

remembering SUCCESS for case PEN-4
remembering SUCCESS for case PEN-4
Name for new case base ---> cb.xy

```

```

storing improved case base in file CB.XY
end of problem solving
NIL
<cl>

```

```
*** end of transcript ***
```

Table 3 and 4 illustrate the run-time to train the receptacle and ball-point pen case base. Each table shows the development of a case base from left to right. If an action is not valid (failure), the label "f" is used in that table entry. Each entry is the amount of time (in seconds) the system takes to come up with an assembly operation.

Table 2 shows that it takes one failure to create a case base capable of solving the receptacle problem. Column four (cb.pen.4) shows that the case library created for solving the ball-point pen (Figure 3b) can be used to solve the receptacle device. The significance here is that our system can be trained to solve a class of problems. In this case, both the receptacle device and the ball-point pen are part of the class of enclosure

problems. Furthermore, if the system is trained to solve a more difficult problem in the same class, it can solve a simpler one without much failure.

Several observations from the experiment are summarized here:

- The more pieces an assembly has, the longer it takes to calculate the next action.
- The more cases a case base has, the longer is the run-time.
- The number of pieces effects the run-time more than the number of cases does.

Other problems our system solved include the interlocking blocks and assembly from industry which has 11 parts. Both of them have been studied by researchers in the ASP field. The run-time cost of the AFI problem is 3317 seconds (or 55 minutes), a rather low figure which we have not seen in other ASP systems.

RELATED WORKS

Traditional Assembly Sequence Planning Methods The ideas on ASP problems described by Bourjault (1984), Homem de Mello (1989), and Sanderson (1987, 1990) have influenced our work in terms of representing the problem and understanding the nature of the problem. However, these methods are highly computationally intensive. Knowledge-based systems normally use heuristics to maximize the efficiency. De Fazio and Whitney (1987) proposed a rule-based expert system to solve ASP problems. Some ASP problems, such as the interlocking blocks, pose constraints that are non-linear and cannot be solved by rules. Our system takes the advantage of the observation that "if ASP problems are generally computationally hard, not every one of them has to be solved from scratch." Thus the first-principled method by Homem de Mello (1989), for instance, can be viewed as the resort we come to when our system fails to retrieve a similar problem. However, most of the time the system can solve ASP problems by using past experiences efficiently, as we have shown in the performance tables.

Case-Based Systems The major finding of this work is that current techniques, such as CHEF (Hammond, 1989a), in CBR address goal-oriented planning, rather than constraint-oriented planning which is more common in the engineering domain. Each case in the case library captures the solution of an entire problem, instead of the characteristic subproblems which comprise the whole. Thus it does not provide a model for ASP problems or other constrain-oriented planning problems. Those CBR systems that did address engineering problem-solving issues (Barletta and Mark, 1988 and Navinchandra, 1988) tend to follow the goal-oriented paradigm also.

CONCLUSION AND FUTURE WORK

The success of our approach relies on two factors: it solves new problems by retrieving subsolutions instead of complete solutions and it keeps a small case library of the

primitives of the problem domain. To achieve the first, we must perform what is called successive reminding. Furthermore, an effective indexing scheme must be used so that at each stage of the assembly process, the system retrieves the right subsolution. To achieve the second, we observe that in the ASP and many other engineering domains, the underlying set of primitives which serve as building blocks for the whole class of problems is a small set. In the ASP case, there are on the order of ten primitive problems. Designing a CBR system with this principle in mind avoids a large case library, and thus achieves efficiency.

We have studied heavily the enclosure problem in this research, since it is the most time-consuming one if first-principled approaches are used (Homem de Mello, 1989), which involve the use of path-planning algorithms. Our future work will include the further study of other characteristic problems. In Homem de Mello's work, there is a classification of what he called preference relations, of which enclosure is one (called the ordering preference). From our experience with the enclosure problem, it is a matter of training a case base with the capability of solving certain preference relations.

The formula for calculating the best case or cases is subject to further study. For all the cases we tried with our current implementation, there are no undesirable behaviors found in the system. But this is not a satisfactory answer on theoretical bases. We are currently looking at mathematical treatments of distance functions in treating similarities (in our case, relevancy). One possible solution is to make the formula calculating how irrelevant a case is to a problem. Thus the best case is the one that is *least* irrelevant to the problem. If such a least distance formula can be shown to satisfy the triangular inequality (Conte and de Boor, 1980), then it is guaranteed that the system will converge. That is, the more experiments we perform with the system, it is better in terms of dealing with the selection of best cases.

References

- [Ashley, K.D. and Rissland, E.L. 1988] Compare and contrast, a test of expertise. In J.L. Kolodner, editor, *Proceedings of the Case-Based Reasoning Workshop 1988*, pages 31–36.
- [Ashley, K.D. and Rissland, E.L. 1988] Weighting on weighting: a symbolic least commitment approach. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 239–244.
- [Barletta, R. and Mark, W. 1988] Explanation-based indexing of cases. In J. Kolodner, editor, *Proceedings of Case-Based Reasoning Workshop*, Morgan Kaufmann.
- [Bourjault, A. 1984] *Contribution à une Approche Methodologique de L'Assemblage Automatisé: Elaboration Automatique des Séquences Opératoires*. Technical Report (Thèse d'Etat), Université de Franche-Comté, Besançon, France.
- [Conte and de Boor 1980] *Elementary Numerical Analysis: An Algorithmic Approach*. McGraw Hill Book Company, third edition.
- [Homem de Mello 1989] *Task Sequence Planning for Robotic Assembly*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pa.

- [De Fazio and Whitney 1987] Simplified generation of all mechanical assembly sequences. *IEEE Journal of Robotics and Automation*, 705–708, December.
- [Hammond, K.J. 1989] *Proceedings of Second Case-Based Reasoning Workshop*. Morgan Kaufmann.
- [Hammond, K.J. 1989] *Case-based Planning: Viewing Planning as a Memory Task*. Academic Press, San Diego.
- [Kolodner, J.L. 1988] *Proceedings of the First Case-Based Reasoning Workshop*. Morgan Kaufmann.
- [Kolodner, J.L. 1989] Judging which is the “best” case for a case-based reasoner. In *Proceedings of the Case-Based Reasoning Workshop*, pages 77–81.
- [Navinchandra, D. 1988] Case-based reasoning in cyclops, a design problem solver. In J. Kolodner, editor, *Proceedings of Case-Based Reasoning Workshop*, Morgan Kaufmann.
- [Reschberger, M. 1990] *Case-Based Reasoning for Assembly Sequence Planning*. Technical Report CSE-TR-90-25, University of Connecticut, Department of Computer Science and Engineering.
- [Riesbeck, R and Schank, R. 1989] *Inside Case-based Reasoning*. Lawrence Erlbaum.
- [Sanderson, A.C. and Homem de Mello, L.S. 1987] *Task Planning and Control Synthesis for Flexible Assembly Systems*, pages 331–353. Springer-Verlag.
- [Sanderson, A.C., Homem de Mello, L.Z. and Zhang, H. 1990] Assembly sequence planning. *AI Magazine*, 11(1):62–81.
- [Schank, R. 1982] *Dynamic Memory: A Theory of Learning in Computers and People*. Cambridge University Press.

Empowering designers with integrated design environments

G. Fischer[†] and K. Nakakoji[‡]

[†]Department of Computer Science and Institute of Cognitive Science
University of Colorado
Boulder CO 80309 USA

[‡]Software Engineering Laboratory
Software Research Associates, Inc
1113 Spruce Street, Boulder CO 80302 USA

Abstract. Designers deal with ill-defined and wicked problems characterized by fluctuating and conflicting requirements. Traditional design methodologies based on the separation between problem setting (analysis) and problem solving (synthesis) are inadequate to solve these problems. These types of problems require a cooperative problem-solving approach empowering designers with integrated, domain-oriented, knowledge-based design environments. In this paper, we describe the motivation for this approach and introduce an architecture for such design environments. We focus on the integration of specification, construction, and a catalog of prestored design objects in those environments for an illustration of how such integrated environments empower human designers. The system component described in detail (called CATALOGEXPLORER) assists designers in locating examples in the catalog that are relevant to the task at hand as partially articulated by the current specification and construction, thereby relieving users of the task of forming queries for retrieval.

INTRODUCTION

Design is an ill-defined (Simon, 1973) or wicked (Rittel, 1984) problem with fluctuating and conflicting requirements. Early design methods, based on directionality, causality, and separation of analysis from synthesis, are inadequate to solve such problems (Cross, 1984).

The research effort discussed in this paper is based on the assumption that these design problems are best solved by supporting a cooperative problem-solving approach between humans and integrated, domain-oriented, knowledge-based design environments (Fischer, 1990). Combining knowledge-based systems and innovative human-computer communication techniques empowers humans to produce “better” products by augmenting their intellectual capabilities and productivity rather than simply by using an automated system that may not be entirely appropriate (Stefik, 1986).

Our approach is not to build another expert system. Expert systems require a rather complete understanding of a problem to start with — an assumption that does not hold for ill-defined problems. In order to produce a set of rules for an expert system, the relevant

factors and the background knowledge need to be identified. However, we cannot fully articulate this information. What has been made explicit always sets a limit, and there exists the potential of breakdowns that call for moving beyond this limit (Winograd and Flores, 1986).

In this paper, we will use the domain of architectural design of kitchen floor plans as an “object-to-think-with” for purposes of illustration. The simplicity of the domain helps in concentrating on the essential issues of our approach without being distracted by understanding the semantics of the domain itself. We first discuss issues with design environments and emphasize the importance of domain orientation and integration of those environments. Then we describe integrated, domain-oriented, knowledge-based design environments based on the multifaceted architecture as a theoretical framework. Next, an innovative system component, CATALOGEXPLORER, is described as an illustration of how such an integrated environment empowers human designers. The system integrates specification, construction, and a catalog of prestored design objects. The synergy of integration enables the system to retrieve design objects that are relevant to the task at hand as articulated by a partial specification and construction, thereby relieving users of the task of forming queries for retrieval. We discuss related work and conclude with a discussion of achievements, limitations, and future directions.

PROBLEMS

Integration of problem setting and problem solving

Integration of problem setting and problem solving is indispensable (Schoen, 1983). As Simon (1981) mentioned, complex designs are implemented over a long period of time and are continually modified during the whole design process. Simon stated that they have much in common with painting in oil, where current goals lead to new applications of paint, while the gradually changing pattern suggests new goals. One cannot gather information meaningfully unless one has understood the problem, and one cannot understand the problem without information about it. Professional practitioners have at least as much to do with defining the problem as with solving the problem (Rittel, 1984).

An empirical study by our research group, which analyzed *human-human cooperative problem solving* between customers and sales agents in a large hardware store (Reeves, 1990), provided ample evidence that in many cases humans are initially unable to articulate complete requirements for ill-defined problems. Humans start from a partial specification and refine it incrementally, based on the feedback they get from their environment.

The integration of problem setting (analysis) and problem solving (synthesis) is not supported in first-generation design methodologies or in traditional approaches of software design (Sheil, 1983). Automated design methodologies fail because they assume that complete requirement specification can be established before starting design.

Retrieval of information relevant to the task at hand

In supporting integration of problem setting and problem solving in design environments, supporting retrieval of information relevant to the task at hand is crucial. Every step made by a designer toward a solution determines a new space of related information, which cannot be determined a priori due to its very nature. Integrated design environments are based on high-functionality systems (Lemke, 1989) containing a large number of design objects.

High-functionality systems increase the likelihood that an object exists that is close to what is needed — but without adequate system support it is difficult to locate and understand the objects relevant to the task at hand (Figure 1) (Nielsen and Richards, 1989; Fischer and Grgenohn, 1990).

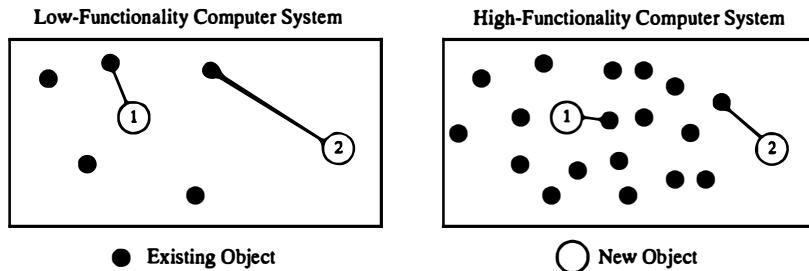


Figure 1: Trade-off Between Accessibility and Usefulness

It is easier to locate existing objects in a low-functionality computer system, but the potential for finding an object closer to what is needed is higher in a high-functionality system. The length of lines represents the distance between desired objects and existing objects.

The task at hand cannot be articulated at the beginning of a design, leading to the inapplicability of conventional information retrieval techniques (Fischer, Henninger, and Redmiles, 1991). In a conventional *query-based search*, a highly specific query has to be formulated. If users can articulate what they need, a query-based search takes away a lot of the burden of locating promising objects (Henninger, 1990).

In *navigational access* provided by a browsing mechanism, users tend to get lost while wandering around in the space looking for some target information if the space is large and the structure is complex (Halasz, 1988). Navigational access requires that the information space has a fairly rigid and predetermined structure, making it impossible to tailor the structure according to the task at hand. Browsing mechanisms become useful once the space is narrowed by identifying a small set of relevant information.

Design environments need additional other mechanisms (as discussed in this paper) that can identify small sets of objects relevant to the task at hand. Users must be able to incrementally articulate the task at hand. The information provided in response to these problem-solving activities based on partial specifications and constructions must assist users in refining the definition of their problem.

Domain orientation

To reduce the great transformation distance between a design substrate and an application domain (Hutchins, Hollan, and Norman, 1986), designers should perceive design as communication with an application domain. The computer should become invisible by supporting *human problem-domain communication*, not just human-computer communication (Fischer and Lemke, 1988). Human problem-domain communication provides a new level of quality in human-computer communication by building the important abstract operations and objects in a given area directly into a computer-supported environment. Such an environment allows designers to design artifacts from application-oriented building blocks of various levels of abstractions, according to the principles of the domain.

Integrated design environments

Design should be an ongoing process of cycles of specification, construction, evaluation, and reuse in the working context. At each stage in the design process, the partial design embedded in the design environment serves as a stimulus for suggesting what users should attend to next. This direction to new subgoals permits new information to be extracted from memory and reference sources and another step to be taken toward the development of the design. Thus, the integration of various aspects of design enables the situation to “*talk back*” to users (Schoen, 1983) by providing them with immediate and clear feedback of the current problem context.

By virtue of the synergy of integration, such environments can partially articulate the user’s task at hand by a partial specification and construction. As a consequence, the users can be provided with the information relevant to the task at hand by the system without forming queries for the retrieval. The use of the information is also supported in the same environment; thereby the system can analyze usage patterns of the retrieved information and use them for refining the retrieval.

A MULTIFACETED ARCHITECTURE FOR INTEGRATED DESIGN ENVIRONMENTS

During the last five years, we have developed and evaluated several prototype systems of domain-oriented design environments (Fischer, McCall, and Mørch, 1989; Lemke and Fischer, 1990). Different system-building efforts led to the multifaceted architecture that will be described in the context of the JANUS system. The domain of JANUS is the architectural floor plan design of a kitchen. The system is implemented in Common Lisp, and runs on Symbolics Lisp machines. Currently JANUS consists of subsystems JANUS-CONSTRUCTION, JANUS-ARGUMENTATION, and CATALOGEXPLORER. Each subsystem supports different aspects of the architecture.

Although we have emphasized the importance of domain orientation, this architecture should not be regarded as a specific framework for a certain domain. To the contrary, we assume that the architecture presented here serves as a generic framework for constructing a class of domain-specific environments.

Components of the multifaceted architecture

The multifaceted architecture for integrated design environments consists of the following five components (Figure 2).

- A **construction kit** is the principal medium for implementing design. It provides a palette of domain abstractions and supports the construction of artifacts using direct manipulation and other interaction styles. A construction represents a concrete implementation of a design and reflects a user’s current problem situation. Figure 3 shows the screen image of JANUS-CONSTRUCTION, which supports this role.
- An **issue-based argumentative hypermedia system** captures the design rationale. Information fragments in the hypermedia issue base are based on an issue-based information system (IBIS; McCall, 1986), and are linked according to what information serves to resolve an issue relevant to a partial construction. The issues, answers, and arguments held in JANUS-ARGUMENTATION (see Figure 4) can be accessed via links from the domain knowledge in other components.

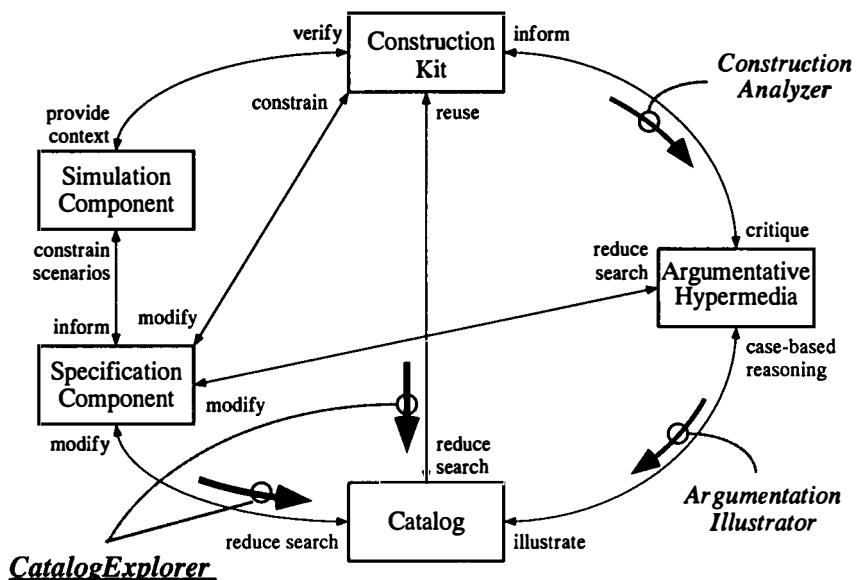


Figure 2: A Multifaceted Architecture

The components of the multifaceted architecture for an integrated design environment. Support for links between the components are crucial for synergy of integration.

- A **catalog** (see Figures 3 and 6) provides a collection of prestored design objects illustrating the space of possible designs in the domain. Catalog examples support reuse and case-based reasoning (Kolodner, 1990; Riesbeck and Schank, 1989).
- A **specification component** (see Figure 7) allows designers to describe some characteristics of the design they have in mind. The specifications are expected to be modified and augmented during the whole design process, rather than to be fully articulated before starting the design. CATALOGEXPLORER provides this mechanism. After specification, users are asked to weigh the importance of each item (Figure 8).
- A **simulation component** allows one to carry out “what-if” games to let designers simulate usage scenarios with the artifact being designed. Simulation complements the argumentative component.

Links among the components

The architecture derives its essential value from the integration of its components and links between the components. Used individually, the components cannot achieve their full potential. Used in combination, however, each component augments the value of the others, forming a synergistic whole.

Links among the components of the architecture are supported by various mechanisms (see Figure 2). The integration enables the system to incrementally understand the task at

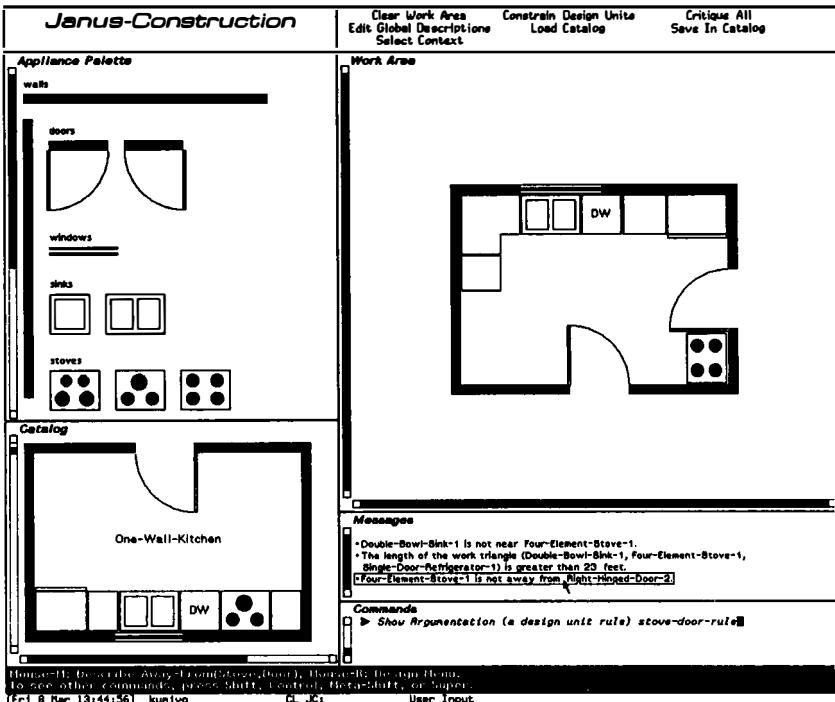


Figure 3: Screen Image of JANUS-CONSTRUCTION

This screen image shows J NUS-CONS RUCTION, the construction component of J NUS. Building blocks (design units) are selected from the *Pale* and moved to desired locations inside the *Work Area*. Designers can reuse and redesign complete floor plans from the *Catalog*. The *Messages* pane displays critiques automatically after each design change that triggers such a critic message (done by CONSTRUCTION ANALYZER). Clicking with the mouse on a message activates JANUS-ARGUMENTATION and displays the argumentation related to that message (see Figure 4).

hand, thereby providing users with the information relevant to the task at hand. The major mechanisms to achieve this are:

- **CONSTRUCTION ANALYZER** is a critiquing component (Fischer et al., 1990) that detects and critiques partial solutions constructed by users based on domain knowledge of design principles. The firing of a critic signals a breakdown to designers (Winograd and Flores, 1986), warning them of potential problems in the current construction, and providing them with an immediate entry into the exact place in the argumentative hypermedia system where the corresponding argumentation lies (see Figures 3 and 4).
 - **ARGUMENTATION ILLUSTRATOR** helps users to understand the information given in an argumentative hypermedia by using a catalog design example as a source of concrete realization (see Figure 4). The explanation given as an argumentation is often highly abstract and very conceptual. Concrete design examples that match the explanation help users to understand the concept.
 - **CATALOGEXPLORER**, described later in detail, helps users to search the catalog space

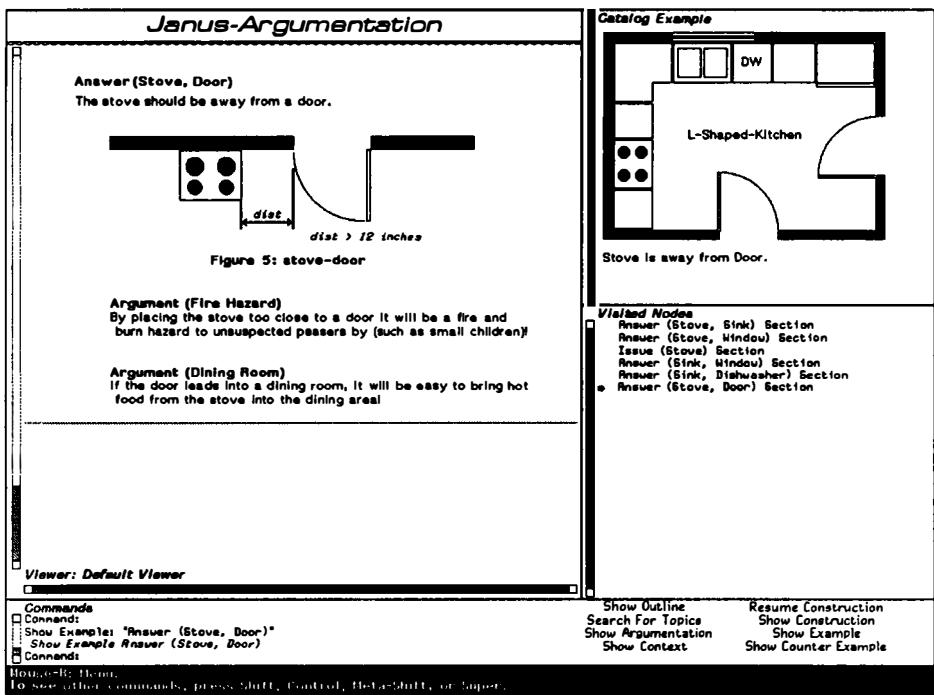


Figure 4: Screen Image of JANUS-ARGUMENTATION

This screen image of JANUS-A GUMENTATION shows an answer to the issue of where to locate the kitchen stove with respect to a door, and graphically indicates the desirable relative positions of the two design units. Below this is a list of arguments for and against the answer. The example in the upper right corner (activated by the "Show Example" command in the *Commands* pane) contextualizes an argumentative principle in relation to a specific design (done by A GUMENTATION ILLUSTRATOR).

according to the task at hand. It retrieves design examples similar to the current construction situation and orders a set of design examples by their appropriateness to the current specification.

Design within the multifaceted architecture

Figure 5 illustrates the coevolution of specification and construction in an environment based on the multifaceted architecture. A typical cycle of events in the environments includes: (1) designers create a partial specification or a partial construction; (2) they do not know how to continue with this process; (3) they switch and consult other components in the system, being provided with information relevant to the partially articulated task at hand; and (4) they are able to refine their understanding based on the *back talk* of the situation. As designers go back and forth among these components, the problem space is narrowed and all facets of the artifact are refined. A completed design artifact consisting of specification and construction may be stored into the catalog for later reuse. Thus, the environment gradually evolves itself by being continually used.

Problem analysis and synthesis are thus integrated in such an environment, following Schoen's (1983) characterization of design activities:

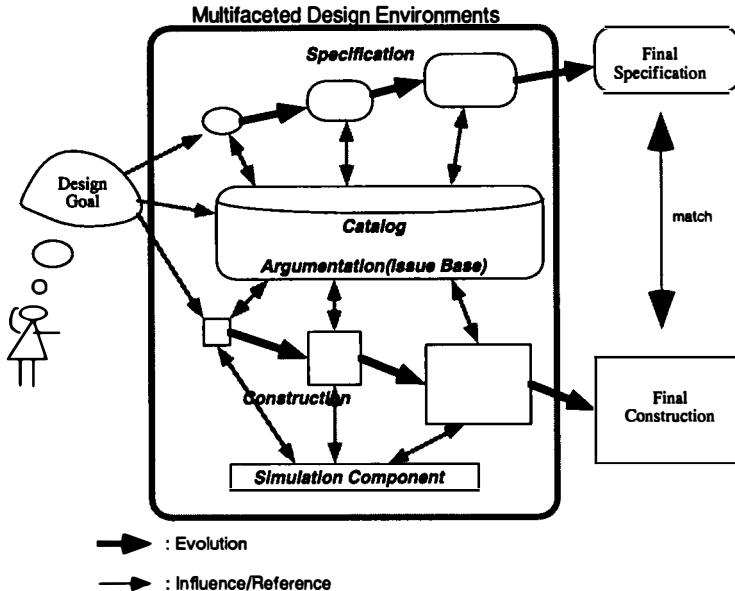


Figure 5: Coevolution of Construction and Specification of Design in Multifaceted Architecture

Starting with a vague design goal, designers go back and forth among the components in the environment. During the process, a designer and the system cooperatively evolve a specification and a construction incrementally, by utilizing the available information in an argumentation component and a catalog and feedback from a simulation component. In the end, the outcome is a matching pair of specification and construction.

Sometimes modification of a specification leads a designer directly to modify a construction, or vice versa. Instead of evolving them, a designer may replace the current construction or specification by reusable design objects. A cycle ends when a designer commits the completion of the development.

The designer shapes the situation in accordance with his initial appreciation of it [construction], the situation “talks back” [critics], and he responds to the situation’s back-talk. In a good process of design, this conversation with the situation is reflective. In answer to the situation’s back-talk, the designer reflects-in-action on the construction of the problem [argumentation].

Schoen’s work provides interesting insights into design processes, but it does not provide any mechanisms to support the approach. Our system-building efforts (McCall, Fischer, and Morch, 1990; Fischer, 1989) are oriented toward the goal of creating these support mechanisms for the theory.

CATALOGEXPLORER

In this section, we describe CATALOGEXPLORER, which links the specification and construction components with the catalog (see Figure 2), followed by a scenario that illustrates a typical use of the system. In the following two sections, we describe the underlying mechanisms used in the scenario in more detail, including the mechanisms of retrieval from specification and retrieval from construction, respectively.

System description

Design objects stored in a catalog can be used for (1) providing a solution to a new problem, (2) warning of possible failures, and (3) evaluating and justifying the decision (Kolodner, 1990; Rissland and Skalak, 1989). The catalog provides a source for different ideas such as commercial catalogs shown by a professional kitchen designer to customers to help them understand their needs and make decisions. For large catalogs, identifying design examples relevant to the task at hand becomes a challenging and time-consuming task.

By integrating specification, construction, and a catalog, CATALOGEXPLORER helps users to retrieve information relevant to the task at hand and, as a result, helps users to refine their partial specification and partial construction. Users need not form queries for retrieving design objects from a catalog because their task at hand is partially articulated by a partial specification and construction.

The design examples in the catalog are stored as objects in a knowledge base. Each design example consists of a floor layout and a set of slot values. The examples are automatically classified according to their explicitly specified features, the slot values provided by a user. Each design example can be (1) critiqued and praised by CONSTRUCTION ANALYZER, and (2) marked with a bookmark, which provides users with control in selecting design examples and forming a personalized small subset of the catalog.

CATALOGEXPLORER is based on the HELGON system (Fischer and Nieper-Lemke, 1989), which instantiates the retrieval by reformulation paradigm (Williams, 1984). It allows users to incrementally improve a query by critiquing the results of previous queries. Reformulation allows users to iteratively search for more appropriate design information and to refine their specification, rather than being constrained to their specified query in the first place (Fischer, Henninger, and Redmiles, 1991).

Based on the retrieval by reformulation paradigm, CATALOGEXPLORER retrieves design objects relevant to the task at hand by using the following mechanisms:

- It provides a specification sheet and a mechanism to differentiate the importance of each specification item by assigning weights to them. It orders design examples by computed appropriateness values based on the specification.
- It analyzes the current construction and retrieves similar examples from the catalog.

A scenario using CATALOGEXPLORER

CATALOGEXPLORER (Figure 6) is invoked by the *Catalog* command from JANUS-CONSTRUCTION (Figure 3). The *Specify* command provides a *specification sheet* (Figure 7) in the form of a questionnaire. After specification, users are asked to assign a weight to each specified item in a *weighting sheet* (Figure 8).

The specified items are shown in the *Specification* window in Figure 6. By clicking on one of the specified items, users are provided with physical necessary condition rules (*specification-linking rules*) for a kitchen design to satisfy the specified item, as seen in the two lines in the middle of the *Specification* window in Figure 6. With this information, users can explore the arguments behind the rules. The shown condition rules are mouse-sensitive, and clicking on one of them will activate JANUS-ARGUMENTATION providing more detailed information. Figure 4 illustrates the rationale behind the rule “*the stove*

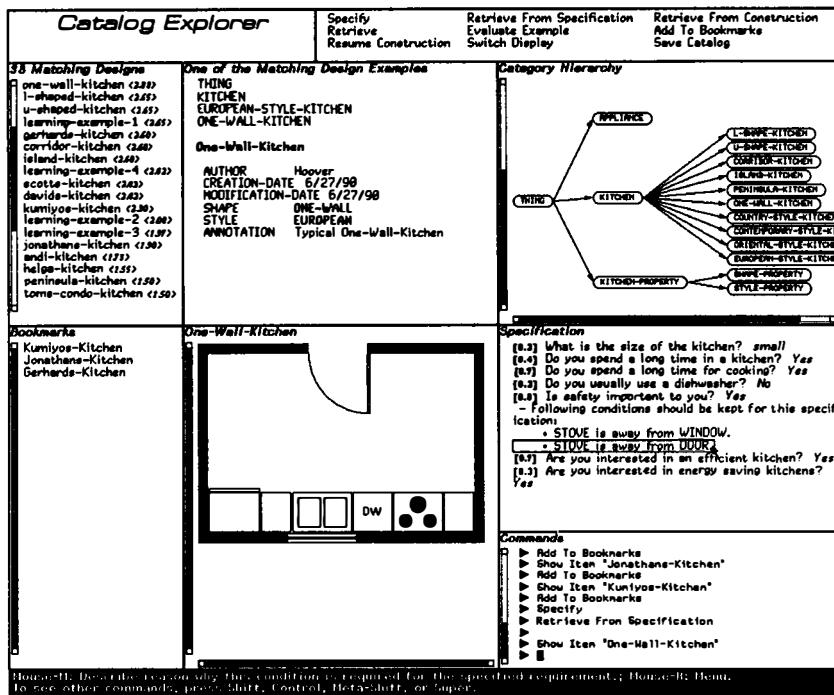


Figure 6: Screen Image of CATALOGEXPLORER

The leftmost *Matching Designs* window lists the names of all matching design examples in the catalog. The numbers following the names represent the *appropriateness* values of each design. The *Bookmarks* window stores some of the previously visited catalog items. The two panes in the middle show one of the matching examples in detail (the top pane shows a set of slot values and the bottom pane a floor layout). The *Category Hierarchy* window shows the hierarchical structure of the catalog. The *Specification* window shows the specified items with assigned weights (see Figures 7 and 8).

should be away from a door if a user wants a kitchen to be safe.” By invoking the *Retrieve From Specification* command, the design examples of the catalog are ordered (see the *Matching Designs* window in Figure 6) by *appropriateness* values to the specified items.

Users can then retrieve design examples similar to their current construction. When invoking the *Retrieve From Construction* command, users are asked to choose a criterion (*parsing topic*) for defining the similarity between the current construction and design examples in the catalog. When users choose “*Design Unit Types*” as a parsing topic, a menu comes up as shown in Figure 9, allowing the user to select all or some of the design unit types being used in the current construction. In Figure 9, a user has selected all appliances that were used in the construction of Figure 3. The system then retrieves examples that contain the specified design unit types.

The above interactions gradually narrow the catalog space, providing users with a small set of examples relevant to the current construction and ordered by the appropriateness to their specification. Users can examine them one by one with a reasonable amount

Specification sheet.			
What is the size of the kitchen?	small	large	Do-Not-Care
Do you spend a long time in a kitchen?	Yes	No	Do-Not-Care
Do you spend a long time for cooking?	Yes	No	Do-Not-Care
Do you usually use a dishwasher?	Yes	No	Do-Not-Care
Is safety important to you?	Yes	No	Do-Not-Care
Is light important to you?	Yes	No	Do-Not-Care
Are you interested in easy plumbing?	Yes	No	Do-Not-Care
Do you have to consider building codes?	Yes	Not so much	Do-Not-Care
Are you interested in an efficient kitchen?	Yes	No	Do-Not-Care
Are you interested in energy saving kitchens?	Yes	No	Do-Not-Care
<input type="button" value="Done"/> <input type="button" value="Abort"/>			

Figure 7: Specification Sheet

The *Specify* command in CATALOGEXPLORER provides a specification sheet in the form of a questionnaire. The questions are derived by analyzing questionnaires being used by professional kitchen designers.

Specify the factor of importance for each specified item. Least								Most	
What is the size of the kitchen?	small								
Do you spend a long time in a kitchen?	Yes								
Do you spend a long time for cooking?	Yes								
Do you usually use a dishwasher?	No								
Is safety important to you?	Yes								
Are you interested in an efficient kitchen?	Yes								
Are you interested in energy saving kitchens?	Yes								
<input type="button" value="Do It"/> <input type="button" value="Abort"/>									

Figure 8: Weighting Sheet for the Specification

After specification, users are asked to weight the importance of each specified item.

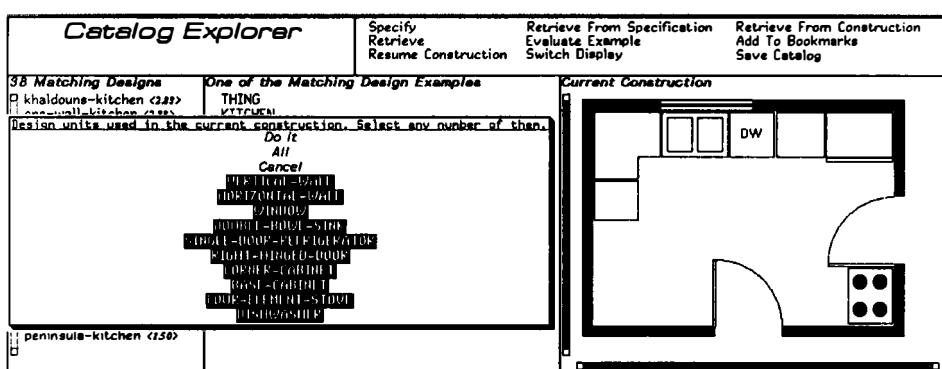


Figure 9: Retrieve From Construction

The *Retrieve From Construction* command with a *parsing topic* “Design Unit Types” analyzes the current construction and provides a list of all the design unit types being used in the construction. Users can then select which design unit types they consider to be most important for locating prestored designs in the catalog.

of effort. If no relevant objects are found, they can modify the specification by either selecting other answers in the specification sheet, or changing the weights in the weighting sheet, or both. After this is done, the *Retrieval from Specification* command will reorder the examples. Users can also use the *Retrieval from Construction* command and choose other criteria for defining the similarity, which will retrieve another set of examples.

Finally, they can decide which example they want to use by bringing it into the *One of the Matching Design Examples* window, and go back to JANUS-CONSTRUCTION with the *Resume Construction* command. JANUS-CONSTRUCTION automatically will show their selected example in the *Catalog* window of JANUS-CONSTRUCTION (Figure 3). Users can refer to this example for getting new ideas on how to proceed with their construction, or they may replace the current construction with the example found.

RETRIEVAL FROM SPECIFICATION

Issues related to specification

In order to use a partial specification for identifying a relevant design object, one must consider the following issues: types of specifications, weighting importance, and multiple contradictory features.

Types of specifications. We have observed that there exist two types of specifications for a design: *surface features* and *hidden features*. For example, the specification “*a kitchen that has a dishwasher*” is a surface feature that explicitly describes the design, whereas “*a kitchen that has less than 100 square feet*,” or “*a kitchen good for a large family*” are hidden features of the design that are not explicitly expressed in the final design artifact (Kolodner, 1990). Surface features are determined by the structure of a design, whereas hidden features are related to functions of the design rather than to the structure (Gero, 1990). Hidden features can be computed or inferred only by using domain knowledge. There are two types of specifications in hidden features, *per se*. Features such as “*a kitchen that has less than 100 square feet*” are objective or judgmental, whereas features such as “*a kitchen good for a large family*” are subjective. A set of formal rules can be defined for deriving objective hidden features. In contrast, subjective hidden features can be inferred only relative to one’s viewpoint. An inference of whether a kitchen design is good for a large family is subject to dispute and may vary across time and society.

In practice, initial customer questionnaires that professional kitchen designers give to their customers often ask questions relating to subjective hidden feature specifications. The experts map these specifications to concrete structural features by using their domain knowledge and experience.

Mechanisms for retrieval of design objects from specifications should, therefore, be different according to their types. Retrieving design examples from the catalog by surface feature specification can be done with a conventional query mechanism. In contrast, for retrieving design examples by hidden feature specifications, the system must have the domain knowledge to infer those features.

Weighting importance. Sometimes specified items contradict each other. If those contradictions are among hidden features, users may not notice the occurrence of the contradictions. Consequently, the system cannot retrieve design examples from the catalog that satisfy their specification because there do not exist such examples. For example, consider the two specifications “*a safe kitchen*” and “*a kitchen that provides easy access to the dining area*.” Although they seem not to contradict each other, they do so in terms of hidden features. As seen in Figure 4, a stove should be away from a door for the first specification, whereas a stove should be close to a door for the second one.

To resolve the contradiction, users must prioritize the specifications and make trade-

offs. They have to differentiate importance of the specifications by assigning a weight to each specification item. If users specify that “*a safe kitchen*” is more important to them, kitchen designs in which the stove is away from a door are more appropriate to the user’s specification than others.

Multiple contradictory features. One design object may have multiple contradictory features; that is, hidden features that semantically contradict each other. For example, there can be a kitchen design in which some relationships of appliances in the example are *good for a large family*, whereas other relationships in the design are *bad for a large family*. In practice, some part of a design may serve contradictory purposes to other parts of the same design.

Mechanisms

Specification-linking rules. CATALOGEXPLORER dynamically infers subjective hidden features of design examples in the catalog by using domain knowledge in the form of *specification-linking rules*. The *specification-linking rules* link each subjective hidden feature specification item to a set of physical condition rules. For example, in the middle of the *Specification* window in Figure 6 two rules are shown (*stove away from a door* and *a stove away from a window*), which are conditions for a kitchen to have a hidden feature “*a safe kitchen*.”

Previous versions of CATALOGEXPLORER required design examples to have explicitly specified values for *good-for* and *bad-for* slots to represent subjective hidden features. This approach relied on the questionable assumption that one could determine a priori that these features will become relevant later. Such features may become obsolete under new circumstances (e.g., an inefficient kitchen design may become efficient by introducing new appliances such as a microwave). Designers cannot articulate all the subjective features of a design, and even if they could do so, such features may be difficult to understand.

The most important aspect of the *specification-linking rules* is that they can be dynamically derived from the content of JANUS-ARGUMENTATION. Suppose the system has the following internal representation for the “*(Fire Hazard)*” argument shown in Figure 4.

$$\neg (\text{Away-from-p STOVE DOOR}) \rightarrow \text{'FIRE-HAZARDOUS'}^1 \quad (1)$$

And the system has the domain knowledge:

$$\text{'SAFETY} \rightarrow \neg \text{'FIRE-HAZARDOUS'}^2 \quad (2)$$

When users specify that they are concerned about safety, the system infers that design examples with a stove away from a door are appropriate to their need by the following inference. First, (1) is equivalent to the following:

¹Symbols such as “FIRE-HAZARDOUS” and “SAFETY” represent concepts as constant values, whereas “STOVE” and “DOOR” represent classes of design units. “Away-from-p” is a predefined predicate computing a distance between two design units and returns true if it exceeds a certain amount.

²This should read as “For a kitchen to be safe, it needs to be *not* fire-hazardous.”

$\neg \text{'FIRE-HAZARDOUS' } \rightarrow (\text{Away-from-p STOVEDOOR})$

(3)

Therefore, by using (2) and (3),

(2) \wedge (3) $\rightarrow (\text{'SAFETY' } \rightarrow (\text{Away-from-p STOVE DOOR}))$

(4)

Appropriateness to a set of specifications. To deal with some of the issues mentioned earlier, CATALOGEXPLORER provides a mechanism for assigning a weight to each specification item and uses the concept of *appropriateness* of a design example to a set of specification items. The appropriateness of a design in terms of a set of specification items is defined as in Figure 10.

S_1, S_2, \dots, S_n is a set of specification items with weights w_1, w_2, \dots, w_n , respectively. For each specification item S_p , let R_{ij} ($j=1 \dots m_i$) be a set of physical necessary conditions specified by a specification-linking rule. Let E be an example design, and define $E(R)$ as follows:

$$E(R) = \begin{cases} 1 & \text{if the condition R is satisfied in E} \\ 0 & \text{otherwise} \end{cases}$$

Then, the *appropriateness* of design E in terms of a set of specifications $S=\{(S_1, w_1), (S_2, w_2), \dots, (S_n, w_n)\}$ is defined as follows:

$$\sum_{i=1}^n \left\{ \left(\sum_{j=1}^{m_i} E(R_{ij}) / m_i \right) \times w_i \right\}$$

Figure 10: Definition of the *Appropriateness* of a Design

As a simple example, suppose a user specified one item: “*Is safety important to you? YES*” with a weight 0.8. The physical necessary conditions of this item are “*a stove is away from a door*” and “*a stove is away from a window*,” as seen in the *Specification* window in Figure 6. Therefore, a kitchen that has a stove away from a door but close to a window gets the appropriateness value of $0.4=(1+0)/2 \times 0.8$.

RETRIEVAL FROM CONSTRUCTION

For retrieving design examples related to a partial construction, one must deal with the issues of matching design examples in terms of surface features of a design, namely, at a structural level. The issues discussed in the previous section, such as partial matching and factor of importance, also hold here.

Domain-specific parsers analyze the design under construction. They represent the user’s criteria for the articulation of the task at hand from a partial construction. In other words, they determine how to define similarities between the partial construction and a design example in the catalog for retrieval of design examples from the catalog.

CATALOGEXPLORER supports the following two parsers. Users have a mechanism to choose which parser they want to use.

- ***Design unit types:*** Search for examples that have the same design unit types as the current construction. The system first analyzes the current construction, finds which design unit types are used, and provides the user with a menu to select some of them (see Figure 9).
- ***Configuration of design units:*** Search for examples that have the same configuration of design units. For example, if the current construction has a dishwasher next to a sink, the examples matching this configuration element will be retrieved.

RELATED WORK

Using catalogs in design raises many problems in *case-based reasoning*. Retrieval techniques used in case-based reasoning systems, however, are often applicable only for domains in which problems can be clearly articulated, such as word pronunciation (Stanfill and Waltz, 1988). Such systems do not support dealing with fluctuation of the problem specification and are inadequate for ill-defined problems.

In JULIA (Kolodner, 1988), problem and solution structures must be articulated in frame representations before starting a retrieval process. *Value Frames* used in JULIA provide the rationale behind a design decision, which can be used for the retrieval of cases. CATALOGEXPLORER needs to integrate mechanisms to support recording of the design rationale for this purpose (Fischer et al., 1991).

Most of case-based reasoning systems require representations of cases to be predetermined, and therefore are not feasible. The approach presented in this paper addresses an indexing problem (Kolodner, 1990) by using more than surface representation of a case and enables the match at more abstract levels of representations. Use of the specification-linking rules can be regarded as a type of analogical matching such as *systematicity-based match* in CYCLOPS (Navinchandra, 1988). In CYCLOPS, however, the explanations associated with cases must be predetermined and cannot be dynamically computed.

The INTERFACE system (Riesbeck, 1988) provides interesting mechanisms for addressing some of the issues relating to matching rules. One of them is the use of *abstraction hierarchies* for dealing with the issue of partial matching, which could be used in CATALOGEXPLORER to support retrieval from construction. Another mechanism is to differentiate the importance of design features. This is similar to the *weighting sheet* in CATALOGEXPLORER, but it requires the features to be linearly ordered. Assigned importance values in our system enable users to deal with more complex contradictory features. Being built for the purpose of constructing a case-based library, the INTERFACE system supported these mechanisms only while storing cases in the library. In our work, the retrieval processes are driven by the user's task at hand, requiring that the weights are determined at the retrieval time rather than at the time when cases are stored. The INTERFACE system supports the creation of such matching rules only in an ad hoc manner. The integrated architecture of CATALOGEXPLORER enables the specification-linking rules to be derived from the argumentation component associating the rules with a clearly stated rationale.

The system allows users to store design examples in the catalog without checking for duplications and redundancies. Other systems store only prototypes (Gero, 1990), or prototypes and a small number of examples that are a variation of them (Riesbeck, 1988).

These approaches allow users to access *good* examples easily and prevent the chaotic growth of the size of the catalog. However, by not including failure cases, these catalogs prevent users from learning what went wrong in the past.

Many case-based reasoning systems support comprehension and adaptation of cases (Kolodner, 1990). CATALOGEXPLORER supports the comprehension of examples by allowing users to evaluate them with CONSTRUCTION ANALYZER. Adaptation is done by the users by bringing an example into the *Work Area* in JANUS-CONSTRUCTION. No efforts have been made toward automating adaptation in our approach.

DISCUSSION

Achievements

By integrating knowledge-based construction, hypermedia argumentation, catalogs of pre-stored design objects, and specification components, several crucial design activities can be supported, such as relevance to the task at hand, the situation talking back, reflection-in-action (Schoen, 1983), and integration of problem analysis and synthesis.

In CATALOGEXPLORER, users gradually narrow a catalog space. The system can dynamically infer subjective hidden features and provide users with an explanation for the inference mechanism. The system retrieves examples similar to the current construction, providing users further directions in proceeding the design or warning them of potential failures. Using the retrieved information they can incrementally evolve a specification and a construction in JANUS. The retrieval mechanisms of the system allow users to access information relevant to the task at hand in a more effective and accurate way without requiring the users to form queries. Control and responsibility of retrieval of information is shared between the user and the system (Fischer, 1990).

Limitations

A major limitation of the current system is the relatively small size of the catalog (less than a hundred examples). Many problems of managing large spaces effectively have not been dealt with. A lack of mechanisms for associating formal representations to arguments forces us to manually derive the *specification-linking rules*. The definition of appropriateness is limited and needs a more sophisticated mechanism such as connectionist networks (Henninger, 1990). The parsers for analyzing partial constructions should be extended to deal with more abstract levels, such as an emerging shape (e.g., L-shape or U-shape) that currently requires to be specified by the user. A combinatorial use of the parsers should be explored.

Future work

Future extensions of integrated design environments based on the multifaceted architecture include:

- *Level of Assembly.* The use of JANUS by kitchen designers has illustrated that the designers work not only with design units, but with higher level abstractions such as cooking centers and clean-up centers. These centers should be integrated into the palette, eliminating clear distinction between the elements in the palette and the catalog. The catalog should contain not only completed designs, but also important partial designs. These extensions will require further consideration on issues such as how to focus on a solution (Kolodner, 1990).

- *Support for Other Transition Links.* A partial specification can be used to determine the set of relevant arguments in the argumentation component, enabling us to dynamically rearrange argumentation space. A link between construction and specification can reduce the set of relevant units displayed in the palette.
- *Extensions of the Architecture.* Our design environment for user interface design (Lemke, 1989; Lemke and Fischer, 1990) has been improved greatly in its effectiveness by having a *checklist* component to help users to structure and organize their design activities. The integration of the checklist into the multifaceted architecture has to be explored further.
- *End-User Modifiability.* In developing design environments, domain knowledge should be built into a *seed*. As users use the environment continually, this seed should be extended. Sophisticated mechanisms for end-user modifiability (Fischer and Grgensohn, 1990) are crucial for this evolution of seeded environments.

CONCLUSION

Design activities incorporate many cognitive issues such as recognizing and framing a problem, understanding given information, and adapting the information to the situation. Integration of problem setting and problem solving is crucial in dealing with ill-defined problems. In this paper we have described mechanisms relating partial specifications and partial constructions to a catalog of prestored designs, thereby retrieving design objects stored in a catalog relevant to the task at hand without asking users to form queries. The system demonstrates the synergy of integrated design environments empowering human designers. The multifaceted architecture developed in the context of these research efforts is a promising architecture for building a great variety of integrated design environments in different domains.

ACKNOWLEDGMENTS

The authors would like to thank the members of the Human-Computer Communication group at the University of Colorado, who contributed to the conceptual framework and the systems discussed in this paper. The research was supported by Software Research Associates, Inc. (Tokyo, Japan), by the National Science Foundation under grants No. IRI-8722792 and IRI-9015441, and by the Army Research Institute under grant No. MDA903-86-C0143.

REFERENCES

- Cross, N. (1984). *Developments in Design Methodology*, John Wiley, New York.
- Fischer, G. (1989). Creativity Enhancing Design Environments, *Preprints Modelling Creativity and Knowledge-Based Creative Design*, Design Computing Unit, University of Sydney, Sydney, pp. 127-132.
- Fischer, G. (1990). Communications Requirements for Cooperative Problem Solving Systems, *The International Journal of Information Systems* (Special Issue on Knowledge Engineering), 15(1): 21-36.

- Fischer, G. and Girsensohn, A. (1990). End-User Modifiability in Design Environments, Human Factors in Computing Systems, *CHI'90 Conference Proceedings* (Seattle, WA), ACM, New York, pp. 183-191.
- Fischer, G., and Lemke, A.C. (1988). Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication, *Human-Computer Interaction*, 3(3): 179-222.
- Fischer, G. and Nieper-Lemke, H. (1989). HELGON: Extending the Retrieval by Reformulation Paradigm, Human Factors in Computing Systems, *CHI'89 Conference Proceedings* (Austin, TX), ACM, New York, pp. 357-362.
- Fischer, G., Lemke, A.C., Mastaglio, T. and Morsch, A. (1990). Using Critics to Empower Users, Human Factors in Computing Systems, *CHI'90 Conference Proceedings* (Seattle, WA), ACM, New York, pp. 337-347.
- Fischer, G., Lemke, A.C., McCall, R. and Morsch, A. (1991). Making Argumentation Serve Design, *Technical Report*, Department of Computer Science, University of Colorado, Boulder, CO.
- Fischer, G., Henninger, S. and Redmiles, D. (1991). Intertwining Query Construction and Relevance Evaluation, Human Factors in Computing Systems, *CHI'91 Conference Proceedings* (New Orleans, LA), ACM, New York (in press).
- Fischer, G., McCall, R. and Morsch, A. (1989). JANUS: Integrating Hypertext with a Knowledge-Based Design Environment, *Proceedings of Hypertext'89* (Pittsburgh, PA), ACM, New York, pp. 105-117.
- Gero, J.S. (1990). Design Prototypes: A Knowledge Representation Schema for Design, *AI Magazine* 11(4): 26-36.
- Halasz, F. G. (1988). Reflections on NoteCards: Seven Issues for the Next Generation of Hypermedia Systems, *Communications of the ACM* 31(7): 836-852.
- Henninger, S. (1990). Defining the Roles of Humans and Computers in Cooperative Problem Solving Systems for Information Retrieval, *Proceedings of the AAAI Spring Symposium Workshop on Knowledge-Based Human-Computer Communication*, pp. 46-51.
- Hutchins, E. L., Hollan, J. D., and Norman, D. A. (1986). Direct Manipulation Interfaces, in D.A. Norman and S.W. Draper (eds), *User Centered System Design, New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, pp. 87-124, ch. 5.
- Kolodner, J. L. (1988). Extending Problem Solving Capabilities Through Case-Based Inference, in J. Kolodner (ed.), *Proceedings: Case-Based Reasoning Workshop*, Morgan Kaufmann, Clearwater Beach, FL, pp. 21-30.
- Kolodner, J. L. (1990). What is Case-Based Reasoning?, Tutorial Text on Case-Based Reasoning, *AAAI'90*, Boston, MA, pp. 1-32.
- Lemke, A.C. (1989). *Design Environments for High-Functionality Computer Systems*, Unpublished Ph.D. Dissertation, Department of Computer Science, University of Colorado.
- Lemke, A.C., and Fischer, G. (1990). A Cooperative Problem Solving System for User Interface Design, *Proceedings of AAAI-90*, Eighth National Conference on Artificial Intelligence, AAAI Press/The MIT Press, Cambridge, MA, pp. 479-484.
- McCall, R. (1986). Issue-Serve Systems: A Descriptive Theory of Design, *Design Methods and Theories*, 20(8): 443-458.

- McCall, R., Fischer, G. and Morch, A. (1990). Supporting Reflection-in-Action in the Janus Design Environment, in M. McCullough et al. (eds), *The Electronic Design Studio*, The MIT Press, Cambridge, MA, pp. 247-259.
- Navinchandra, D. (1988). Case-Based Reasoning in CYCLOPS, in J. Kolodner (ed.), *Proceedings: Case-Based Reasoning Workshop*, Morgan Kaufmann, Clearwater Beach, FL, pp. 286-301.
- Nielsen, J., and Richards, J.T. (1989). The Experience of Learning and Using Smalltalk, *IEEE Software*, May, pp. 73-77.
- Reeves, B. (1990). Finding and Choosing the Right Object in a Large Hardware Store—An Empirical Study of Cooperative Problem Solving among Humans, *Technical Report*, Department of Computer Science, University of Colorado, Boulder, CO.
- Riesbeck, C. K. (1988). An Interface for Case-Based Knowledge Acquisition, in J. Kolodner (ed.), *Proceedings: Case-Based Reasoning Workshop*, Morgan Kaufmann, Clearwater Beach, FL, pp. 312-326.
- Riesbeck, C. K., and Schank, R. C. (1989). *Inside Case-Based Reasoning*, Lawrence Erlbaum Associates, Hillsdale, NJ.
- Rissland, E. L., and Skalak, D. B. (1989). Combining Case-Based and Rule-Based Reasoning: A Heuristic Approach, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (Detroit, MI), Morgan Kaufmann, Palo Alto, CA, pp. 524-530.
- Rittel, H. W. J. (1984). Second-generation Design Methods, in N. Cross (ed.), *Developments in Design Methodology*, John Wiley, New York, pp. 317-327.
- Schön, D. A. (1983). *The Reflective Practitioner: How Professionals Think in Action*, Basic Books, New York.
- Sheil, B. A. (1983). Power Tools for Programmers, *Datamation*, February, pp. 131-144.
- Simon, H. A. (1973). The Structure of Ill-Structured Problems, *Artificial Intelligence*, 4: 181-200.
- Simon, H. A. (1981). *The Sciences of the Artificial*, The MIT Press, Cambridge, MA.
- Stanfill, C. and Waltz, D. L. (1988). The Memory-Based Reasoning Paradigm, in J. Kolodner (ed.), *Proceedings: Case-Based Reasoning Workshop*, Morgan Kaufmann, Clearwater Beach, FL, pp. 414-424.
- Stefik, M.J. (1986). The Next Knowledge Medium, *AI Magazine*, 7(1): 34-46.
- Williams, M.D. (1984). What Makes RABBIT Run?, *International Journal of Man-Machine Studies*, 21: 333-352.
- Winograd, T., and Flores, F. (1986). *Understanding Computers and Cognition: A New Foundation for Design*, Ablex Publishing Corporation, Norwood, NJ.

Steering automated design

L. Colgan,[†] P. Rankin[‡] and R. Spence[†]

[†]Department of Electrical Engineering
Imperial College
London SW7 2BT UK

[‡]Philips Research Laboratories
Redhill Surrey RH1 5HA UK

Abstract. We describe the rationale and design of an advanced interface linking an engineering designer to a complex design facility whose novelty resides in its provision of both automated design and knowledge-based advisory systems. The creation of the system was hampered by a lack of knowledge as to how automated design and advisory systems would be used, a lack of experimental controls, and the high cost of implementation. Despite these factors, a powerful industrial design system was created which fluently integrates conventional interactive design with automated design. The approach adopted in the design of the constituent components of the interface is described, including comment on prototyping needs and evaluation. Finally, our methodology is critiqued, and useful outcomes identified.

1. THE DESIGN OF ARTIFACTS AND SCHEMA

Visualisations are rarely used to communicate insights in mid-flight so that the investigator can guide the computation.

Frederick P. Brooks, Jr, *CHI'88 Proceedings*, p. 7

The design of an artifact (such as a toy or a silicon chip) or a scheme (such as a project schedule) requires, in essence, a creative act on the part of the human designer followed by the choice of numerical values for the components comprising the designed object. Thus, a structural designer might choose a particular approach, say a suspension bridge, to link two islands, and then choose the thicknesses of the steel members, the strength of the cables and many other parameter values to ensure that the properties of the bridge - the weight to be supported, the deflection under load and the stability, to name but a few - are satisfactory.

The characteristics of structural design are shared by many engineering fields, including electronic circuit design, the context of this paper. In this field a specification of required performance is received from a customer. Candidate solutions are proposed by the human designer in the form of circuit structures comprising many components, each described by one or more parameters (for example, resistance or capacitance). Equally, the circuit exhibits many performance properties such as amplification, freedom from

noise, manufacturing cost and heat dissipation. Unfortunately, each of the parameters (as many as 100) affects many of the properties, often significantly so (Figure 1). The task of the designer, which is to adjust parameter values to ensure that every property satisfies the customer's specification, is therefore an extremely difficult one. Humans are notoriously poor at the search through multi-dimensional parameter space. It is made especially difficult if, as is usually the case, conflicts (trade-offs) occur between different performance properties.

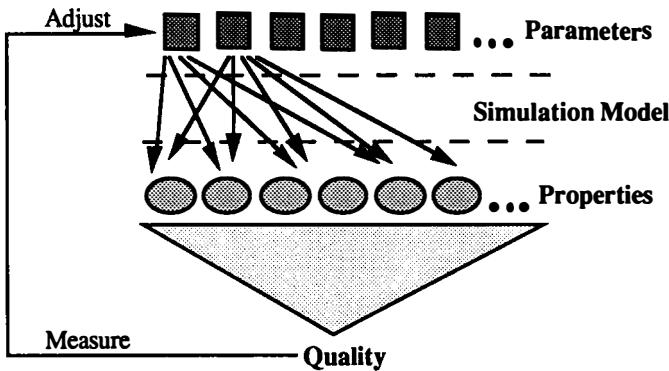


Figure 1. Characteristics of the Design Process

2 AUTOMATED DESIGN

Over the past two decades many computer routines have been devised which automate the adjustment of parameter values. They are often called optimisation algorithms because their purpose is so to influence the properties of a designed object that the object can be considered to be optimum in some sense. In outline, the user of an optimisation algorithm first describes a model of the object being designed (in our case an electronic circuit) which defines the computable performance arising from a particular choice of parameter values. Next, a single scalar measure of the quality of the circuit is defined by an **objective-function** which actually describes the *discrepancy* between achieved and desirable performance. It is this discrepancy which the optimisation algorithm attempts to minimise. Since there is no hard limit to its value, the objective-function is referred to as 'soft'. By contrast, '**hard**' **constraints** in the form of upper and lower limits on aspects of performance are often prescribed. Finally, the designer identifies those parameters - called **design variables** - which the optimisation algorithm is free to adjust.

All this information is then submitted to the optimisation algorithm which, step-by-step, adjusts parameter values, gradually improving the quality of the circuit until the objective-function is deemed either to have reached a minimum or be insensitive to further parameter adjustment. During this process, algorithms usually form approximations to the gradients of the objective function with respect to design variables to set the direction and

step size of parameter movement. Although many minima may exist, more than one may be acceptable to the designer. In summary, the advantage of an optimisation algorithm is that it relieves the human designer of the colossal task of choosing parameter values which deliver a 'good' circuit.

3. COGNITIVE DIFFICULTIES

Unfortunately, optimisation algorithms find little use in real industrial circuit design. The reasons are mainly cognitive in nature and include the following.

- (a) It is very difficult to define a scalar measure (the objective function) which takes all properties of interest into account and provides a measure of the discrepancy between the ideal and the already achieved. In the past, designers have only been asked to quantify informally what is 'best'.
- (b) At the start the designer has poor insight into the interrelations between important properties. At best only first-order effects are appreciated.
- (c) Typically, optimisation programs demand a complete specification of all performance requirements, whereas the typical designer would prefer to introduce requirements incrementally as his or her knowledge of the designed object accumulates.
- (d) The designer may not know at the start which parameters have strong effects upon properties and might, therefore, be suitable as designable parameters. Additionally, some parameters may have a similar influence on the objective function, causing a redundancy or non-orthogonality in the variables which may severely impair the algorithm operation.
- (e) If an optimisation is carried out in 'batch mode' a great deal of information about the progress of the design is lost: it could be of immense benefit to the designer in building up insight, for example to expose the conflicts and trade-offs in the problem.
- (f) Conventional optimisation programs require the problem to be specified in mathematical notation, which is completely foreign to the designer, whereas engineering designers tend to prefer graphical representation. In any case, learning and using a new syntax adds to the designer's cognitive load.
- (g) In many stages of design the designer thinks qualitatively, a mode poorly supported by current user interfaces.

We were particularly convinced that the effectiveness of optimisation could be significantly enhanced if the progress of optimisation could be observed by the designer and modified if deemed necessary. This conviction was reflected in the name assigned to the proposed system: CoCo - the Control and Observation of Circuit Optimisation. Additionally, for our investigation to have real value it was considered essential for the CoCo electronic circuit design system to be capable of use by real circuit designers engaged in the design of real circuits.

4. CONSTRAINTS ON THE SOLUTION

The constraints imposed upon the invention, exploration and development of the new interface were severe, and strongly influenced the route taken in the interface's development.

4.1 Novelty

The *novelty* of the application is one source of difficulty. The use of optimisation in industry - or, indeed, anywhere - is so minimal that no established and reliable body of knowledge and expertise is available to guide the interface design. For example, it is not clear what information the designer wishes to see or can usefully observe as the optimisation proceeds or is replayed, or how the user would wish to interact during the monitoring and guidance of the optimisation. Although there are points of similarity between the control and observation of circuit optimisation on the one hand and, on the other, existing examples of process control, the essential difference between a *design* task and the *control* task leaves many questions unanswered.

This novelty has two repercussions. One is that no 'controls' are available against which any proposed system can be tested. The other is that potential users of an optimisation system face a learning curve, and in some cases may be resistant to a change in their working habits, especially when under the typically tight deadlines in industrial environments where the optimisation system is targetted.

4.2 Context

One approach that might be adopted to harness optimisation for the benefit of industrial designers is to create a 'simple' laboratory system which would allow controlled studies to focus on isolated aspects of interface issues. However, this would carry the danger that results would not refer to realistic engineering use of optimisation. Only when a reasonably complete picture of regular use of optimisation has been established can controlled laboratory studies begin to make a useful and relevant contribution to knowledge

Our approach was to implement a complete exemplar circuit design environment incorporating optimisation. This implied the use of large amounts of existing code (more than 350k lines), blending in new user interface code to the same professional standard (150k lines were created during the four year project lifetime), and solving very complex system communication problems. While stronger industrial relevance was then assured, the impact of extending an existing CAD system on project development effort is clear. Another consequence of our decision to create an exemplar system was the need to have an effective system of prototyping. With a completely novel system there are few if any guidelines that allow a rigid specification to be created *ab initio*. Economically effective trials using 'mock-ups' provide valuable direction.

4.3 Expandability

Any engineering design system that does not anticipate change and extension is doomed. Such considerations led to the decision to incorporate two expert advisory systems in the circuit design facility. One expert system provides mathematical advice about the choice of optimisation algorithm. The other incorporates electrical knowledge, offering the kind of assistance an experienced technician sitting beside a master designer might give. These expert systems play an essential role in the use of optimisation and are considered in a companion paper (Gupta and Rankin, 1991). Systems which provide such knowledge-

based assistance to engineering design are few in number (Arora and Baenziger, 1986; Ketonen et al, 1988; Baker et al, 1989).

5. EVOLUTIONARY APPROACH TO INNOVATION

CoCo's architecture and interface aimed to bridge the gap between industrial designers versed in electrical engineering and the numerical skills practised by the mathematical community. This paper concentrates on the user-interface elements in this concept, but tries to place their development in context.

The creation of a large system usually takes place in two stages. The first, 'conceptual' stage in which the essential structure and functionality is identified, requires the vision of developers having deep experience of both the design domain and optimisation techniques: users are a poor source of direction at this stage, having no experience with either the power or the difficulties of optimisation techniques.

It is in the second stage that users, exposed to a system of gradually increasing sophistication, which is always capable of supporting real problem-solving, can provide the feedback essential to the improvement of the user interface. This feedback was obtained by gradually extending and validating a growing 'core system', a process achieved by the incremental addition of largely independent functional subsystems. Such an approach offered the combined benefits of a constantly available working system and considerable scope for subsystem innovation via independent prototyping without risking the whole.

6. THE CoCo DESIGN TOOL

The CoCo system, shown diagrammatically in Figure 2, comprises a number of modules of which three are interfaces, two are knowledge based and two are involved with numerical processes.

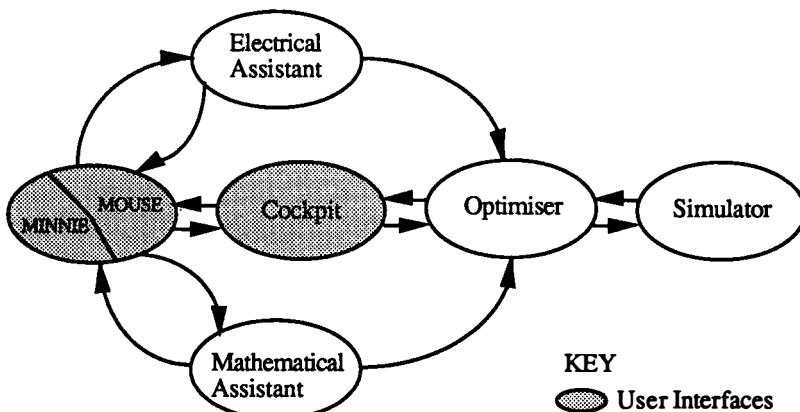


Figure 2. CoCo System Concept .

The kernel of the system is a robust optimisation engine (the 'Optimiser') which can repeatedly call the **Simulator** with new trial values of designable variables chosen by the algorithm and receive updated objective function and/or constraint values for the latest version of the design. MINNIE was an existing graphical interface allowing a circuit designer to sketch a design and easily examine its predicted performance. The **MOUSE** module allows the designer to define performance specifications, pose an optimisation problem and view the results. Two knowledge-based subsystems interact with the main system. An **Electronic Assistant** acts as would an experienced technician, automatically running rules to monitor 'common-sense' but unvoiced electrical requirements as the optimiser changes the design. The **Mathematical Assistant** encapsulates the numerical art of selecting an appropriate optimisation algorithm, tuning its parameters and diagnosing the reasons for its termination. The **Cockpit** is a novel graphical interface which allows the designer to observe the progress of automated design and intervene when judged appropriate.

The submodules of CoCo are now described in more detail, focusing upon the three user interface parts.

7. MINNIE: A GRAPHICAL ENVIRONMENT FOR MANUAL DESIGN

User interfaces in CAD systems for analogue electronics echo a metaphor which is familiar to the circuit designer - the workbench. Designers 'connect up' a network of components by drawing a circuit diagram (a 'schematic') on a workstation screen, 'attach current/voltage driving sources' to define the circuit performance of interest, and 'hook probes' onto interesting points in the circuit by pointing to locations in the circuit diagram. A CPU-intensive simulation of the circuit is then performed, whereupon the designer views presentations of the predicted circuit performance as if on an oscilloscope.

An existing circuit design environment called MINNIE developed from a concept established in 1977 (Spence and Apperley) was chosen as the basis for an attempt to integrate circuit optimisation into the existing design flow. MINNIE comprises 150k lines of code to support three modes of designer activity:

In the **Drawing Mode** schematics such as that of a CMOS amplifier shown on the left of Figure 3 can be constructed with components taken from libraries accessed through menus shown on the right of Figure 3. Inter-relationships between component parameter values can also be defined to maintain design rules during subsequent modifications.

In the **Analysis Mode** simulation options can be chosen from pop-up menus, and test probes assigned to circuit locations by a pointing action. A wealth of circuit properties, including user-defined functions, can be computed, any of which can be the subject of automated design.

Simulated circuit performance is displayed and available for measurement in the **Results Mode**. Slider bars on the screen can allow exploration of the effect of any designated independent variable. A key feature of MINNIE is the ability to display results in both quantitative and qualitative form on the schematic to 'electrically animate' it, thereby enhancing the designer's insight into circuit functioning.

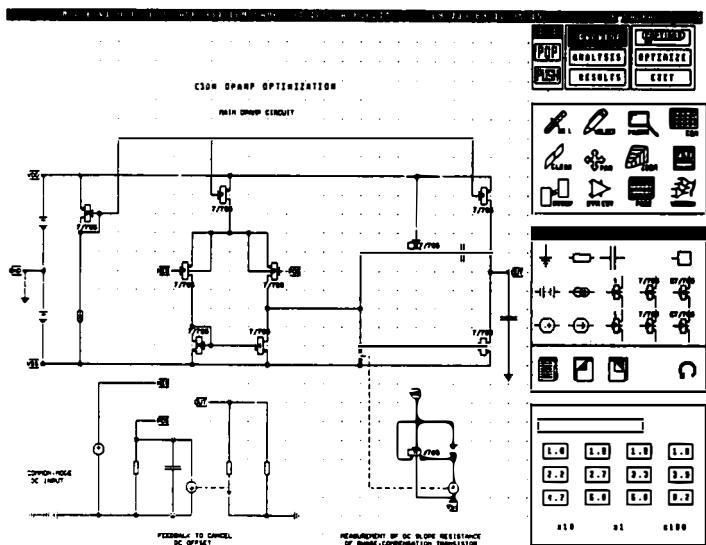


Figure 3. CMOS operational amplifier.

8. MOUSE: A GRAPHICAL ENVIRONMENT FOR DESIGN AUTOMATION

8.1 New Activity States

The original MINNIE system formed the starting point for the work reported here. Effort would not permit a complete re-write for the additional optimisation facilities, so that extensions were forced to use the same style of menus and user interaction as the original. Code generators were used to create menu procedures, enabling menus to be in open, closed or 'hibernated' states. Hibernated menu branches for a main system mode can be re-opened down to their last state as left by the user, or fully closed when, for instance, a new circuit design is started. Picks which cannot be actioned by a child menu are passed to its parent. In all, about 100k lines of new code and 90 new menus were created to provide the interface from which batch optimisations could be composed, submitted and reviewed off-line.

This user-interface system, 'MOUSE', added two further modes to 'MINNIE'. One is a Specifications Mode, allowing the designer to define the desired circuit performance, the other an Optimisation Mode in which design variables are identified and design priorities decided. Performance improvements produced by the optimisation can be compared against the performance requirements in the Results Mode. When the results of optimisation are deemed satisfactory, the best values of the design variables can be updated in the circuit schematic. The main design cycles between these five states are depicted in Figure 4.

Note that the tight coupling between modes of a single program makes for a fast response and encourages the exploration of new design ideas. The states of activity in each

of the modes are automatically remembered - one pick returns the last menu situation in the selected mode, eliminating much menu traversal.

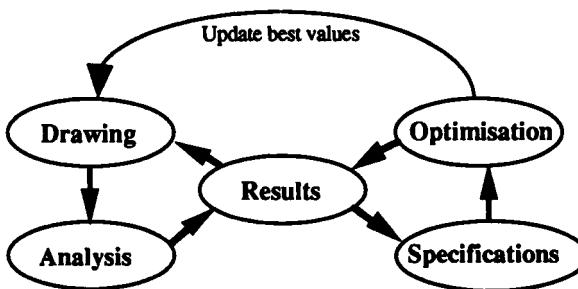


Figure 4. MOUSE - Main cycles between states:

Version control is applied to all objects - such as schematics, test conditions and specifications - held in the design's database. Dependent files such as simulation or optimisation results are keyed to the version numbers of their parents. Recall of an old design therefore also brings its associated analysis set-ups, specifications, optimisation profile and last selected views of results. Without this attention to 'housekeeping' details and a deliberate attempt to keep the screen uncluttered and context-sensitive, the designer would quickly be overwhelmed by the mass of data and actions involved, or afraid to experiment creatively.

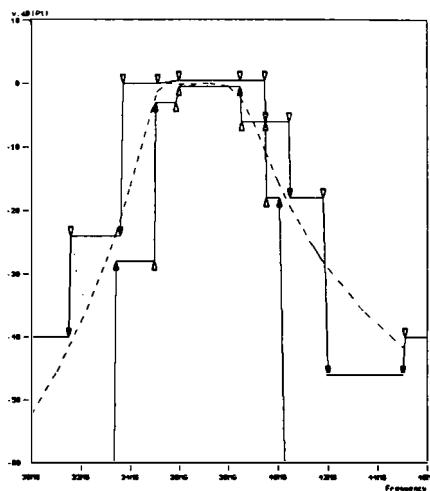


Figure 5. Graphical specification of a filter frequency response

8.2 Specification Entry and Optimisation

Although performance specification is recognised as a distinct activity in the field of software design, it is rarely the case in hardware, of which analogue circuit design is an

example. Prior to the availability of MOUSE, analogue circuit designers had used CAD systems which supported only the familiar manual modification-and-test cycle approach to design, so had not entered into an explicit dialogue concerning required circuit performance. Moreover, previous experimental optimisation systems had neglected the complexities involved in gathering specifications: usually text entry or form-filling was offered, requiring an understanding of the simulator syntax (Shyu and Sangiovanni-Vincentelli, 1988)

In MOUSE's Specification Mode, targets and limits can be attached to any previously-simulated aspect of circuit performance, without the need to redefine how these quantities are to be measured or to be sampled as functions of any independent variables such as time, frequency or temperature. A template of upper, lower and ideal specifications can be drawn in a special graphical editor against a 'backdrop' of the current simulated circuit performance (Figure 5), creating any 'don't care' regions where required. Templates may be defined by the user along any number of independent variables with mutual consistency in the specifications automatically enforced. Alternatively, performance limits may be generated by drawing an ideal response and giving the allowed deviation, or by typing values into a table. For nodal quantities, such as voltage, circuit locations can be picked on the schematic and specifications entered via pop-up menus. 'Meter' icons are then created on the schematic to indicate in a familiar form the current performance relative to lower, ideal and upper markers (Figure 6).

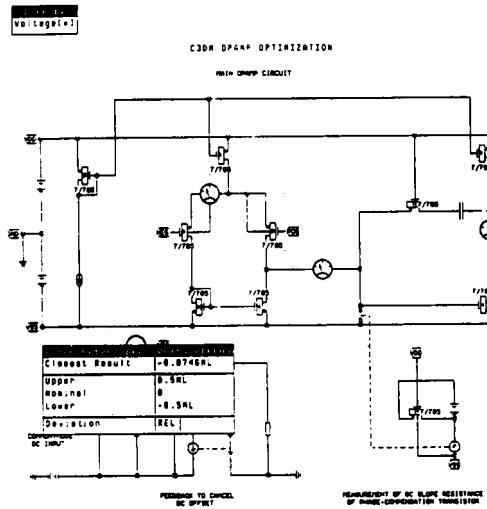


Figure 6. DC specification meters

Many optimisation algorithms can only minimise a single scalar objective function, which in CoCo is the discrepancy between desired and currently achieved circuit performance. This single number represents a balance between several, and often conflicting, sub-objectives, each having its own priority (or 'weight'). In the

Optimisation Mode these design priorities are represented by weights applied within an hierarchical sum of objectives. Weighting at each level can be set independently by pulling or fixing 'weight bars' on the screen (Figure 7). While a particular weight is being set, the bars alter dynamically to preserve relative ratios of other weights on a given level. Of course, the circuit may not be able to deliver the performance priorities entered as weights, in which case weights might be manually adjusted and the optimisation repeated. Special algorithms for multi-criteria optimisation may be more effective in achieving the desired balance of priorities than those using only a scalar objective function (Brayton, Hachtel and Sangiovanni-Vincentelli, 1981; Lightner, Trick and Zug, 1987).

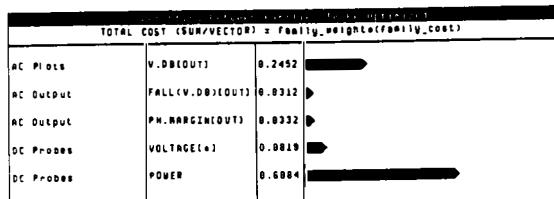


Figure 7. Performance family weighting

As an alternative to the use of weights to prioritise objectives, performance requirements can be expressed as nonlinear constraints. Constrained optimisation algorithms first try to satisfy these mandatory requirements before improving the figure of merit represented by the objective-function. Other quantities of minor concern to the designer can be passively monitored. Although these alternative representations are available, designers still sometimes find it difficult to set the relative stringency of conflicting circuit requirements. In manual design, they clarify such issues with Marketing after discovering the inherent trade-offs involved in a design. New user interface concepts to aid this dialogue are needed.

Design variables then need to be identified for the Optimiser to adjust - 'probing' the circuit diagram in the Optimisation Mode to achieve such identification proved natural for the designers. Parameters of individual components, global circuit factors, or variables of the manufacturing process can all be selected as design variables. The final stages of preparation for an optimisation include activating the Mathematical Assistant to advise on the choice of algorithm, its parameter values and termination criteria, and activating the Electrical Assistant to generate a set of alarms which monitor changes in the circuit caused by optimisation.

Rarely does the first optimisation provide a final design. Designers prefer to explore the potential of a circuit with a sequence of optimisations. Different design strategies might be employed, addressing the most difficult requirements first, or the most important ones, or those which are fastest to simulate. Sometimes it is more efficient to decompose the problem into a sequence of sub-optimisations rather than pose one single but very complex

optimisation. Further study of the interface required to support what is in effect a very high-level planning dialogue is indicated.

The MOUSE environment, with its simulator and optimiser, forms a usable product (Rankin and Siemensma, 1989) valued by designers. However, despite its sophistication, MOUSE gives no feedback apart from CPU usage during the optimisation. The designer therefore has no opportunity either to observe or to interact with the Optimiser in any way. These functions are provided by the Cockpit (Section 10).

9. KNOWLEDGE-BASED SUBSYSTEMS

A master designer never has the time or patience to voice all the constraints and factors which he/she subconsciously takes into account during the iterative process of manual design. Use of a general-purpose optimisation algorithm is therefore likely to produce a design which is unacceptable in the designer's eyes. To guard against this situation, a Lisp-based expert system has been developed. In accordance with the manufacturing technology, sensible bounds on design variables can be allocated. This 'Electrical Assistant' also runs rules to find certain types of component, functional component clusters, or nodes in the electrical network. Additional simulated circuit measurements and testing routines are generated which can then passively monitor the correct electrical functioning of circuit objects during optimisation without on-line interaction with the Lisp subsystem. In effect, a set of electrical 'alarms' are automatically 'wired' into the optimisation problem which can in turn signal the Cockpit of any violations. The designer then has the choice of ignoring the alarms, or modifying the optimisation problem to explicitly satisfy the extra constraints.

Experts in the mathematics of optimisation are too few to help every designer in person. An algorithm must be selected which is suited to the problem (Singhal et al, 1989), and other numerical details decided. A separate expert system has therefore been written in Smalltalk to encapsulate this mathematical knowledge. This Mathematical Assistant interacts minimally with the user as its inferences are in a domain which is foreign to the Electrical Engineer. The natural inheritance of common numerical strategies employed by different algorithms available in libraries is exploited to encode the knowledge required for appropriate algorithm selection, choice of values for its parameters and diagnosis of its termination condition.

These two knowledge-based subsystems act as pre- and post-processors for the CPU-intensive optimisation process, deliberately avoiding speed penalties incurred from their real-time interaction. Further details are described in a companion paper (Gupta & Rankin, 1991).

10. THE COCKPIT

It is through the Cockpit interface that the designer is able to monitor the progress of optimisation and, if judged appropriate, to intervene and change the performance requirements, the choice of design variables, the objective function weights and many other quantities including the circuit itself. Figure 8 shows a typical view presented by the

Cockpit, the subwindows of which will now be discussed individually. The overall concept for the interface was arrived at through prototyping in HyperCard™. Without the Cockpit, the designer would be overwhelmed by the mass of data produced during optimisation and could not interact with the design synthesis process.

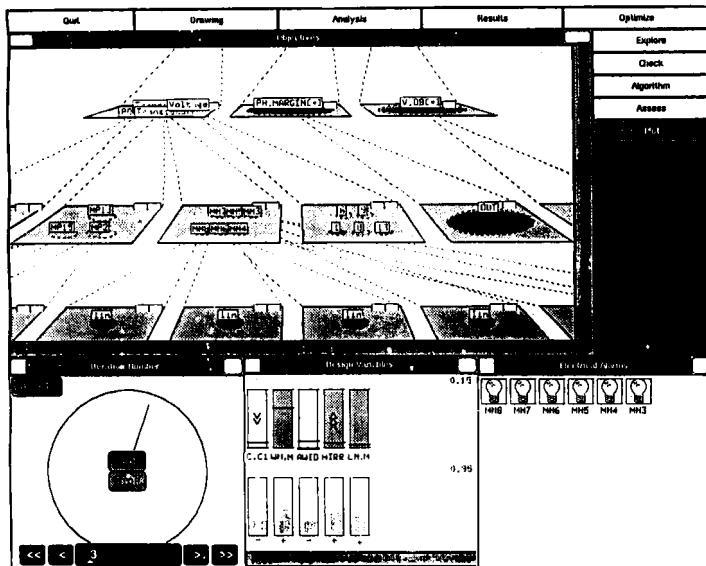


Figure 8. The Cockpit

10.1 Sheets

In the main window is a pseudo-3D presentation of a tree-structure composed of linked sheets. This structure mirrors the strong sense of hierarchy of issues in the designer's mind. One such sheet, positioned at the top of the tree, is shown in Figure 9. On the sheet, seen in perspective in Figure 9a, is a single circle, coloured red, whose area is proportional to the objective function, thus representing a measure of the discrepancy between desired and actual performance. It therefore shrinks in size in response to the optimisation until, ideally, it is of zero area. This circle represents, in qualitative terms, the highest level of abstraction of the properties of the designed object. The use of circles to encode data is not new, having been promoted by Sir Edward Playfair in 1801 (Tufte, 1983): we have added programmability, colour, dynamics and interaction.

10.2 Qualitative vs Quantitative Displays

If more detail is required, a click on the 'orientation' button will switch to a plan view of the sheet (Figure 9b), whereupon the numerical value of the objective function can be displayed. Additionally, via a pull-down menu, the history of the objective function over time can be recalled (Figure 9c). Thus, within a single sheet, two representational modes

are available and can be employed simultaneously: one offers a choice between qualitative and quantitative viewing, the other a choice between current state, recorded history and progress as it occurs. Suppression of detail, to revert to the display of Figure 9a, is achieved by again clicking on the “orientation” button.

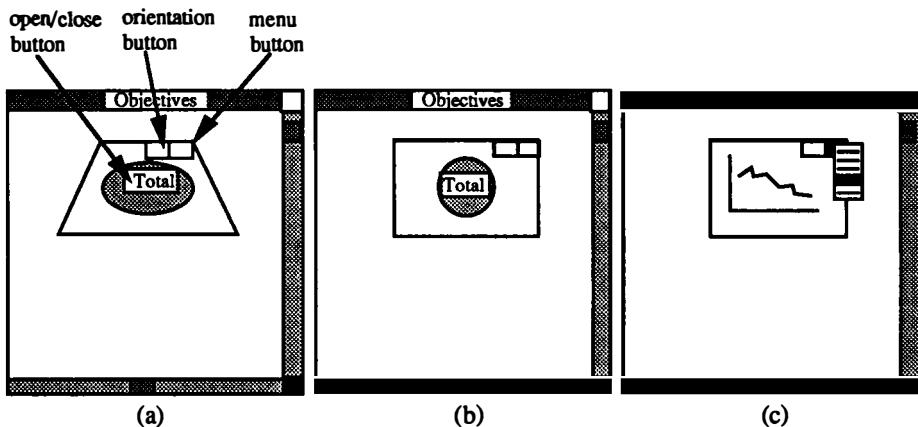


Figure 9. Sheet Functionality

10.3 Abstraction hierarchy

The objective function was composed from weighted components each reflecting different properties of the circuit: for example power consumption or speed. In turn, each of these sub-objectives may themselves be related to different properties, possibly associated with specific circuit locations, again with the relative importance expressed by weights, and so on. There may well be many such levels including, at the lowest level of detail, voltages and currents at different time/frequency sample points in the simulation domain. In this way we have a hierarchy of levels, sheets at one level being the parents of sheets one level lower. This hierarchy is made overt in the display illustrated in Figure 8.

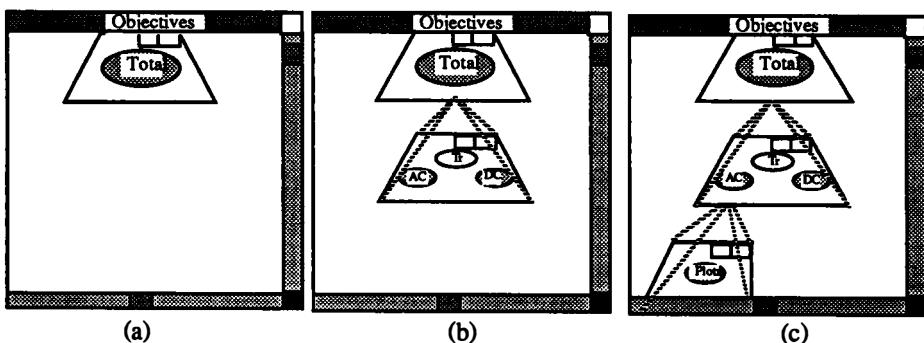


Figure 10. Customising the display of sheets

At any stage during automated design the designer may wish to focus selectively upon one or more sub-objectives at any level of detail. For this reason the Cockpit offers, in addition to an overall view of the hierarchy as illustrated in Figure 8, the ability to selectively display or suppress one or more sheets at a selected level. Thus, a click on the 'open/close' button (indicated in Figure 9a) will cause a sheet or a complete branch to be 'folded up' into its parent sheet. A further click on this button will unfold the display of all contributory sheets at lower levels. Figure 10 shows a sequence illustrating the opening up of a branch. In addition, the choice of detail illustrated in Figure 9 is available on all sheets, and provides an easy way to locate the residual sources of dissatisfaction in a design. At any one time, therefore, the residual sources of dissatisfaction in the current design can be rapidly located in the overall hierarchy.

If the hierarchy is extensive, the scrolling facility helps to avoid some of the problems associated with compressing too much information onto the screen. Zoom facilities are planned.

10.4 Objectives, Constraints and Alarms

The value of the objective function (or, at lower levels, of constituent objectives) is not the only information encoded in the hierarchy of sheets. It was mentioned earlier that, according to the importance the designer associates with them, design subgoals can be posed in three ways: as (1) hard constraints, (2) soft weighted objectives or (3) alarms. As all three classes of requirements may be present in a particular design, they must be displayed, though in a prioritised order, on the sheet hierarchy. In order to filter the information, the violation of hard constraints takes precedence. If all constraints on a sheet are satisfied then objectives are next in priority for display, alarms having the lowest priority. Thus, if a constraint violation occurs at a particular level it should be indicated (by a filled rectangle of a fixed size, in contrast to the objective circle) at that level and at all parent levels, and be accompanied by the suppression of the objective circle, as illustrated in Figure 11.

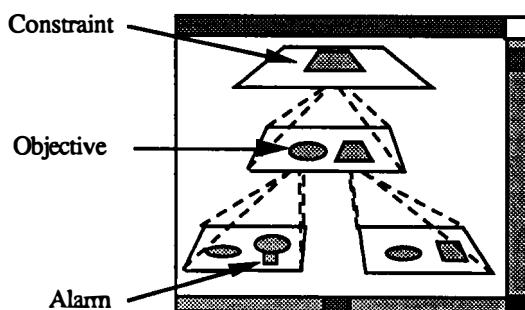


Figure 11. Precedence filtering to display circuit requirements.

A constraint violation noticed on a sheet can be traced down the hierarchy to see where the problem has arisen. Alarms, which constitute 'advice' open to interpretation by the

designer, are indicated on the sheet on which they occur, but their activation does not cause the suppression of either constraint or objective symbols on parent sheets. Thus, Figure 11 illustrates a situation in which the objective circle visible on the second sheet is suppressed on the top sheet. Since the sheet containing an alarm may currently not be visible on the tree (it may have been scrolled out of sight or folded up), the fact that an alarm has been activated can always be seen by the highlighting in the Electrical Alarm Window (Figure 8) of the Cockpit. Different classes of rules run by the Electrical Assistant produce many individual alarms, so these are presented, again hierarchically, in the Electrical Alarms Window, where text explaining the status of each alarm or group of alarms is available.

10.5 Sheet Geography

For global properties of the circuit which integrate a number of aspects of circuit behaviour, their representation as abstract circles, squares and lightbulb icons (representing alarms), is appropriate. However, as more detail is presented, designers tend to have geographic associations with their mental map of the circuit. For this reason a symbolic 'road map' view of the circuit can be used, when appropriate, to place encoded values on a sheet: in this way objectives can be more readily associated with the circuit locations at which they are significant.

10.6 Design Variables Window

Values of design variables need to be displayed so that the magnitude and direction of change can easily be perceived. Frequently, the discrepancy between the actual circuit performance and the desired performance is sensitive to only a few of these variables, and insensitive to others. In this situation, the user might wish to be made aware of this sensitivity and take over control, selecting new design variables or dropping weaker ones to improve the efficiency of the process. With many parameter attributes to be communicated to the user, the aim should be least obtrusion and minimal cognitive processing, so that the designer can get a feel for design variable activity at a glance. This necessarily involves presenting data so that trends can be identified. If a variable is always stuck against one of its bounds, for example, then the optimisation session may need to be restarted with different bounds, one less design variable, or a structural circuit change.

In the Design Variables Window (see Figure 8) design variable values, mapped between lower and upper bounds, are represented qualitatively by bars. The initial values are marked by horizontal lines, whilst design variables that have hit bounds are indicated by thickened lines. Trends are indicated by the numbers of chevrons on the bars (the number of chevrons representing the magnitude of change from the last value). Detailed quantitative data can always be accessed via pull-down menus, for example, to view graphs of values against iteration. This visualisation of design variable data reduces clutter on the screen by displaying much information in a small space, and encourages the user to engage in pattern detection: strong design variables whose values are regularly changing will exhibit many chevrons and be immediately apparent.

11. USER STUDIES

Before producing any new CAD system, the division of responsibility between roles best left to the human and those the computer should play must be clear. To ensure that circuit design activities were appropriately supported by the CoCo system we needed to increase and confirm our understanding of the circuit design process. Therefore, as the development of the CoCo system proceeded, we attempted to model the cognitive circuit design process. In particular we needed to establish those aspects of the process for which human cognitive limitations impede progress and those aspects in which a machine could overcome these limitations.

It should be noted that formidable problems are associated with any attempt to carry out user studies in a complex engineering field (Colgan and Brouwer-Janse, 1990). Not least is the fact that the design of a circuit usually takes place over a period of several months. Yet, to allow reliable inferences to be drawn, psychologists often advocate consistent, controlled experiments which, in the context of non-trivial circuit design, are simply not possible. In the spirit of the gradual enhancement which characterised the development of the CoCo system it was judged appropriate to undertake a continuous qualitative evaluation of evolving prototypes, as advocated by Carroll and Rosson (1985). Our evaluation procedures were refined as the project progressed.

11.1 Initial User Studies

Early in the project we investigated the conventional, manual approach to circuit design. Two subjects were used, one from an educational and the other from an industrial environment. One provided a retrospective review of circuit design by talking through a circuit he had designed previously. The other provided an introspective view of circuit design by designing a circuit and thinking aloud as he did this. Both sessions were audiotaped and transcribed.

11.2 User trials on MOUSE

When the first version of CoCo to offer optimisation was available, two activities were made possible. One was a dialogue with circuit designers to obtain feedback regarding their use of the system, an activity which is extremely important when no other product exists as a control or basis for comparison.

The other activity was a second set of user studies designed to extend our model of circuit design to include the availability of optimisation. Designers were asked to think aloud during design sessions, and their verbalised thought processes were audiotaped. Analysis of the data resulted in a design 'trace' from which a pattern of strategies, goals, subgoals and verification of hypotheses could be seen emerging in a cyclic fashion.

Data from both sets of studies have been mapped into a model discussed elsewhere in these proceedings (Colgan and Spence, 1991). Comparison of the resulting models has allowed us to suggest how the system affects the circuit design process and to make suggestions for refining the user interfaces, specifying the functionality required from the two knowledge-based systems and forecasting future enhancements to CoCo.

11.3 User Trials of the Cockpit

As it has evolved, the form and detail of the Cockpit display has benefitted from critical assessment by a small group of engineering designers and psychologists, some involved in its design. Planned for the immediate future is a structured interview with two designers in order to obtain a first independent assessment of the many details of the Cockpit presentation. Simultaneously, on the basis of some experience of Cockpit use, we have initiated the SuperCard™ implementation of separate facilities for laboratory-based experimentation of key aspects of the user interface. The Cockpit is now approaching a level sufficiently engineered to be tested in an industrial setting. A fair appraisal must be based on the use of the completed CoCo system to solve a real design problem, an exercise involving considerable effort by experimenters and subjects.

12. CONCLUSIONS

12.1 Engineering Design

Experience with the development of CoCo confirmed the need for a cognitive model of design on which incremental enhancements can be based and against which suitable metaphors can be evaluated. The resulting goal-plan model provided useful information in this respect and, together with other observations, has already suggested desirable changes and additions to the CoCo system. Extensions to the model to account for Cockpit use and the presence of advisory systems is, however, necessary. In view of the justifiable suspicion with which industrial designers regard 'toy' prototypes in the context of difficult design problems, we are satisfied from our experience that, despite the considerable engineering effort required to implement working prototypes, such effort was essential to permit a useful dialogue over the requirements for the CAD system. Note that working prototypes may have to reuse large bodies of existing CAD software and thus be constrained to follow the existing user interface style for consistency.

12.2 Evolutionary approach

Judging from the reactions of circuit designers to the introduction of optimisation into their design flow, the evolutionary approach involving the simultaneous prototyping of the system and its subsystems has proven to be effective. That this judgement is subjective is an inescapable consequence of the very nature of the invention and creation of interfaces to large, novel and complex tools. As Apperley (1990) has remarked,

The designer of the "interface-in-the-large" is likened to an architect who must bring experience and standard "techniques-in-the-small" to bear on the complete problem to produce a tasteful, appropriate, functional and watertight solution. Many solutions are possible - the 'best' is a subjective choice'

In the development of MOUSE, user trials have demonstrated, at least within CoCo, that one cannot approach the user until a serious kernel system is available with which to

tackle realistic design tasks. The overall success, therefore, depends heavily upon the vision of the architect who creates this initial kernel. Only after such a supportive setting has been implemented can deeper issues be progressively investigated, layer by layer. One consequence is that the approach can be seriously impeded if anything other than a 'clean', engineered system, free from system and interaction bugs, is offered at each user trial. Another, more serious difficulty, is posed by the gulf between the stage wherein feedback is obtained from a 'story-boarding' prototype (for example, using HyperCard™), and the stage at which a reasonably powerful and robust system can be presented to the designer in a user trial. There is a pressing need for effective prototyping tools at an intermediate level.

12.3 Achievements

Many outcomes of the CoCo project can be identified. One is the embodiment, within MOUSE, of a large measure of integration between manual and automated design, valued by the designers. Another is the provision, in the form of CoCo, of a catalyst and vehicle for the development of realistic user models of design. A third is a demonstration, involving a first version of the complete CoCo system, of a CAD system enhanced both by automated design and the provision of knowledge-based assistants. Additionally, the Cockpit has provided, for evaluation and later improvement and extension, a novel visualisation medium for engineering designers: it exemplifies the future convergence we see between supervisory interfaces for the monitoring and control of processes (for example, power plants or automatic pilots) and interfaces for the computer-aided description and simulation of designs.

12.4 Future Work

Our next goal is the generalisation, to other design domains, of our solutions for interactively coupling automation aids and knowledge-based assistance to manual design environments. In particular, the Cockpit concept shows promise for graphically supporting problem-solving in any complex domain where a hierarchy of issues is present, and may even be helpful in fields outside design (for example, database navigation). In the electrical field, it would be interesting to see whether an object-oriented metaphor for hardware design brings the same benefits as in software design.

Our attempts to raise the level of the designer's dialogue with a CAD system has led us into areas where performance specifications and priorities must be made explicit to the system. This has underlined the need for further work on interfaces to support trade-off dialogues and design strategy planning aids. In parallel with this, the continuing development and refinement of user models of design will ensure input from real circuit designers and deepen our knowledge of design.

The potential of the Cockpit as a configurable starting point for design must be evaluated. It could provide a medium in which to formulate the optimisation problem as well as observe its progress. Presented with a dynamic hierarchy, the user might select and move symbols until the appropriate hierarchy has been configured. In this way, the Cockpit provides a configurable, object-oriented view of design issues which may help the designer accurately express and manipulate a design problem in its early stages. This crucial concept-forming phase of design is poorly supported by present CAD tools.

Acknowledgements. Paul Jennings and Steven Hart gave useful advice during the programming of MOUSE by Keith Hollis. We are indebted to Dr. Maddy Brouwer-Janse for her guidance in user-evaluations, and to Professor Mark Apperley for helpful discussions. Financial support for the research was provided by Philips Research Laboratories, and by the United Kingdom's ESRC/MRC/SERC Cognitive Science/HCI Initiative (Grant number SPG 9019856). Finally, without the trust, assistance and sympathy for our aims shown by many circuit designers this work would not have been possible.

REFERENCES

- Apperley, M. D. (1990). Practical interfaces to complex worlds. (Panel) *CHI'90 Conference Proceedings* Seattle, Washington, April 1-5, pp 257-260.
- Arora, J. and Baenziger, G. (1986). Uses of Artificial Intelligence in design optimisation. *Computer Methods in Applied Mechanics*. 54:303-323.
- Baker, K. D., Ball, L. J., Culverhouse, P.F., Dennis, I., Evans, J.St.B.T., Jagodzinski, A. P., Pearce, P.D., Scothern, D.G.C. and Venner, G.M. (1989) A Psychologically Based Intelligent Design Aid, *Eurographics Workshop in Intelligent CAD*.
- Brayton, R.K., Hachtel, G.D. and Sangiovanni-Vincentelli, A. (1981). A survey of optimization techniques for integrated circuit design, *Proc IEEE*, 69(10):1334-1362.
- Brooks, F. P. (Jr.) (1988). Grasping reality through illusion. Plenary Address. *CHI'88 Conference Proceedings* Washington, DC, May 15-19, pp. 1-11.
- Carroll, J. M. and Rosson, M. B. (1985). Usability Specifications as a tool in Iterative Development, in: H. R. Hartson (ed) *Advances in Human-Computer Interaction*. Ablex, Norwood, New Jersey.
- Colgan, L. and Brouwer-Janse, M. (1990). An analysis of the circuit design process for a complex engineering application, *Interact '90*, Cambridge, UK, 27-31 August, pp. 253-257.
- Colgan, L. and Spence, R. (1991). Cognitive modelling of electronic design. These proceedings.
- Gupta, A and Rankin, P. (1991). Knowledge assistants for design optimisation. These proceedings.
- Ketonen, T., Lounamaa, P. and Nurminen, J. K. (1988). An electronic design CAD system combining knowledge-based techniques with optimization and simulation, in J. S. Gero (ed.) *Artificial Intelligence in Engineering: Design*, Elsevier, Amsterdam.
- Lightner, M. R., Trick, T. N. and Zug, R. P.(1987). Circuit Optimization and Design. in A. E. Ruehli (ed.) *Circuit Analysis, Simulation and Design*, Elsevier, North-Holland, Amsterdam.
- Rankin, P.J. and Siemensma, J.M. (1989). Analogue circuit optimization in a graphical environment', *Proc. IEEE ICCAD'89 Conf.*, Santa Clara, November, pp. 372-375.
- Singhal, K., McAndrew, C.C., Nassif, S.R. and Visvarathan, V. (1989) The CENTER Design Optimization System, *AT & T Technical Journal*, May/June, pp. 77-90.
- Shyu, J.M. and Sangiovanni-Vincentelli, A.L. (1988). ECSTASY: A new environment for IC design optimization, *Proc. IEEE ICCAD-88*, Santa Clara, November, pp. 484-487.
- Spence, R. and Apperley, M. D. (1977). The interactive-graphic man-computer dialogue in computer-aided circuit design, *Trans IEEE on Circuits and Systems*, CAS-24, 2:49-61.

Tufte, E. R. (1983). *The Visual Display of Qualitative Information*, Graphics Press, Cheshire, Conn., p. 43.

An intelligent tutorial system for computer aided architectural design

P. J. Scott,[†] B. R. Lawson[‡] and J. Ryu[§]

[†]Department of Psychology
The University of Sheffield
Sheffield S10 2TN UK

[‡]Department of Architecture
The University of Sheffield
Sheffield S10 2TN UK

Abstract. This paper presents a prototype system called ICADT developed at Sheffield University which uses artificial intelligence and flexible learning techniques to help teach users about the use of the GABLE 4D™ Computer Aided Architectural Design environment. The system generates small scale HyperCard™ tutorials from a central knowledge base as they are required by the user to explore some aspect of the CAAD environment. This project draws upon ideas from a number of areas: as in conventional Intelligent Tutoring Systems, the ICADT project keeps an internal model of each user and type of user. This model is used not only to keep track of the progress of users but also to help control future interactions; like many Intelligent-Help systems the project works from a simple query by the user about a topic upon which they wish to receive instruction; and like Hypermedia work the output of the project is a flexible learning 'stack' of instruction with a number of paths through the material available to the user. However, the core of the project is an AI system implemented in Prolog which maintains a conceptual network of knowledge related to the CAAD system, and from which it generates the tutorial materials.

INTRODUCTION

Computer Aided Design as applied to architecture (CAAD) includes the creation of three dimensional models of design proposals. Such models may be used to investigate the appearance, organisation and, in some cases, performance of buildings or groups of buildings. CAAD therefore has the potential to create a fundamental change in the way designers work and think, and its advent has introduced major training and re-training needs for designers. Inevitably, good CAAD systems are now very sophisticated and

consequently complex. Thus no matter how well designed the user interface, the learning process is non-trivial. Often the problem facing a new user of CAAD is not just in terms of understanding the syntax of commands and what they do, but rather knowing how the system works, what concepts it employs, and what combination of commands will achieve a particular design end. This problem can become quite severe in the case of students who do not have the time, nor their institutes the resources, for expensive training courses (see Lawson 1986). In many cases, design schools not specialising in CAAD research, also have little in house staff expertise.

These learning problems are compounded by the fact that designers in general, and architects in particular, can find the process of learning to use a CAAD system a rather frustrating experience. This frustration arises from an unfortunate mismatch between the learning style of a designer and the tutorial style of existing CAAD packages. Designers are generally educated through a process of 'learning by doing'. They expect to use their creativity to solve a series of problems which in turn bring forward a series of theoretical issues. CAAD training courses have generally not been written by design educationalists and offer instruction on a system-oriented basis, thus failing to identify the real and varied needs of their users. In functional terms, CAAD users may be casual or regular, and may input data or only exploit data. In role terms, they may be design executives, office managers, technicians, draftsmen etc. Thus it is possible to see a whole range of subsets of system features on which a user needs training, which will vary according to the organisation of the individual design office.

It is therefore an urgent priority to develop and evaluate more flexible approaches to CAAD learning. The project discussed here builds upon the substantial existing resources in terms of CAAD expertise, software and hardware deriving from the GABLE 4D™ CAAD system developed in the Sheffield University Department of Architecture, and now available throughout higher education in Britain. Training architecture students to use the CAAD system is an integral part of an architecture degree and traditional methods of teaching GABLE by means of lectures, tutorials, worked examples, and hands-on experience with online help are already well-established, but they are expensive on time and resources as they rely on intensive use of expert human demonstrator assistance.

Designers expect to use their creativity to solve design problems and are resistant to fixed tutorial material that has been prepared to introduce concepts in a structured and rigid fashion. Both students and commercial users of such a system require not a syllabus, but rather support when they encounter a specific problem for a specific purpose. Conventional on-line help or Computer Assisted Learning (CAL) support packages are woefully inadequate to this task. They are insensitive to different users and to the same user's changing needs and it is a major research problem to introduce a useful context sensitivity

mechanism. One of the most serious limitations of this existing technology is the difficulty it poses for the material's author who must write general help for specific needs about a complex system which can change rather quickly. Unfortunately, the sequence of instructional material in traditional computer teaching is 'frozen' in computer presentation code. Recent research into authoring systems (Nicolson, Scott and Gardner, 1988) has indicated that a much more powerful approach is to generate computer teaching specifications at the higher level of 'design pseudocode'. This specification is then used to generate specific teaching material in whatever form is required. The level of design pseudocode for CAL is seen as an appropriate intermediate level upon which to reason about specific forms of CAL presentational program, but it is not the highest level of analysis. An even higher level of the authoring process lies with the specification of the knowledge-to-be-taught as a network of interconnected concepts. The acquisition and exploitation of this conceptual network knowledge base remains the central problem for Intelligent Tutoring Systems (ITS) research (see for example Sleeman and Brown, 1982). Most tutoring systems have a network of concepts to represent the domain, a much smaller net to represent the student's knowledge, and a large collection of heuristics which can be used to generate some teaching interactions. The idea of a computer tutor for a CAAD environment is clearly an ideal solution to the teaching problem. A computer ITS would act much as the human demonstrator, observing and critiquing user performance, offering help and advice wherever appropriate. Unfortunately this ideal appears to be well beyond the current state of the art. Current ITS projects are only successful in limited ways in experimental systems and in carefully selected domains. The domain of design is not yet easily tractable to an AI analysis. Nevertheless, recent research has suggested that some aspects of this ITS development, short of a full-scale tutor, can be applied directly to the computer teaching authoring problem. We show one approach to this in our tutor.

The key innovative aspect of this project is in the use of AI techniques for CAAD knowledge representation and for user modelling to create a prototype computer-based Intelligent Tutoring System which allows CAAD users to acquire system skills in a manner more sympathetic to their learning style, more appropriate to their functional tasks, and more suited to their organisational needs. The prototype system, the Intelligent CAD Tutor (ICADT), is implemented in Prolog and HyperCard™ and runs on an Apple Macintosh™ micro. It supports work on the GABLE 4D™ CAAD environment on the Apollo Domain.

The GABLE system, originally developed at Sheffield University, comprises a number of sub-systems for handling two dimensional drafting, abstract object modelling, intelligent building modelling and terrain modelling. The systems all interact, can exchange data, and share a common interface design and command structure both in terms of dictionary and syntax. The most typical and representative of all these sub-systems is the Object

Modelling System (OMS) which has normally been taught first to undergraduate architecture students at Sheffield University. The prototype ICAD Tutor has therefore been developed primarily to assist with the teaching of this sub-system.

The Needs of CAAD Users

We have identified four forms of knowledge needs across all classes of system user : concept-based; task-based; design-based, or help-based.

(i) The first approach to tutorial generation is driven by a **concept-based** analysis where the user wants to explore a known sub-area of the systems knowledge. For example, the user may simply ask about the use of the 'move' command or the concept of 'dimensioning' elements.

(ii) More usually however the user is engaged in a specific **task** that requires a range of concepts which may not necessarily be known to the user. So, for example in teaching about the use of the CAAD system to produce different types of 3D views (perspective, isometric etc.) from a user's design model, the user must follow a series of stages (setting lens angle, specifying target and eye points, etc.) each of which may require the knowledge of other commands and concepts.

(iii) If the user cannot identify a task or concept they may be aware that their problem is *analogous* to some other **design** problems that they can expect the system to recognise. They might reasonably expect to be able to indicate a drawing as an example and have the system infer from this the tasks and concepts involved.

(iv) Finally, the tutor might be expected to constantly observe and evaluate the user's performance and compare its assumptions about their goals and solution paths with its own. If so then it might offer general **help** even if users did not know they needed it.

These forms of knowledge can be seen as progressively more complex and requiring progressively more sophisticated technical solutions up to a full Intelligent Tutoring System which is able to respond to a help-based need. We have begun to prototype solutions to the above needs in a number of complementary projects. Here we report on the first and the second of these. The latter two forms of knowledge are the subject of a further research programme which is currently underway.

THE PROTOTYPE ICAD TUTOR

Architectural Computer Aided Design as it is instantiated in the GABLE 4D™ system is a very interesting problem domain: it has a very large space of knowledge; this knowledge is about a closed environment (a micro-world); the knowledge is reasonably uniform, but structured and accumulative; it has a well specified syntax, semantics and pragmatics of

commands and actions; and finally, unlike many tutorial domains it is characterised by user and task driven learning where there is no fixed pedagogy desirable or enforceable. These domain characteristics make it particularly suited to the approach we are using here.

Recent approaches to the development of intelligent courseware (such as, for example, the feature networks of Webb, 1988) have advocated the representation of the knowledge to be taught as a semantic network. A tutorial is then drawn from this network as a 'fragment' of knowledge and presented to the user. The user's state of knowledge can be modelled as a subset of this network and can, in principle, be used to guide tutorial selection and assembly and can be updated according to user performance.

The current ICAD Tutor system consists of four modules: a concept network knowledge base; a tutorial builder system; a user record maintenance system; and a tutorial presentation and performance monitor. This project prototypes each of these modules to produce a working version of an architectural CAD tutor. In the experimental system the first two modules are implemented in Prolog and the latter two are implemented in Apple's HyperCard™. These systems cohabit under the Macintosh Multifinder and pass control messages to each other via an external file interface. The basic structure of the system is illustrated in figure 1.

The **concept network knowledge base** models the necessary CAAD knowledge - high level conceptual descriptions of tools, commands, tasks, features of drawings and so on. This occupies the upper half of figure 1. Combined in the tutor are three classes of knowledge: concepts of the GABLE commands specifically; more generally about Computer Aided Design; and most generally about principles of architectural design. These concepts are all related together as functionally or structurally dependant, more or less primitive and so on. Coupled with each concept is a primitive instructional unit, or topic, from which tutorial sessions may be assembled. These topics are currently implemented as simple screens of graphical and textual information, stored in topic data-files, giving help and instruction about specific aspects of the concept. Clearly the *development* of this network and its *maintenance* is a major project in its own right, and one which is receiving a great deal of interest in the AI community. The current project has developed a concept network of only a couple of hundred topics. The topic primitives are produced using the GABLE IDS (Integrated Drafting System) module. We envisage the development of a support system to assist with the maintenance of a larger set of knowledge as a future research project. This is outlined in dashed lines in figure 1.

The **tutorial builder system** extracts a fragment of the concept network which is appropriate to the user's current needs and state of knowledge, and along with the net itself is the main subject of this current research. This system is labelled 'Tutorial Builder' in figure 1. Two of the forms of knowledge needs across all classes of system user identified

above have been included in the system: concept-based and task-based needs. The *concept-based* approach to tutorial generation is appropriate where the user wants to explore a known sub-area of the system's knowledge. More usually however *task-based* knowledge is required - the user may want to perform a specific task without putting too much efforts on understanding all related concepts. What is required is a series of instructions to follow preferably step-by-step manner. However, for both of these needs the user must know the correct question to ask of the system.

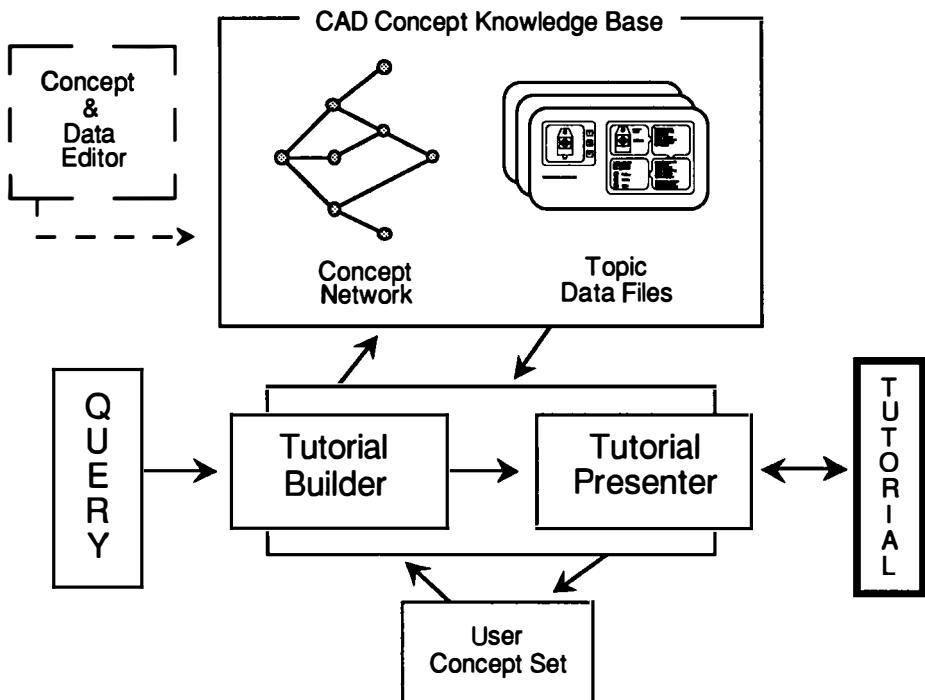


Figure 1. The Structure of the ICADT System

The **tutorial presentation and performance monitor** presents the tutorial, assesses user progress and updates the user model. This system is labelled 'Tutorial Presenter' in figure 1. The presentation and performance module could clearly be intelligent itself in its control of user interaction, and indeed this is the level at which most existing tutoring research effort has so far been expended. However, the current version of the tutor includes no real performance monitor as it is physically distinct from the user's performance on the CAAD workstation - future work is aimed at integrating the two systems. The **user record maintenance** mechanism models each user's current knowledge as a subset of the system concept network. This is labelled 'User Concept Set'

in figure 1 as it consists primarily of a list of the topics that the user has viewed in their history of interaction with the system. This model has predictive value towards monitoring the teaching process, and must allow for a range of different users. The range of system users is very great, from second-year undergraduate architecture students to the managers of commercial architectural practices, and each class of user may need a different sort of model. For example, the 'knowledge fragment' elicitation algorithm used to develop a tutorial in the specification system above may need to be different for different classes of 'student' user. The development of sophisticated user modelling facilities is central to all intelligent human-computer research, so the work reported here relies on only the most simple interpretation of this model.

As can be seen from figure 1 the input to the system is a system topic about which instruction is required. This is the user's query about some aspect of the system. The output is normally a Hypertext tutorial linking up the query topic with an appropriate network fragment for the user's perusal. Appropriateness is judged by the tutorial builder with reference to the current topic set which is stored for this user. The network fragment is shown to users graphically and is used as a 'Map' through which they may explore the connections of this concept inside the knowledge net for the system.

CONCEPTS AND TUTORIAL TOPICS

The basic element of the ICADT system is a 'concept' which is a single word or short phrase denoting an important primitive chunk of knowledge about the design environment. Examples of simple concepts range from general knowledge about features of the system such as the shapes and role of "the screen cursor" to the use of specific commands like the "align" verb. These concepts are richly interconnected with each other in a conceptual network. All the concepts in this network are directly related to TOPIC units which can be seen as their explanation. Topic units are basically screens of information, stored in topic data-files, giving help and instruction about specific aspects of the concept. The topic is always headed by a single graphic summary which is an iconic representation of the topic with the minimal text required to discuss it.

Figure 2 shows an example of the topic summary for "the screen cursor" concept. Whenever requested, the topic card can provide a number of further explanations - usually screens of text. Some of the textual entries are supplemented by "hot-words" and glossary entries on specific pieces of text. A hot-word is a textual link from some simple word or phrase to a further explanation of the issue, whilst a glossary is a simple collection of definitions of some of the terminology which is used in the text. Both are accessed via the Macintosh's excellent WIMPs point and click facilities.

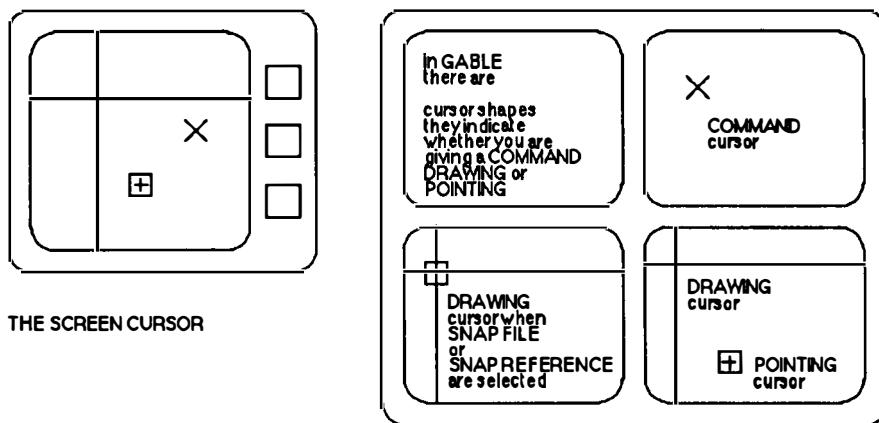


Figure 2. A topic summary.

THE CONCEPT NETWORK

The concept network is implemented as a large Prolog program of database facts that record the set of concepts which the system designers have encoded and the relationships between them. There are currently seven different classes of link in the network.

Examples of conceptual links:

MIXING COLOURS	acts_on	COLOURS
LINESTYLE	attribute_of	LINE
DIMENSONG TO THE GRID	expanded_by	DIMENSONG COMMANDS
SPECIFY ELEVN LINE	illustration	SPECIFY
ALIGN	involves_act	POINTING
SLIDE	is_part_of	DIMENSONG COMMANDS
POINTING	needs_concept	THE SCREEN CURSOR

In the examples given above: the mixing of colours acts upon or has power over colours; lines have attribute called linestyle; the concept of dimensioning to the grid is expanded by the more general concept of the dimensioning commands; an illustration of the concept specify is the multiple concept of specifying an elevation line; the act of aligning involves the act of pointing; the concept of the slide command is structurally a part of the system's dimensioning commands; and finally the basic connective link "needs_concept" states that pointing cannot be fully understood without prior understanding of the concept of the role and function of the screen cursor. Now some of these links may be read as tutorial prescriptions (such as the latter which notes that screen cursor concept should be taught

before the pointing concept) but most are simply structural or functional comments about the system and its use which could be used to lead the learner (for the stated reason) between the given concepts in exploring the system.

Each network link is explicitly marked as suited to a certain level of tutorial user. The fragment extraction algorithm can only use topics that are within a certain range around the current level of the user. In this way the complete novice is protected from confusing (advanced) detail and the expert is protected from irrelevant (introductory) detail. It should be noted that all information is available to users who ask specifically about it, but the tutor decides on how suitable each link is to explain another topic with this user-level metric.

INTERACTING WITH THE ICAD TUTOR

There are essentially five stages in a typical interaction with the ICADT system. First the user must log on to be identified; then they must issue a query of the system; the query must connect with a fragment of the concept network; this fragment must be turned into a tutorial; which must finally be presented to the user. There is also an underlying help system that the user can access at any time. The help system and all of the user's other contact with the system is conducted via the Apple HyperCard™ interface.

1. Identify the User. The system maintains a record of all interactions with it as a separate logging facility for experimental purposes - to help us evaluate its use. However it also keeps an individual record for each user that effects their interaction. So it is important that the user both logs in and logs out of the system so that their individual record is correctly maintained. For the evaluation of the system which is currently underway, the system is being used by small groups of students who are learning about the CAAD environment. In this case each group has a separate group user name which identifies their collective user model. The user model is stored in an external file where it may be independently accessed by both the tutorial presentation system (HyperCard™ front-end program) and the tutorial specification system (Prolog background program).

2. Query. The main purpose of the system is to enable the user to ask a sensible query about some aspect of the design system. The current interface simply provides the user with a list of the concept names that the system knows about and offers them the opportunity to select one of these as their query. This query is then passed to the Prolog background program for tutorial construction.

3. Select the Network Fragment. The job of the Prolog system is to take the user's query topic together with other relevant information which may affect the contents of the tutorial (ie user level or a list of already seen topics) and take out from the concept network a fragment of concepts around the query concept which may be used to help explain it. At the moment the system grows network connections around the query guided

by some simple pruning heuristics and the user level flags on each concept link. All of the links have a direction which can be used to indicate which concepts are dependent on the current one. So the links are typically seen as a collection of concepts conceptually interconnected with the query concept and upon which understanding about it depends. The selected network fragment is basically a lattice but suitably pruned, it may be drawn as a tree with the query as its root and with some linked concepts as the branches and leaves. Concepts which are already part of the current user's model are also included but are marked as 'seen'.

4. Construct Tutorial. The tutorial network fragment thus selected is then passed over to the HyperCard™ tutorial presentation system. Each node in the tree has an associated set of topic units the first of which is a simple iconic topic summary. The appropriate topics are collected together from the topic database - a large collection of independent files. The network fragment is itself drawn as a tree and each node of the tree is connected via a Hypertext link to the topic summary card. Already 'seen' topics are marked as secondary on the map - so that the 'new' topics are clearly salient. Next the concept links between the nodes in the tree are used to link up all the topics. The tutorial is complete.

5. Manage Tutorial. Control is finally returned to the user who may then explore the topic region that is made available to them. The key features of the tutorial product from the user's point of view are that it is a graphical Hypertext with an overview map (Figures 3 and 4) to guide navigation, explicit vertical and horizontal links between topics, hot-word and glossary links.

Figure 3 shows the overall tutorial map that is presented to a user who is rated as a beginner who has entered a query about the CAAD system command "Align". The query concept is presented at the right of the screen with a tree of prior and explanatory concepts connected to it. The user is able to go directly to the topic for any of these concepts by clicking with the mouse cursor on the relevant node of the tree. However they are encouraged to move from left to right through these concepts. It should be noted that concepts that the user has already encountered (having "seen" their topic explanations - possibly in the context of some quite different tutorial), are highlighted with a box.

The tree shown in Figure 4 is the tutorial map for the same query but for the next level user and illustrates that the "low-level" topics are removed. Also a slightly more sophisticated topic "Undo" that complicates the explanation has been added. In principle the tree drawing algorithm can cope with very large trees which it scales as necessary - but the pruning algorithm (and the nature of the stored knowledge currently in the database) means that very large tree explanations are avoided.

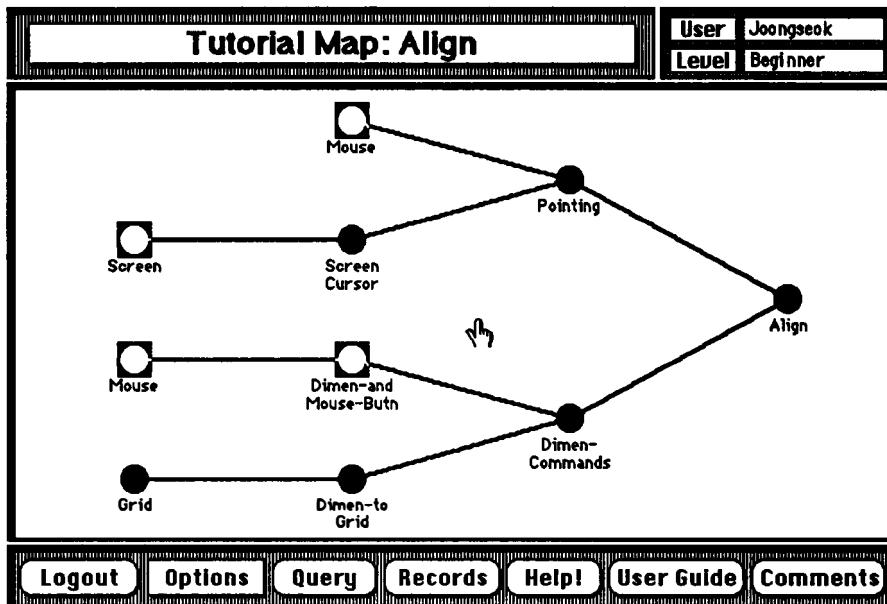


Figure 3. A tutorial map constructed for "Align" for a beginner level user.

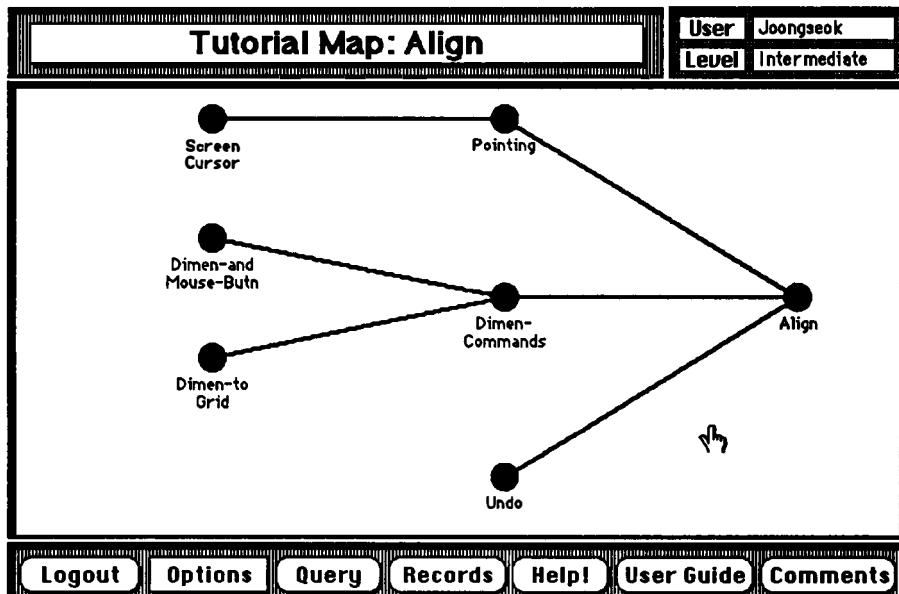


Figure 4. A tutorial map for "Align" for an intermediate level user.

The buttons ranged along the bottom of the screen in figures 3 and 4 illustrate the range of options open to the user at this point - aside from clicking on the tree itself. There are two sorts of button in the tutor: the rounded rectangle buttons are clicked once to have an effect; the rectangular buttons are clicked to reveal a sub-menu of further choices. From here users may simply log out of the tutor; change some options which are preferences about how the system works; they may execute another query; look at the records of their performance; ask for help; go to the tutor's user guide; or make some comments on the workings of the system. The most typical action at this stage is to select a node in the tree to examine. The result of selecting a node on the tree is to take the user to the topic summary card for this concept. An example card is shown in figure 5. The main options available to the user from here are again provided in the row of buttons along the bottom of the screen. They may return directly to the map at any stage. They may proceed left or right through the tree. In both figures 5 and 6 the button to move "Left" through the tutorial tree is greyed out, indicating that it is an unavailable option as the user is currently at a 'left-most leaf'. This is a rectangular button, meaning that there are alternatives for the user to choose, because moving left may involve a choice of more than one branch. The button for moving right is rounded as it can only involve one choice because it is always heading towards the 'root'. The next button "User Guide" will provide them with help on the use of the tutorial system. Alternatively they may wish to find out more about this particular topic through the "More Info" button. Moving down into the current topic usually leads the user into some text which says more about the meaning of the iconic picture given in the topic summary. An example from the simple "Screen" topic shown above is given in figure 6. The text can be quite extensive and is shown in a scrolling field. Some parts of the text may themselves require further explanation. Simple critical words or phrases are marked with an asterisk denoting that they are hot-words that have links to further simple entries - usually these are glossary definitions of the word's meaning in the CAAD system.

Overall, the user need never be aware that the tutorial is constructed dynamically from a distributed database representation. When the user has finished with the tutorial they may issue a further query or log off and in either case changes to the user's model are recorded (according to their use of the tutorial) as are any comments or suggestions they have made during the use of the system.

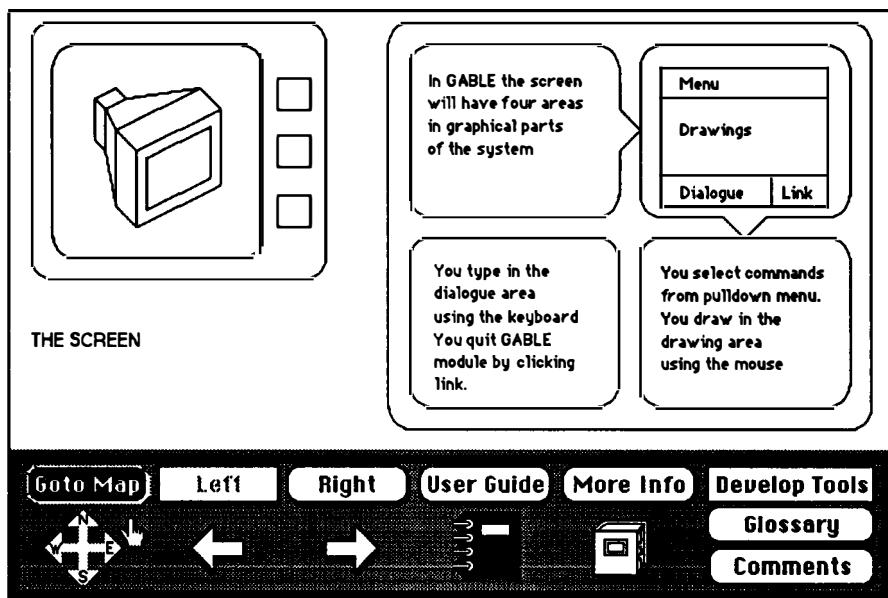


Figure 5. The "Screen" topic summary card.

The GABLE* screen is divided into four major areas.

THE MENU

This is the area where you point and select most commands. Some commands are placed at the corners of the drawing screen as icons (ex: ZOOM, PAN, FULL/ALL REDRAW).

THE DRAWING WINDOW

This window contains all objects you have added. You can have several windows at the same time. At the bottom of the drawing window there is a status line which displays current status of the drawing (scale, snap, tolerance* etc). If you click any of these items you can change the current status.

[Goto Map] Left Right User Guide Hide Info Develop Tools
Glossary Comments

Figure 6. An example of "More Info" for the "Screen" topic (text and hot word links).

ADVANTAGES OF THE CONCEPT NETWORK APPROACH

Firstly it is important to note that this is a very simple and yet powerful technique. A set of concepts-to-be-taught (learned about) and the explicit connections between them is at the core of all systems aimed at intelligent tutoring. This project has adopted a version of this technique and shows how it may be most easily applied. As an authoring technique it bestows three principle advantages: a) It frees a tutorial author to concentrate on higher-level issues (ie specifying the knowledge-to-be-taught) rather than the low-level presentation details (how each screen should appear); b) One investment of authoring time covers a vast range of possible tutorial material, because teaching is generated by system for individual's learning need when it is needed; c) The teaching resource is much easier to change, it is not 'frozen' into CAL program code. As a tutorial is generated from a uniform base - the base information may be upgraded and changed in a piecemeal and incremental fashion.

Secondly, the approach offers two principle pedagogical advantages; a) The system supports individualised teaching based on the record of user knowledge as subset of all domain knowledge; b) One representation covers a large range of possible presentational styles. We have chosen to emphasise the production of Hypertext-like tutorials that have a large element of user control, but many alternative forms for the product could be envisaged - chunking the information and controlling its access and use in different ways.

Finally it is important to note that we are benefiting from a couple of pragmatic advantages also. To begin with the problem is tractable because it relies on robust state-of-the-art techniques rather than inviting the complexities of most tutoring system projects. Secondly, all of the techniques used are readily and incrementally extensible, offering great possibilities for future developments (eg. a better user model, an improved interface including some natural language input, etc).

The tutorial system is currently undergoing an evaluation which involves comparing the performance and understanding of second-year undergraduate university students who have been tutored with and without access to this system. There are both qualitative and quantitative aspects to the evaluation where students' feelings towards and performance in the CAAD system will be examined.

FUTURE DEVELOPMENTS

Many simplifying assumptions have been made here to develop a prototype tutor that provides a practical solution to a real problem. The scope of the knowledge covered is very narrow; the model of the user employed is quite limited; the initial interface with the program - via a single word query that must be an existing concept - is primitive; we do not

fully exploit the richness of the knowledge net; and the quality of the levels of 'explanation' which are generated is uneven. The process of future development of this prototype involves the gradual relaxation of some of these limits and assumptions.

However, in general we have provided a reasonable solution to the representation and access of 'concept' and 'task' based knowledge. The most challenging future developments lie in the development of design based and help based systems. We have argued earlier that one of the needs of users of an ICAD Tutor is to learn in a more design-based way. That is, in a situation where the user has an idea of some design or subset of a design which they want to model on the CAAD system. The problem facing them is 'how do I use the system to create this design?'. Ideally the user would like to look through a catalogue of designs, find one with aspects sufficiently like their problem and then be tutored on the aspects of the CAAD system directly relevant to constructing that design. This is a very ambitious goal which is made more credible by some of the powerful features of the current environment in which we are working. The GABLE™ CAAD system allows the user to record all of the actions involved in performing any task and store this information in a 'record/playback file' for future use. This record file is a complete record of their design activity that can even be re-executed/played back to reproduce the design. We envisage a front-end to the tutor which can take the recording of the file indicated as the analogy by the user, recognise the commands and command structures which are responsible for the features chosen and hand on a set of concepts to ICAD Tutor for the construction of a suitable tutorial. This tutorial would be built in the manner already described earlier in this paper, pruned according to the user model, and displayed as normal.

We recognise that CAAD users learning through the techniques described in this paper are likely to get stuck whilst actually using the CAAD system since they are quite likely to perform actions beyond their current understanding. Such a situation may be very difficult to recover from without expert help. We are therefore also conducting a parallel project to that described here to develop an expert 'help based' system which monitors the users actions in the CAAD system by means of the recording file. The objective of this project is to instruct the user how to recover the situation in order that they may continue without the serious loss of motivation which inevitably results from data loss. Again, we expect this expert help system to be able to generate queries for the ICAD Tutor should the user wish not only to recover from a problem, but to understand both the recovery procedure and how the situation developed. We hope to report upon the expert help system and its links to the ICAD Tutor at a future date.

In conclusion then, the tutorial system we have discussed here is aimed at providing an economical and innovative solution to the problem of teaching about computer aided architectural design using artificial intelligence techniques.

REFERENCES

- Lawson, B. R. (1980) *How designers think*. Architectural Press.
- Lawson, B. R. (1986). Teaching CAD at Sheffield University. *Proceedings ECAADE Education in Computer Aided Architectural Design in Europe*, Rome September, 78-87.
- Nicolson, R. I. and Scott, P. J. (1986) Computers in Education: the software production problem. *British Journal of Educational Technology*, 17(1), 26-35.
- Nicolson, R. I., Scott, P. J. & Gardner, P. H. (1988) The intelligent authoring of computer assisted learning software. *Expert Systems*, 5(4), 302-314.
- Sleeman, D. and Brown, J. S. (eds) (1982) *Intelligent Tutoring Systems*. Academic Press, London.
- Webb, G. I. (1988) A knowledge-based approach to computer-aided learning. *International Journal of Man-Machine Studies*, 29, 257-285.

Designer's Workbench: a tool to assist in the capture and utilization of design knowledge

B. A. Babin and R. Loganantharaj

Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette LA 70504-4330 USA

Abstract. In most cases, the result of a design activity is only the final detailed design of a desired object. The knowledge utilized while developing the design for an object is discarded after the design has been completed. Recently, researchers have begun to focus more attention to the issue of keeping this design knowledge. The process of acquiring and representing this knowledge is generally referred to as *design knowledge capture*. To address the knowledge capture problem, we are developing the Designer's Workbench which will assist in the design of complex objects. The Designer's Workbench uses the theory of plausible designs with a few extensions to capture a significant amount of design process knowledge. The extensions serve to make retention of the remaining design process knowledge easier. The physical knowledge associated with a design is represented through the use of CAD tools. The function of an object is represented at a high level. Behavioral knowledge can be represented using a causal network together with the associated temporal constraints. We provide a number of methods to access a design experience once it is stored in the memory. Overall we believe that the system is capable of capturing a good amount of design knowledge.

INTRODUCTION

At the abstract level a design task involves finding a consistent assignment for a set of variables that together define the object desired and satisfy the functional, structural, performance, and other constraints placed upon the object. Often the initial constraints are either incomplete, ill-defined, or inconsistent. In such cases, the initial specification must be modified by extending, clarifying, or refining it.

In most cases, the result of a design activity is only the final detailed design of the desired object. The knowledge utilized while developing the design for an object is discarded after the design has been completed. This philosophy of design places its emphasis solely on the end product and its representation while ignoring the design knowledge which details the way in which the product was developed. Thus the final version of an object design represents only a small part of the knowledge generated by the design process.

The knowledge associated with the design process also includes the history of the design, the rationale behind the design, and expertise gained in the process. This additional knowledge can be useful in the future to help to understand the design as it was completed, make modifications to the design, or provide assistance in the design of new objects.

Recently, researchers have begun to focus more attention to the issue of keeping this design knowledge. The process of acquiring and representing this knowledge is generally referred to as *design knowledge capture*. Notice that this is a two part problem. In addition to developing a representation for design knowledge, a method of obtaining this information from the design process must be found. The former problem considered alone poses a formidable challenge while the latter problem serves only to compound the difficulty. We refer to the process of simply representing design knowledge as *design knowledge retention*. Our efforts will be directed at establishing a model for design knowledge capture even though we know that there are areas where we will have to settle for design knowledge retention.

To address the knowledge capture problem, we are developing the Designer's Workbench which will assist in the design of complex objects. The system will have the following features:

- (a) the design process will be organized under a specific, yet extremely flexible, framework,
- (b) the design framework will serve to document the history of a design as it evolves,
- (c) the plausibility of a design will be established as it is being developed,
- (d) the physical knowledge associated with an object can be captured,
- (e) the functional and behavioral knowledge associated with an object can be incorporated into the overall knowledge framework,
- (f) previous designs can be used to assist in the design of new objects, and
- (g) a hypermedia environment will be used to incorporate the various types of knowledge associated with the design of a complex object.

Before we give any details on the Designer's Workbench, we first provide a general description of the domain of interest, followed by a brief overview of the process of design, and then more information on issues related to design knowledge capture. We then present the specific domain which we will concentrate on followed by the approach that we will use in developing the Designer's Workbench. Finally, we summarize our work and present the conclusions that we have reached to this point.

COMPLEX OBJECTS

In order to more fully understand the process of design knowledge capture, we have chosen to study the process of designing *complex objects*. Loosely stated, a complex object is anything with a nontrivial design consisting of a number of potentially complex components and their interconnections. Examples of complex objects include table lamps and toasters at the relatively simple end and jet engines and space vehicles at

the more complex end. Complex objects have also been referred to in the literature as devices, artifacts, and systems.

This definition is not intended to set definite boundaries on a class of entities called complex objects. The distinction is made here only to convey the idea that we are most interested in entities whose complexity goes beyond the limits of comprehension and memory retention of the human mind. These complex entities require a structured design representation in order to be documented and understood. The definition we present allows a large number of diverse types of entities to be classified as complex objects. Simon (1969) provides an interesting look at the nature of complex systems.

While developing the Designer's Workbench we will focus on one particular type or class of complex object. This is necessary in order to provide us with actual complex objects to study. We are doing this to avoid using toy or fabricated domains and instead focus on a real-world domain. The class of complex objects we have chosen to concentrate on is solar domestic water heating systems. More information about this domain and why it was chosen will be given later.

Although we will be examining a specific domain while developing the Designer's Workbench, we will still be developing a tool which can be applied to other complex object domains with limited modifications. That is, the system will be based as much as possible on domain-independent features but may utilize domain-dependent features to enhance its performance.

Now that the domain of interest has been introduced, we will examine the design process and design knowledge capture as it relates to complex objects. Hereafter a complex object is simply referred to as an object.

AN OVERVIEW OF DESIGN

Design is an activity which plays an important part in our everyday lives. Virtually everything that we interact with is the end product of a design activity which involves some form of creativity. We wake up in a dwelling that is the result of an architectural design. The car we drive is partly the result of a mechanical design. The roads we travel on and traffic signals we encounter are likely part of a traffic flow design. The list of designs and types of designs goes on and on.

Next, we examine what the term *design* actually means. This is followed by a broad look at the process of design. We conclude this section by examining the structure of the design process.

What Is Design?

The first issue of design to consider is actually defining what is meant by the term *design*. As mentioned earlier, there are numerous types of design. Since we are interested in studying the process of design as it relates to complex objects, we only present definitions of design which apply to this domain. These definitions can be generalized to apply to other domains and types of design.

Mostow (1985) states that the purpose of design is to construct a structure (artifact) description that achieves certain goals. The artifact designed must satisfy a given functional specification, meet requirements on performance and resource usage, and

satisfy design criteria on the form of the artifact. Also, the resulting design must be realizable within a specified target medium. Finally, the design process itself must remain within boundaries such as length, cost, and available design tools. Mostow further states that other results, such as capturing design rationale, may be produced by the design process.

Other definitions describing design as a restricted constraint satisfaction problem are given by Daube and Hayes-Roth (1989), Eastman (1981), and Goel and Chandrasekaran (1989a).

The Design Process

Once the desire for a particular object has been established, how is the design for this object created? The design process can be problematic since there exist many different areas of design each potentially with many different methods. The design process can be either structured or ad hoc. Any ad hoc design activity should be avoided in cases where the object to be designed has some degree of complexity. However in design projects involving only a few people, structured design methods are often ignored despite the benefits which would result (Eastman, 1978).

A structured design framework is needed to manage the design of complex objects. Ideally, every design activity would be carried out in a structured fashion. Design methods which can be employed within a structured design framework are *intuitive*, *systematic*, and *knowledge-based*. Intuitive design (Eastman, 1970) is the method that human designers naturally use in solving design problems. In this method, knowledge about solving design problems is obtained from direct or learned design experience. Systematic design emphasizes the logical aspects of design behavior. Unfortunately, the process of systematic design is not very well understood. While there exist systematic design processes for some specific classes of objects, there is no generic method for systematic design. All of the systematic design procedures developed thus far fall into the category of *routine design* — the design of objects that have a known form or design method which is simply instantiated to the specific problem at hand. A knowledge-based approach to design uses facts, reasoning abilities, and case histories to solve a design problem.

The Structure of the Design Process

It is imperative to identify design projects which would benefit from a structured design process and to use this process in creating those designs. The overall structure and control of the design process are important aspects to address while developing a model of the design process.

There is considerable agreement that most design problems (especially complex ones) are best solved in a top-down manner. Top-down design methods have been proposed by Freeman and Newell (1971), Brown and Chandrasekaran (1985), and Steinberg (1987). The structure generally associated with top-down design is a hierarchy. In general, a design usually begins with a vague set of requirements for a desired object. These initial requirements are then progressively refined until a specification of an object which meets the requirements is obtained. During this process of refinement, requirements may be added, deleted, modified, or further defined. By further defining

the requirements, an abstract design becomes more specific. The specifications for an object are used to create a detailed design.

The *theory of plausible designs* (TPD) (Dasgupta, 1991; Aguero, 1987; Aguero and Dasgupta, 1987; Hooton, et al., 1988) is a design paradigm for evolutionary design. A design method based on TPD is referred to as *plausibility-driven*. An important feature of a plausibility-driven design method is that it documents the plausibility of a design as it is developed. A plausibility-driven design method may be viewed as combining top-down refinement methods with the facilities to verify design decisions at each level of refinement. The documentation generated for a design consists of the design decisions made throughout the course of the design process and the evidence or rationale supporting these decisions.

As with other design methodologies, TPD views design as “a specification of a set of interacting functional, structural, and performance constraints” (Hooton, et al., 1988) which must be satisfied by the resulting object or system. The design of an object begins with a set of possibly imprecise *goal constraints* which specify the requirements for the desired object. These goal constraints are then refined into more specific constraints which, in turn, may be further refined. This process continues until constraints corresponding to features actually present in the design of an object are obtained. These constraints are called *design facts*.

Through the use of *plausibility statements* — a representation for constraints which allows for reasoning about them — the satisfaction of the goal constraints can be determined from the existence of the design facts. The plausibility statement for a particular constraint contains information which may include evidence for or against the constraint and/or more specific constraints that must be satisfied in some prescribed manner in order for the more general constraint to be satisfied. Through top-down refinement the plausibility of a design is established. A design is verified in a bottom-up fashion beginning with the satisfaction of the design facts.

While plausibility statements usually deal with how a constraint is satisfied (or why it could not be satisfied), a second type of plausibility statement can be used to document the *desirability* of another constraint. This provides a means for documenting not only *what* constraints must be satisfied and *how* they are actually satisfied but also for justifying *why* certain constraints were included in the design.

It is important to note that the use of the term “constraint” with respect to TPD has a somewhat broader definition than is the case in some other design contexts. In TPD, a constraint refers to any requirement or specification for an object. Constraints can be expressed with varying degrees of specificity. That is, one constraint may be extremely restrictive while another constraint may incorporate some flexibility.

Besides simply organizing the design process, another goal of a structured design method is to provide a means for capturing the design knowledge which is associated with a particular design. The next section describes the importance of design knowledge capture along with the components of design knowledge.

DESIGN KNOWLEDGE CAPTURE

In this section we present the issues associated with design knowledge capture (DKC). We look at the significance of DKC, what we really mean by the term *capture*, the

types of knowledge we must be able to acquire and represent, and what to do with this knowledge once we have it.

The Importance of Capturing Design Knowledge

Capturing design knowledge is crucial when this knowledge is likely to be needed in the future. This is especially the case when an object is part of a project whose lifetime is expected to exceed the time of involvement of the designers. Design knowledge capture is necessary in general because the designers of an object may not be available for future consultation or they might forget their reasoning. The design and engineering expertise which is retained can also be used to assist in future design projects (Freeman, 1988).

There are particular design situations which can benefit substantially from the capture of design knowledge. One of these situations is the development of NASA's Space Station. The environment for designing, testing, operating, and maintaining the Space Station has three important characteristics which make design knowledge capture crucial (Wechsler and Crouse, 1986). First, as the design of the Space Station evolves, design decisions made at one point may be changed at a later time. By using stored design information, these modifications can be made in a more sound and confident manner. Second, the Space Station Program will employ many people throughout its lifetime. When someone leaves the program, the design knowledge associated with that person must have been captured or it may be lost. When someone enters the program, the design knowledge available serves to educate that person on the history and current state of the design project. Third, developments in technology will provide powerful and hardly-imagined ways to use this design information. Another situation with the same characteristics is the Hubble Space Telescope Design/Engineering Knowledgebase Project. The sheer magnitude of these design projects coupled with operational lifetimes on the order of decades makes DKC indispensable (Freeman, 1988). Because of this fact, NASA has committed itself to become a leader in researching and developing DKC methods.

Defining “Capture”

Before we set off to examine the components of design knowledge, let us pause for a moment to consider what the term *capture* means. Webster's New International Dictionary (second edition, unabridged) defines capture as the “act of seizing by force, or getting possession of by power or by stratagem.” The key here is the idea of getting possession by strategem.

It is necessary to interpret this definition as it applies to computers. Capture is the process of acquiring and storing data in computer-interpretable form (Wechsler and Crouse, 1986). The stored data must be organized in such a way as to associate meaning with it. Thus, design information must be stored in a meaningful manner for it to really be useful.

Types of Design Knowledge

Although the scope and types of knowledge to be captured cannot at present be rigorously defined, DKC is aimed at including both design objects and designer's knowledge

(Freeman, 1988). Design knowledge goes beyond the actual design of an object to include the specifications which must be satisfied as well as how and why they were satisfied (Crouse and Spitzer, 1986). The components of design knowledge for an object include its

- (a) physical representation,
- (b) functional representation,
- (c) behavioral representation,
- (d) initial requirements and how they are refined into the specifications,
- (e) design specifications and how they are satisfied in the detailed design,
- (f) design decisions,
- (g) design alternatives,
- (h) design rationale,
- (i) constraints on the design process itself, and
- (j) post-design evaluations

(Goel and Chandrasekaran, 1989a; Addanki and Davis, 1985; Mostow, 1985).

We divide design knowledge into four basic categories: *physical*, *functional*, *behavioral*, and *design process*. Physical knowledge is concerned with the static characteristics of an object. Functional knowledge is concerned with the functionality of an object, while behavioral knowledge is concerned with the dynamic characteristics of an object that meet the functional requirements. The remaining components of design knowledge together comprise design process knowledge. DKC efforts usually focus on issues associated with the capture of design process knowledge since it is this type of knowledge which is often discarded after a design is completed.

A physical representation describes an object in terms of the components from which it is made, how these components are put together, and other characteristics pertaining to the static nature of an object. Thus the physical representation describes "what the object is". A functional representation describes the purpose to which an object is put, i.e. "what the object does". A behavioral representation describes how an object achieves its purpose, i.e. "how the object does it". The behavioral aspects of an object invariably depend on the physical aspects of that object since it is the physical object which actually performs the actions described in the behavioral description.

The design requirements describe an object to be designed at an abstract level. The initial requirements are often vague, ill-defined, and even inconsistent. These requirements are then progressively refined until a sufficiently detailed design specification is produced. The process of refining the initial design requirements into the final design specifications should be documented to show how the design of an object evolved. This knowledge can prove valuable in answering questions about how an object was designed. Also, the way that the specifications are satisfied in the detailed design should also be documented to prove that they are in fact satisfied and to provide assistance in analyzing the effects that a design modification may have in reference to the specifications.

Design decisions, alternatives, and rationale are separate, but interrelated, concepts. Design decisions are the actual decisions made during the design of an object. Design

alternatives are design choices that were considered but not chosen. Design rationale explains why the decisions were made as they were. The reason behind a particular design decision is documented by giving evidence supporting the decision or perhaps reasons why other alternatives were *not* chosen. Just as documenting good design choices is important, documenting unsuccessful design choices is necessary to avoid repeating them in the future.

Constraints on the design process itself are the restrictions imposed on resources available while an object is being designed. These restrictions provide limits on resources such as time, money, and personnel available. By knowing the conditions under which a design was produced, it may be easier to see why certain design choices were made.

Post-design evaluations are important in documenting the goodness of a design. Just as capturing the design knowledge utilized in creating the design for an object is crucial to understanding the completed object, post-design evaluations are crucial in determining the worth and actual validity of the reasoning used in the design process. The evaluations of previous designs can be used to help improve the design of a new object. While good ideas and decisions may be identified, it is more likely that design mistakes will receive the most attention in the post-design analysis. The presence of post-design evaluations allows us to learn from the successes and failures of others.

All of this information about an object can be gathered together to form a particular *design experience* (DE) (Babin and Loganathanraj, 1990). While it is important to have a representation capable of capturing a DE, a representation alone is not enough. If we simply design an object and capture the design knowledge for that object, we have a DE with limited usefulness. The DE would only be consulted when a question is raised about the specific object that is represented. Thus some type of organization of DEs is needed to make them more useful.

Organizing Design Knowledge

A memory organization for DEs is necessary to fully exploit the design knowledge captured for different objects. Specifically, DEs of physically or functionally similar objects should be (conceptually) organized together. Obviously many more categories of similarity (and dissimilarity) are possible. This organization allows a designer to study past DEs in order to help in a current design task which is similar in some respect. A designer can make decisions based on the decisions made in past designs. The representation allows previous successes to be copied where desired and also guards against repeating previous failures. Persons not involved in a design activity would also benefit from this type of organization by being able to examine classes of objects or particular objects that are of interest.

The desire to use DEs to assist in future design projects suggests that case-based reasoning (CBR) methods may be applicable. Background information on CBR or, more generally, memory-based reasoning can be found in (Hammond, 1989), (Kolodner, 1988), (Kolodner, 1987), (Stanfill and Waltz, 1986), and (Schank, 1982). While a DE is essentially a design case, the term *design experience* is used to convey the idea that it represents the total design effort, not just the resulting design. Capturing the DE for an object makes CBR unnecessary because of the richness of information available. CBR methods are suited to situations where design process knowledge is scarce and

one must rely almost exclusively on the outcome of a design activity.

We now present the domain which we will use for studying design knowledge capture along with the major features which led us to choose this domain.

THE APPLICATION DOMAIN

The class of complex objects on which we have chosen to demonstrate our design knowledge capture techniques is solar domestic water heating systems (SDWHSs). There are several features of SDWHSs that make the domain appealing to use as an example for developing design knowledge capture techniques:

- (a) A SDWHS is a complex object consisting of many interconnected components. These components must work together along with the sun to make the system operate as intended. The overall complexity can vary greatly between systems but remains tractable for this type of project.
- (b) There are numerous generic types of SDWHSs which have been developed over the years. These generic types serve as generalizations to be considered when designing a SDWHS for a particular situation.
- (c) Although there are generic types of SDWHSs, there are a multitude of factors to consider while designing a SDWHS. These factors include system capacity, climate, site layout, and cost, to name a few. Routine design methods may be applicable to some of the subproblems associated with a SDWHS design, but the complex nature of the overall design process all but rules out the possibility of developing a routine design method for SDWHSs. This observation is further supported by the possibility of evolutionary SDWHS design.
- (d) Since SDWHS design is not routine, each design needs to be documented in order to record the decisions, rationale, and alternatives considered during the design process.
- (e) When designing a SDWHS, previous design experiences can be examined to provide valuable insight not only into configurations which were successful and the rationale behind them but also into configurations which were not successful, not chosen, or not examined and why.
- (f) The process of capturing design knowledge for SDWHSs appears to be easily transferable to other types of complex objects.

Because of these major features, we feel that SDWHS design provides an excellent example to use for studying design knowledge capture. The details of SDWHSs are not important to elaborate upon at this time, instead they will be exposed as needed in future reports.

We next look at our approach to design knowledge capture by describing the Designer's Workbench which we are currently developing.

THE DESIGNER'S WORKBENCH

While efforts have been made in different areas of design representation, design analysis, object representation, and memory organization, no significant attempt at "putting it all together" into an integrated approach has been made. The overall goal of this research is to develop a system — the Designer's Workbench — which will assist in the design of complex objects. The system will have the following features:

- (a) The design process will be organized under a specific, yet extremely flexible, framework. The framework provides a method for specifying constraints on an object and various verification techniques to prove that the constraints on an object have been satisfied.
- (b) The design framework will serve to document the history of a design as it evolves. Design constraints, decisions, and alternatives will be recorded. In addition, the way in which the constraints were satisfied will be documented.
- (c) The plausibility of a design will be established as it is being developed. The relationships between design constraints will be explicitly specified in order to determine how one design decision affects the rest of the design.
- (d) Some physical knowledge is captured by the design constraints. In addition, the physical representation of an object produced by the design process is linked to the satisfaction of the design constraints and the plausibility of the design.
- (e) Functional and behavioral knowledge are captured to some degree by the design constraints. To assist in verifying a design, additional functional and behavioral knowledge representations may be incorporated into the overall knowledge framework.
- (f) Previous designs which are similar in some respect to the current design project can be retrieved so that the alternatives considered and decisions made can be analyzed.
- (g) A hypermedia environment will be used to provide the user with an easy way to access the many different types of knowledge associated with the design of a complex object.

We will now give an overview of how we plan to represent the four types of design knowledge: *design process*, *physical*, *functional*, and *behavioral*. We will then describe our methods of indexing this information. This section concludes by examining the extent to which true design knowledge capture is realized by this system.

Representing Design Process Knowledge

Design process knowledge consists of

- (a) the initial requirements for an object and how they are refined into specifications,
- (b) design specifications and how they are satisfied in the detailed design,
- (c) design decisions,
- (d) alternatives which were considered,

- (e) rationale for decisions made,
- (f) constraints on how an object was designed, and
- (g) post-design evaluations.

By including a measure of time in the documentation of this information, the evolution of the design can be traced.

We have chosen to use a dependency directed network (DDN) to capture the top-down refinement process and to establish the plausibility of a design as it is being developed. Each node in the network is essentially a plausibility statement similar to that used in the theory of plausible designs. We extend the plausibility statement concept to include more types of design knowledge.

At the heart of each node is a design constraint. A constraint pertains to some physical, functional, and/or behavioral aspect of an object which must be satisfied in the detailed design. A constraint may also describe some restriction on the design process itself. The method of top-down refinement begins with a set of possibly imprecise and even inconsistent goal constraints. These constraints are progressively refined to become more specific until they become design facts. A DDN captures how a constraint is refined into more specific constraints and shows the dependency relationships between the constraints. The method or source used to verify a constraint is explicitly represented. Design decisions can be justified by including plausibility statements which document why certain constraints are desirable.

If a constraint has more than one viable method of satisfaction, the designer may choose to explore the options available. We accommodate this by providing a means of expressing alternative satisfactions for a constraint. Of the alternatives, all but one are considered to be inactive with respect to a particular version of a design. The designer may switch between alternatives, but only one can be considered active at any given time. This condition is necessary to ensure that, at any given time, a version of a design is deterministic.

We allow the designer to describe the rationale behind the decisions which were made while refining a constraint. This rationale includes justifying why a constraint was satisfied as it was and why other alternatives were not chosen. This is not to be confused with the purpose of a desirability constraint which specifies why a constraint was included in the design but nothing about its refinement. A plausibility statement may also contain comments which informally document the intent of the associated constraint and provide other information or notes that may be relevant. In addition, provisions will be made to include post-design evaluations on the plausibility statement level.

A designer begins by specifying the initial goal constraints for the object desired. As the design process progresses, constraints can be created, refined, inactivated, deleted, or restored. The difference between deleting a constraint and inactivating it is that a deleted constraint is removed from the design because it has been deemed no longer necessary while an inactive constraint is still under consideration, as in the case of a design alternative. In any case, no constraint is ever completely thrown away since its existence will be recorded in the design history. This allows deleted constraints to be restored if necessary.

Aside from the constraints on the design process itself, all of the constraints refer to physical, functional, and/or behavioral aspects of an object. Thus a great deal of

physical, functional, and behavioral knowledge is contained within the constraints. In addition to this, separate representations for these types of knowledge may be used to provide more information which is not easily represented in constraint form and to show how constraints are satisfied. We now look at how other representations can be integrated into the overall knowledge framework.

Representing Physical Knowledge

The physical knowledge associated with an object refers to the static characteristics of that object. Physical knowledge includes decompositional knowledge, structural knowledge, and physical properties. The physical representation used for an object is dependent on the domain to which the object belongs. By choosing to study solar domestic water heating systems we are able to develop a general physical representation scheme which we feel is applicable to a wide range of complex objects. We now elaborate upon the types of information which must be included in a scheme for representing physical knowledge for complex objects and suggest methods of realizing such a representation.

Decompositional knowledge arises from the fact that an object may be composed of parts, each of which may be composed of subparts, and so on. A collection of parts may be thought of as a component of an object. This type of information is hierarchical in nature as long as different instances of the same component or part are treated as being separate. Relationships between parts in the hierarchy are expressed through HAS-PART and PART-OF links. An object is ultimately realized by a collection of parts which are elementary or are themselves complex objects with their own designs. A part is considered elementary if it is nondecomposable with respect to a certain level of abstraction. A design may be represented at various levels of abstraction. By supporting levels of abstraction, a design can be examined at a particular level of detail without including unnecessary and perhaps confusing lower-level details. Therefore, although a part (component) is not elementary, it may be treated as such at a certain level of abstraction. A part which is elementary at one level of abstraction may be decomposable at the next more specific level of abstraction. This provides a way to examine a design in as much detail as is desired. Post-design evaluations can also be included on the component level.

Physical properties are represented by describing elementary objects in terms of size, shape, material, color, weight, etc. Components may also have similar properties which are derived from the properties of its parts. This information can be represented by using a frame-based approach. By constructing a library of parts which includes physical properties, the task of determining this knowledge is eliminated except when a new part is created or introduced.

Structural knowledge concerns the positions of the components with respect to each other and the nature of the interconnections between the components. This type of information is difficult to represent, especially if it is needed for structural or functional reasoning purposes.

Many mechanical designs today are developed with the assistance of computer-aided design (CAD) software. CAD software has evolved from computer-aided drafting (also referred to as CAD) applications where the computer was used as an electronic drafting table. Users soon desired more complex tools which would provide further assistance

in the design process. Features such as wire-frame modeling, solid modeling, part and symbol libraries, and structural analysis are now readily available (Encarnacao, 1987). CAD tools have also been developed to help automate certain design procedures and to provide simulation methods for design verification. Such is the case in integrated circuit design (Begg, 1984).

Due to its obvious applicability, we have chosen to employ CAD techniques to represent physical knowledge. We initially plan to concentrate on using the representational capabilities of such a system. That is, a SDWHS will be represented as a three-dimensional entity composed of elementary objects chosen from a library of parts and arranged into an abstraction hierarchy which specifies their interconnections. This approach will help to simplify the complexity of the physical representation while demonstrating how the physical representation can be incorporated with the other types of design knowledge. Later we may consider integrating analysis and simulation procedures which prove to be useful.

In summary, we will utilize the physical representation tools which are available. If necessary we will fill in any information gaps which exist. In addition, a link between the physical knowledge and the design process knowledge will be established to verify design facts which reference the physical representation of the object.

Many of the bottom-level constraints — the design facts — will concern physical aspects of the object design. The satisfaction of these constraints depends on the existence of specific physical characteristics in the final design. In such a case, the plausibility statement refers directly to the physical representation. In this way the physical representation is linked to the DDN.

Representing Functional Knowledge

The *function* of an object is the purpose to which it is put. A functional representation must be able to effectively express the functional knowledge associated with an object. The representation employed can depend on factors such as the domain chosen and the amount of knowledge required. A single representation incorporating both functional and behavioral knowledge may be used, or separate functional and behavioral representations may be desired. In the case of separate representations, the functional and behavioral knowledge would need to be linked in some manner to establish the connection between a function and the behavior which realizes it.

One common approach to representing functional knowledge is to use a function-to-object mapping. This is accomplished by assigning an object to one or more functionality classes. In this way, objects which perform similar functions are grouped together in a class. These classes are defined according to the needs of the domain. An object with more than a single function may be assigned to more than one functionality class. Whenever a specific function must be realized, objects in the closest corresponding functionality class can be examined for appropriateness.

This approach to representing functionality is relatively simple and is independent of any particular behavioral representation. We feel that this type of representation combined with the functional knowledge present in the constraints is adequate for many situations including our domain.

Representing Behavioral Knowledge

The behavioral knowledge associated with an object refers to the dynamic characteristics of that object. In the context of a design, the *function* of an object is the purpose to which it is put while the *behavior* of an object is the way in which this function is achievable. For example, the behavior of hands rotating about a point on a clock provide a means for achieving the function of a clock — telling the time of day. Thus functions of objects are realized through behaviors. The behavior of an object can be described in terms of the behaviors of its components. The behaviors of the components may in turn be described in terms of the behaviors of their subcomponents, and so on. The structure of an object is used to relate how the behaviors of the components together accomplish the overall behavior which realizes the function of that object.

The popular approach to describing behavior is to use causal networks. The notable works in this area are from Chandrasekaran et al. (Sembugamoorthy and Chandrasekaran, 1986; Goel and Chandrasekaran, 1989b). They represent a device in terms of its structure, function, and the behavior that achieves the function. A behavior is represented as a causal network with each node representing a (partial) state of an object and each link between a pair of states representing a causal relation. Each causal link is explained by the function or behavior of a component and can be qualified by a set of conditions, called assumptions, that must be satisfied before the causal link can hold. Generic knowledge is also used as an element in the behavioral description to explain how the intended function is achieved. In such cases a causal link refers to that generic knowledge.

Chandrasekaran's approach, though elegant and very useful to capture the functional knowledge of a object, lacks several aspects necessary to fully capture the concurrent behaviors of different components of a device. We have extended the basic scheme to include the temporal aspects of behavior (Babin and Loganathanraj, 1991).

We want to emphasize here that different functional and behavioral representation schemes may be used in conjunction with our design knowledge capture framework. The scheme used is likely to vary between domains as different types of functional and behavioral models are required. In cases where functional representation schemes already exist, the focus would be on integrating the existing schemes into the design knowledge capture framework.

This is accomplished in much the same way that the physical representation for an object is linked to its constraints. For example, the satisfaction of a behavioral constraint may depend on the existence of a model which can simulate the behavior of (some component of) the object. Thus the behavioral representation chosen can be linked to the behavioral knowledge present within the DDN.

Indexing Design Knowledge

The ability to create new designs is only a part of the Designer's Workbench. When a design experience is completed it is added to the memory where the design experiences are stored. From there, the indexing and retrieval methods provided take over. All or part of stored designs may be retrieved for any one of a number of reasons:

- (a) to allow criticism of the design by persons not actually involved in the design process,

- (b) to introduce new members of a design team to the current status and history of the design,
- (c) to determine the effects of a design modification on the rest of the design,
- (d) to diagnose functional, behavioral, and structural problems which arise,
- (e) to serve as a permanent reference for the object in the future, or
- (f) to assist in the design of other objects which are similar in some respect.

Since there are so many diverse types of knowledge associated with the design of an object, different types of indices will be needed to access this information. We plan to include constraint indexing mechanisms, a part/component/object index, a decompositional index, and a classification based on functionality. Other types of indices may be included if the need arises.

These indices will be implemented through a combination of direct, navigational, and query-based retrieval methods. Each type of index will be implemented using the method or methods which are appropriate. A direct index provides the quickest access, but the exact identifier of the information desired must be known. A navigational index allows a user to browse through various levels of information in order to find what is desired. A query-based index provides a way to access information whose location is unknown or is spread across many areas of the design memory.

The constraint indexing mechanisms will allow a designer to find constraints similar to the one from a current design project and provide a means to represent the conceptual structure of an object. These constraints can then be analyzed to determine, among other things, (1) if the satisfaction method chosen previously is applicable in the current situation, (2) other alternatives considered but not chosen, and (3) the rationale behind the decision made. This information can provide valuable insight that would have otherwise been lost.

A design management structure will help to further organize plausibility statements in terms of components and constraint types (e.g. cost, performance). This conceptual structure provides a number of advantages. First, the level of abstraction introduced helps in conceptualizing the overall organization of the design constraints. Second, this level of abstraction provides efficient navigation through the constraints for an object. Third, post-design evaluations can also refer to conceptual entities, such as components, instead of only referring to specific constraints.

The design management structure will be able to indicate whether or not all of the constraints dependent on a particular component have been satisfied. Components which do not satisfy all of the constraints placed upon them must be refined or modified to satisfy these constraints or else the constraints themselves must be modified. This applies to the physical, behavioral, and functional constraints placed upon a component.

The part/component/object index is a simple direct indexing mechanism to be used when the name, identification number, or some other key is known. The user can then directly access the part, component, or object desired. The decompositional index maintains the HAS-PART and PART-OF links described previously and allows the user to traverse the component-subcomponent abstraction hierarchy.

The functionality-based classification is a multi-level walk-through type of index where the objects are grouped according to similarities. This index is similar to a Yellow

Pages type of index except that many levels — from general to specific classifications — are possible. This will allow a user to progressively narrow down object classes until a sufficiently specific one is reached, and then peruse the designs in that class (possibly using other indices).

Is This Design Knowledge Capture?

Before closing this introduction to the Designer's Workbench, let us pause for a moment to consider whether or not this system is actually capable of performing true design knowledge capture. Remember that design knowledge capture means that the design knowledge which is stored must come as a direct result of the design process, not through a posteriori reflection or extra-design effort. To make this determination we must examine each of the components of design knowledge as they relate to this system.

First of all we look at design process knowledge. Due to the nature of the DDN and its plausibility statements, the design constraints and methods of satisfaction result directly from the design process. However, design decisions connected with the specification of desirability constraints, alternatives considered, rationale, and constraints on how the object was designed are not necessarily direct products of the design process. Although the designer is not forced to provide this information, we have made it easy to do so. With a bit of initial discipline, we feel that providing this information can become second nature to a designer.

As far as physical knowledge is concerned, the resulting design must provide this information to be considered complete. The only issue to consider is the amount of detail that is supplied. By specifying the level of detail required in the original design constraints, the design process will produce this information.

For functional knowledge, the object must be proven to perform the function for which it is intended to be considered complete. This is usually proven through the specification of behavioral knowledge or through well known function-to-structure mappings. The function of an object is realized through its behavior. Thus the amount of functional and behavioral knowledge generated will depend on the amount necessary to prove that the object performs correctly.

So, what is the verdict? A good amount of design process knowledge is captured from the design process and the rest is relatively easy to provide. Physical and functional knowledge must be generated to the extent that they are needed to satisfy the goal constraints. Additional design constraints may be imposed in order to capture more detailed information. Therefore, although the Designer's Workbench does not perform complete design knowledge capture, we feel that it comes reasonably close.

SUMMARY AND CONCLUSIONS

In this paper we have proposed the Designer's Workbench — a system to assist in the design of complex objects which provides a scheme for design knowledge capture. We have chosen to study design knowledge capture in the context of design solar domestic water heating systems. We presented the major features which make this domain attractive for this application.

The Designer's Workbench uses plausibility statements for the purpose of design verification. A significant amount of design process knowledge can be captured by using plausibility statements organized into a dependency directed network (DDN). We use a DDN since it is able to represent the top-down structure of design which is commonly used. The extensions we described serve to make retention of the remaining design process knowledge easier. The physical knowledge associated with a design is represented through the use of CAD tools. The function of an object is represented at a high level. Behavioral knowledge can be represented using a causal network together with the associated temporal knowledge.

We provide a number of methods to access a design experience once it is stored in the memory with other design experiences. One drawback of our scheme is that design process knowledge cannot be stored unless it conforms to the plausibility statement (constraint) format. Our research into constraint representation and indexing methods should prove useful in other design disciplines. In the future, this research may provide alternative formats to allow other types of constraint-based design knowledge structures to be included in the case base.

We plan to create a working model of the system which includes most, but not all, of the aspects of design knowledge capture that we have presented within. The working model will include a plausibility-driven design method, a CAD-based physical representation scheme, a history mechanism, and knowledge indexing capabilities. The remaining aspects, such as including a scheme for representing behavioral knowledge, will be incorporated into the system soon after.

Overall we believe that the system is capable of capturing a good amount of design knowledge. Whenever additional effort is needed in order to retain design knowledge, we have attempted to minimize the effort required.

REFERENCES

- Addanki, S. and Davis, E. (1985). A representation for complex physical domains, In *Proceedings of the Ninth International Conference on Artificial Intelligence (IJCAI-85)*, Los Angeles, CA, August 1985.
- Aguero, U. (1987). *A Theory of Plausibility for Computer Architecture Designs*, PhD thesis, The University of Southwestern Louisiana, 1987.
- Aguero, U. and Dasgupta, S. (1987). A plausibility-driven approach to computer architecture design, *Communications of the ACM* 30(11):922-932, November 1987.
- Babin, B. and Loganathanraj, R. (1990). Capturing design knowledge, In *Proceedings of the Fifth Conference on Artificial Intelligence for Space Applications*, Huntsville, AL, May 1990.
- Babin, B. and Loganathanraj, R. (1991). Representing functional knowledge, In *Proceedings of the Fourth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, Kauai, HI, June 1991.
- Begg, V. (1984). *Developing Expert CAD Systems*, Koran Page, London, 1984.
- Brown, D. C. and Chandrasekaran, B. (1985). Expert systems for a class of mechanical design activity, In J. S. Gero (ed), *Knowledge Engineering in Computer-Aided*

- Design*. Elsevier Science Publishers B.V. (North-Holland), 1985.
- Crouse, K. J. and Spitzer, J. F. (1986). Design knowledge bases for the space station, In *Proceedings of ROBEXS '86*, June 1986.
- Dasgupta, S. (1991). *Design Theory and Computer Science*, Cambridge University Press 1991.
- Daube, F. and Hayes-Roth, B. (1989). A case-based mechanical redesign system, In *Proceedings of the Eleventh International Conference on Artificial Intelligence (IJCAI-89)*, Detroit, MI, August 1989.
- Eastman, C. M. (1970). On the analysis of intuitive design processes, In G. Moore (ed), *Emerging Methods in Environmental Design and Planning*. MIT Press, Cambridge, MA, 1970.
- Eastman, C. M. (1978). The representation of design problems and maintenance of their structure, In Latombe (ed), *Artificial Intelligence and Pattern Recognition in Computer Aided Design*. North-Holland Publishing Company, New York, 1978.
- Eastman, C. M. (1981). Recent developments in representation in the science of design, In *Proceedings of the Eighteenth ACM and IEEE Design Automation Conference*, Washington, D.C., 1981. IEEE.
- Encarnacao, J. L. (1987). CAD technology: A survey on CAD systems and applications, *Computers in Industry* 8(2/3):145–150, April 1987.
- Freeman, M. S. (1988). The elements of design knowledge capture, In *Proceedings of the Fourth Conference on Artificial Intelligence for Space Applications*, Huntsville, AL, November 1988.
- Freeman, P. and Newell, A. (1971). A model for functional reasoning in design, In *Proceedings of the Second International Conference on Artificial Intelligence (IJCAI-71)*, London, England, September 1971.
- Goel, A. and Chandrasekaran, B. (1989a). Functional representation of designs and redesign problem solving, In *Proceedings of the Eleventh International Conference on Artificial Intelligence (IJCAI-89)*, Detroit, MI, August 1989.
- Goel, A. and Chandrasekaran, B. (1989b). Use of device models in adaptation of design cases, In *Proceedings of the Second Case-Based Reasoning Workshop*, Pensacola, FL, June 1989. DARPA.
- Hammond, K. (ed) (1989). *Proceedings of the Second Case-Based Reasoning Workshop*, Pensacola, FL, June 1989. DARPA.
- Hooton, A., Aguero, U., and Dasgupta, S. (1988). An exercise in plausibility-driven design, *Computer* 21(7):21–31, July 1988.
- Kolodner, J. (1987). Extending problem solving capabilities through case-based inference, In *Proceedings of the 4th Annual International Machine Learning Workshop*, 1987.
- Kolodner, J. (ed) (1988). *Proceedings of the First Case-Based Reasoning Workshop*, Clearwater, FL, 1988. DARPA.

- Mostow, J. (1985). Toward better models of the design process, *AI Magazine* 6(1):44–57, 1985.
- Schank, R. C. (1982). *Dynamic Memory – A Theory of Reminding and Learning in Computers and People*, chapter 2, Cambridge University Press, 1982.
- Sembugamoorthy, V. and Chandrasekaran, B. (1986). Functional representation of devices and compilation of diagnostic problem-solving systems, In Janet L. Kolodner and Christopher K. Riesbeck (eds), *Experience, Memory, and Reasoning*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1986.
- Simon, H. A. (1969). *The Sciences of the Artificial*, The MIT Press, Cambridge, MA, 1969.
- Stanfill, C. and Waltz, D. (1986). Toward memory-based reasoning, *Communications of the ACM* 29(12):1213–1228, December 1986.
- Steinberg, L. I. (1987). Design as refinement plus constraint propagation: The VEXED experience, In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, August 1987.
- Wechsler, D. B. and Crouse, K. R. (1986). An approach to design knowledge capture for the space station, In *Proceedings of Space Station Automation II, Vol. 729*, pages 106–113. SPIE, 1986.

Unsupervised learning of design rules aided by system derived heuristics

S. Matwin,[†] F. Oppacher,[†] U-M. O'Reilly[‡] and B. Pelletier[†]

[†]Intelligent Systems Research Group
Carleton University

[‡]Ottawa Machine Learning Group
University of Ottawa
Ottawa Ontario K1N 6N5
Canada

Abstract. This paper describes HERALD (HEuristic Rule Analysis and Learning for Design), a two stage learning system consisting of ECMA (Extended Conceptual Modelling Assistant) and LEADER (LEArning DEsign Rules). HERALD provides design assistance for conceptual modelling in the Extended Entity-Relationship Model (ER) domain. Its first component, ECMA, using an extended case-based model and the expert as oracle, provides designers with suggestions for design completion by developing and refining heuristics reflecting modelling strategies. These heuristics are subsequently used by LEADER which, non-interactively, learns design rules via an explanation-based paradigm. Once approved by experts, LEADER's rules are directly incorporated into the underlying knowledge-based design tool which HERALD supports. This work addresses what we perceive as a shortcoming of some AI-based design tools, which require detailed and explicit knowledge of design. In our approach the problem is circumvented by combining the explanation-based learning (EBL) with case-based learning. The latter type of learning, rather than requiring a codified theory of design, uses readily available design cases.

0. INTRODUCTION

Learned knowledge is essential for providing good assistance interactively or for learning rules which can enhance an underlying design tool. HERALD (HEuristic Rule Analysis and Learning for Design) is an integrated, two component system. We briefly present the overall architecture showing the system's interactions with expert designers and the link between ECMA (Extended Conceptual Modelling Assistant) and LEADER (LEArning DEsign Rules). The individual architectures of ECMA and LEADER are shown. We describe how ECMA, the case-based component, represents and learns the heuristic

knowledge supplied to LEADER. LEADER is then presented detailing how it uses heuristics, a design rule learning strategy, and model construction sequences to define design rules. Our conclusion is that it is beneficial, if not essential, to learn design rules and heuristics in this complementary fashion.

1. LEARNED KNOWLEDGE FOR DESIGN ASSISTANCE

HERALD has been developed for a tool called Modeller. Modeller, a research development of Cognos, Inc., is an assistant to a human designer for the construction of Extended Entity-Relationship (ER) models and databases. It has a graphical interface for drawing ER models, and a rule-based expert system module, whose primary purpose is to clarify and tidy the design of a corresponding physical data schema. Constructing an ER model is the step in the DB design process that expresses semantic structure and the constraints imposed by a real world situation. It stresses complete logical definition of a system.

The basic components of an ER model are entities - the objects in a system, and relationships - the behavioral interactions in a system which evolve over time. The model illustrates entities as rectangles and relationships as diamonds between directed arcs. The cardinality of a relationship refers to the number of objects of one entity in the relationship that can be related to objects of the other entity in the relationship, and vice versa. A minimum and maximum cardinality can be specified (Tsichritzis and Lochovsky, 1982). An ER model can express the existence of multiple relationships between the same two entities and define the nature of any relationship including its cardinalities. Further, the attributes of an entity (i.e. properties that all data instances share) and keys or subkeys (Tsichritzis and Lochovsky, 1982) (aspects which make a data instance unique) are included in ER models.

Modeller has fifteen assertional commands which support entity definition, relationship definitions (though two entities are restricted to only one relationship), key, attribute, subkey attribute, domain and cardinality (the precise enumeration of a relationship) definition (Tauzovich, 1990). Relationships can be explicitly taxonomic ('*is Role of*', '*is Natural subclass of*'), explicit dependencies ('*is an Extension of*', '*is an Association of*', '*is a Characteristic of*'), or simply generic ('*is Related to*'). Syntactically, i.e. with respect to sequence and context, command choices are totally unconstrained.

HERALD focuses on the conceptual design phase of Modeller since the expert system component provides only limited assistance with model construction. For example, it does consistency checking, applies uniform defaults to unlabeled cardinalities, and detects 'traps', which are portions of the model that match certain patterns indicating a probable error. The present knowledge of this component is basically restricted to an explicit formulation of the semantics of Modeller's command repertoire. Since command choices are unconstrained with respect to sequence and context, the expert system component can verify certain proposed designs but can give no advice on how to design a model.

ER modelling is a creative design task. Modeller is easy to use but specifically intended only for experienced and knowledgeable system analysts. HERALD enhances Modeller by 1) providing construction suggestions based on design expertise and 2) deriving rules of design from examination of design sequences which can be added as short-cuts, macros or fixes in Modeller's rule-base. To do this requires component systems which can learn

design knowledge. We propose that ECMA, which can learn heuristics helpful in design completion, supply these heuristics as a set of knowledge for use by LEADER. LEADER is an explanation-based learning component which formulates rules which uncover goals and plans used in design. LEADER is an unsupervised learning system that learns new design knowledge to enhance an initial theory of development of ER data models. The task is achieved by observing an human expert designer through his/her interactions with the Modeller tool, by using heuristics, and explanation-based learning.

Before presenting how learning occurs in HERALD, we present an overview of the interaction HERALD has with the human expert and of the type of knowledge the system learns: First, a design expert uses Modeller to build ER models (Cases). ECMA analyses those models through an interaction with the design expert and produces as output heuristics predicting potential intention (goal) or strategy that a designer might use at a given point of a design.

Later on, LEADER takes the construction sequence of models produced by Modeller and analyses the way they were built; it looks at the complete user session, not only at the final model. To perform its analysis, it uses an incomplete theory of design techniques as well as incomplete knowledge about the domain being modelled. If LEADER fails to explain the strategy used by the designer at a given point of the user session, it uses heuristics produced by ECMA to obtain advice about the goal and/or the strategy the designer might be pursuing at that point. This advice helps LEADER to infer new goals and/or new strategies which are capable of explaining a larger part of the user session (hopefully, the entire session). This learning is unsupervised because LEADER requires no interactions with experts.

Thus, LEADER's (and HERALD's) output is design goals and design strategies. This new knowledge is accumulated and later on verified by a design expert and added to the knowledge already used by LEADER. The design knowledge acquired by LEADER is then incorporated into Modeller to enhance interactions with future (possibly novice) ER model designers.

In Section 4 we discuss how a modification of an ER representation of a database is learned by LEADER and ECMA interacting with each other. In that case, the system learns a "macro" design rule for generalizing a relationship of a ER schema.

2. HERALD = ECMA + LEADER

Fig. 1 shows HERALD with its two components. Fig. 2 shows ECMA in more detail. ECMA has an extended case-based reasoning (CBR) model (Riesbeck, 1988), (Hammond, 1988), (Kolodner and Riesbeck, 1989). Models are stored in graph-theoretical notation as cases. The system accepts an incomplete model (design in progress) and by finding the most similar case and adapting its solution, produces suggestions (in assertion form) as to what should be added next.

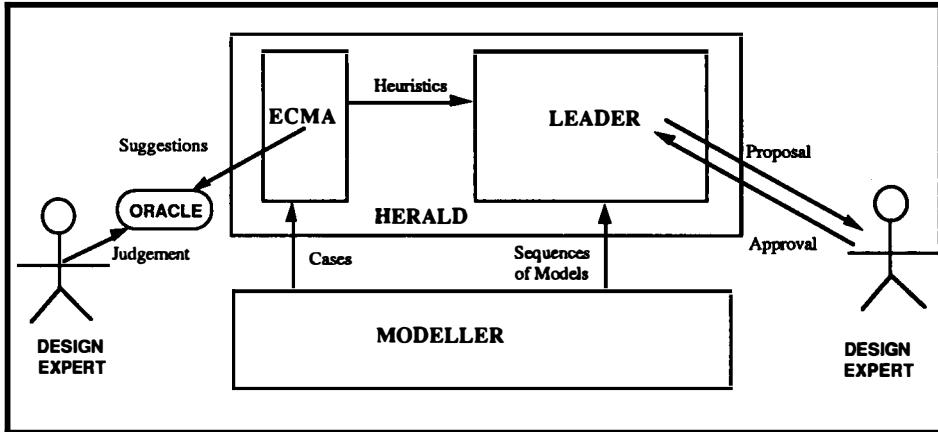


Fig. 1. Architecture of HERALD.

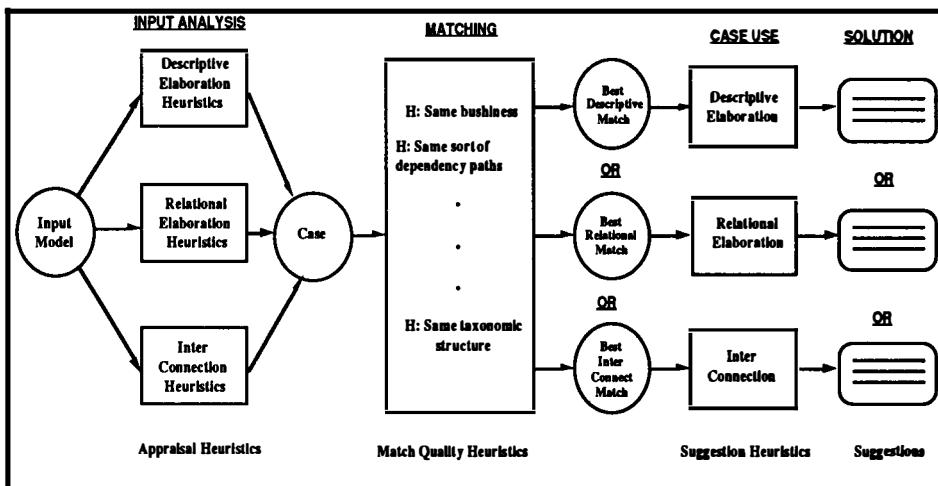


Fig. 2. Architecture of ECMA.

The standard input analysis, matching procedures and case-use rules in a CBR system cannot be directly given to ECMA by experts due to the creative nature of ER modelling. The system learns these by using the expert as oracle. During a training stage it produces, by using its existing heuristics, suggestions. The oracle by merely rewarding or penalizing suggestions supplies ECMA with sufficient feedback to adjust its heuristic knowledge (O'Reilly and Oppacher, 1990).

LEADER is shown in more detail in Fig. 3. The Interaction Observer watches the expert designer, with no disturbance, and produces the ordered sequence of actions corresponding to the user session. The Action Executor executes in sequence all actions of

the given session, and produces the corresponding ordered sequence of models. The Interaction Analyzer takes the sequence of models as input and, using the current knowledge about the modelling (Approved Knowledge), builds a goal structure (called 'goal graph') over the user session. This structure identifies the part of the user session that can be explained with the current approved knowledge (the unexplained part is therefore also identified).

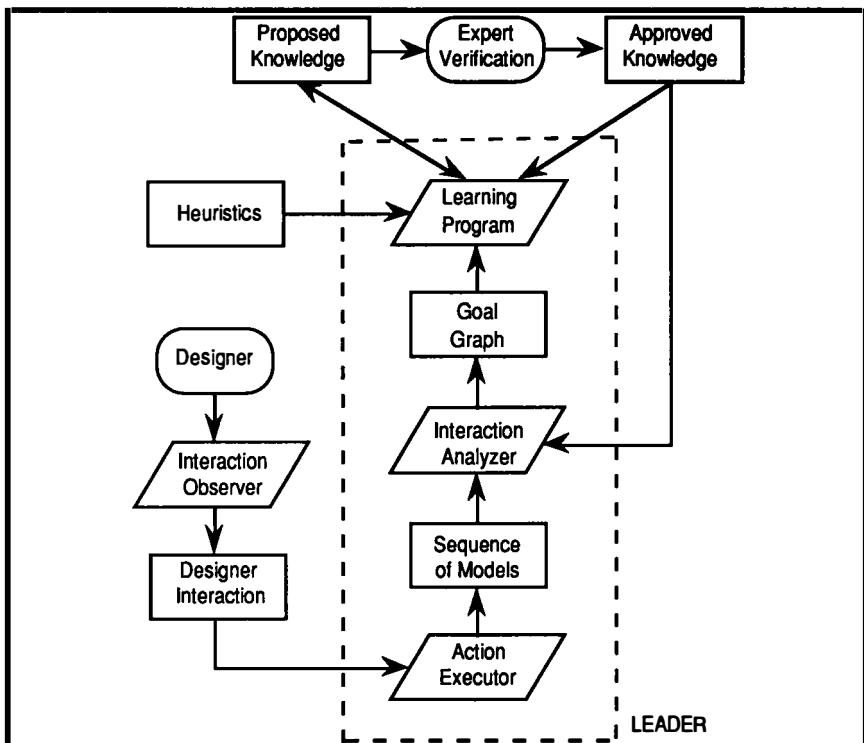


Fig. 3. Generic architecture of LEADER.

The Learning Program is the kernel of the entire system: this is where most of the learning is being performed. Using the current approved knowledge about the modelling, and using learning heuristics (some produced by ECMA), it analyzes the goal graph and infers new knowledge to augment the design theory.

The entire process will be applied to different designer sessions (and possibly to different designers as well) and the resulting candidate knowledge for each of these sessions will be accumulated.

After the system has been functioning for a while, a human expert in database design will examine the accumulated proposed knowledge to verify it and add the filtered knowledge to the current approved knowledge. His/her task is to eliminate incorrect or redundant information, and to refine the correct information, merging it with the already approved knowledge. This merging may further eliminate redundancies, refine knowledge, and even repair any incorrect information not previously detected.

3. LEARNING HEURISTICS IN ECMA

ECMA derives two sorts of heuristics: strength-based and definition-based. Strength-based heuristics are predicate hypotheses about some aspect of the domain which can be verified in graph terms. Such heuristics undergo adjustments to their strengths each time they are factors in an outcome to which the oracle reacts. Penalization by the oracle results in a heuristic which returns true being weakened and a heuristic which returns false being strengthened. Reward is treated in an analogous manner. All strengths are adjusted, at present, by having the amount of adjustment normalized to the relative performance seen thus far by the heuristic. ECMA is implemented in Smalltalk V-286 and the design of heuristics as objects allows any adjustment method to be specified for any set or particular heuristic.

For example, the system has one heuristic (the 'relationally complete' heuristic) that suggests a descriptive elaboration of the partial model when there are approximately equally many entities (nodes) and relationships (edges) in a subgraph.

The second type of heuristic is definition-based. These heuristics use feedback to adjust the bounds which define a concept. In this way the system acquires the definition of a concept. The definition used by the system is in graph terms but what also arises is a semantic definition in terms of the domain. For example, the 'long enough taxonomic path' heuristic tests the length of a taxonomy path against its current definition of what long-enough is. If the path tests negatively but feedback from the oracle indicates that the path was sufficiently long then the definition of long-enough is adjusted.

We tackle the problem of getting the initial heuristic knowledge by requiring only loose concept definitions and estimates of their contextual importance from experts. We then use the expert as an oracle to give the system feedback to refine this knowledge. During this training phase the expert merely indicates **like** or **dislike** of, or **indifference** to the system's suggestions as to how to complete or continue a design. Using this feedback, the system then adjusts the heuristic knowledge (either the bounds of a concept definition or the importance associated with a heuristic). This is shown in Fig. 4. The heuristics which arise out of the training mode express both the syntactic, primitive operational concepts in the domain and their higher level semantic definitions.

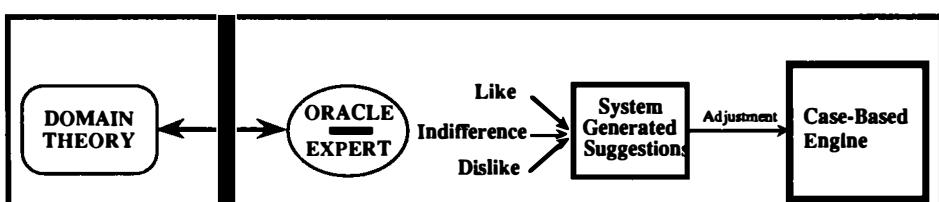


Fig. 4. Using the Expert as an Oracle to Learn Heuristics.

ECMA implicitly assumes goals in the form of the heuristics (i.e. strategies) it forms for use in analysis, matching and suggesting. The analysis heuristics and suggestion

heuristics it derives reflect the best way to understand a model in design terms rather than application terms since they have been acquired with an expert's interaction. Thus they will be useful to LEADER as it, unsupervised, learns new goals of design modifications, and rules that achieve those goals.

4. UNSUPERVISED LEARNING OF DESIGN RULES IN LEADER USING ECMA-DERIVED HEURISTICS

This section shows through an example how a new goal rule (i.e., a triplet: <plan precondition, plan, plan postcondition>) is inferred by LEADER observing an unknown modification to an ER model and using heuristics suggested by ECMA. The chosen modification (Fig. 5) consists of replacing an entity ('teacher') with a more general one ('dept_employee'), while preserving its relationship ('related'), by creating a role relationship (ER models are shown before and after the goal achievement).

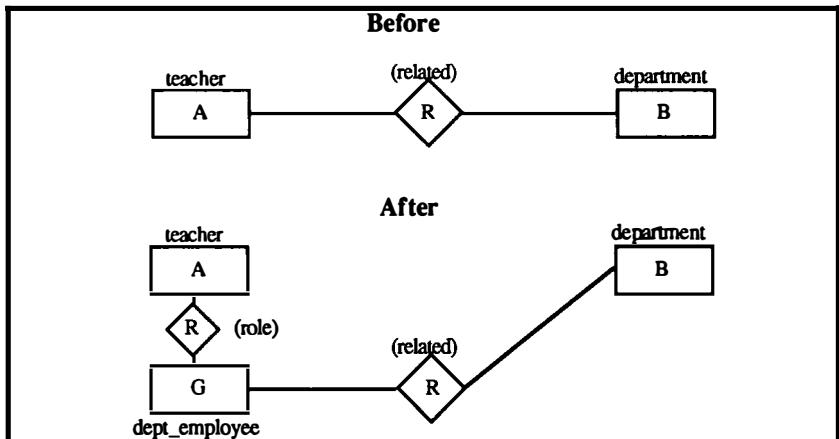


Fig. 5. Modifications of an ER model illustrating the goal "Replacement of an entity with a more general entity".

Assume that some user session is recorded by the Interaction Observer (Fig. 6: the action 'addE' creates an entity, 'addR' creates a relationship, 'chgEN' changes the entity name, and 'delR' deletes a relationship). The model (produced by the Action Executor) obtained by the sequential execution of the first nine actions is shown in Fig. 7.

```

(a1: addE(teacher),
 a2: addE(department),
 a3: addR(teacher, department, related),
 a4: addE(university),
 a5: addE(student),
 a6: addE(employee),
 a7: chgEN(employee, dept_employee),
 a8: addR(teacher, dept_employee, role),
 a9: addR(dept_employee, department, related),
 a10: delR(teacher, department),
 a11: addE(department, university, related),
 a12: addE(teacher, student, related))

```

Fig. 6. User session used by LEADER to produce a new goal rule.

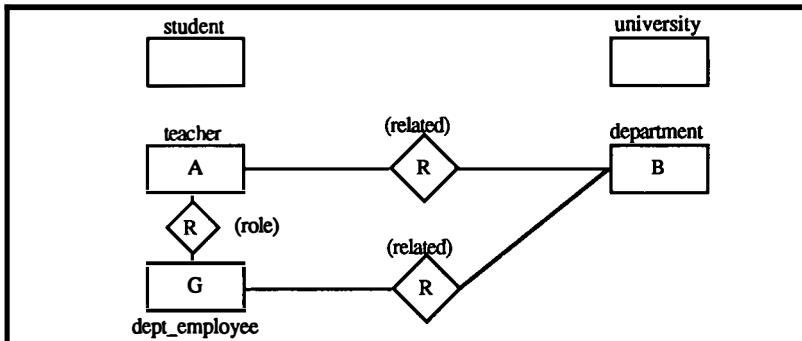


Fig. 7. Model obtained by executing the first nine actions of Fig. 6.

The next step is to identify the part of the user session that can be explained with the current design theory. The explanation is provided by the Interaction Analyzer. Assuming that the design theory contains only the goal rule shown in Fig. 8, the user session of Fig. 6 cannot be entirely explained. However, almost all expected actions of the goal rule's plan are realized (via actions a8 to a10); only the expected action 'addE(G)' (creation of the more general entity) is not.

```

generalize(A) <=
  < entity(A) & entity(B) & relation(A, B, R),
  (
    addE(G),
    addR(A, G, role)
    addR(G, B, R),
    delR(A, B),
    entity(A) & entity(B) & entity(G)
    & relation(G, B, R) & relation(A, G, role) >

```

Fig. 8. Goal rule for generalizing an entity by creating a role relationship.

When LEADER must identify the part of the user session that can be explained with its current design theory many explanations are possible since many goals may be present. For instance, after executing action a3 (Fig. 6), the goal might be to either elaborate a relationship or to elaborate the description of an entity. ECMA can say which goal is the most likely because it has been trained by experts and has heuristic knowledge of how to interpret designs. It might suggest, using the heuristic in Fig. 9, that at this point (a3), the goal is to generalize the relationship.

ECMA is able to indicate this preference from its previous learning. The learning occurs in the following manner: The case library contains completed models which have been parsed and are represented as graphs. The input model is preprocessed to locate a focus cluster. This is a subgraph which is the largest connected component containing the edge which represents the most recently defined relationship in the input model. The focus cluster may be preprocessed to locate its subgraphs by taxonomy, dependence or connectivity.

ECMA must decide whether the input model requires elaboration or generalization. A group of heuristics for each possibility is tested with the input model. One of the heuristics tested is (Fig. 9) : if a taxonomic path of 2 or more nodes exists (in the example of Fig. 7, the path between 'teacher', 'dep_employee' and 'department'), and no generalization relation is present then generalization of a particular node in the path is a goal. The particular node is chosen by the mapping to cases. Through its training with experts ECMA has found this heuristic useful and since its concepts are instantiated in the example, (A = 'teacher', G = 'dept_employee'), it suggests generalization.

Name: '*Recognizing Entity and Relation Generalization*'

Type: Generalization

If a taxonomic path of two or more nodes exists such that one edge in the path is a generalization relation, then the goal was the generalization of a particular node in the path. More precisely, if two entities A and B are related and A is on a taxonomic path to G, and G is related to B, then: 1) A is the entity generalized, 2) G is the result of this generalization, and 3) the relation between G and B is the result of relational generalization.

Fig. 9. An ECMA Heuristic To Recognize Generalization

The assertions that are responsible for introducing the 'dept_employee' node into the model are a6 and a7. By back-substituting constants to variables, LEADER generates a candidate goal rule for achieving the same goal, 'generalize(A)', but using another plan (Fig. 10).

```

generalize(A) <=
  < entity(A) & entity(B) & relation(A, B, R),
  ( addE(X),
    chgEN(X, G),
    addR(A, G, role),
    addR(G, B, R),
    delR(A, B),
    entity(A) & entity(B) & entity(G)
      & relation(G, B, R) & relation(A, G, role) >

```

Fig. 10. Candidate goal rule produced to explain a previously unexplained user session.

Later on, goal rules of Fig. 8 and 10 can be inductively generalized by the LEADER to produce a more structured goal rule (Fig. 11).

```

generalize(A) <=
  < entity(A) & entity(B) & relation(A, B, R),
  ( create_entity(G),
    addR(A, G, role),
    addR(G, B, R),
    delR(A, B),
    entity(A) & entity(B) & entity(G)
      & relation(G, B, R) & relation(A, G, role) >

create_entity(G) <=
  < ~entity(G), addE(G), entity(G) >

create_entity(G) <=
  < ~entity(G), (addE(X), chgEN(X,G)), entity(G) & ~entity(X)>

```

Fig. 11. Candidate goal rules produced to enhance the design theory.

5. CONCLUSION

We have presented a multi-strategy (case-based and explanation-based), closed loop learning method that is applied to database design. The method combines a teacher-trained, case-based component (ECMA) which learns design heuristics and provides suggestions for completing designs. These suggestions are useful to humans and, as well, the second component, LEADER, which does unsupervised learning of design rules. ECMA makes the incomplete theory on which LEADER is based more complete. The second component, LEADER, employs an explanation-based learning approach, in which designs and their modifications are explained in order to identify the salient parts. These salient parts are then subject to generalization. The output of LEADER, together with the ECMA heuristic that led to the rule proposed by LEADER, and with an expert user's evaluation of the learned rule, are fed back to ECMA. The selection process in ECMA can then be rewarded or penalized depending on the success or failure of the proposed heuristic. Rules proposed by LEADER are intermittently submitted to experts for a final verification before being included in the Modeller knowledge base.

6. ACKNOWLEDGEMENTS

The work described here has been supported by Cognos Inc. under the Ontario Technology Fund program. Activities of the labs involved are supported by Natural Sciences and Engineering Research Council of Canada, the Government of Ontario (URIF Program), Cognos Inc, and the Canada Centre for Remote Sensing. We thank B. Tauzovich and R. Stanley for their comments and suggestions on how to improve the paper.

7. REFERENCES

- Hammond, K. J. (1988). Case-Based Planning, *Case-Based Reasoning Workshop*, pp.17-20.
- Kolodner, J. and C. Riesbeck (1989). Cased-Based Reasoning, Tutorial, *IJCAI*.
- O'Reilly, U.-M. and F. Oppacher (1990). Using the Expert as an Oracle for Knowledge Acquisition in Case-Based Reasoning Systems, *Knowledge Acquisition Workshop*, *AAAI*, Boston, MA.
- Riesbeck, C. K. (1988). An Interface for Cased-Based Knowledge Acquisition, *Case-Based Reasoning Workshop*, pp.312-326.
- Tauzovich, B. (1990). An Expert System for Conceptual Data Modelling, in Elsevier Science Publisher, B.V. [North-Holland] (eds), *Entity-Relationship Approach to Database Design and Querying*, pp.205-220.
- Tsichritzis, D. C. and F. H. Lochovsky (1982). *Data Models*, Prentice-Hall Inc.

A knowledge acquisition system for conceptual design based on functional and rational explanations of designed objects

O. Katai,[†] H. Kawakami,[§] T. Sawaragi[†] and S. Iwai[†]

[†]Department of Precision Mechanics

Kyoto University

Yoshida Honmachi, Sakyo-ku, Kyoto 606 Japan

[§]Department of Information Technology

Okayama University

Tsushima-naka, Okayama 700 Japan

Abstract. An EBL-based system for acquiring conceptual design knowledge in physical systems was proposed and implemented based on Value Engineering Methodologies and Axiomatic Design Approaches. The key idea behind this system is that such knowledge can be acquired by analyzing existing designed objects which can be regarded as the result of rational decisions and actions. In this system, the structural features of designed objects are analyzed by domain specific knowledge to yield a systematic explanation of how they function and attain their design goals and why they are used for attaining the goals. The "how" explanation results in a generalized version of the Functional Diagram used in Value Engineering from which the object in question can be interpreted via two kinds of design rationalities, i.e., teleological and causal ones. Namely, a designed object is regarded as being consisted of a hierarchy of design goals (primary functions), subgoals (subfunctions), structures and substructures toward attaining those goals. We applied the EBL method to this hierarchical model to acquire general design knowledge which is operational at various phases and fields in the conceptual design. Through organizing domain-specific knowledge of this EBL system according to the above hierarchical model, various levels of knowledge can be extracted by a single positive instance. The quality of the extracted knowledge is then discussed with reference to its level in the hierarchy of acquisition.

The "why" explanation gives us a deeper understanding of the designed objects from which we can then extract meta-planning or strategic knowledge for selecting rational plans from among other possible alternatives. This deep explanation is obtained by regarding the object in question as being the result of a sequence of strategically rational decisions and actions which are subject to the principles of "good" design that are formalized as an axiomatic system in the Axiomatic Design Approach.

INTRODUCTION

It is generally acknowledged that a detailed systematic analysis of existing designed objects is beneficial not only for their improvement but also for the novel design of objects. **Value Engineering** introduced by Miles (1961) provides a systematic method of analysis for these improvements and novel designs. The analysis, the so-called **Functional Analysis**, focuses on the functional composition of objects yielding the so-called **Functional**

Diagram by which we can deduce new ideas for improvements or design that are not constrained by the existing parts or components. Even though the knowledge embedded in the Functional Diagram concerning the functional composition, structural composition and function-structure relations may be beneficial to the improvement of the object analyzed, it lacks "generality", i.e., it is not utilizable for the improvement and new design of other objects. This is because the Functional Diagram (i) applies only to the particular object analyzed and (ii) is dependent on the particular way it was derived, i.e., it is dependent on the particular method of analysis used and also on the person or the group of people who derived it. In order to solve problem (ii), Bytheway (1971) proposed a systematic method called FAST (Functional Analysis System Technique). Even by the use of this method, however, a need still exists for standardizing the representation of the Functional Diagram. Moreover, taking into account problem (i), we arrive at the conclusion that it is necessary to introduce a computer-supported systematic method of Functional Analysis for deriving "standardized" and "generalized" Functional Diagrams.

In this research, we propose an **EBL (Explanation-Based Learning)** system which derives "standardized" and "generalized" Functional Diagrams from which we can extract various kinds of knowledge for conceptual design of physical systems. The **Explanation-Based Generalization (EBG)** of the object based on first order logic resolves problem (i), and the standardization and comprehensiveness of the domain-specific knowledge used to derive the explanation resolves problem (ii).

The above explanation shows how the design goal, i.e., the primary function of the object analyzed, is attained by the use of functional composition together with the structures supporting these functions and subfunctions. This type of explanation is, however, a shallow explanation of the object. For the deep understanding of the object, it is necessary to explain the reason "why" these functional and structural compositions of the object were selected from among other alternative choices. In this research, we will introduce a method of explanation based on the **Axiomatic Design Approach** by Suh et al. (1978) whose key idea is that design should be guided according to several principles of "good design" which are derived from an axiomatic system. From this deep explanation, we can acquire various strategic or meta-planning knowledge for conceptual design by which the order of attaining design goals and the way of resolving interaction among design goals can be clarified. This "why" explanation is derived from the same basis as that of the "how" explanation, the two styles of explanation sharing the common basis.

In the second section, we introduce a hierarchical way of modeling of designed objects from the Value Engineering Perspective. In the third section, we develop an EBL system for acquiring various design knowledge which extracts knowledge from designed objects by the use of their functional analyses based on the domain theory derived from the modeling. In the fourth section, we show how the generalized functional diagram is generated. We also introduce various levels of knowledge extraction from this diagram, and the quality of the acquired knowledge is discussed in relation to its respective level of acquisition. In the fifth section, we develop a system for acquiring "deep" or strategic design knowledge based on the Axiomatic Design Approach. This is based on the "why" explanation of designed objects, i.e., the reason why the designed objects "necessarily" have the structural features they do.

HIERARCHICAL MODELING OF DESIGNED OBJECTS AND THE MULTILAYERED STRUCTURE OF DESIGN KNOWLEDGE

Hierarchical Modeling of Designed Objects

In Value Engineering, Functional Analysis techniques analyze how the design goal (the primary function) is attained by the use of functional and structural composition of objects. The analysis results in the so-called Functional Diagram on which basis we can improve the object in question by searching for better structural components or a completely new structural composition. By focusing on the functional composition of the object, the modification or structural alteration is not constrained by the existing structural composition or by the components of the object. Figure 1 shows the Functional Diagram of a sensing device (Figure 2) used to measure the velocity of melted lead (Iwasaki, 1985). This diagram shows that the primary function, "to measure the velocity of melted lead," consists of three subfunctions, "to detect ...", "to transmit ..." and "to transduce ...", and that the second subfunction also consists of three primitive subfunctions F2, F3 and F4. Each primitive subfunction is supported by a set of structural entities. For instance, each of the leafs F1 - F5 of this diagram is attained by a substructure of the device as shown in the semantic network in Figure 2. From the above perspective of Value Engineering, the process of design can be interpreted as a successive selection of "ends" and "means" which correspond to design goals (primary functions), subfunctions, and substructures such as primitive components, materials, etc. (Watanabe, 1984).

Rationality in Design and the Multilayered Structure of Design Knowledge

The above selection process can be conceived as comprising two kinds of rationalities, teleological and causal, as follows:

Teleological Rationality: the rationality on the selection of the functional composition of design.

Causal Rationality: the rationality on the selection of the structural composition of design for attaining the subfunctions.

In order to explain these rationalities of designed objects, we will need the following kinds of design knowledge:

Planning Knowledge: knowledge on how to attain a goal (primary function) by combining subgoals (subfunctions).

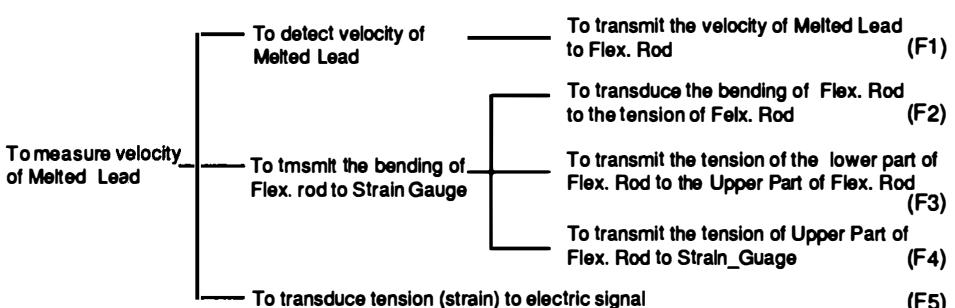
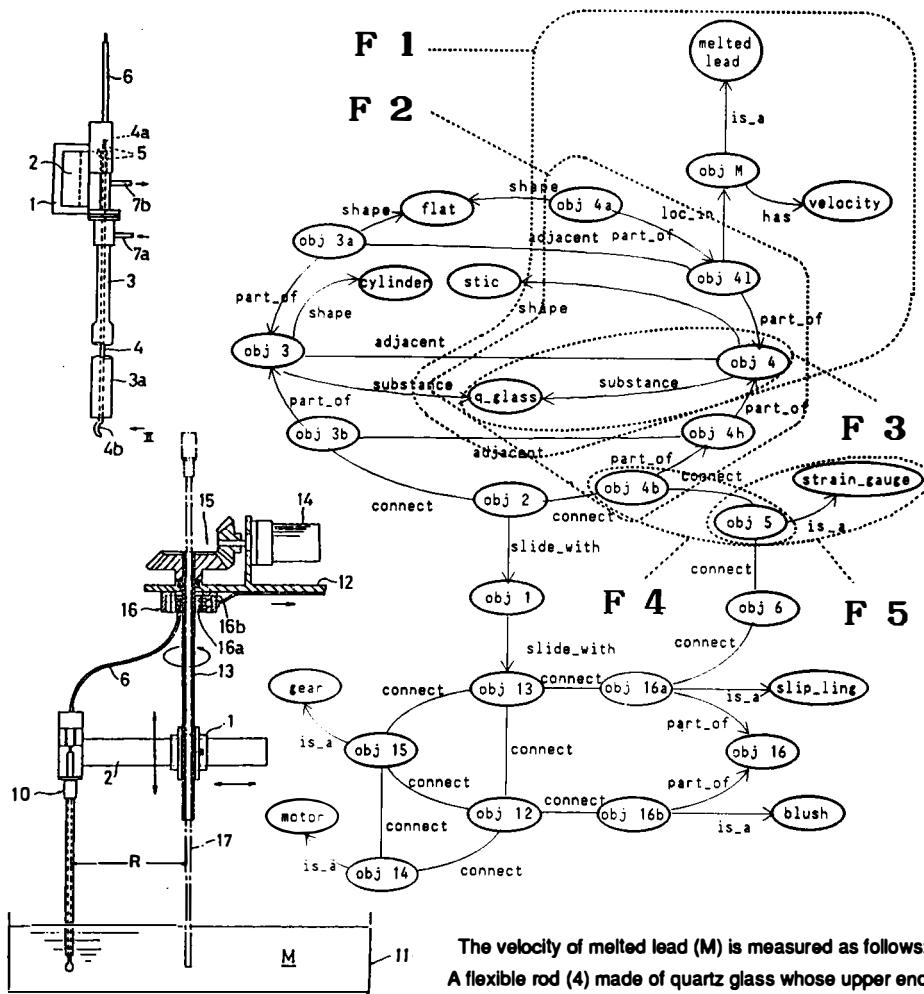


Figure 1. Functional Diagram of the "flexible rod" for measuring the velocity of melted lead



obj1 :Slider	obj11:Tank
obj2 :Arm	obj12:Base
obj3 :Tube (Quartz Glass)	obj13:Pivot
obj4 :Flexible Rod (Quartz Glass)	obj14:Motor
obj5 :Strain Gauge	obj15:Bevel Gear
obj6 :Electric Wire	obj16:Brush and Slip ring
	objM :Melted Lead

The velocity of melted lead (M) is measured as follows:
A flexible rod (4) made of quartz glass whose upper end (4b) is supported by the arm (2). The spoon-like lower end (4a) is placed in M, so that the mass flow of M bends the rod producing tension on its surface, since the rod is made of an elastic material. Finally, the strain gauge (5) attached to 4b transduces tension (strain) as an electric signal.

Figure. 2 The structure and behavior of the flexible rod

Physical-Law Knowledge: knowledge on causal laws which may be used to explain the sufficiency of a substructure for attaining a subfunction.

Planning Knowledge mediates (relates) a certain set of possible primary functions (**Goal Space**) with a certain set of possible functions (**Function Space**). Physical-law Knowledge mediates Function Space with a certain set of possible Structure-attributes (**Structure-Attribute Space**).

Figure 3 shows the above multilayered structure of design knowledge.

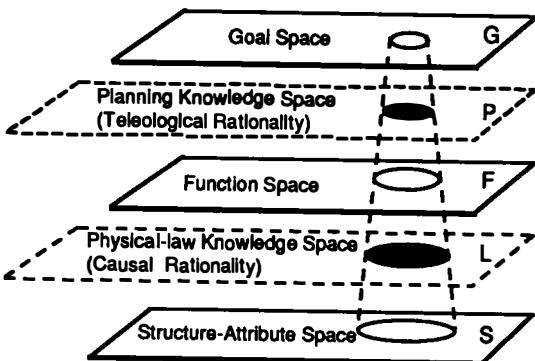


Figure 3. Multilayered structure of design knowledge

CONSTRUCTION OF THE EBL SYSTEM

Functional Analysis and EBL

As mentioned in the Introduction, it is necessary to introduce a systematic method to derive "standardized" and "generalized" Functional Diagrams. We will construct, in this section, a method based on EBL (Explanation-Based Learning) introduced by Mitchell et al. (1986). By the use of this method, Functional Analysis can be done in a quite systematic way and the resultant Functional Diagram contains all the essential knowledge and no other surplus information on design, i.e., it shows a "general" way for attaining the primary function. We adopted a two-step strategy for acquiring design knowledge. The first step produces a standardized and generalized Functional Diagram from which, in the second step, various levels of conceptual design knowledge are then extracted.

Domain theory of the EBL system

In order to derive the above Functional Diagrams, it is necessary for the EBL system to have the following kinds of domain-specific knowledge (**Domain Theory**): Planning Knowledge, Physical-Law Knowledge, Knowledge on Structure-Attributes. In the following, we will confine ourselves to the case of conceptual design of sensing devices (Katai et al., 1984).

The Domain Theory of this EBL system, illustrated in Figure 4, consists of five kinds of knowledge, each of which is given as a set of sentences in fact form or Horn clause form as follows:

(1) **Goal Space G:** The goal of sensing devices is to measure the amount of specified physical quantity together with auxiliary goals such as "being adapted to the environment," etc. In the above example, the goal is not only to measure the velocity but also "to be resistant to high temperature" and "to localize the velocity measurement". As the primary goal, the "GC", "to measure fluid velocity," is selected from among the GC's in the space G.

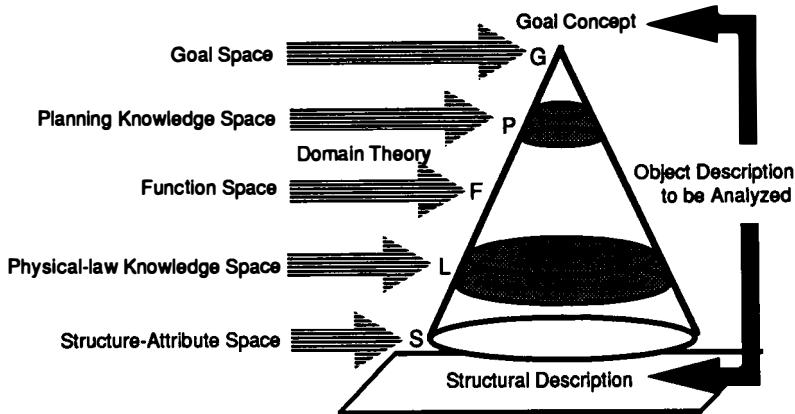


Figure 4. Domain theory of the proposed EBL system

(2) **Physical-law Knowledge Space L:** The process of measuring physical quantities usually consists of causal chains of physical laws as illustrated in Figure 5 which shows the causal process in the above velocity measurement. In this EBL system, each of the causal laws (such as M₁, M₂, ...) is encoded as an implicational law (in Horn clause form) whose conditional and conclusive parts are set to be the most general. By instantiating these general components by unification as shown in the figure, M₁, M₂, ..., and M₅ constitute a causal chain for measuring the velocity in terms of change in resistance of the strain gauge. For implementing the knowledge in this space, we referred to several technical dictionaries (Hix and Alley, 1958) on physical laws and effects in which concise explanations of each law or effect are described.

(3) **Function Space F:** Functions common to the area of sensing devices are to detect, to compare, to balance, to transmit, to transduce, etc. These functions are attained by sequences of sub-functions or physical laws, (p_p)'s, or their combinations. Each knowledge in this space is on the general laws which hold among these functions and physical laws. For instance, a physical quantity P of an entity X will be transmitted to R of Z when there is a physical law relating X with P to Z with R, or there is an entity Y with a quantity Q which mediates X with P and Z with R.

A crucial problem in Functional Analysis in VE is how to formalize and represent various kinds of functions. The usual practice is to formalize them as the combination of verbs and nouns such as "to <verb> to <noun>". The Japan Value Engineering Society (1982) examined a large number of functional verbs and selected about 140 representative ones to stand for various kinds of functions. These verbs were classified into four categories such as those on effect, those on change, those on movement, and combinations thereof. Based on this analysis, we encoded these verbs using CD theory (Schank, 1972) by examining the conditions on the subject, those on the attributes of the objects (nouns), and those on the paths of influence from the subject to the object (Kawakami, 1987). From this analysis, we could derive a hierarchical structure of these functional verbs by which the mismatching of functional representations in deriving the Functional Diagram can be prevented. Also the Society examined the nouns for functional analysis, which led them to impose certain restrictions on the nouns, such as "they should be material nouns and be quantitative," and so

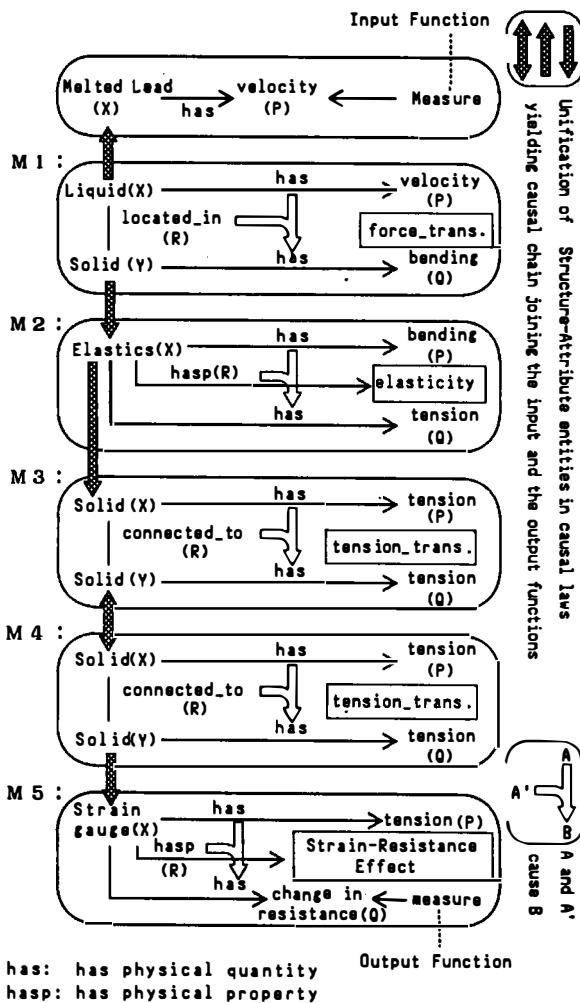


Figure 5. Causal chain explanation of the fluid velocity measurement

on, and to introduce a categorization of these nouns, both of which are also useful for preventing mismatching. In the world of sensing devices, however, these verbs and nouns are more confined and can be rigorously treated since we have a well-defined domain theory of physics which allows for the use of restricted terms and in which the relations among these terms have been fully investigated.

(4) Planning Knowledge Space P: One typical plan for measuring a physical quantity is to detect it and then to transmit it to the transducer which then transforms it into a specified type of quantity such as electrical current, voltage, etc. Another typical one is to balance and compare it with a standardized quantity and then to transform their difference into a specified type of quantity. The knowledge in this space is prepared based on a comprehensive list of

sensing methods, most of which were selected and coded by referring to the JIS Manual for measurement instrumentation (JIS, 1989), which was also used to control or standardize the technical vocabulary in the domain theory of spaces P, F, L and S.

(5) Structure-Attribute Space S: This space contains all the necessary knowledge on facts and general laws such as the inheritance of attribute values via "is-a" or "part-of" relations that are commonly accepted or should be utilized in the phase of structural design. Figure 6 shows a portion of this domain knowledge, where the predicate "dt" stands for "domain theory", "p_p" for "physical laws", "hasp" for "has property", etc. Each "dt" sentence with a single argument represents a fact, and that with two arguments represents a Horn clause whose second argument is a sufficient condition for the first argument being the case.

This multilayered structure of domain theory makes it possible to rather independently prepare domain-specific knowledge in each space, hence we can set the domain theory of EBL modularly and comprehensively. The chunking mechanism of EBL makes the compilation of these modularized pieces of knowledge yield various kinds of operational knowledge which are then utilized to derive functional explanations of the objects analyzed. This is the reason why we can extract various kinds of operational design knowledge from a

Menu for choosing a Training Example	Menu for choosing the Domain Theory
<pre> File Edit Find Windows Fonts Eval Example Theory Explain EBLw ListOfDomainTheory dt(hasp(X, Y), [substance(X, Z), hasp(Z, Y)]). dt(hasp(X, Y), [part_of(X, Z), hasp(Z, Y)]). dt(is_a(X, Y), [+((atom(Y))), is_a(X, Z), is_a(Z, X)]). dt(hasp(quartz_glass, heat_resist)). dt(hasp(quartz_glass, collision_free)). dt(hasp(glass, transparency)). dt(hasp(metal, heat_conduct)). dt(hasp(metal, elec_conduct)). dt(phase(glass, solid)). dt(substance(iron, metal)). dt(is_a(ammeter, elec_device)). dt(is_a(strain_gauge, elec_device)). dt(hasp(elec_device, elec_conduct)). dt(substance(X, Y), [part_of(Z, X), substance(Z, Y)]). dt(connect(X, Y), [part_of(X, Y)]). dt(p_p(L, velocity, S, bend), [loc_in(S, L), phase(L, liquid), phase(S, solid)]). dt(p_p(S, bend, S, tension), [hasp(S, elastic)]). dt(p_p(S1, P, S2, P), [connect(S1, S2), +(S1==S2), phase(S1, solid), phase(S2, solid)]). dt(measure(X, P), [detect(X, P, Y, Q), transduce(Z, R), trans(Y, Q, L, Z, R)]). dt(detect(X, P, Y, Q), [p_p(X, P, Y, Q)]). dt(transduce(Z, tension), [is_a(Z, strain_gauge)]). dt(trans(Y, Q, L, Z, R), [p_p(Y, Q, Z, R)]). dt(trans(Y, Q, [(X,P)L], Z, R), [p_p(Y, Q, X, P), trans(X, P, L, Z, R)]). </pre>	<pre> File Edit Find Windows Fonts Eval Example Theory Explain EBLw ListOfExample ex(connect(o13, o15)). ex(connect(o13, o16a)). ex(connect(o14, o15)). ex(part_of(o3, o3a)). ex(part_of(o3, o3b)). ex(part_of(o4, o4a)). ex(part_of(o4, o4b)). ex(part_of(o16, o16a)). ex(part_of(o16, o16b)). ex(through(o4, o3)). ex(loc_in(o4a, om)). ex(slide_with(o1, o2)). ex(is_a(o5, strain_gauge)). ex(is_a(o14, motor)). ex(is_a(o15, gear)). ex(is_a(o16, slip_ling)). ex(is_a(o16b, brush)). ex(substance(o3, quartz_glass)). ex(substance(o4, quartz_glass)). ex(substance(om, lead)). ex(phase(om, liquid)). ex(shape(o3, stic)). ex(shape(o3a, flat)). ex(shape(o4, pipe)). ex(shape(o4a, flat)). ex(has(om, velocity)). </pre>

Figure 6. A portion of the domain theory of the EBL system and the description of the training example

simple structure such as the flexible rod shown later.

The EBG Inference Part

The inference part of the system is implemented compactly, as a meta-interpreter of prolog, by extending the EBG inference engine proposed by Keder-Cebelli and McCarty (1987) and Hirsh (1987). This inference engine generates a functional explanation tree (Functional Diagram) of an arbitrarily given designed object, which is then generalized to produce a **Generalized Functional Diagram (GFD)**.

In the EBG process, each "dt" sentence in the above Domain Theory is treated as a fact sentence. By referring recursively to the second arguments (sufficient conditions) in the "dt" sentences, an explanation tree is generalized. The engine is implemented using LPA prolog on Mac II, on which exists the facility for visualizing the generation process of GFD with the use of various fonts, that for extracting design knowledge from a partial GFD (without completing the generation process), and that for storing the current part of GFD, which will be used for the visualization, together with other user-friendly interface facilities, as shown in Figure 7. These facilities are quite important for revising the domain theory when failure or misleading explanations occur.

The style of inference here is a hybrid one, with the knowledge in Space P providing top-down inference and that in Space S bottom-up inference. This is in distinct contrast with the traditional derivation of Functional Diagrams by Functional Analysis such as in FAST where only bottom-up (structure-based) analyses can be utilized. This EBG inference and the above multilayered and comprehensive structure of domain theory thus standardize the resultant Functional Diagrams, making it as unbiased and generalized as possible.

DESIGN KNOWLEDGE ACQUISITION BY THE EBL SYSTEM

Generation of Functional Diagram and Its Generalization

The input to the proposed EBL system is the semantic network representation of the structure of a sensing device as the training example (cf. Figure 2) and its primary function is the GC, the goal concept. The explanation tree derived from the system shows "how" the primary function is attained by the use of plans, subfunctions, physical-laws, structure-attributes; by referring to the domain knowledge in Spaces P, F, L and S; by supplementing to the original semantic network (Figure 2) with necessary terms from the domain knowledge; and by filtering out unnecessary terms which have no use in attaining the primary function as shown in Figure 8. This network is then abstracted by generalizing terms which are dependent on the particular object in question, yielding the GFD, as shown in Figure 9(a).

Figure 10 shows the GFD obtained by our EBL system for the sensing device shown in Figure 2. Compared with the manually-made diagram in Figure 1, the new diagram is more systematic and standardized. It contains all the necessary information for the measurement and no other surplus information. In the figure, "transmit(Y1, bend, Z1, tension)" shows that the amount of bending in the structural component Y1 is transmitted to the amount of tension in component Z1. Containing the more general information of how to measure fluid

Construction of the explanation tree is processing

File Edit Find Windows Fonts Eval Example Theory Explain EBLW

```

construct
  current_meter
    measure(om, velocity)
    detect(om, velocity, o4a, bend)
    p_p(om, velocity, o4a, bend)
      loc_in(o4a, om)
      phase(om, liquid)
      phase(o4a, solid)
        part_of(o4, o4a)
        substance(o4, quartz_glass)
        is_a(quartz_glass, glass)
        phase(glass, solid)
    transduce(o5, tension)
      is_a(o5, strain_gauge)
    trans(o4a, bend, [o4a, tension])|_173697, o5, tension)
      p_p(o4a, bend, o4a, tension)
        hasp(o4a, elastic)
        substance(o4a, _226093)
          part_of(_226092, o4a)
          substance(_226092, _226093)
        hasp(_226093, elastic)
    trans(o4a, tension, _173697, o5, tension)
    phase(om, liquid)
  
```

Slanted characters indicate the structural components of the training example which were unified with the domain theory

Bold-faced characters represent the part of the explanation tree on which the EBG inference system are now working

NEXT **ABORT**

File Edit Find Windows Fonts Eval Example Theory Explain EBLW

```

construct
  current_meter
    measure(om, velocity)
    detect(om, velocity, o4a, bend)
    p_p(om, velocity, o4a, bend)
      loc_in(o4a, om)
      phase(om, liquid)
      phase(o4a, solid)
        part_of(o4, o4a)
        phase(o4, solid)
        substance(o4, quartz_glass)
        phase(quartz_glass, solid)
        is_a(quartz_glass, glass)
        phase(glass, solid)
    transduce(_5682, _5683)
    trans(o4a, bend, _5681, _5682, _5683)
    phase(om, liquid)
  
```

You can chunk the bold-faced part of this tree into a compiled knowledge

SURE TO SHIFT MICRO ?

YES **NO**

Figure 7. Illustration of the interaction between user and the EBL system

Supplemented from the Domain knowledge in Space S

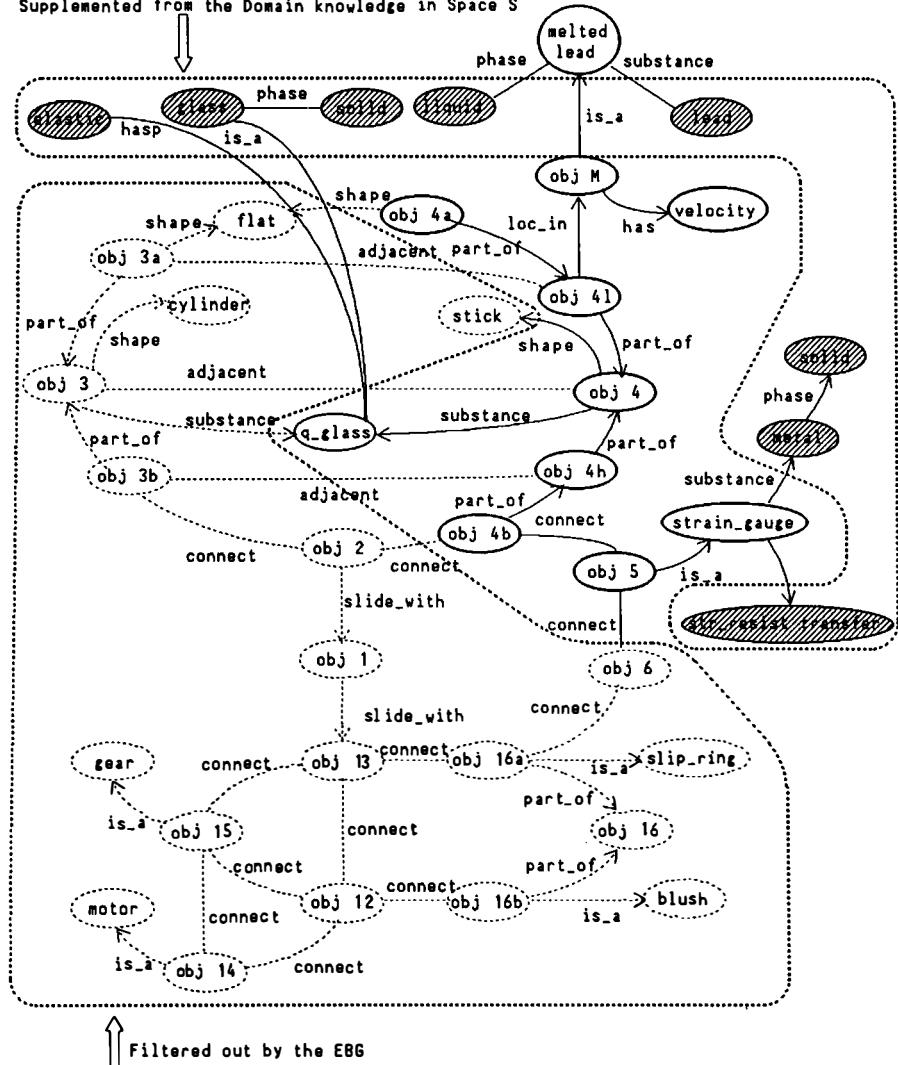


Figure 8. Deletion and augmentation of entities in the original semantic network by the EBL process

velocity "in general," the new diagram is not confined to measurement of melted lead. Moreover, it should be noted that the terms (arguments of "dt") located in space L constitute a "causal chain" joining (L1,velocity) to (Z1,cir), whose existence are supported by the terms in space S.

It should also be noted that the new diagram is based on the causal chain explanation which refers to the underlying physical laws. This style of explanation makes the functional diagram complete and put it in full detailed-form from which all the other possible explanations at various levels of abstraction can be extracted. Moreover, this prevents the

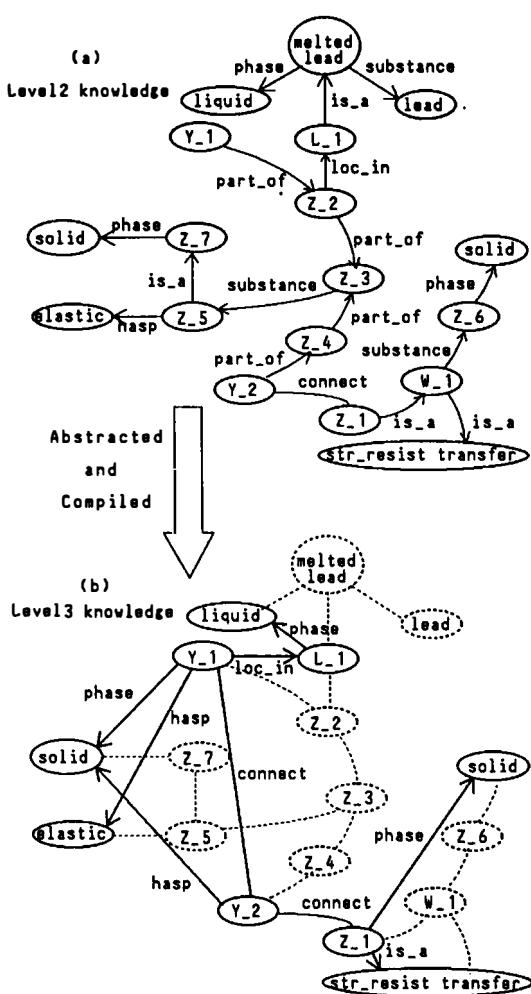


Figure 9. Abstracted and compiled description of the training example

structure which naturally reflects the hierarchy of the design model in Figure 3. We set three levels in the hierarchy of knowledge extraction from the Generalized Functional Diagram, by which different kinds of knowledge are acquired as shown in Figure 11.

The first level (**Level 1**) is set as the top (uppermost) level in Space F (Function Space). In this case, we obtain associational knowledge relating the primary function to the component functions whose combination attains the primary function. In other words, we obtain the general planning knowledge for attaining the primary function. Namely, we can use the EBL system to extract the general knowledge which is embedded implicitly in the mediating spaces between GC and the level of extraction. The uppermost broken line (---) in Figure 10 indicates this level of knowledge acquisition, and the acquired knowledge

occurrence of the multiple explanation problem in EBL. Namely, as mentioned by Rajamoney and DeJong (1978), the problem usually caused by the lack of domain knowledge required for selecting the true causal explanation from among other alternative explanations. Also as Hempel (1965) observed in his theory on scientific explanation, the causes of this problem are the underdeterminedness of the main causes and the optional localization of overdetermined causal chains. The detailed causal explanation used in our functional analysis provides a mechanism for selecting the true and main causal chain without localizing it. When a problem arises due to the lack or imprecision of the domain theory, we can use the user-friendly visualization facility to our explanation system to search for the problematic part of the domain theory.

Knowledge Acquisition from the Generalized Functional Diagram

The Generalized Functional Diagram has a hierarchical

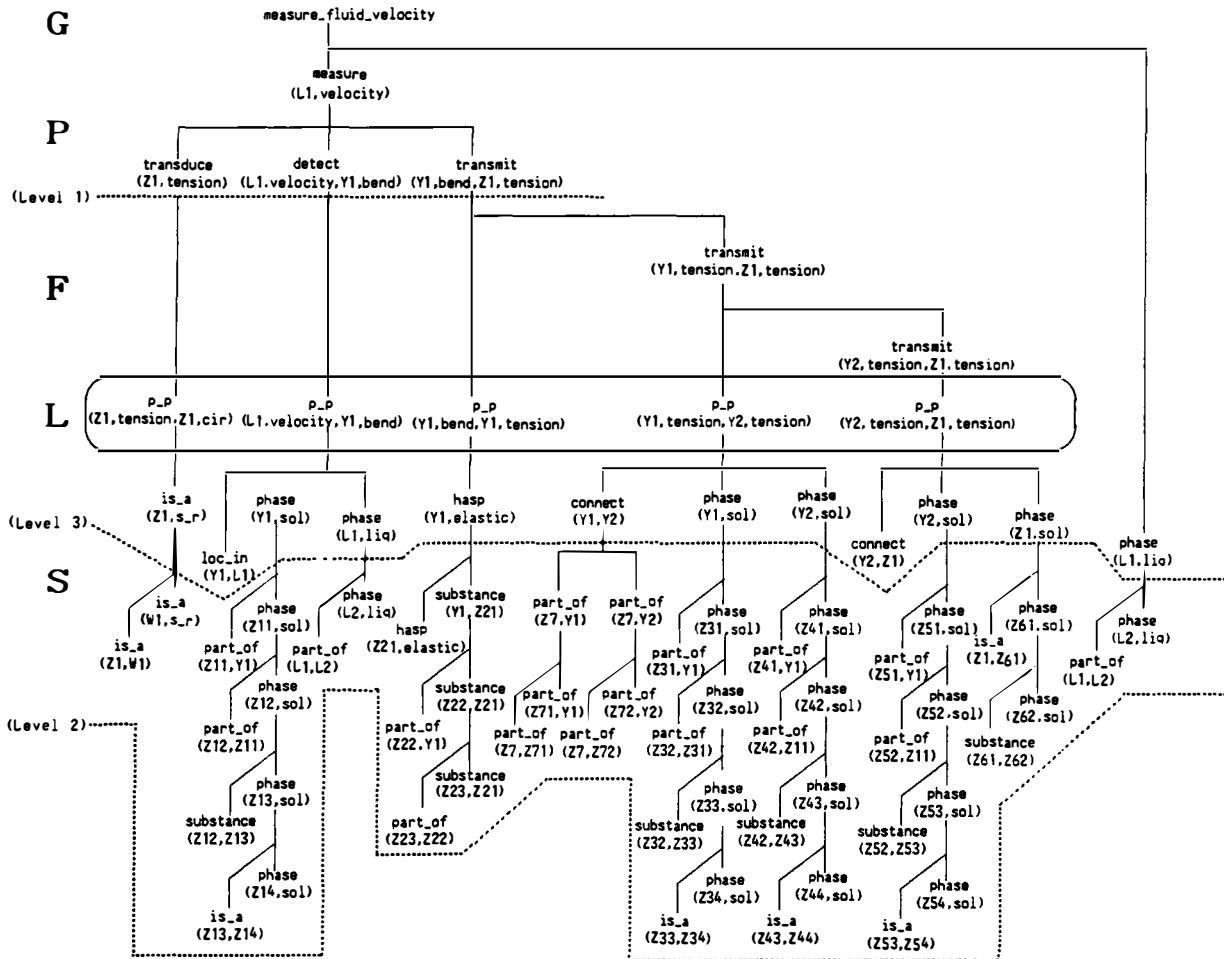


Figure 10. The resultant Generalized Functional Diagram for the example in Figure 2

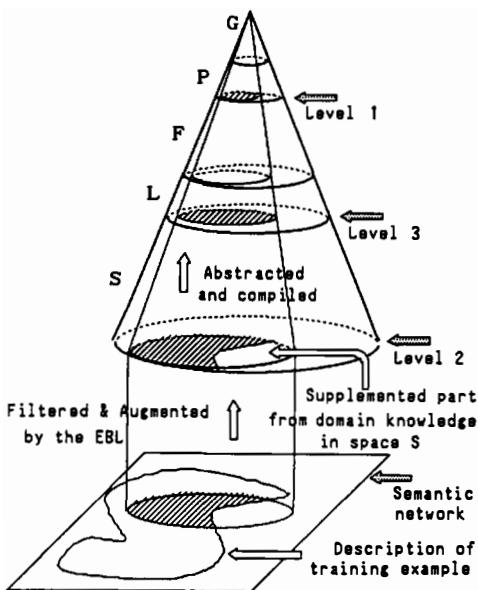


Figure 11. Three levels of knowledge extraction from the GFD

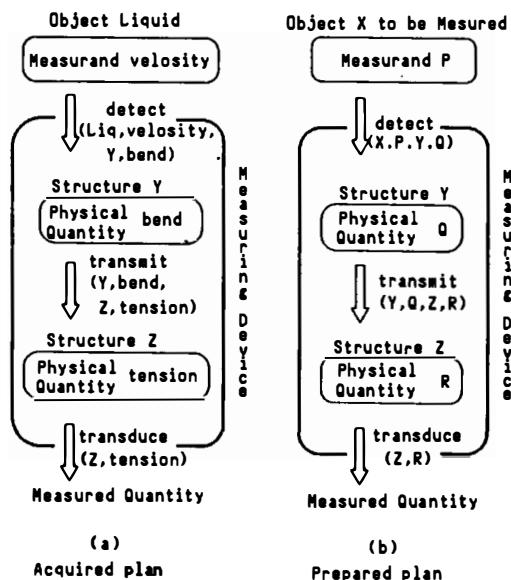


Figure 12. Acquired knowledge (a) for measuring fluid velocity , which can be regarded as a specialization of domain knowledge (b) prepared in the planning knowledge space P

arrived at is shown in Figure 12(a). This knowledge can be readily seen as a specialization of the original planning knowledge (Figure 12(b)) in the Planning Knowledge Space.

In general, the acquired knowledge has teleological and/or causal rationalities when the mediating spaces P and/or L (Planning Knowledge Space and/or Physical-law Knowledge Space) of these rationalities are located between GC and the level of extraction. It is quite difficult to prepare the domain knowledge in Space P beforehand such that it has causal rationalities; to do so we would have to examine a large number of cases of usage for each physical law. Hence, we have adopted an approach in which the domain knowledge in Space P is rational only in the teleological sense. This knowledge becomes causally rational when compiled with the knowledge on Space L or S by the EBL process. Hence, the acquired knowledge (in Level 1) in Figure 12(a) is causally rational, although it is derived from the planning knowledge prepared in Space P shown in Figure 12(b) which is not necessarily causally rational.

In design problem solving, the domain knowledge is usually not distinct but rather unbounded, hence it is quite difficult to prepare a comprehensive and complete domain theory beforehand. The above chunking property of EBL suggests the possibility of the domain theory of EBL to have self-organizing and learning properties by which

simple and fragmentary knowledge prepared in the domain theory can be gradually compiled to yield a complex and comprehensive domain knowledge. If we supplement the domain theory with these pieces of compiled knowledge, we will run into another kind of multiple-explanation problem mentioned, i.e., we will derive another compiled (chunked) causal chain explanation. Hence, what we need is a new facility of the inference system which enables us to refer to the original componential knowledge whose compilation yielded the knowledge utilized in the chunked explanation.

The second level (Level 2) of knowledge extraction is the bottom (lowest) level of Space S (Figures 10 and 11), by which we can acquire the most detailed or concrete structure which reflects in fidelity the original structure of the input instance (Figures 2 and 10). The obtained structure is shown in Figure 9(a), which is yielded by filtering out the irrelevant components from the original structure in Figure 2 and by supplementing the relevant components from the domain knowledge in Space S as shown in Figure 8.

The third level (Level 3) of extraction is set as the top (uppermost) level in the Space S (cf. Figures 10 and 11) which yields the most general (abstract) structure (and attributes) for measuring fluid velocity. The knowledge in this level can be regarded as being the abstracted and compiled version of the knowledge in Level 2 as shown in Figure 9. This abstraction and compilation are supported by the domain knowledge in Space S and are guided by the domain knowledge in Spaces P, F and L as shown in Figure 11.

Levels 2 and 3 thus provide knowledge which can be utilized in the structural design phase while Level 1 provides knowledge for the functional design phase. The knowledge acquired by Level 3 is expected to be more abstract and less efficient than that acquired by Level 2.

The system implemented thus far is in prototype phase, and a few more examples of sensing devices were analyzed, yielding similar functional diagrams and various operational design knowledge.

ACQUISITION OF META-PLANNING KNOWLEDGE BY AXIOMATIC DESIGN APPROACH

Introduction of Axiomatic Design Approach

The above explanation shows "how" the GC (the primary function) is attained by the use of subfunctions, physical laws, structures and attributes. In order to explain the reason "why" they are used for attaining the GC, we have to examine the above "how" explanation structure from certain principles of good design to which strategically rational processes of design should be subject.

To derive this kind of explanation, we have to take into consideration the other primary functions as well such as "to be resistant to high temperature" and "to localize the velocity measurement." Namely, we have to search for the "deep" explanation of designed objects which shows the necessity for the extra structures and attributes which were not used in the explanation in the Generalized Functional Diagram obtained in the previous section. We will employ the following Axiomatic Design Approach to derive the above explanations.

Suh et al. proposed an axiomatic design approach based on the idea that design process

should not remain in the field of experience and artistic skill but should be guided by a formal axiomatic system as follows (Suh et al., 1978):

Axiom 1: In good design the independence of functional requirements is maintained.

Axiom 2: Design is optimized by minimizing information content.

Each functional requirement corresponds to a GC or a primary function in designed objects; the information content represents the complexity of the object in terms of its description and production.

The first axiom guides the process so that the resultant design becomes more complex; supplementary functions and structures are sometimes required to maintain the independence of design requirements (GCs). The second axiom guides the resultant design so as to be as simple as possible. For instance, in an advanced refrigerator design, the two functional requirements, i.e., "to cool food" and "to store food", should be independent of each other. To solve this problem, it increases the number of doors, thus preventing the loss of refrigerated air whenever a door is opened to put in or take out foodstuffs (cf. Figure 13(a)). When the doors are set on top of a refrigerator, however, as in the case of a "deep freeze", the interaction between the two GCs becomes quite small, and they can be regarded as being wholly independent of each other. In this latter case, according to Axiom 2, the number of doors should be as few as possible, hence just using a single door is sufficient and optimal (cf. Figure 13(b)).

Extraction of Meta-Planning Knowledge

Usually, these axioms are not applied directly to design processes but instead their corollaries such as "Decouple or separate parts of aspects of a solution if functional requirements are coupled or become interdependent in the design", "Integrate functional requirements in a single part of solution if they can be independently satisfied in the proposed solution", "Conserve materials and energy", etc. are used to guide the processes (cf. Suh et al. (1981)). These corollaries can be readily seen to be compiled and operationalized version of the two axioms.

In this research, we do not utilize these corollaries to derive the deep explanation structures, but instead utilize the original axioms directly. From the resultant explanation structure, we derive the meta-planning knowledge which can be regarded as the compiled and operationalized version of the axioms.

The "deep" explanation process is as follows:

(a) Search for the most simple structures with the least set of attributes (simplified design) which attain the most primary design requirement (GC). This search is done according to Axiom 2. In the above example, the GC is the measurement of fluid velocity, and the resultant simplified design of the device is shown in Figure 14(a).

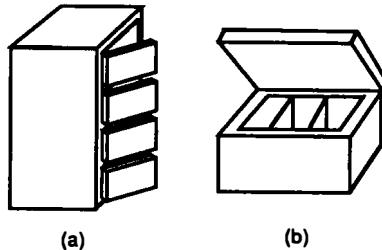


Figure 13. An illustrative example of Axiomatic Design Approach

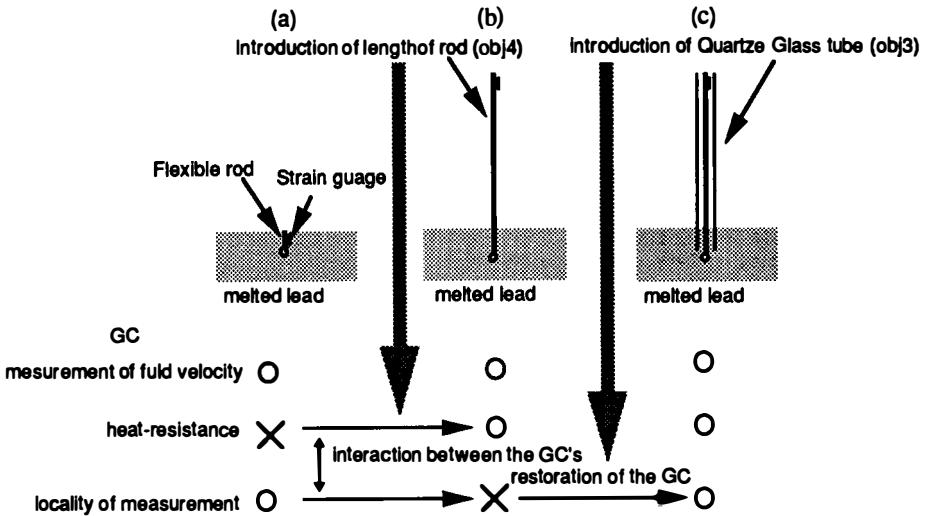


Figure 14. Successive generation of modification and explanation yielding meta-planning knowledge for the system in Figure 2

(b) Check if the other design requirements (GC's) are satisfied. In the example, the other design requirements (GC's) are "to be resistant to high temperature" and the "locality of measurement". In this case, the former one is shown to be unsatisfied.

(c) If some of the GC's are not satisfied, search for supplementary structures and attributes (in the original design) which support the unsatisfied GC's. In this case, the length (attribute) of the rod is found to support the GC, "to be resistant to high temperature".

(d) It should be noted, however, that this satisfaction may have side effects, i.e., it may cause dissatisfaction of some of the satisfied GC's due to the functional dependence (interaction) between them. Actually, in this case, the locality of measurement becomes dissatisfied as shown in Figure 14(b).

(e) According to Axiom 1, some substructures or attributes exist which can be used to resolve this interaction. So, we must now search the semantic network in Figure 8 for the substructures (or attributes) which restore the dissatisfied GC's. In this case, as shown in Figure 14(c), the dissatisfied GC, "the locality of measurement," is satisfied by supplementing the design with a tube enclosing the rod. This then satisfies all the GC's.

(f) From these successive explanations, we can derive the deep explanation structure of the sensing device shown in Figure 14.

These explanations show us how we can acquire strategic or meta-planning knowledge on the way how the interactions between the primary functions (GC's) occur and the way how they can be resolved; that is, an operationalized version of the design knowledge in Axioms 1 and 2 is derived.

In the above process, the simplification of the explanation structure in step (a) can be done in the following way by using the "goal-oriented" filtering and abstracting property of the EBL process and also certain minimization operations on semantic networks as shown in Figure 15.

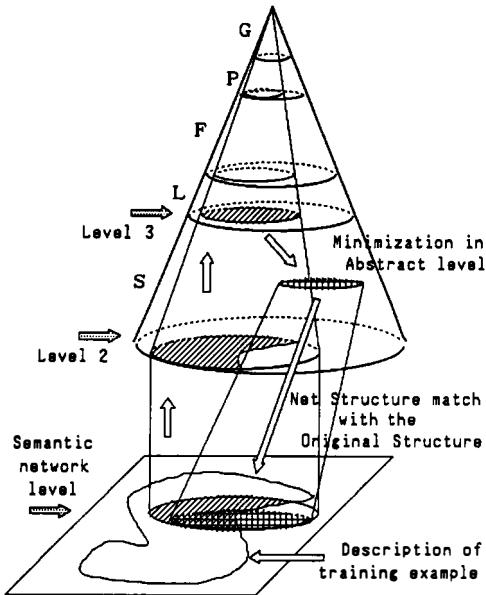


Figure 15. Process of acquiring meta-planning knowledge

be valid (effective). This results in the most general (abstract) semantic network in Figure 9(b) which is effective for measuring fluid velocity in general in the same manner as the example analyzed.

(a.2) We minimize (simplify) the general semantic network which consists of abstract structures and attributes used in the GFD by referring to simplification rules on links and attributes in abstract semantic networks. These rules are given as order relations among link predicates and attribute predicates such as shown in Figure 16. In this case, the network shown in Figure 17 is derived from the network in Figure 9(b).

(a.3) Then we translate the resultant simplified "abstract" semantic network into a concrete network which shows how the original (existing) structures and attributes are simplified as shown in Figure 18. This is done by matching the abstract network with the original concrete network via the abstract-concrete hierarchy of link predicates and attribute predicates.

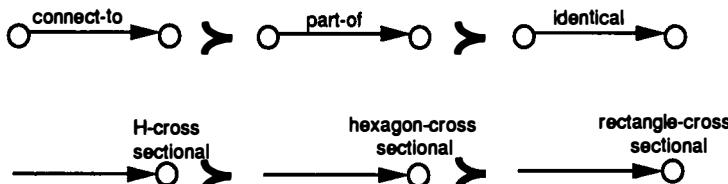


Figure 16. An example of the order relation on structural simplicity among link and attribute predicates

(a.1) We derive the GFD and extract the structural knowledge in Level 3 (cf. Figure 10(b)) which shows the most general structure and attributes attaining the GC (the primary function) "to measure fluid velocity (in general)." In this knowledge representation, as mentioned above, irrelevant features in the semantic network in Figure 2 (for measuring fluid velocity) are filtered out and only the relevant features for the measurement which are implicit in the original structure are supplemented, due to the goal-oriented filtering property of the EBG method used. Also, in this knowledge representation, all the link predicates and attribute predicates in the original semantic network are generalized as much as possible along with the abstract-concrete hierarchy among these predicates as long as the underlying physical laws are shown to

The concrete form (structure) of the resultant network is already shown in Figure 14(a).

CONCLUSION

We have introduced a hierarchical model of designed objects, which can also be regarded as a model of "well-organized" (without backtracking) design process guided by two kinds of rationalities, i.e., teleological and causal rationalities. Even though backtracking is inevitable in actual design problem solving, it is quite important to have a well-organized rational model of design and design process, by which unnecessary backtracking can be avoided. Suh's axiomatic design approach is also based on the same principle. The rationality behind their approach is strategic rationality which guides the selection of design goals and the way of attaining several goals simultaneously. We have shown that the combination of these perspectives provides an effective way for acquiring various design knowledge which can then be used to guide a more well-organized process of conceptual design.

Acknowledgments

The authors are grateful to the colleagues of KSS (Knowledge System Shell), KSA (Knowledge System Architecture) and PGF (Power Generation Facility) Working Groups of ICOT for their stimulating suggestions.

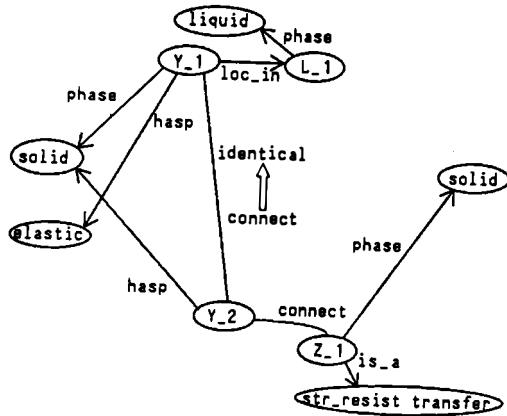


Figure 17. Minimized abstract structure description

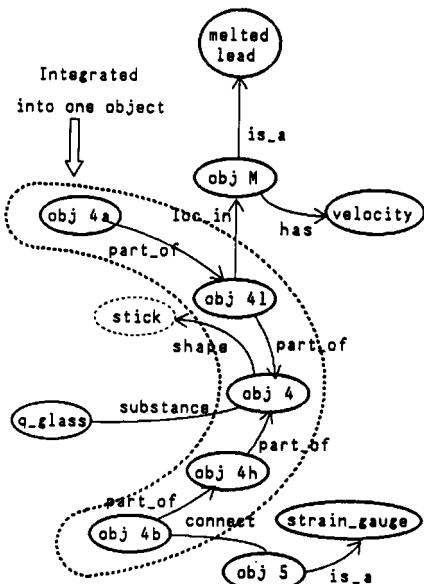


Figure 18. Minimized concrete structure obtained by matching the minimized network (Figure 17) with the original network (Figure 2)

Also, the authors are grateful to Professor Tadataka Konishi of the Department of Information Technology, Okayama University and Mr. Kazuki Shibata of the Division of Precision Mechanics, Graduate School of Kyoto University for their collaboration.

REFERENCES

- Bytheway, C. W. (1971). The Creative Aspect of FAST Diagram, *Proc. of the SAVE Conf., An advanced course of instruction for the Value Engineers*, pp.484-454.
- Hempel, C. G. (1965). *Aspects of Scientific Explanation and Other Essays in the Philosophy of Science*, The Free Press.
- Hirsh, H. (1987). Explanation-Based Generalization in a Logic-Programming Environment, *IJCAI-87*, pp.221-227.
- Hix, C. F. Jr. and Alley, R. P. (1958). *Physical Laws and Effects*, John Wiley, New York.
- Iwasaki, Y. (1985). A Measuring Device for High Temperature Fluid, *Japanese Patent, Showa 60- 10580* (in Japanese).
- The Japan Value Engineering Society (1982). *Techniques for Formalization of Functions* (in Japanese).
- JIS (The Japan Industrial Standards Society) (1990). *JIS Manual Z8103* (in Japanese).
- Katai, O., Sawaragi, T., Iwai, S. and Sue, W. (1984). An Intelligent Interface System for the Analysis and Design of Sensing Devices, *Proc. of IECON '84*, pp.503-508.
- Kawakami, H. (1987). Studies on the Structuralization of Functional Analysis and Structural Representation of Objects for Conceptual Design, *Undergraduate Thesis*, Department of Precision Mechanics, Kyoto University (in Japanese).
- Kedar-Cebelli, S. T. and McCarty, L. T. (1987). Explanation-Based Generalization as Resolution Theorem Proving, *Proc. of the 4th Intl. Workshop on Machine Learning*, pp.383-389.
- Miles, L. D. (1961). *Techniques of Value Analysis and Engineering*, McGraw-Hill Inc.
- Mitchell, T. M., Keller, R. M. and Kedar-Cebelli, S. T. (1986). Explanation-Based Generalization: A Unifying View, *Machine Learning 1*, pp.47-80.
- Rajamoney, A. and DeJong, G. (1987). The Classification, Detection and Handling of Imperfect Theory Problems, *Proc. of the IJCAI'87*, pp.205-207.
- Schank, R. C. (1972). Conceptual Dependency: A Theory of Natural Language Understanding, *Cognitive Psychology 3*, pp.552-631.
- Suh, N. P., Bell, A. C. and Gossard, D. C. (1978). On an Axiomatic Approach to Manufacturing and Manufacturing System, *Trans. ASME J. Engg. Ind.*, 100,
- Suh, N. P., Bell, A. C., Wilson, D. R. and Rinderle, J. R. (1981). Exploratory Study of Constraints of Design by Functional Requirements and Manufacturing, *Annual Progress Report to NSF, Grant No. DAR-77-13296*.
- Watanabe, D. (1984). Theoretical Researches on Value Engineering—A Design Theoretic Approach, *Proc. of the 17th Japan VE Conf.*, pp.117-124, (in Japanese).

Learning in design: an EDRC (US) perspective

Y. Reich, R. Coyne, A. Modi, D. Steier and E. Subrahmanian

Engineering Design Research Center
Carnegie Mellon University
Pittsburgh PA 15213 USA

Abstract. We identify four dimensions along which learning processes can play a role in design tasks: the generation and refinement of knowledge using machine learning techniques; knowledge acquisition, the transfer of knowledge to the machine during system development; the transfer of knowledge in a design support system back to its user; and the development of design repositories via the systematic recording, analysis, classification and re-use of design knowledge. We introduce six design systems at the EDRC at various stages of development that collectively span the learning dimensions mentioned above. We describe their individual domains, tasks, structures, performance and planned extensions. We also discuss the issues encountered in developing these systems and the solutions proposed to resolve them.

INTRODUCTION

Design has been characterized as an ill-structured problem (Simon, 1981). This characteristic is the result of three main causes: (a) design problems, when posed, are almost always under-specified; (b) design problems require the use of diverse knowledge sources from several disciplines in order to generate solutions; and (c) design problems are limited by the resources available for finding solutions. Attempts to aid and improve the process of design have proceeded by structuring design problems, as much as possible, using analytical and experimental methods. However, despite these efforts, the need to solve larger problems, changes in design objectives and the introduction of new methods and tools constantly add to the complexity of design problems. Additionally, as certain design problems (or parts thereof) become better understood through experience, the level of complexity of problems tackled tends to increase, thus sustaining the general ill-structured character of design problems.

The complexity of design problems has traditionally been addressed either by research to gain a better understanding of design domains, or by the development of better normative, experimental, mathematical and computational methods. Human learning through these

endeavors has been primarily responsible for progress in engineering and design. The challenge for computational research in design is to support learning by the use of computational mechanisms that allow for the generation, accumulation and dissemination of learned design knowledge.

In our approach we adopt a strategy to incorporate learning in design support environments where human designers interact with computational design agents in a setting that accommodates and integrates the capabilities of both. Furthermore, since the respective roles of these participants are likely to evolve and shift through time and accumulated experience, the design agents should be capable of adapting to such changes and evolving to improve their performance. In design settings that involve groups and/or multiple tools, the ability to do this is even more crucial.

Learning, the process of transferring knowledge from one source to another, is applicable in the context of both individuals and organizations. While individual designers learn from their experiences, resulting in the incorporation of new advances in their domain, organizations evolve through collective learning that leads to changes in the nature of the workplace, including the introduction of new support systems for aiding the design task.

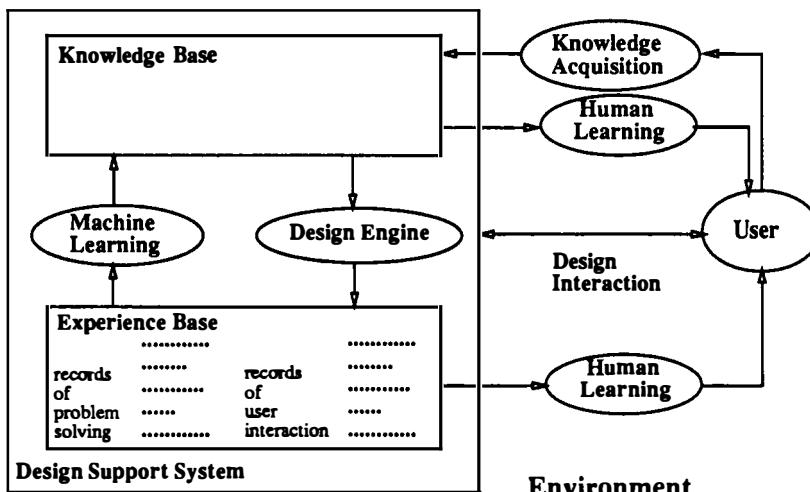


Figure 1: Learning in the context of a design-support setting

Improving design practice by supporting the learning processes involved is thus a reasonable working goal to adopt. With regard to this, we envision a design-support setting in which special emphasis is placed on learning. We believe that integrating learning in computational design support systems will result in better environments for design. Figure 1 illustrates the relationship between a user and a design support system and the various learning processes that occur within this setting. (A rectangle in the figure denotes knowledge and an ellipse denotes a process which either generates or transfers knowledge.) These processes include:

- (1) *Machine learning.* The availability of a large knowledge base and an experience

base provides an opportunity to incorporate within the support system a variety of machine learning techniques for converting experience into knowledge. These techniques can be applied either autonomously or interactively.

- (2) *Knowledge Acquisition*. This refers to the process of transferring knowledge from domain experts to the design system. Both traditional knowledge-acquisition techniques and computational tools can play a role in this process. The knowledge once encoded is then available for dissemination.
- (3) *Human learning*. This takes place along two channels: first, through facilities for browsing the knowledge and experience bases, and second, through facilities that aid interaction with the design system, e.g., devices that support exploration of the design space via experimentation.
- (4) *Organizational memory creation*. With regard to the above forms of learning, the cumulative results and advances in the field of machine learning have much to offer. However, a significant learning process that Figure 1 fails to capture adequately, and that traditional machine learning has not focused on, is the development of a organizational design memory that serves to augment and amplify the memories of the individual problem-solving agents¹.

Given our view of learning in a design setting, we elucidate some of the potential benefits of approaching learning in such an integrated manner. These include shorter system-development times, easier system maintenance, incremental system development, improved system performance (with experience), more rapid rates of improvement in user performance, and easier management and reuse of the knowledge generated in organizational design settings.

In summary, we root our view of learning in our understanding of the nature of design and the need to introduce design support systems that are capable of collaborating with human designers in performing design tasks. Based on this view, we envision a design-support environment that encompasses several forms of learning. In the next sections, we describe a number of design systems under development at the EDRC that currently incorporate learning or plan to. Figure 2 depicts these systems, which may be divided into two groups: those that function within single-tool settings and those that function within either multiple-tool or group settings. Our objectives in presenting these systems are simple: to describe their individual tasks, structures and performance, and to identify how closely they resemble our envisioned design support and learning environment. We conclude with a summary that contrasts the systems described along two dimensions: the knowledge each system learns (or will learn) through problem solving and the strategy employed in developing the system.

LEARNING IN SINGLE-TOOL SYSTEMS

CPD-Soar

The task domain of CPD-Soar is the design of distillation sequences. Given a feed mixture of known conditions, i.e., the feed specifications, CPD-Soar determines the set of all splits that should be applied to the mixture to isolate its components and the order in which

¹The general environment is also a potentially rich information source and learning directly from it is also a significant process. We believe that this form of learning is important especially in the kind of group settings that characterize large numbers of engineering design tasks.

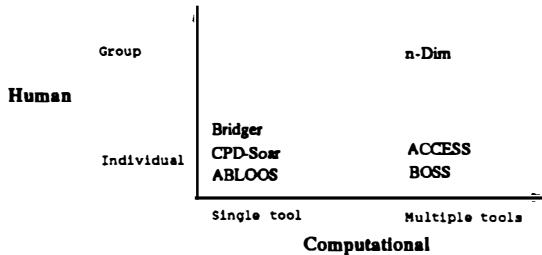


Figure 2: EDRC learning-related projects

these splits should be performed.

CPD-Soar has been implemented within Soar (Laird *et al.*, 1987), an integrated architecture for knowledge-based problem solving, learning and interaction with external environments. All tasks in Soar are formulated as search in problem spaces. Applying an operator to a current state generates a new state and a goal is achieved when a desired state is reached. The task is accomplished when the top-level goal is attained. All problem solving in Soar revolves around a number of decisions: what problem space should be searched to attain a goal, what state should the search proceed from and what operator should be applied to the state. All problem solving in Soar thus takes place in a sequence of decision cycles. Therefore, the number of decision cycles required by the system is a good metric of the effort required to accomplish a particular task. The knowledge required to make the decisions is acquired from Soar's long-term memory which has been realized as a production system. In the event that a unique decision cannot be made during the course of problem solving, e.g., more than one operator may be applicable to the current state, a subgoal is automatically created by the system. Such a situation, known as an impasse, usually arises because of incomplete or inconsistent knowledge. The purpose of the subgoal is to search for knowledge within some other problem space that will allow the system to resolve the impasse. Subgoals may occur within subgoals, thus resulting in a hierarchy. Soar learns from its experiences in resolving impasses by constructing productions, known as chunks, for insertion into its long-term memory. The chunks, which summarize the problem solving that occurred in the subgoals, are created whenever results are generated. These results form the actions of the chunks and the conditions are the pre-impasse situation upon which the results depend. The firing of these learned productions in future situations that are similar will prevent the system from subgoaling in order to make the relevant decisions, thus allowing for more effective and efficient problem solving.

As has already been implied, the CPD-Soar project views design as an act of search through multiple problem spaces. In this respect it resembles the ACCESS project. A detailed description of the structure and working of CPD-Soar is presented in (Modi and Westerberg, 1989). In terms of the larger picture presented in Figure 1, CPD-Soar can be categorized as a single-agent design system that employs a number of very simple external devices to aid it in its task. Its only interaction with a user is to obtain the problem specifications.

The system controls its search through the application of a number of heuristics commonly used in distillation-sequence design for the selection of the split to apply next. However,

since these heuristics discriminate on the basis of different attributes, ties and conflicts among competing splits can and do arise in many cases. In these situations, the system resorts to a one-step lookahead search to break the impasse. Learning thus allows the system to acquire search-control knowledge whose application prevents the system from performing lookahead searches in order to resolve impasses among competing splits. Learning therefore improves computational efficiency within and across problem-solving trials. Hence, the learning is internal to the system with no learned knowledge being transferred either to the user or any other external entity.

The development of CPD-Soar has contributed to our understanding of the design activity and how best to create systems in support of it in a number of ways. In our domain, as with research in most fields, questions that have to be addressed are often recognized only as work in the area progresses. We discuss two issues whose import was realized as a direct consequence of the development of CPD-Soar.

First, the knowledge learned by CPD-Soar is over-specific, thus preventing its transfer across tasks. The full benefits of learning will only be realized when the knowledge learned is employed in a situation not exactly the same as the one it was acquired in. In an attempt to understand how more general chunks can be learned by a system such as CPD-Soar that operates in a highly numerical domain, another system, Interval-Soar, has been developed. Interval-Soar performs simple tasks in an arithmetic domain and demonstrates how a system can abstract from numeric data points to intervals of numbers. A full description of Interval-Soar is given in (Modi and Westerberg, 1989) and a description of how the functionality of Interval-Soar can be embedded within CPD-Soar to further improve the performance of the latter can be found in (Modi, 1991). As has been indicated for the case of CPD-Soar, the problem solving required for design tasks can often be simplified by the use of an approximate and/or abstract model. However, design systems usually require the specific approximation or abstraction to be supplied *a priori* by an external agent. Furthermore, engineering design tasks are also characterized by the existence of a multiplicity of such models. These models differ along several dimensions: accuracy, precision, scope and computational overhead. A more detailed discussion of this issue in the context our system is presented in (Modi *et al.*, 1990). Design systems that generate their own simplified models or can select among given competing models are likely to perform their tasks more effectively than those that cannot.

Second, the external devices that CPD-Soar employs to aid in solving its problem are simple lisp functions that do numerical tasks such as determining the vapor flowrate through a column and computing the mole fractions of the components in a stream. However, the separations design task, like many other engineering design tasks, is characterized by the availability of a large number of powerful tools to help solve it. Providing CPD-Soar with the ability to use these tools, which range from simple numerical packages to highly sophisticated process simulators, and the ability to learn from their use, should increase its problem-solving efficiency considerably. In this regard, the ACCESS project will be instrumental in improving our understanding of how this should be done. The use of multiple agents and tools to perform a design task gives rise to the important question of how the labour should be divided among the various problem-solving and computational entities. The development of systems such as CPD-Soar that ship part of their computational effort onto other agents can help in answering this question as well as the all important implied one, "What part of the task should be left to humans?"

CPD-Soar was developed within an architecture that is comprised of a small number

of distinct mechanisms intended to provide support for a large number of the capabilities required by a problem-solving agent. These mechanisms include a single representation scheme for all knowledge (attribute-value objects), a uniform framework for conducting search (problem spaces), a single device for generating goals (impasse-driven subgoaling), a single procedure for making decisions (preference-based decision making), a uniform scheme for interacting with the external world (input and output functions) and a single learning mechanism (chunking). The use of such an architecture offered advantages during system development as well as strengths in the resulting application.

The most obvious advantage gained in using Soar to develop CPD-Soar was the avoidance of having to construct the mechanisms provided by the architecture before the main development of the system began. Problem-space search, impasse-driven subgoaling and preference-based decision making were all found to be extremely useful for task performance in CPD-Soar. Furthermore, because the use of Soar to develop the system required a good understanding of the application domain, a thorough analysis of the task was carried out. In the case of CPD-Soar, this resulted in the discovery of a set of new and powerful heuristics for the task domain. Modi (1991) describes this in depth.

Our experiences in constructing both CPD-Soar and Interval-Soar indicated that the strengths of systems developed within Soar are many. Some of these include the ability to bring all relevant knowledge to bear when doing any part of the task, the ability to both work on and learn from a wide range of tasks, the easy integration of new knowledge into memory through both manual insertion and learning and the transfer of learned knowledge across different tasks. However, although the chunking mechanism was employed in a wide variety of learning situations, lots of experimentation with the task representation was often required in order to create useful chunks. In some cases contorted searches through multiple problem spaces were required.

Bridger

Bridger is a system that aids in performing preliminary design of cable-stayed bridges. The description of cable-stayed bridges used by Bridger is very elaborate and has been drawn from existing bridges; each bridge is described by a set of 60 properties.

Bridger's architecture is based on a sequential view of design composed of problem analysis, synthesis, design analysis, redesign, and evaluation (Reich, 1991; Reich, in press). The system is intended to provide support for all except the first task. The ultimate goal of Bridger is to provide a direct mapping between specifications and design descriptions. This approach is similar in spirit to General Design Theory (Yoshikawa, 1981).

Bridger represents artifacts by a list of property-value pairs. This representation can potentially be extended to structured representations of objects where each component is described by a list of property-value pairs. This representation restricts the scope of Bridger to domains where artifacts can be described in this form. In many domains however, artifacts in the preliminary design stage can be modeled by such a representation.

Bridger contains two main subsystems: synthesis and redesign. The synthesis system is responsible for synthesizing several candidates from a given specification. It also augments its knowledge by design examples that are supplied by the user or rejected by the analysis system. Bridger uses concept formation to incrementally create a hierarchical classification

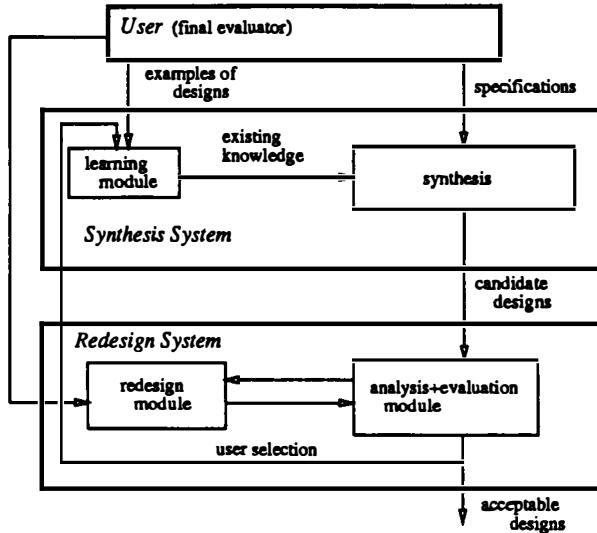


Figure 3: An overview of Bridger's architecture

knowledge structure from the examples (Reich and Fenves, 1991). Concept formation ability is believed to be fundamental to design (Reich and Fenves, 1991). The more examples Bridger incorporates, the better the quality of the knowledge generated. The classification hierarchy generated is used in synthesis of new designs. Synthesis can be viewed as a refinement process aided by models that are abstract classes of examples (Reich, 1990).

The redesign system receives candidate designs from the synthesis system and analyzes them with the use of a finite-element procedure. The analysis is restricted to linear strength and deflection calculations, but can be extended to incorporate additional concerns such as constructibility. The results are then displayed on the screen and the user can perform redesign actions until an appropriate design is obtained. An automatic redesign module is currently under development using the knowledge acquisition system Protos (Bareiss, 1989). On receiving the analysis results, this module will retrieve the best design modification for the bridge. The user can override the redesign modifications and supply explanations that enhance the redesign knowledge. The results of the redesign system are acceptable designs. From there, the user can select a partial set for training the synthesis system further.

Learning performs three functions in Bridger. First, it is used to automatically construct synthesis knowledge from bridge examples. No other means is used to derive synthesis knowledge. Second, learning is used to automatically reconstruct redesign knowledge from examples of redesign scenarios obtained from the synthesis system and additional weak domain knowledge obtained from the user. Third, the user selection of appropriate designs as a feedback on Bridger's performance can be viewed as a scheme for tuning Bridger to its user's style. In summary, learning improves both the accuracy and the efficiency of design.

Bridger is based on the hypothesis that learning systems for complex domains can be built by following a two-stage process (Reich, in press). First, the intended task of the system

is carefully analyzed and decomposed into smaller tasks. Next, the knowledge required to solve each subtask is obtained using a well defined learning technique. These well-defined learning techniques are called *generic learning tasks* (Reich and Fenves, 1989). Although, our experience with the complete Bridger architecture has thus far been limited to a single domain, its synthesis module has been tested in many domains and has demonstrated good performance (Reich, 1989). In this regard, we believe that the Bridger approach is an effective path to the use of learning in design.

The task decomposition required in order to match generic learning tasks to well-defined domain subtasks raises the important issue of how best to integrate these well-defined subsystems into a complete system. In Bridger this problem is solved by the use of a number of interfaces that transfer data between the subsystems and verify that the information transferred is consistent with the input required by the receiving subsystem. It is expected that this integration issue will be more prominent in the case of organizational design.

Bridger covers a small part of the framework presented in Figure 1. It supports the design activity of a single user, it records problem solving scenarios in the restricted form of input specification-design description pairs, and it employs user interaction to both acquire redesign knowledge and to obtain feedback on its own performance.

ABLOOS

ABLOOS is a framework for layout design that evolved as a hierarchical extension of the LOOS (Flemming *et al.*, 1989) approach to layout synthesis. It allows a layout task to be hierarchically decomposed, and for the LOOS methodology to be applied recursively to layout subtasks at appropriate levels of abstraction within the hierarchy; layout solutions for the subtasks are then recomposed to achieve an overall solution. In ABLOOS, layout proceeds as a partially automated design process that composes a given set of 2D objects into one or more layouts of rectangles. Application of the process to a layout design task produces a set of alternatives that capture trade-offs in terms of domain-specific, expert knowledge about performance criteria or constraints. With this approach, ABLOOS has produced high quality solutions on classes of industrial layout design tasks (e.g., analog power board layout involving 60 components with multiple complex constraints on their placement) (Coyne, 1991; Doreau, 1990).

The ABLOOS approach is based on the assumption that design is characterized as an exploration process in which design task formulation and the generation of intermediate solutions must occur concurrently. Thus an environment is called for where designers can incrementally and experimentally formulate design problems and efficiently generate (partial) solutions that reflect the current formulation. Such design environments must be extremely flexible to support adaptation to particular design tasks, and this can be done currently by building *computer supported design environment* (Coyne and Subrahmanian, 1990), that integrate human and computer design agents based on their respective strengths. The hypothesis is that such design environments must and can evolve to encompass new knowledge and experience generated and compiled during their use; and that the knowledge elicited by using these environments on classes of problems over time provides an empirical base for developing and testing design theories and methodologies, and for the study of learning in design.

The architecture of ABLOOS is designed around a graph-based relational representation

which facilitates the systematic generation of candidate solution layouts and a separate tester which supports evaluation of candidate solutions over a wide range of criteria. The generator and tester, combined with an appropriate controller, yield an overall architecture which functions as a form of hierarchical generate-and-test (G&T) (Coyne, 1989; Stefik *et al.*, 1983). In ABLOOS, overcoming the disadvantages of standard G&T crucially depends on three forms of abstraction built into our approach: the suppression of dimensional variations that results directly from the representation used; the suppression of detail that occurs when partial solutions are evaluated before they are expanded in all possible ways; and the suppression of complexity that results from hierarchically decomposing the design task.

The design process depends on a human/computer partnership where the computer's speed and memory capabilities are used to enumerate a space of alternative designs and to rapidly process complex evaluations; the human-designer's judgment is used to create a set of evaluations, their levels of importance, and the decomposition structure appropriate for the particular task. The task can be conveniently reformulated in terms of the latter, allowing for rounds of formulation and solving that ultimately converge to the solution (which may be a set of alternatives.) We believe that the generalized representation for layout (sub)tasks in ABLOOS will support the storing of taxonomies of evaluation and decomposition decisions in a persistent knowledge-base where they can be browsed, retrieved and reused for various purposes.

Unlike the other current design-learning systems described earlier, ABLOOS was *not* developed in order to experiment with learning techniques in design. But, as stated, it performs well on complex layout problems from industry, and it exemplifies a cooperative human/computer design environment where knowledge about design can be discovered and accumulated through the everyday solving of design problems. The environment becomes a potential repository for design knowledge at all levels, perhaps even the record of individual design sessions. With design knowledge stored in properly structured representations, it will be possible to analyze, refine, and generalize design knowledge and facilitate its exchange between human designers and computational agents in the environment. (All of these types of knowledge exchange are shown in Figure 1.)

Therefore, there are specific kinds of knowledge in the environment, or needed in the environment, that stimulate the need for human as well as machine learning. For instance, human designers doing layout tasks can access and learn from the evaluation and decomposition knowledge stored for similar problems in their domain; in the context of a design office this offers a natural process for capturing and teaching the expertise and experience that comprises the intellectual capital of the firm. Aside from the opportunities for human to human learning, and human to machine learning, the structure of ABLOOS as an interactive G&T design process suggests the applicability of a couple of specific techniques from machine learning. These may enable self learning by the machine and machine to human learning in ABLOOS:

- *Interactive induction:* ABLOOS involves interactive acquisition of evaluation knowledge from designers, knowledge that is often implicit and only accessible through recognition in context. There are several machine learning techniques that have been developed to aid this type of acquisition, enabling knowledge to be extracted more quickly and in an operationally effective form (Buntine and Stirling, 1988; Clark, 1990). We intend to examine the applicability of these and related techniques to

enhance our approach to evaluation-knowledge extraction and encoding in ABLOOS.

- **Knowledge Compilation:** This is a term for the application of learning techniques (such as variations of explanation-based learning) to optimize a problem-solver's performance by compiling its inefficient, explicit knowledge representation into more efficient implicit forms (Mostow and Bhatnagar, 1987). In a G&T process this takes the form of *test incorporation* (Dietterich and Bennett, 1986) where evaluations are advanced to preconditions on generation. Variations of test incorporation are possible(Braudaway and Tong, 1989): a test may be regressed back into the generator to achieve early pruning without affecting the correctness of the problem-solver; the generator may be modified so that it enumerates only those values which satisfy a particular problem constraint. The modular architecture of ABLOOS, where there is a clean separation of *generation* from *evaluation*, appears to be a prime candidate for the application of these techniques, subject to the complicating issues involved in adapting machine learning techniques to an existing design system, or vice versa.

LEARNING IN MULTIPLE-TOOL SYSTEMS

ACCESS

The ACCESS project concentrates on the coordination of multiple computational design tools within an integrated environment (Steier, 1990). We assume that an integrated environment is available to allow a centralized controller to invoke a diverse (in terms of languages, interfaces, and physical location) set of tools. ACCESS addresses the issue of what mechanisms should be inside that controller so that it can *learn* to select the tools that make progress towards high-quality designs while consuming the least possible computational resources. The hypothesis we are testing is that the acquisition of this knowledge will be facilitated by providing a design system with a set of general problem solving and learning mechanisms, such as those contained within the Soar architecture. Because we assume an interactive environment, the controller should be able to acquire this knowledge both from the advice of human designers and its own design experience.

The intended organization of the system we are building is shown in Figure 4. We use as an example environment FORS (Papanikolopoulos, 1989), which allows the user to visually specify the computational path, i.e., the tool sequence needed to transform the specifications into a design. Given a path, FORS uses a set of primitives included in the Distributed Problem Solving Kernel to invoke and synchronize distributed processes. ACCESS will be the knowledge allowing Soar to monitor the user's interactions with FORS along with the results of the tools invoked, and eventually to use the tools accessible through FORS on its own. Like CPD-Soar, ACCESS casts design tasks as search in multiple problem spaces. ACCESS proposes and selects operators based on abstracted versions of the available design data, and the operators selected are implemented by the tools to make progress through problem spaces. As work on ACCESS has only recently begun, the current implementation of ACCESS operates only on an abstracted (simulated) version of a set of seven building design tools from the IBDE project (Fenves *et al.*, 1990).

We would like ACCESS to use all relevant available knowledge while solving design problems. One source of knowledge is experimentation. Exhaustive forward search (applying

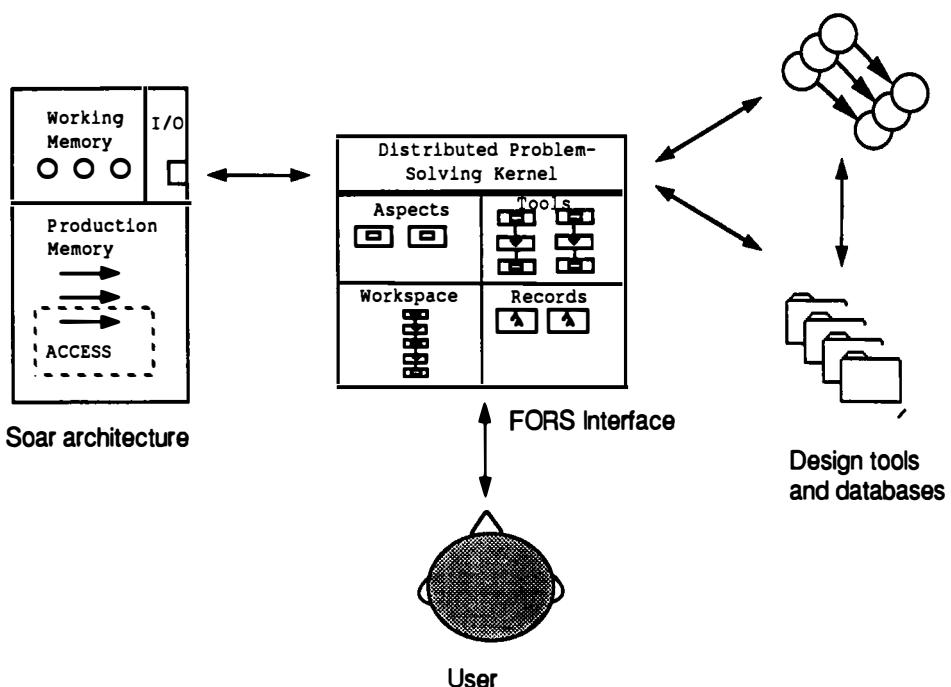


Figure 4: The intended use of ACCESS.

tools whenever possible until a desired aspect is produced) works in smaller design spaces, while in larger spaces, more search control may be required. This can be provided in the form of means-ends analysis, where the current state is compared to the desired state, and the operator that reduces the most critical difference between the two is selected. Both exhaustive search and means-ends analysis have been implemented in ACCESS by adding the appropriate method increments about the design tools to the universal weak method implicitly embodied in Soar's problem-solving mechanisms (Laird, 1984). Since another source of knowledge is the user, ACCESS includes a user-guided error recovery mechanism (Laird, 1988). This permits the user to direct ACCESS' attention to important features of the situation that caused an error, so that the system can learn to avoid that error in the future.

In terms of the more general picture in Figure 1, ACCESS studies the records of problem solving and user interaction to add tool selection knowledge to the design knowledge base. This knowledge reduces the problem solving effort, measured in terms of Soar decision cycles, significantly. For example, the number of decision cycles needed to find a path to a solution for the seven building design tools during exhaustive search is reduced from 556 without learning to 141 with it, which translates into a reduction in elapsed time for the tool selection on a DEC 3100 from 369 seconds down to 100 seconds.

Our long-term objective is to have ACCESS function as an intelligent controller for a design environment working in tandem with a user for complex engineering domains. By

giving the controller the ability to learn from experience, we expect it to assist designers in learning the long but routine tool selection sequences that characterize current design practice. Work is currently underway to have ACCESS work with actual design tools and to learn responses to violations of constraints detected during iterations of evaluation and redesign.

BOSS

The BOSS program (Bridger On Synthesis Specialists) is based on the premise that complex designs require a group of specialists. To investigate what is required to manage these specialists, BOSS was created. BOSS manages a group of specialists operating within a single domain. The specialists differ in their background experiences. Each specialist in the group knows best how to solve problems within its own experience, and to a lesser extent, how to solve problems outside its expertise. BOSS learns to allocate problems to its specialists based on their individual experiences.

BOSS' overall architecture, which embeds the same mechanisms employed within Bridger, is presented in Figure 5. The system is composed of a manager and a collection of specialists. On receiving a new specification, the manager selects the best specialist for performing this design. After performing the design, the specialists returns the solution to the managing system to be evaluated. If the solution is accepted, the design terminates, otherwise the manager selects another specialist to perform the task. The task characteristics and the specialists' performance are used as training examples to enhance the manager's knowledge about its specialists. Gradually, BOSS identifies which class of problems each of the specialists can best solve.

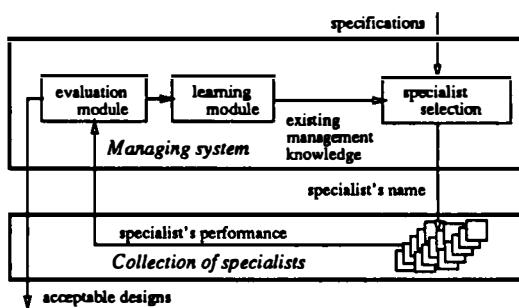


Figure 5: An overview of BOSS' architecture

In BOSS, designs are described by property-value pairs. In the current version, both the manager and the specialist systems employ the same list of properties to describe the task. However, since the roles played by the two kinds of systems in solving the task are fundamentally different, the manager should be capable representing the task using a different set of properties than the specialists.

BOSS allows the exploration of management styles and an analysis of their influence on the overall design performance. The overall performance of BOSS is governed by several

parameters. Some of these include the number of specialists, the level of expertise of the specialist, the performance accuracy which the manager is willing to accept, and the ability of the manager to explore tradeoffs between available resources and their expenditure. In the present version, the manager determines whether to accept a solution generated by a specialist by determining if the accuracy of the solution satisfies a certain threshold. Incorporating the cost of generating a design into the evaluation criteria for accepting or rejecting a design is planned as an extension to the manager's abilities.

BOSS expands the scope of Bridger to include learning 'how to manage' knowledge. At the moment though, this type of learning is restricted to problems that can be described by the same lists of property-value pairs at both the specialist and management levels. BOSS addresses the same issues as Bridger in relation to Figure 1.

n-Dim

n-Dim is a proposed design environment to support groups performing design (Subrahmanian *et al.*, 1991). Single designer/machine integration is a fundamental building block for designing large, complex human/computational environments for tasks involving multiple designers(teams), many domains and disciplines, and an array of tools. Current design practice and mechanisms often fail to adequately capture information generated during design and do not provide for making this information easily available for future designs. During the design of an artifact within an organization, designers continually negotiate and build models of the organization, the project, the functional requirements of the artifact, and of design systems in order to simulate the performance of the artifact. n-Dim is based on the hypothesis that intelligent, selective use of the computational medium for modeling in design provides an opportunity to support construction, organization, negotiation and sharing of designer's models to aid both present and future design.

The organization of n-Dim is such that it allows the users to build models of objects that correspond to particular views of the design problem such as the project schedule or functional decomposition. For each of these views it provides a language for constructing models. The languages are based on a basic node-link hypertext model where links for aggregation (Part-of, Element-of), generalization/specialization (Is-a), instancing (Instance-of), and association (Flows-to, Connected-to) are provided. The underlying principle in the design of these languages is that for a given view, such as project modeling, the language should provide for the incremental generation of models by allowing the user to develop and manage the model-structures at different levels of abstraction. The explicit purpose of n-Dim is to facilitate the collaboration and co-ordination of groups within a design task. This objective requires facilities for easy exchange of information which in turn entails the need for providing explicit structures for describing the models used in the task and their progression over time. Besides being a means of communication among designers, these models allow for the systematic recording of design history. The basic n-Dim system, besides having information models of design, will also support the invocation of computational tools that can act on these models to perform a basic design task. More details can be found in (Subrahmanian *et al.*, 1990).

The n-Dim system is currently being implemented and the role of learning in the system is essentially speculative. But we discuss it here because learning in n-Dim hopes to address questions that are different from the questions raised by machine learning approaches that

are intended to be incorporated within the design environment (Figure 1). Machine learning approaches are directed towards improved performance and identification of new knowledge in single designer contexts. All learning methods require data from which the new information can be generated to either improve performance or to identify new concepts and relationships. The data for learning in n-Dim is encoded as collections of models and their progression over time. Within this progression, the design decisions that are made can be captured in terms of trade-offs and their. A computational design environment that supports collaborative design and records design history will provide a unique database for understanding and developing empirical model of design. This understanding will help in modifying design environments to suit the needs of designers. This approach constitutes an important type of learning in design made possible by the use of computational media, but distinct from machine learning.

Some of the aspects of design that can be learned from such a potentially rich database on design are:

- Patterns of communication can be studied in the context of a particular design project. The project specific relational data can be used to identify authority structures, functional roles and communication structures in group settings. Further, the relationships between task structures and group structure can be studied for suitability (Mantei, 1981; Scacchi, 1984).
- The database can facilitate longitudinal studies of multiple design projects. The longitudinal data could be bolstered with additional observational and questionnaire-based data to achieve methodological triangulation in design research.
- Longitudinal data would also permit the study of emergent, spontaneous forms of behavior and patterns of interaction among designers. The evolution over time and their initial task related patterns as they impact on design effectiveness should lead to a better understanding of the structure and behavior of the design groups (Bendifallah and Scacchi, 1989).

These studies could potentially use methods of machine learning for the analysis of patterns of behavior. Additionally, the environment can be made more effective by improved intelligent control over tools by machine learning approaches such those described in ACCESS.

Approaches such as those of Bridger and ABLOOS that provide for building taxonomies of existing designs and problem classes would aid in building appropriate knowledge repositories in particular domains within the design environment. Since individual design systems can be considered to be building blocks of group design environment, machine learning techniques that improve the performance of the former can also be expected to contribute to the overall effectiveness of the latter.

DISCUSSION

The systems described in the previous sections provide a broad exposure to our subject of inquiry: the use of learning in design. The presentation of each project has concentrated on the organization, and observed or potential benefits, of learning. The different origins and emphases of the projects raise two different sets of issues: (a) what kinds of knowledge needs to be learned; (b) how can learning be integrated into design systems.

With regard to what kinds of knowledge need to be learned, several were identified.

Learning synthesis knowledge from examples of existing/generated designs.

Many domains include large bodies of historical data of existing designs that were generated using domain knowledge. Furthermore, the performance of these designs has been observed over time. Substantial and inexpensive domain knowledge can be accumulated by reformulating design knowledge implicit in the body of existing designs into explicit synthesis knowledge. This can help in alleviating the knowledge acquisition and maintenance bottlenecks encountered in developing design system. Since current techniques for learning from examples usually exploit only syntactic similarity, they will first have to be modified to employ domain knowledge as well. In Bridger, this issue has been explored in the context of acquiring synthesis knowledge from existing and newly generated examples of cable-stayed bridges.

Learning to choose appropriate models for the task.

Most engineering artifacts are characterized by the existence of multiple models which differ both in level of approximation and abstraction. Given that the selection of a model can have a significant impact on the computational effort required in solving a problem, choosing the most appropriate model becomes an important decision. Hence, learning the conditions under which a model should be selected can thus play a significant role in improving problem solving efficiency. In CPD-SOAR, this issue has been explored in the context of mathematical models which are a pervasive characteristic of most engineering design tasks.

Learning problem decompositions.

Complex design problems can be decomposed in a variety of ways to yield different solutions. The choice of a specific decomposition over another influences the efficiency of the process and the quality of the solution. In complex problems, decompositions can rarely be pre-specified. Learning can be used to gradually acquire decomposition knowledge. In ABLOOS, learning of layout design-task decompositions is proposed as a mechanism for acquiring knowledge from users and supporting knowledge transfer between users.

Learning to select appropriately tools in a multi-tool design environment.

Realistic design support systems contain collections of tools for performing various aspects of design. In realistic complex environments where hand-wired control is ineffective and hard, learning is the only mechanism for gradually accumulating control knowledge. The major issue that needs to be addressed is the building of abstract models of tools. These models are then used to assess which tool should be selected next for application within a specific design scenario. ACCESS addresses this issue in the context of the IBDE system.

In the world of design tools that evolve through learning, it is imperative that one be able to model what each tool can best do without having to analyze the tool's experience. This model would include the expected performance and cost of using the tool. BOSS creates such models for different instantiations of Bridger.

Learning organizational knowledge.

The role of learning in a human group setting is of a different nature from the role of learning in a purely computational environment, whether it be a single tool or multiple tools. The essential type of learning that will take place in a group setting is about the environment itself. It has been argued that the most essential capital of a firm is its design process knowledge (Capaldi, 1990). If this is so, learning at the level of improving the design process requires the ability to reuse the knowledge of previous design experiences in both a technical and

an organizational sense. To achieve this objective, repositories of experiential knowledge of designs that contain the process by which the designs were generated will have to be made available. The data collected on the design processes in these repositories can be used for analyzing and modifying the design process. In this context all of the machine learning methods that have been developed and described in the context of single designers can be incorporated for maintenance and generation of new knowledge. n-Dim is an attempt to provide a shared computational environment for supporting the building of organizational knowledge.

With regard to how learning can be integrated into design systems, the following strategies were identified.

Build non-learning design systems first and then embed learning mechanisms within them. There is a distinct trade-off involved between the flexibility of a learning method and its tractability. Will it be possible to modify and extend specific learning methods to work in conjunction with the representations and processes essential to an existing design system? Alternatively, can the representations and mechanisms of the design system be recast (without compromising its scope or performance) to allow existing machine learning techniques to be directly applied. Our guess is that some adjustment will be required from both directions to span the gap, but we believe it will be well worth the effort involved. ABLOOS and ACCESS (in its integration of existing tools) are examples of existing design systems within which learning techniques are to be embedded.

Build design systems, using an integrated architecture, that have embedded within them a limited number of simple learning mechanisms.

The use of an integrated architecture that embodies a fixed set of mechanisms, including one for learning, can have a significant impact both in the development of a design system as well as its abilities. The use of such an architecture must typically be preceded by a careful analysis of the task. In some instances, as was indeed the case with CPD-Soar, this can result in useful domain knowledge being acquired about the task. Furthermore, it is conjectured, if the set of mechanisms provided by the architecture is correct, developing a design system within that architecture will provide a valuable step towards understanding design and perhaps even formulating theories of design. CPD-Soar and ACCESS are design systems developed using an integrated architecture.

Build design systems that employ multiple machine learning techniques.

In complex design domains, it might be necessary to use several learning mechanisms to support design knowledge acquisition. In general, the ability to systematically match learning techniques to design tasks requires a good understanding of both. In turn, this requires a thorough understanding of the knowledge representations and problem-solving strategies required by the task and the learning techniques that support them. The generation of a taxonomy of representation/problem-solving/machine learning techniques could aid in facilitating this matching. Bridger, BOSS, and n-Dim are systems that integrate or will integrate a number of distinct machine learning mechanisms.

CONCLUSIONS

We have discussed the essential roles of learning in design, and described research projects that begin to support these roles. Table 1 is a summary of these roles and the

Table 1: A summary of research projects

	CPD-Soar	Bridger	ABLOOS	ACCESS	BOSS	n-Dim
<i>What kinds of knowledge are learned?</i>						
learning from existing/generated designs	•	•	✗	•	•	✗
learning to choose models	•	•				✗
learning problem decomposition			✗			✗
learning tool selection				•	•	✗
building organizational knowledge						✗
<i>How is integration achieved?</i>						
applying learning to existing systems			✗		•	
using an integrated architecture	•			•		
integration of several learning techniques		•			•	✗
Key:						
• - an implemented primary concern of a system						
✗ - a non-implemented primary concern of a system						
○ - an implemented secondary concern of a system						
✗ - a non-implemented secondary concern of a system						

projects discussed in this paper. Our experience raised several issues in relation to what kinds of knowledge need to be learned for supporting this role, and described several means of supporting these learning activities.

ACKNOWLEDGEMENTS

This research has been supported in part by the Engineering Design Research Center, a National Science Foundation Engineering Research Center, and a Sun Company Grant for Engineering Design Research.

REFERENCES

- Bareiss, R. (1989). *Exemplar-Based Knowledge Acquisition*. Academic Press, Boston, MA.
- Bendifallah, S. and Scacchi, W. (1989). Work structures and shifts: an empirical analysis of software specification teamwork. In *Proceedings of the Eleventh International Conference on Software Engineering*, pages 260–270, Pittsburgh, PA.
- Braudaway, W. and Tong, C. (1989). Automated synthesis of constrained generators. In *Proceedings of the International Joint Conference on Artificial Intelligence*, page , Detroit, MI.
- Buntine, W. and Stirling, D. (1988). *Interactive Induction*. Technical Report TIRM-88-030, The Turing Institute, Glasgow.
- Capaldi, B. (1990). Designing the future. EDRC Design Lecture, September, 1990.
- Clark, P. (1990). *Machine Learning: techniques and recent developments*. Research Memoranda TIRM-90-041, The Turing Institute, Glasgow.
- Coyne, R. F. (1989). *Planning in Design Synthesis: Abstraction-Based LOOS (ABLOOS)*. PhD Proposal, Engineering Design Research Center, Carnegie-Mellon University.
- Coyne, R. F. (1991). *ABLOOS: a computational design framework for layout synthesis*. PhD thesis, Department of Architecture, Carnegie Mellon University, Pittsburgh, PA.

- Coyne, R. F. and Subrahmanian, E. (1990). Computer supported creative design: a pragmatic approach. In Gero, J. and Maher, M., editors, *Modeling Creativity and Knowledge-Based Creative Design*, Lawrence Erlbaum Associates, New York. (forthcoming).
- Dietterich, T. and Bennett, J. (1986). The test incorporation theory of problem solving (preliminary report). In *Proceedings of the Workshop on Knowledge Compilation (AAAI)*, Oregon State University.
- Doreau, M. (1990). Personal communication. Digital Equipment Corporation, Chelmsford, MA.
- Fenves, S. J., Hendrickson, C., Maher, M. L., Flemming, U., and Schmitt, G. (1990). An integrated software environment for building design and construction. *Computer-Aided Design*, 22(1), 27–36.
- Flemming, U., Coyne, R. F., Glavin, T., H., H., and Rychener, M. (1989). *A generative expert system for the design of building layouts (final report)*. Technical Report EDRC 48-15-89, Engineering Design Research Center, Carnegie-Mellon University.
- Laird, J. (1988). Recovery from incorrect knowledge in Soar. In *Proceedings of the National Conference on Artificial Intelligence*, pages 618–623.
- Laird, J. E. (1984). *Universal Subgoaling*. PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA.
- Laird, J. E., Newell, A., and Rosenbloom, P. S. (1987). SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33(1), 1–64.
- Mantei, M. (1981). The effect of programming team structures on programming tasks. *Communications of the ACM*, 24(3), 106–113.
- Modi, A. K. (1991). *On the Development of an Engineering Design System within an Integrated Problem-Solving and Learning Architecture*. PhD thesis, Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, PA. (forthcoming).
- Modi, A. K. and Westerberg, A. W. (1989). Integrating learning and problem solving within a chemical process designer. Presented at the Annual Meeting of the American Institute of Chemical Engineers, November 1989, San Francisco, CA.
- Modi, A. K., Steier, D. M., and Westerberg, A. W. (1990). Learning to use approximations and abstractions in the design of chemical processes. In the Working Notes of the AGAA-90 Workshop, AAAI-90 Conference, July 1990, Boston, MA.
- Mostow, J. and Bhatnagar, N. (1987). Failsafe – A floor planner that uses EBG to learn from its failures. In *Proceedings of the International Joint Conference on Artificial Intelligence*, page , Milan, Italy.
- Papanikolopoulos, N. (1989). *FORS: Flexible Organizations*. Master's thesis, Carnegie Mellon.
- Reich, Y. (1989). *Combining Nominal and Continuous Properties in an Incremental Learning System for Design*. Technical Report EDRC-12-33-89, Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, PA.

- Reich, Y. (1990). Converging to "Ideal" design knowledge by learning. In Fitzhorn, P. A., editor, *Proceedings of The First International Workshop on Formal Methods in Engineering Design*, pages 330–349, Colorado Springs, Colorado.
- Reich, Y. (1991). *Building and Improving Design Systems: A Machine Learning Approach*. PhD thesis, Department of Civil Engineering, Carnegie Mellon University, Pittsburgh, PA. (forthcoming).
- Reich, Y. (in press). Design knowledge acquisition: Task analysis and a partial implementation. *Knowledge Acquisition*. An earlier version of this paper appears in the Proceedings of The Fifth Banff Knowledge Acquisition for Knowledge-Based Systems Workshop, 1990.
- Reich, Y. and Fenves, S. J. (1989). *Integration of Generic Learning Tasks*. Technical Report EDRC 12-28-89, Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, PA.
- Reich, Y. and Fenves, S. J. (1991). The formation and use of abstract concepts in design. In Fisher, D. and Pazzani, M., editors, *Computational Approaches to Concept Formation*, Morgan Kaufmann, Los Altos, CA. (in press).
- Scacchi, W. (1984). Managing software engineering projects: a social analysis. *IEEE Transactions on Software Engineering*, SE-10(1), 45–59.
- Simon, H. A. (1981). *The Sciences of The Artificial*. MIT Press, Cambridge, MA, 2 edition.
- Stefik, M., Aikins, J., Balzer, R., Benoit, J., Birnbaum, L., Hayes-Roth, F., and Sacerdoti, E. (1983). Basic concepts for building expert systems. In Hayes-Roth, F., Waterman, D. A., and Lenat, D. B., editors, *Building Expert Systems*, pages 59–86, Addison-Wesley Publishing Company, Inc., London.
- Steier, D. M. (1990). Intelligent architectures for integration. In *Proceedings of the IEEE Conference on Systems Engineering*, page , IEEE Press, Pittsburgh, PA.
- Subrahmanian, E., Davis, J. G., and Westerberg, A. W. (1990). *A Preliminary Description of a Shared Computational Environment in Design*. Technical Report, Engineering Design Research Center, Carnegie-Mellon University, Pittsburgh, PA.
- Subrahmanian, E., Westerberg, A., and Podnar, G. (1991). Towards shared computational environments for engineering design. In Sriram, D., editor, *Lecture Notes on Collaborative Product Development*, Springer-Verlag Heidelberg. (in press).
- Yoshikawa, H. (1981). General Design Theory and a CAD system. In Sata, T. and Warman, E., editors, *Man-Machine Communication In CAD/CAM, Proceedings of the IFIP WG5.2-5.3 Working Conference*, pages 35–57, North-Holland, Amsterdam.

An experimental evaluation of some design knowledge compilation mechanisms

D. C. Brown and M. B. Spillane

Artificial Intelligence Research Group
Department of Computer Science
Worcester Polytechnic Institute
Worcester MA 01609 USA

Abstract. This paper describes an experimental evaluation of the use of some simple learning mechanisms in design expert systems. These Knowledge Compilation methods act to improve the knowledge in the system so as to improve its efficiency and effectiveness.

INTRODUCTION

Despite their successes, expert systems still have limitations to overcome. One limitation being addressed is the brittleness displayed by some expert systems at the edge of the domain knowledge. To compensate for this, research is currently striving to add learning capabilities to expert systems. A learning component will enable the systems to extend their problem solving capabilities beyond the original bounds of the knowledge base (Michalski 1986).

Since expert systems have been developed to emulate a human expert, it seems natural to look at humans as a model for learning techniques. One learning technique which has been associated with the process that expert designers use when designing is *Knowledge Compilation* (Brown & Sloan 1987).

This research concentrates on routine design expert systems. We hypothesize that a task becomes routine for expert designers through Knowledge Compilation. This is the process by which knowledge is refined, in order to improve problem-solving efficiency (Brown 1989a).

We are studying the changes to one particular type of design knowledge, *design constraints*. A design constraint is a test of an attribute value or relationship. Design

constraints are transformed by three methods: *Constraint Migration*, *Constraint Inheritance* and *Constraint Absorption & Relaxation* (Brown & Breau 1986) (Brown & Sloan 1987). Each of these methods has been successfully implemented in individual design expert systems, (Sloan & Brown 1988), (Horner & Brown 1990) and (Meehan & Brown 1990), respectively. These methods are founded on plausible human mental activities (Brown 1989a).

Until now, the effects of these mechanisms on the performance of a design expert system had not been proven empirically. The goal of this research was to examine the effects of knowledge compilation on design expert system performance (Spillane 1990).

Of the three knowledge compilation mechanisms listed above only two have been included in this work. The original intent was to combine the three mechanisms into one design system. However time limitations prevented the realization of this goal. The two that were combined into one system were Constraint Inheritance and Constraint Absorption & Relaxation. Constraint Migration will be added in the future.

ROUTINE DESIGN ACTIVITY

In Routine Design Problem Solving the knowledge and the problem solving process needed are known from experience (Brown & Chandrasekaran 1989). A designer gradually learns how to solve the problem efficiently as he or she repeatedly produces a particular design object. The knowledge evolves from theoretical principles of design to an efficient, more concise form which allows the designer to solve these problems in a routine manner.

This "how to design" knowledge is used in our expert system, expressed in the Design Specialists and Plans Language (DSPL) (Brown & Chandrasekaran 1989). DSPL reflects the hierarchical structure of routine design knowledge. The human designer's "how to" knowledge is in the form of alternative design plans. DSPL is a domain independent, task-level language; meaning it is best suited for design problems at the routine design level.

DSPL is made up of cooperating agents: *Specialists*, *Plans*, *Tasks*, *Steps*, *Constraints*, *Redesigners*, *Failure-Handlers*, *Sponsors* and *Selectors*. The structure of DSPL reflects the designers decomposition of the design problem through its hierarchy of Specialists. Each Specialist represents one of the designer's subproblems. It contains a set of Plans, any one of which might be used to solve the subproblem. Each plan is a sequence of actions that orders the design activity.

This knowledge compilation study has focused on how constraints might be generated and how they might change over time (Brown & Sloan 1987). The domain chosen to demonstrate the prototype system is a ball point pen (BPEN). BPEN was selected because it is not too complex, yet it is not simple enough to be obvious. The dependencies between attributes allow for ample constraints.

The BPEN has five parts: the head, refill, spring, body, and pushing-component. There are 52 attributes to be given values, under the control of 33 constraints. Constraints, for example, limit the length of the refill relative to the body, and check that the slot cut in the body is not too wide relative to the diameter of the body. Requirements determine such things as the size of the body and the maximum allowable diameter for a pen.

CHARACTERISTICS OF THE KNOWLEDGE

Knowledge exists in many forms. In routine design, the knowledge used is in a very concise and efficient form, such as plans. This knowledge, which has been compiled through experience, is considered *surface* knowledge due to its highly refined form and ease of use. The designer's experience allows her (or him) to problem-solve in an "off the top of the head" manner.

The converse of this surface knowledge is *deep* knowledge. Deep knowledge may contain first principles, basic relationships, and knowledge about function. A designer would use deep knowledge as he or she "figured out" the problem solution. Deep and surface knowledge are relative terms, in that some knowledge may not be very deep, i.e., not quite first principles, yet is deep relative to some other piece of knowledge which is more readily usable.

DSPL has a Generic Object Knowledge Base (GOKB) which contains deep knowledge about objects (Horner & Brown 1990). Automatically building surface knowledge from deep knowledge can be accomplished through Knowledge Compilation.

KNOWLEDGE COMPIILATION

Although DSPL has been developed to easily express routine design knowledge, knowledge acquisition may not be able to capture all of the knowledge used in design. Also, over a period of time the design requirements may drift towards the outer bounds of the original knowledge. It is therefore highly desirable to create a self-adjusting, or adaptive, knowledge base which can learn from its own experience. As a means to this end, research is striving to add knowledge compilation to design expert systems (Brown 1989b).

Compilation has been used to describe processes that reduce the amount of knowledge used in the problem solving process (Anderson 1986) (Mostow 1989), and to describe the addition of relevant pieces of knowledge which alter the problem solving process (Keller et al 1989) (Pazzani 1986). The common thread amongst the uses of the term "Knowledge Compilation" is the ability of the compilation process to refine the problem-solving to enhance its performance.

Knowledge compilation often involves a change in the representation level of the source knowledge. A piece of knowledge changes from a deeper context-independent level to a more easily used form, in our case a design constraint (Horner & Brown 1990).

Some systems generate rules or failure suggestions from deeper knowledge (Bratko et al 1989), (Keller et al 1989), (Kwauk & Brown 1988). Other compilation processes refine the surface knowledge to make it more efficient (Meehan & Brown 1990), (Sloan & Brown 1988), (Tong & Franklin 1989).

Three types of adjustments to a design constraints have been proposed: *Constraint Inheritance*, *Constraint Migration*, and *Constraint Absorption & Relaxation* (Brown & Sloan 1987). Constraint inheritance identifies and creates missing constraints. It represents the designer using deeper knowledge to learn or remember a missing piece of design knowledge. The piece of design knowledge will be converted into a new constraint. Constraint migration relocates existing constraints within the design knowledge, resulting in a more efficient use of the constraint. As the knowledge base is refined, some of the constraints may become redundant. These constraints are subsequently "absorbed" into the design knowledge. During the design process a constraint may fail by a small amount. If this constraint is not critical to the design, it may be "relaxed", and the design process allowed to continue.

Constraint Inheritance

Constraint inheritance is a "deep to surface" form of compilation. It augments the existing surface knowledge by adding a new design constraint. The Constraint Inheritance System has successfully demonstrated the concept of constraint inheritance as a method of knowledge compilation.

Constraint inheritance in DSPL uses *Design Expectations* to identify the need for constraint knowledge (Horner & Brown 1990). It is the designer's prediction of a design attribute's value. The expectations are not looking for exact values but are more like approximations of what the design attribute should be. When an expectation is violated, the system reasons with the generic object knowledge to find a relationship involved with the expectation. Once a relationship is found, an *Inherited Constraint* is formed and inserted into the design knowledge (Horner & Brown 1990).

The Generic Object Knowledge Base (GOKB) supplies the system with general knowledge about the design object. This knowledge is not the same as the DSPL design knowledge, as it contains part decomposition and spatial relationship knowledge about the design object. The GOKB contains declarative knowledge about the design object versus the problem-solving knowledge in the DSPL knowledge base.

Constraint Absorption & Relaxation

Both Migration and Absorption are considered "surface to surface" compilation methods. Absorption and Relaxation are based on forgetting and judgementally "bending the rules". These activities occur when there has been a long run of constraint successes or failures. Based on the design history of constraint execution, the system may reason about absorbing or relaxing the constraint (Meehan & Brown 1990).

Redundant knowledge may be caused by poor knowledge acquisition, an expert with incorrect knowledge, or by another compilation mechanism such as Migration (Brown 1989b). By including a mechanism such as Absorption to eliminate the redundant constraints, the design knowledge should not become cluttered by unnecessary constraints. As a redundant constraint would rarely fail, its behavior would trigger the absorption mechanism.

Once activated the Absorption mechanism looks for a constraint earlier in the design process that may duplicate or subsume the constraint in question. Another possible cause for a consistently true constraint is that the preceding calculations may ensure its success. Such a constraint is then considered redundant and absorbable. These cases are hard to discover in general.

Another type of absorption is called *Automatic Absorption*. This can occur when no redundant knowledge is found, but the constraint continues to succeed for many design runs. Automatic absorption requires a greater number of consecutive successes than the other approach.

Constraint Relaxation responds to a constraint which consistently fails by a small amount. This may be caused by a small error in the design knowledge. When a designer encounters such a situation it may be possible to allow the attribute's value to pass, knowing it will not adversely effect the overall quality of the design object. An expert designer has the ability to identify which constraints in the design process must be unyielding and which may be relaxed. If the expert identifies a constraint as relaxable, it is called a "soft constraint", otherwise it is a "hard constraint" (Meehan & Brown 1990).

Relaxation occurs only if the constraint is soft and if the design attribute value fails by less than the relaxation limit stipulated by the expert designer. The design quality is monitored through a *Design Confidence Factor* which regulates the amount of relaxation allowed within a design run.

There are two types of relaxation. One is *Temporary Relaxation* which is effective only for the current design run. The other is *Permanent Relaxation* which is effective for all subsequent design runs. When a constraint succeeds through temporary relaxation for 80% of the previous design runs, it becomes a candidate for permanent relaxation.

MERGING THE TWO SYSTEMS

The purpose of this research is to determine the effectiveness of the compilation methods, both alone and together. Prior to this work each compilation method had been implemented separately, but not together. The new system combined two existing prototype systems, Constraint Inheritance (CI) and the Constraint Relaxation & Absorption System (CRAS). The resulting Combined Compilation Method System (CCMS) was used in this experiment (see Figure 1).

There were many inconsistencies between CI and CRAS. Each system used a different version of the basic DSPL interpreter. CCMS used the DSPL interpreter from CI as its base interpreter. The inheritance version of the interpreter used PCL (Portable

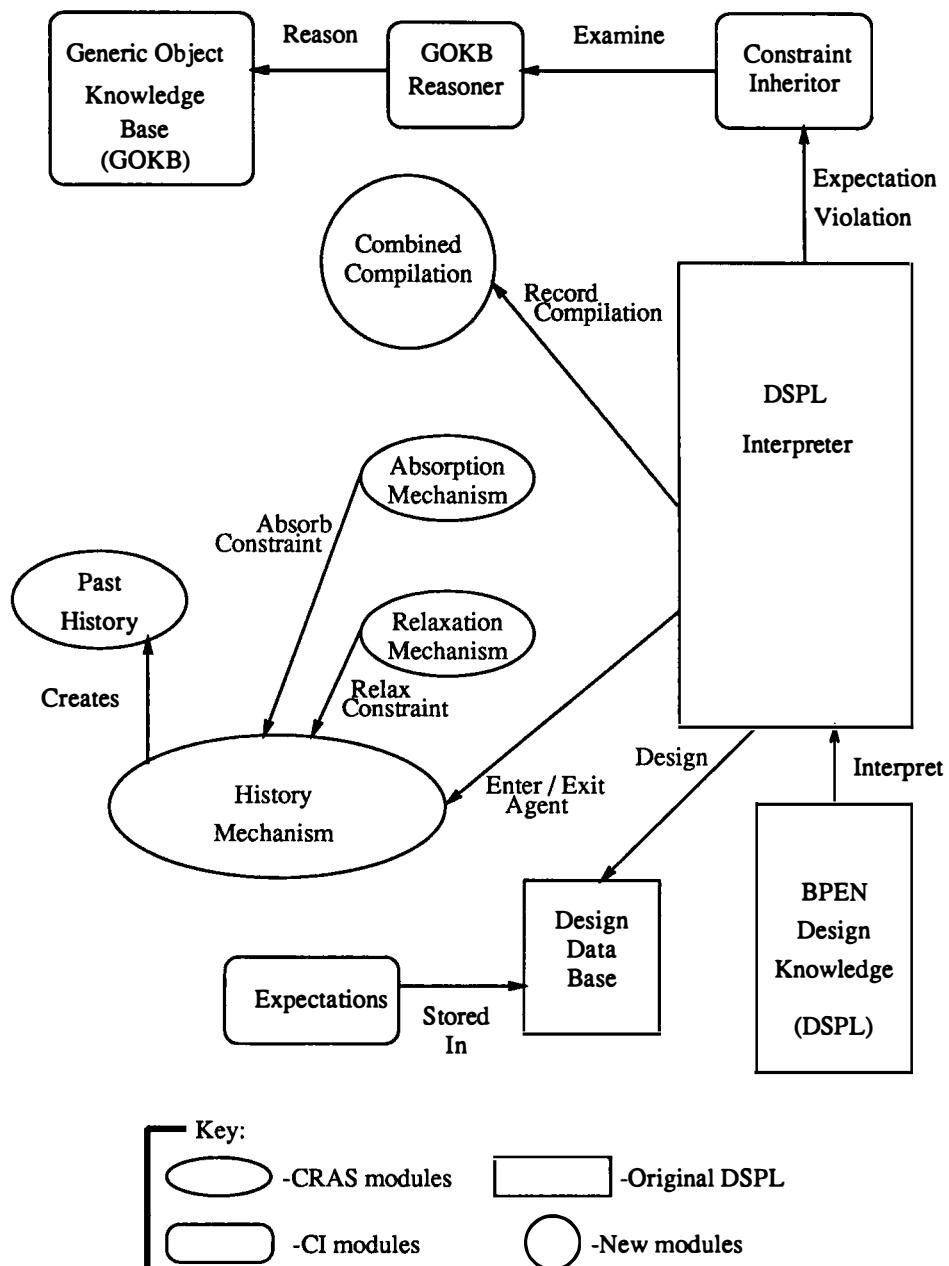


Figure 1: Combined Compilation Method System

Common Loops) to represent the Design Data Base. PCL is an object-oriented language. As PCL (or CLOS) will be used in our future research, it was chosen for CCMS. The rest of the DSPL interpreter is written in Common LISP.

In CRAS, temporary relaxation is a part of the DSPL interpreter, and is always active. As this was inconsistent with the test plan, which included design runs with no compilation, the temporary relaxation mechanism was disabled. Additional tests were run to demonstrate the effects of temporary relaxation.

EVALUATION CRITERIA

By including compilation in a design expert system, the problem solving ability should improve with experience. What constitutes an improvement? Improvement in other learning systems has been measured in a variety of ways, from recording CPU time, to counting internal system operations, or some combination. In order to develop some notion of change in a system's performance the data is normalized within its own system and/or compared to similar measurements taken before learning occurred (Markovitch & Scott 1988) (Minton 1988) (Pazzani 1986) (Prieditis & Mostow 1987) (Tambe & Newell 1988) (Tong & Franklin 1989) (Wogulis & Langley 1989).

Informative comparisons have been made between the learning system's result and that of a system which included expert coded knowledge or known solutions (Pazzani 1986) (Tong & Franklin 1989). This type of comparison was able to illuminate the performance gap between a system in which the knowledge base is completely derived through knowledge acquisition and a system which starts with less knowledge but is capable of learning. However, in some areas such as Explanation-Based Learning or Chunking, the system performance did not always improve with the inclusion of the learning mechanism (Ellman 1989) (Minton 1988) (Tambe & Newell 1988).

Improvements can be measured by the result and by the process. A more *effective* system would produce a better solution to the problem, or would be able to produce a design where none was possible before. A more *efficient* system would solve the problem in less time or with a shorter design path. Improving the quality of a solution is not a major goal of this research. Our primary focus is on the efficiency of the design expert system. There is usually no single solution to a design problem. The solution's quality depends on one's perspective or goals, as well as the purpose for which it was designed. Consequently, quality is difficult to measure.

Some researchers have measured CPU time as well as the internal activity of the system (Minton 1988). Others found that measuring the activity of a particular piece of knowledge was sufficient (Markovitch & Scott 1988) (Pazzani 1986) (Tong & Franklin 1989) (Wogulis & Langley 1989). In some research, only the CPU time was needed (Prieditis & Mostow 1987). Often it is beneficial to calculate the amount of time spent on a particular step in the system (Minton 1988) (Tambe & Newell 1988).

For our system, measuring CPU time alone will not clearly indicate the effects of compilation. The activity of the design agents must be measured, as well as the CPU

time. One measurement is the length of the design path, i.e., the total number of design agents used in the design run. Other measurements include the number of successful and unsuccessful constraints, both before and after compilation occurs.

These measures reflect changes in the system's effectiveness and efficiency, but they do not highlight changes in the quality of the solution. In DSPL the final problem solution must satisfy all of the applicable constraints as well as any of the design expectations associated with the knowledge base. These checks will determine the solution quality.

TEST PLAN

Part of our goal is to associate a compilation method with a particular type of change in system performance. This can be found by testing each mechanism separately and noting the performance changes.

The testing sequence is run three times. CCMS is run once for Inheritance, and once for Relaxation & Absorption, and once again for both Inheritance and Relaxation & Absorption together. These tests are called *the Inheritance Test Run*, *the Relaxation & Absorption Test Run*, and *the Combined Test Run*, respectively.

Phases

There are three phases to the test scenario: *Pre-Compilation Phase*, *Compilation Phase* and *Post-Compilation Phase*. During the Compilation Phase a set of design requirements are submitted to the system to cause compilation. This set of design requirements is called the *Compile Set* of design requirements. Both before and after the Compilation Phase a different set of design requirements are run. This is called the *Control Set*. No compilation occurs while the Control Set is being run, i.e., the mechanisms are not functioning during Pre- and Post-Compilation Phases. The Pre-Compilation Phase produces benchmark data. It is used as a guide to judge the effects of compilation. As these design runs are made without any of the compilation methods in use, it represents a system with a static knowledge base.

All learning by the design expert system occurs during the Compilation Phase. Changes to the design knowledge base are caused by Inheritance, by Relaxation & Absorption, or by Inheritance and Relaxation & Absorption together.

In the Post-Compilation Phase none of the compilation mechanisms are in use. The design runs are made to gather results to be compared to the Pre-Compilation results. The same design runs are made in the Pre-Compilation and Post-Compilation phases. This provides a consistent basis from which a performance evaluation can be made.

Design Requirements

Each run of a DSPL design expert system requires a set of design requirements for input.

Design requirements contain the designer's specifications for a design object. Each set of design requirements is different from the next. As the test plan calls for multiple design runs, multiple sets of design requirements are needed. Each set of design requirements is different, so that the same object is not being designed in each run. The Control Set contains ten sets of design requirements, which will potentially produce ten different design objects. The Compile Set contains five sets of design requirements, which should produce five different design objects.

In the Control Set, the design requirements were arbitrarily chosen. There was no specific intent for these requirements. However, the design requirements in the Compile Set were selected with the objective of maximizing compilation in order to make results more obvious. Under normal operating conditions, compilation occurs slowly over a long period of time. The decision to compress compilation into several design runs was made because of time and resource limitations.

RESULTS & CONCLUSIONS

The results from the tests have provided positive indications that knowledge compilation can enhance system performance. Although every design run does not display an increased performance, on the average the improvement in system efficiency and effectiveness is notable. Table 1 shows the average changes in the system's behavior.

Constraint Inheritance Results

As inherited constraints directly influence the path taken through the design agents, they may change the order in which the design agents are executed. By altering the design path, the outcome of the design process is changed.

Constraint Inheritance can prevent an unacceptable design from being produced, and may allow it to be designed correctly. This occurs because the inherited constraints can force the design process to use a different path. Another desired effect of Constraint Inheritance is finding a failure earlier in the design process, thus preventing unnecessary effort. The inherited constraints check attribute relationships earlier in the design process, and identify incorrect relationships.

A drawback of adding Constraint Inheritance to a design expert system is that the inheritance process takes time. However, while using the resultant knowledge base the execution time was increased in only 50% of the runs. Thus increases in execution time are not prohibitive. The potential gains from constraint inheritance are worth the trade-off in time.

Constraint Absorption Results

Constraint Absorption reduces the number of agents used in a design run. The design process is made shorter by eliminating redundant constraints from the design path, thus

Table 1: Changes in the System's Behavior.

Pre & Post-Compilation Phase Results

Test Run	Avg. Time Pre-Comp.	Avg. Time Post-Comp.	Change in Time	% Change in Time	Change in Design Path
Inheritance	554.7	501.2	-53.5	9.64% speed up	+3.1 agents
Relaxation & Absorption	554.7	471.7	-83.0	14.96% speed up	-2.9 agents
Combined	554.7	529.8	-24.9	4.49% speed up	-4.5 agents

Non-Compiled & Compilation Phase Results

Test Run	Avg. Time Non-Comp.	Avg. Time Compilation	Change in Time	% Change in Time	Change in Design Path
Inheritance	510.4	710.8	+200.4	39.26% slow down	+35.8 agents
Relaxation & Absorption	510.4	786.4	+276	54.07% slow down	+1.4 agents
Combined	510.4	1064.6	+554.2	108.58% slow down	+36.6 agents

reducing the execution time for the design run. Constraint Absorption does improve the efficiency of the design expert system. However, there is no increase in efficiency while the absorption mechanism is activated, because the execution time increases while absorption is operating. Regardless of whether or not absorption is running, the design path is still shorter.

Constraint Relaxation Results

Relaxation was able to complete a design which previously had been non-producible. A design run without relaxation ended in failure, after some backtracking and redesign effort. Relaxation of the failing constraint eliminated the need for backtracking and redesign, and allowed the design to be completed. This increases the space of problems which the system is able to solve.

Combined Mechanisms Results

When Constraint Inheritance and Relaxation & Absorption are activated in the same design run, the behavior of the system is similar to the sum of the individual system's behavior. The increased efficiency is not evident in every design run, but can be found in a group of many design runs. When averaged over many design runs, the positive effects of compilation are most obvious.

An interesting result from the combined mechanisms test is that compilation activity is dependent upon the experience a system encounters. That is, given the same starting system, compilation will not always occur in the same sequence. It depends heavily on which design paths are used. Because inheritance and relaxation change the design paths, different design states were entered.

Some Future Directions

The next step would be to merge the Combined Compilation Mechanism System with NOMAD, the Constraint Migration system (Sloan & Brown 1988). There are many implementation issues which need to be resolved before the systems can be joined. The most critical being the design histories. Each system has its own version and they are significantly different. Once the systems are combined, a test plan similar to the one used in this research, should be executed. It will be very interesting to see how constraint migration will change the performance of the expert system.

The other important task for knowledge compilation research is to experiment with a real domain. BPEN is small enough for the development of a prototype system, but not large enough to obtain definitive test results. The Combined Compilation Mechanism System, with or without migration, should be tested on a large domain. It is hoped that by using a real design domain that the space of designs would be large enough to support extensive testing. This type of testing would produce far more conclusive results.

Summary

This experiment has shown that knowledge compilation is a valid form of learning for design expert systems. The test runs have exhibited many of the behaviors we anticipated. The compilation mechanisms, Constraint Inheritance and Constraint Relaxation & Absorption, have improved the design knowledge and the performance of the prototype system, BPEN.

One disadvantage of compilation is that the execution time for a design run may increase during compilation. However, the improved efficiency of the design knowledge and the quality of the resulting design object outweigh the disadvantage of additional execution time.

The testing done on BPEN has produced very positive indications that our hypothesis about the performance of knowledge compilation in a design expert system is very close to correct. However, the results here are more suggestive than they are conclusive. Using a larger, real domain would be a move towards more conclusive results. However, in summary, we feel that our compilation methods have been empirically validated using the prototype system.

Acknowledgement: This work was supported in part by NSF grant DDM-8719960.

REFERENCES

- Anderson, J. R. (1986), "Knowledge Compilation: The General Learning Mechanism", *Machine Learning: An Artificial Intelligence Approach*, (Eds) R. S. Michalski et al, Vol. 2, Morgan Kaufmann, pp. 289-310.
- Bratko, I., Mozetic, J., & Lavrac, N. (1989), *KARDIO: A Study in Deep and Qualitative Knowledge for Expert Systems*, The MIT Press, Cambridge MA.
- Brown, D. C. (1989a), "Compilation: The Hidden Dimension of Design Systems", *Intelligent CAD, III*, (Eds) H. Yoshikawa & F. Arbab, North-Holland, to appear.
- Brown, D. C. (1989b), "Adjusting Constraints in Routine Design Knowledge", *Proc. of NSF Engineering Design Research Conf.*, University of Massachusetts, pp. 347-362.
- Brown, D. C. & Chandrasekaran, B. (1989) *Design Problem Solving: Knowledge Structures and Control Strategies*. Morgan Kaufmann Publishers.
- Brown, D. C. & Breau, R. (1986), "Types of Constraints in Routine Design Problem-Solving", *Proc. of 1st Int. Conf. on Applications of AI to Engineering Problems*, Southampton University, UK.
- Brown, D. C. & Sloan, W. N. (1987), "Compilation of Design Knowledge for Routine Design Expert Systems: An Initial View", *Proc. of ASME Conf. on Computers in Engineering*, New York, NY, pp. 131-136.

- Ellman, T. (1989), "Explanation-Based Learning: A Survey of Programs and Perspectives", *ACM Computing Surveys*, Vol. 21, No. 2.
- Horner, R. & Brown, D. C. (1990), "Knowledge Compilation Using Constraint Inheritance", *Proc. of the Artificial Intelligence in Engineering Conf.*, pp. 161-176.
- Keller, R., Baudin, C., Iwasaki, Y., Nayak, P., & Tanaka, K. (1989), "Compiling Special-Purpose Rules from General-Purpose Device Models", Report No. KSL 89-49, Knowledge Systems Laboratory, Department of Computer Science, Stanford University.
- Kwauk, R. & Brown, D. C. (1988), "Generating and Applying Failure Recovery Suggestions in Hierarchical Design", *Artificial Intelligence in Engineering: Diagnosis and Learning*, (Ed) J. S. Gero, Computational Mechanics Publications, Southampton, UK, pp. 29-50.
- Markovitch, S. & Scott, P. (1988), "The Role of Forgetting in Learning", *Proc. of Int. Conf. on Machine Learning*, pp. 459-465.
- Meehan, E. & Brown, D. C. (1990), "Constraint Absorption and Relaxation Using a Design History", *Proc. of ASME Design Theory and Methodology Conf.*, Chicago, IL.
- Michalski, R. S. (1986), "Understanding the Nature of Learning: Issues and Research Directions", *Machine Learning: An Artificial Intelligence Approach*, (Eds) R. S. Michalski et al, Vol. 2, Morgan Kaufmann, pp. 3-26.
- Minton, S. (1988), "Quantitative Results Concerning the Utility of Explanation-Based Learning", *Proc. of 7th National Conf. on Artificial Intelligence*, AAAI-88, Vol. 1, pp. 564-569.
- Mostow, J. (1989), "Towards Knowledge Compilation as an Approach to Computer-Aided Design", *NSF Engineering Design Research Conf.*, University of Massachusetts, Amherst, MA.
- Pazzani, M. (1986), "Refining The Knowledge Base of a Diagnostic Expert System: An Application of Failure-Driven Learning", *Proc. of 5th National Conf. on Artificial Intelligence*, AAAI-86, Vol. 2, pp. 1029-1035.
- Prieditis, A. & Mostow, J. (1987), "PROLEARN: Towards a Prolog Interpreter That Learns", *Proc. of 6th National Conf. on Artificial Intelligence*, AAAI-87, Vol. 1, pp. 494-498.
- Sloan, W. N. & Brown, D. C. (1988), "Adjusting Constraints in Routine Design Knowledge", *Seventh National Conf. on Artificial Intelligence*, Workshop on AI in Design, AAAI-88, St. Paul, Minn.
- Spillane, M. B. (1990) *Knowledge Compilation for Design Expert Systems: A Performance Evaluation* , M.S. Thesis, Computer Science Department, WPI, Worcester, MA 01609.
- Tambe, M. & Newell, A. (1988), "Some Chunks Are Expensive", *Proc. of Int. Conf. on Machine Learning*, pp. 451-458.

- Tong, C. & Franklin, P. (1989), "Tuning a Knowledge Base of Refinement Rules To Create Good Circuit Design", *Proc. of 11th Int. Joint Conf. in Artificial Intelligence*, IJCAI-89, Vol. 2, pp. 1439-1445.
- Wogulis, J. & Langley, P. (1989), "Improving Efficiency By Learning Intermediate Concepts", *Proc. of 11th Int. Joint Conf. on Artificial Intelligence*, IJCAI-89, pp. 657-662.

A data model for design databases

C. M. Eastman, A. H. Bond and S. C. Chase

Graduate School of Architecture and Urban Planning
University of California at Los Angeles
Los Angeles CA 90024-1467 USA

Abstract. This paper reviews the premises and current status of work on EDM, a data model for engineering product databases. A new data model is defined, and its application to the modeling of a portion of an intelligent CAD system demonstrated. Application of the data model to CAD database schema definition is discussed, as well as for translation between databases. This work summarizes and extends the previously published work on EDM, presented in (Eastman, Bond and Chase, 1991a) and (Eastman, Bond and Chase, 1991b).

INTRODUCTION

In its twenty years of existence, computer-aided design (CAD) has evolved through a series of "generations", each generation being the solution of a current major impediment to full utilization. As individual representation problems involving geometry and surfaces have been resolved, the more ambiguous issues of component relations and function have arisen, in parallel with issues of abstraction and support for development sequences. These newer problems are ill-defined; there is a growing sense that traditional approaches that focus on parts of a CAD system will not lead to the currently needed breakthroughs involving integration.

At the level of system architecture, CAD systems have become recognized not as end-user tools but as platforms for the development of domain-specific design applications. The capabilities developed to date have adequately supported those design tasks that can be prespecified as compositions with a fixed set of functions, based on a well-defined set of components, such as P&ID piping layout, electronic component design, and architectural interior and furniture layout. All CAD efforts today seem to involve attempting to define a design domain as a "kit-of-parts" and integrating the tools needed for this closed specification.

A number of large integrated CAD systems have been attempted that offer more comprehensive support of design, that do not impose such a closed specification. Their goal was to represent a design model at multiple levels of abstraction, supporting a variety of functions, coding of manual procedures and practices, and support for a variety of

manufacturing technologies that change over time (Miller, Southall and Wahlstrom, 1979), (Ulfsby, Meen and Oian, 1982), (Atwood, 1985), (Kemper, Wallrath and Lockemann, 1987). These efforts have proved to be very expensive and challenging to develop, and even more expensive to maintain. Such efforts especially fail where elements are not fixed but open-ended and can be defined recursively, as in architectural or mechanism design. The complexities of such systems, spanning division- or enterprise- level design environments, become overwhelming.

From the complexity that these systems display to both their designers and users, it is apparent that the underlying tools and concepts now being employed are too weak to respond to the problems at hand.

DATA MODELING

The idea of a *conceptual data model* for defining the behavior to be included in a database was proposed by Sibley, among others (Sibley, 1974). He described it as a model of the real world concepts and relations that are to be encoded into a database schema. A survey of this line of effort is provided by Peckham and Maryanski (1988). This is to be distinguished from a logical model of data, that defines the logical structure of data and the mathematical operations on that data (Codd, 1970). Effort has gone into merging these two views, developing conceptual models that are rich enough to capture the relations and issues in some application area, in a formal language that has a clear logic or mathematical basis. Several such data models have been defined for database design: NIAM (Van Griethuysen 1983), RM/T (Codd, 1979), and others. All were developed for use in general database conceptual model applications.

The motivations and needs leading to the development of data models for database theory are the same as those in integrated CAD and CAM. The need is for a general method for representing the logical structure of products to be designed within a CAD system, and in particular, to define the structure and relation of data within a product domain. Such a direction is motivated by the slow progress of current efforts. A more abstract, formally based approach appears warranted if major progress is to be achieved.

While the motivations for using data models in CAD/CAM are the same as elsewhere, we believe that CAD data incorporates logical conditions not encountered in most other database areas, and that will probably lead to different data modeling needs:

- (a) CAD data defines a conceptual object that does not yet exist. A business database typically carries data corresponding to entities in the real world, with the real world imposing integrity conditions on what values can be accepted. In contrast, CAD data is: incomplete for most of the design process and lacking integrity. Integrity is added incrementally.
- (b) Design data in many product domains involves highly diverse structures, where the structure depends upon decisions regarding the technology and functions associated with the design components. Thus CAD data is incorporated in a schema that is defined incrementally as design proceeds.

(c) In most database applications, there is a fixed level of description needed for individuals.

In design, however, the level of specification of the product is open-ended. A design can always be further specified as to new components, or standard items can themselves be designed, to the molecular level and beyond. Completeness does not determine when design is finished, and an individual can always be turned into a class and be further defined, e.g., iterative refinement.

Other differences are also involved, some of which we expect to uncover as we proceed. An engineering data model also offers an opportunity to address some as yet unresolved problems in the conceptual design of computer-based design systems. What is to be provided in a base environment for the day when design knowledge, in form of expert systems, intelligent applications, or analysis packages, come in "shrink-wrapped packages", allowing users to mix and match the diverse tools and embedded knowledge to be included in a design application? How should such modules be organized so that they effectively communicate with one another?

Consideration of the relation between creative design and computational design tools. CAD seems always based on the capability to apply pre-packaged knowledge about a design domain to a current task; what about the development of design tools that allow designers to innovate and define new knowledge within the context of one project and generalize it for future use in others?

There have been previous efforts to apply data modeling languages to CAD/CAM. These typically adopted an existing data model and attempted to apply it to design (Batory and Kim, 1985), (Shaw et al, 1989), (Cornelio and Navathe, 1989). In these cases, the goal was to illuminate through data modeling the structure of design and engineering problems. National standards groups have also embarked on engineering data modeling efforts (Smith, 1986), (Geilingh, 1988), in order to define general structures that can serve in the translation of data between different CAD models.

THE EDM APPROACH

Rather than accept as given any existing data model, we felt it was first necessary to examine afresh the structure of CAD data and the semantics involved. Pre-selection of an existing data model would bias our ability to interpret CAD data using only the semantics supported by that data model. Our intention is to define a small set of structures that capture the semantics of design and engineering information. We chose sets and first order logic as a neutral but powerful meta-language for defining these structures.

The lower level primitives are based on the premises that all design data can be represented as sets of data and relations over the sets. The relations are defined by and come from many sources: (i) definitions and identities, e.g., Ohm's Law; (ii) rules of composition, e.g., joint and connectivity conditions; (iii) issues of representation, e.g., wellformedness rules for solids modeling; (iv) design rules, e.g., spacing conditions in VLSI; and others.

All design systems make varying assumptions about which relations are embedded in data structures and which are represented explicitly in operations or tests. All translation or neutral intermediate database efforts that we are aware of attempt to define some covering

of most of these assumptions. The choices become political. We believe that there are large numbers of ways to define structures having the inherent redundancy of CAD models. We can find no way to abstractly characterize these issues without extracting *all* relations from their embeddings and examining them explicitly. Thus we treat all data as unique variables, without even equalities being embedded in structures. We use *constraints* as a general means to define the relations between data variables. Here, a constraint is an expression over a set of data that evaluates to true or false. This view is the opposite from most database efforts. There is no one canonical model nor issues of normalization. All redundancies between views are maintained and managed by constraints.

The condition in design is that constraint relations are highly complex. A major issue is to organize them in a manageable manner. Thus our effort has been to package constraints into structures in a form that makes their behavior both apparent and tractable.

With the primitives we have introduced, we then define four structure types that consist of both data objects and relations between them. These high level structure types are meant to depict both abstract conceptual design information as well as low level production based information. These structures have evolved as we have proceeded, as various semantic conditions have been integrated into the model.

A data model for product modeling has several related uses. Its structures are the basis for defining a product model in some application domain relevant to CAD. The structures are composed to define the data and knowledge in that area needed for or used in design. The product model also can be viewed as a specification for a CAD database or alternatively as a language for defining a design database schema. Last, data loaded into such a product model database, along with the entire database structure, would represent a particular design.

An alternative use is in defining a backend neutral database for translating between existing CAD models. In this case, the product model would be used to capture the semantics of one CAD database and translate it into the format needed by another. Issues of logic arise in this application because one database may not store data that is needed in another, but the needed data can be derived from the stored data. Also, it may be desirable to load data from multiple supposedly consistent applications, so as to translate into a third application. Integrity checking on such data is required, to determine that the two sets of data are logically consistent. We consider the requirements for these two uses to be similar.

EDM STRUCTURES AND RELATIONS

EDM defines a set of abstract data structures for modeling the data used in different areas of product modeling. The structures are abstract, extendible and respond to the set of criteria which were set forth in (Eastman, Bond and Chase, 1991a). Structures are either *base elements* or *forms*.

Design variables are introduced as having a defined domain which is a set of values that the design variable can have. Constraints can then restrict this set of values.

We use *forms* to represent EDM data models. Forms are arrangements of design variables and constraints. Forms are represented in the object language of EDM by ground, i.e., completely instantiated, literals in predicate logic. The different types of form will be

defined in the next subsection. Every form will have a unique name as its first argument. This can be used within another form to refer to this form.

There is no specified language in which constraints are expressed. In general, constraints will simply have a constraint name and a set of actual parameters.

Syntax of EDM structures

- (a) There are symbols, i.e., identifiers, e.g., a, b, X, Y.
- (b) There are some reserved identifiers: domain, aggregation, constraint, FE, composition, accumulation.
- (c) ‘:’ denotes an association between a variable name and its domain or aggregation, *Variable:Dname* or *Variable:Aggname*.
- (d) Curly brackets ‘{’ and ‘}’ indicate an arbitrary length set of sub-expressions {<list of Values>}, e.g., {X,Y,c}.
- (e) Square brackets '[' and ']' indicate an arbitrary ordered list of sub-expressions. In this case, we used a generic name and a subscript notation, [subscripted identifier], e.g., [*vertex:point*].

In defining the types of structure that exist in EDM, we use some additional constructions with corresponding syntactic conventions. Sets of expressions can be denoted by

{*Setexp / Setexp*} , e.g., {*s1 / s2*}

where the vertical bar denotes alternatives.

Base Elements of EDM

The primitive data element is the *domain*, a class definition comprised of a set of possible values. The value of a domain is its set of values. Domains are grouped as sets into higher level structures, called *aggregations*. A domain may be used multiple times in the definition of an aggregation, e.g., to define the multiple points used to define a polygon. Thus each use of a domain is named in an aggregation and the named domains are denoted as *variables*. Variables may refer to other aggregations to define complex variables, such as a polygon. The value of an aggregation is the Cartesian product of its constituent domains.

Domains a named type and a set of possible values:

domain(Dname, {Values}) (1)

The value set may be explicitly enumerated (for example, color names) or denoted implicitly by lower and upper bounds over some rational scale (weights).

Aggregations a named aggregation and a set of variable-domain and variable-aggregation name pairs:

aggregation(Aname1, {Variable1:Dname1 / Variable2:Aname2}) (2)

The value of a variable is its domain set or the value of its associated aggregation. The value of an aggregation is the Cartesian product of its domains. A variable denotes the particular use of the Dname in some context. Multiple variables may exist associated with the same domain. For example, an aggregation may be a line, parameterized by two points: $\text{aggregation}(\text{line}, \{\text{Pt1:2coord}, \text{Pt2:2coord}\})$, where 2coord is an aggregation: $\text{aggregation}(\text{2coord}, \{\text{xcoord:real}, \text{ycoord:real}\})$, where real is the set of rational real values.

Constraints A *constraint definition* has the form:

$\text{constraint}(Cname, Exp, [Evariables])$ (3)

where $Evariables$ is simply the list of variables occurring in the expression Exp . Thus, a constraint is a predicate definition: a named logical expression, Exp , evaluating to true or false. The variables $[Evariables]$ occurring in the Exp must be uniquely identified.

The expression is in some constraint language, and is not part of the EDM specification. For example, if the host language is Prolog, then, we might have, for example:

```
parallel(L1, L2):- gradient(L1, G), gradient(L2, G).
gradient(L, G):- G is (LY1 - LY2)/(LX1 - LX2).
```

Constraints are used by applying them to aggregations to form higher level abstract data structures. A *constraint use* includes the path variable definitions and correspondences to variables, in the context of its use. In specifying constraints over data, we use predicates containing variables which take their values from the data being constrained. Since these data may be located in more than one aggregation, we need to refer to them using paths. A path is simply a sequence of names leading from the top level aggregation to the data element referred to:

$Ccall ::= Cname[Pvariables]$ (4)

where

$Pvariable ::= Fename.Pvariable \mid Aname.Avariable \mid Variable:Dname$

$Avariable ::= Aname.Avariable \mid Variable:Dname$

For example,

```
aggregation(al, {Poly1:polygon, Poly2:polygon})
aggregation(polygon, {Edge1:line, Edge2:line})
parallel(Poly1.Edge1, Poly2.Edge2)
```

These paths are defined by sequences of variables occurring in aggregations such that each successive variable is in a subaggregation included in the one before, and such that the last variable in the path is a domain variable of the lowest subaggregation. The Pvariable domains of the constraint definition must be a subset of the Evariable domains of the constraint call, for all Pvariables and Evariables. Pvariables are matched with Evariables by order.

EDM Forms

Given the above base elements, the three EDM forms and a relation are defined below.

Functional Entities (FEs) The FE is the singular data element within the EDM system. It consists of an aggregation and includes procedural semantics encoded as constraints:

$$FE(Fename1, \{Fename2\}, \{Cname[Pvariable1]\}, Aname1) \quad (5)$$

An FE is a composite structure named *Fename1*, composed of a set of FEs, *{Fename2}*, which this one specializes (see discussion of generalization and specialization below), an aggregation *Aname1*, and a set of constraints *Cname* over the variables in the aggregation. We also say that each *Fename2* is a *parent* of *Fename1*.

Here we introduce the *inclusion* relation. Inclusion corresponds to set membership of one structure in another. A structure is *immediately included* in another structure if its name occurs in that structure. Inclusion is the transitive closure of immediate inclusion. A structure is *included* in another structure if it is immediately included in that structure, or in any other structure which is included in the latter. Thus, a variable is included in an FE if it is included in the FE's aggregation or a parent of the FE. We call the set of variables included in a structure its *variable inclusion set*. A constraint is included in an FE if it is a member of the FE's constraint set, or included in a parent of the FE. The *constraint inclusion set* of a form is the set of all constraint calls included in the form.

A condition for the wellformedness of an FE is that the Pvariables of an FE's constraints must form a subset of the FE's included variables, i.e., all constraint variables in an FE are members of the FE's variable inclusion set.

Figure 1 shows a simple FE consisting of only an aggregation. An FE that is made up of both a set of FEs and variables is shown in Figure 2. In the figure the set *{Fename2}* consists of several subFES: *FE2*, *FE3* and *FE4*. As an example, a polygon could be defined as a simple FE:

$$FE(poly, \{\}, \{two_connected[poly.line.Pt1, poly.line.Pt2], line\})$$

The FE can be considered an abstract data type, possibly composed of other abstract data types. These can represent abstract or physical objects. The intended use of FEs is to include variables that have the same self reference. The constraints consist of definitions of measurements and other integrity constraints involving values referring to the same item.

Generalization Generalization is broadly recognized as an important form of data abstraction. We introduce a new definition of generalization as a relation between EDM data elements which can be used in defining their contents. It is a partial order relation between two EDM forms that can be defined by the *subsumption* relation and the *value sets* of the forms.

Subsumption of EDM structures corresponds to the subset relation between their substructures. Intuitively, structure *A* subsumes structure *B* (denoted $A \Rightarrow B$) if *A* can be considered equivalent to a subset of *B*. The addition of variables or constraints to a structure is considered a case of subsumption, i.e., the new structure is subsumed by the original one.

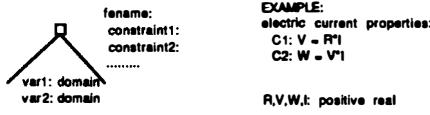


Figure 1. A simple functional entity (FE) structure.



Figure 2. A complex FE structure.

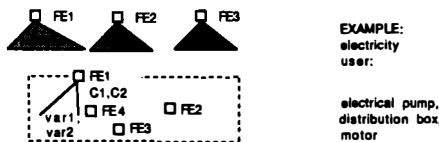


Figure 3. The generalization structure, as related to a complex FE structure.

In general, subsumption between two forms is determined by examining their variable and constraint inclusion sets.

The value set of a form can be considered the subset of the Cartesian product of the variable inclusion set that satisfies the form's included constraints. For a more complete definition, we first define the *constraint product set* to be the set of all variable assignments that satisfy a constraint call. The constraint product set for a set of constraint calls is the intersection of the constraint product sets of the individual constraint calls. The value set of a form is thus defined as the constraint product set of its constraint inclusion set and variable inclusion set.

The generalization relation, denoted $A << B$ (read "B is a generalization of A"), indicates that $\text{value_set}(A)$ is a subset of $\text{value_set}(B)$ (for variables in both forms). In order for the generalization relation to exist between two forms, the subsumption of their variable inclusion sets must hold. By inspection, it is evident that if a form is subsumed by another, it is also a generalization of that form. We note that it is often extremely difficult to determine when the generalization relation occurs. It is usually not possible to determine if adding a constraint actually reduces the value set of a form. Because of this, subsumption alone can be utilized as a less complete but more determinable definition of generalization.

Specialization is the inverse relation to generalization. It is not defined as a separate relation, but is embedded in FE definitions, i.e., an FE is a specialization of each of its parent FEs. Through specialization, the variables and constraints within an FE may come from many other FEs, which have been specialized into a lattice of FE definitions. We diagram the generalization relation as shown in Figure 3. Parent FEs are identified by an inverted funnel. The specialized FE carried within another FE is identified by name.

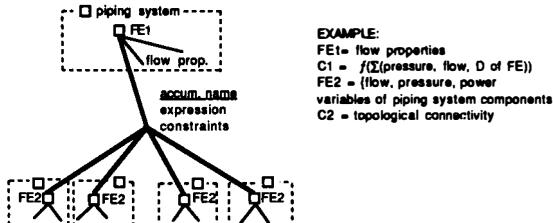


Figure 4. The graphic presentation of an accumulation, which relates the FEs within several other FEs.

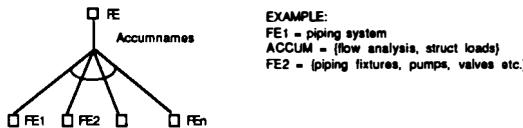


Figure 5. The graphic presentation of a composition.

Generalization is the means to organize definitions about conceptual classes of objects and instances of those objects. Instead of making a strict distinction regarding class and instance, we recognize that specifications in design can be made at any level of specificity and shared across multiple sub-classes or across instances of physical objects. There is no definite closure on a specification in design, as new variables representing new performances may always be added. Thus new aggregations or FEs may be added even to base FEs, that have all variables bound to values. This range of capabilities is fully supported by the generalization relation as defined here.

Accumulations

Another relation in engineering data involves the accumulation of detailed functional information into more aggregated forms:

$$\text{accumulation}(\text{Accname}, \text{Fename1}, \{\text{Cname1}[\text{Pvariable1}]\}, \{\text{Fename2}\}, \{\text{Cname2}[\text{Pvariable2}]\}) \quad (6)$$

An accumulation is a set of relations named *Accname*, defined by a set of constraints *{Cname1}* over both *Fename1* and the set of FEs, *{Fename2}*. In addition, an accumulation includes a constraint set *{Cname2}* that defines the preconditions that the FEs must satisfy. That is, the constraints define the necessary relations between FEs if they are to possess the constraints defined in the expression, *{Cname1}*. Figure 4 diagrams an accumulation. It graphically shows the relation over FEs, possibly nested within other FEs.

The interpretation of an accumulation is that it defines a relationship between separate FEs. The dependency may involve any type of relation: for functioning, for manufacturing, for allowing maintenance, for aesthetics, etc. An accumulation is based on a mathematical

function (possibly a complex algorithm) that computes one or more new properties from a set of FEs to define a new FE. Most often the accumulation derives values bottom-up in a many-one direction. These are typically evaluations. Sometimes they are processed top-down, to define performance goals to be achieved. The accumulation includes restrictions in the form of constraints based on relations that the referenced FEs must satisfy. These constraints would be those that define the wellformedness of a piping or structural system, or the size constraints between the parts of an assembly, for example.

Compositions

Multiple accumulations often apply to the same set of FEs. In such cases, the one-to-many relation corresponds to an assembly which is evaluated by its corresponding set of accumulations. As in normal parlance, we call such a relation a *composition*:

$$\text{composition}(\text{Comname}, \text{Fename1}, \{\text{Fename2}\}, \{\text{Accname}\}) \quad (7)$$

A composition is a named structure between one FE and a set of FEs that compose it. The set of accumulations specify the standard functions of the composition, and the composition's wellformedness constraints. The set $\{\text{Fename2}\}$ is the replacement to Fename1 , defining it in more detail (Figure 5). The information gain of the composition is that the precondition constraint sets of the accumulations over the set of FEs are resolved all together. This set of constraints is probably normalized and reduced to a much smaller set than those defined by the accumulations separately.

Composition and its inverse relation *part-of* have been proposed structures in most other product models (Shaw et. al. 1989), (Cornelio and Navathe, 1989). Part-of defined separately loses most of its semantic meaning. We always derive it from composition.

Discussion

Given each of these structures, their relation to one another can be interpreted:

- (a) An FE is a specialization of an aggregation that restricts its possible value combinations by imposing constraints. The constraints allow combinations of values that satisfy the definitions of the properties held in the FE.
- (b) Accumulations define relations between disjoint FEs, especially for the support of some function or intention. They involve a one-to-many relation.
- (c) Compositions are sets of accumulations corresponding to a set of intentions or functions, that apply to a one-to-many set of FEs. A composition typically corresponds to a technology, a way of composing some physical or abstract object in support of a defined set of functions. Thus, Wankel and cylindrical engines are alternative technologies having similar functions and capabilities. The composition is the efficient encoding of the set of accumulations over a set of FEs.

Constraint relations are highly complex. We have four classes of relation that allow characterization of certain information encountered in design:

- (a) complex domain relations: (in FEs) for specifying definitions, and relations among redundant sets of data;
- (b) class defined lattices: (generalizations) for specifying relations among definitions and partial specifications, particularly as defined by users;
- (c) accumulations derived across sets of data: for defining performance accumulations across systems of components, such as structural, electrical; also for aggregating shape, mass properties and other data;
- (d) technologies that package functions: (compositions) for defining assembly classes that respond to a set of functional criteria.

The three forms are premised on the assumption that all design data can be represented as sets of variables and constraints across those variables. This is in contrast to current translation and archiving efforts that attempt to define specialized data structures that embed identity and other constraints within a semantic domain, such as geometry.

EXAMPLE

We develop here a small detailed example using the EDM information structures, to illustrate their use. The result is a detailed, logical structure equivalent to a schema of a CAD database. The example presented here is of a building wall as found in residential construction, an example with which most readers are intuitively familiar. Here, a wall is defined as a partition between two spaces. The spaces may be empty and occupiable (rooms) or not occupiable (a duct chase).

In this section, we use the EDM structures to define a product model of walls that responds to the needs of design, including drawing generation and thermal performance. Even for this small example, the data model is fairly complex, and we present it decomposed as three interrelated structures: the FE and composition structure, the structure of geometric data and a structure supporting thermal analysis. The structures are presented in diagrammatic form here, with detailed definitions included in the Appendix.

A High Level FE Structure

The graphic notation is that used in Figures 1 to 5. A wall can be composed in many ways. Probably the most common form of wall in U.S. construction is what we call a core wall. The compositional structure of a core wall, in this example, is:

wall segments: solid regions of wall that provide structure and barriers and are of homogeneous construction;

openings: voids or special fillers in a wall that have a purpose distinct from wall segments, such as allowing human passage or transmission of light; examples are doors and windows;

pass-thrus: FEs such as electrical wiring, piping and ductwork that use the wall for routing.

Wall segments can be further decomposed into:

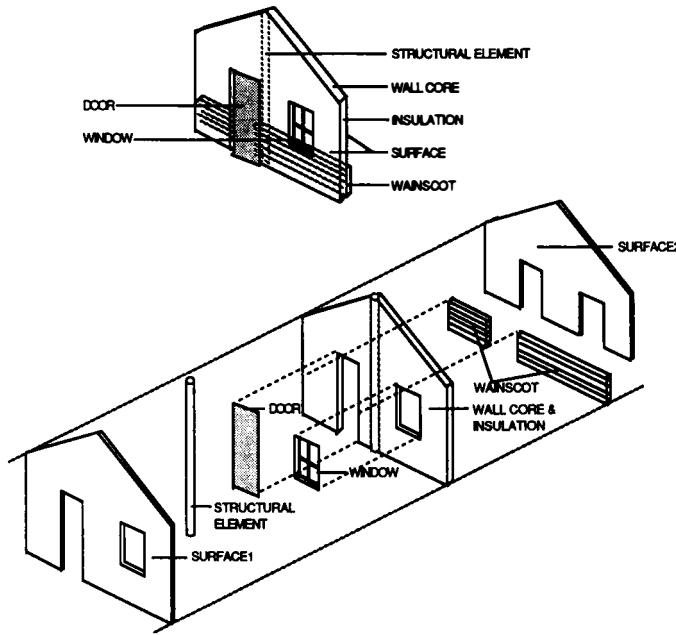


Figure 6. The set of FEs making up a core wall, according to the given compositions.

a core: a self-supporting (and possibly able to bear external loads) portion, which may have hollow interstices;

insulation: material used to fill core interstices to change functional characteristics of the wall segment;

surfaces: layers of material added to both sides of the core to change functional characteristics of the wall. These include visual appearance, light and sound reflection. Surfaces range from lath, gypsum board and plaster to wallpaper and paint. Multiple surfaces may be built up over the core on both sides. Constraints apply to the order of surfaces.

A wall with this structure is shown in exploded isometric format in Figure 6. The component FEs can be readily defined. The FEs carried by the wall, the segment, core surfaces etc. and how they are managed is more difficult to lay out.

The overall structure of EDM forms is presented at the top level in Figure 7. The compositions partition a wall segment into components with distinct functions. Lighting analysis relies on the properties of surfaces and transmission through openings; structural considerations rely only on core properties, for example. Rules of composition are reduced in complexity by these structures. For example, wall segments and openings are disjoint;

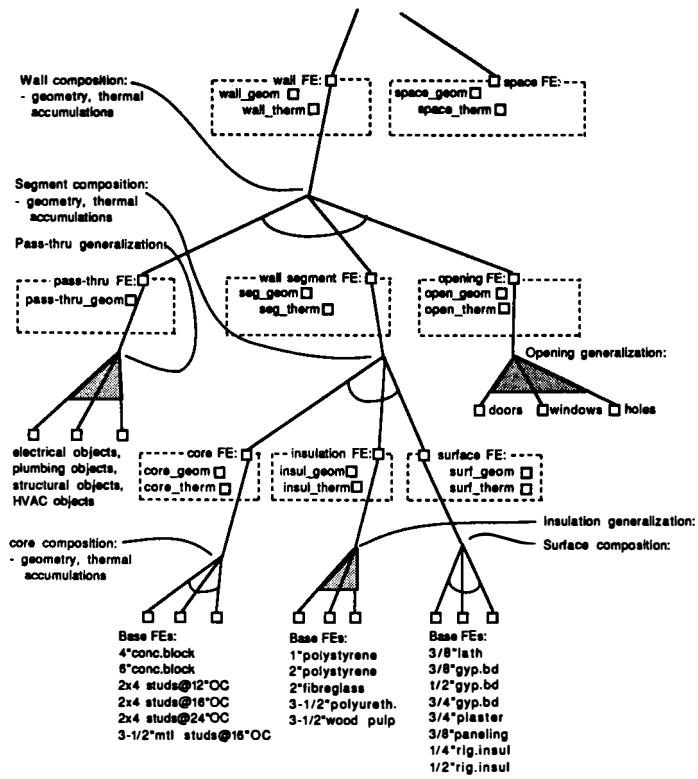


Figure 7. Graphical representation of the FE structures for core walls of buildings.

pass-thrus may not coincide with openings. A subset of the physical components that can be used in walls is also shown in Figure 7.

Several of these FEs are decomposed further. Cores of walls are decomposed into the individual FEs used in different types of construction. Openings are decomposed into the FEs composing a door or window. Some leaf FEs are not made up of compositions, but rather generalizations. They are entities that are not composed, but selected, and the leaves are the alternative specializations from which a specific instance can be selected.

Shape FE Structures

It is important to note that the various FEs for a product model may be defined by different design disciplines, in consideration of different functions. The descriptions used by multiple disciplines, such as geometry, are where important coordination between FEs is required. Geometry is the most important FE in most design disciplines. Some of its constraints are based on fundamental physical principles, for example, with regard to solids being disjoint in space-time. Geometry also provides the presentation-level descriptions needed for

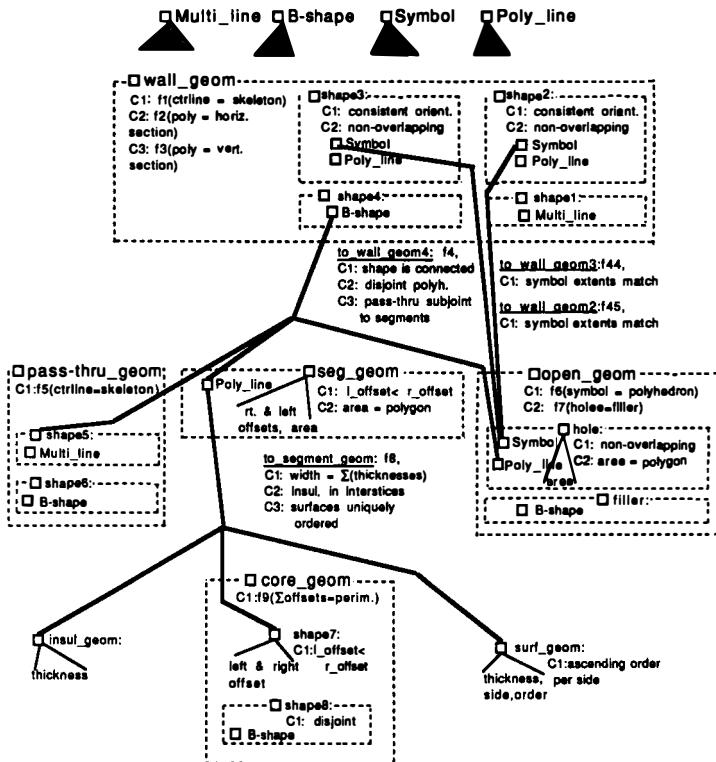


Figure 8. The geometry FEs and their accumulations for the core wall example.

different forms of drawing, used in fabrication and some forms of evaluation, such as building code checks. Geometry also is needed for certain other analyses. In buildings, geometry is the dominant FE, as the provision of space is usually the principal purpose of the product. These descriptions become embedded FEs carried by the wall FE and its components.

Like most FEs, geometry tends to be managed at several levels of detail, using different descriptions. The various descriptions are used for different analyses and evaluations. As revisions occur, the integrity of these different geometric descriptions must be maintained. Integrity within FEs is managed by FE constraints, those between FEs by accumulations.

A set of geometry FEs are shown diagrammatically in Figure 8 and defined more fully in the Appendix. They rely on several pre-defined FEs used to define geometry: sets of lines (*Multi_lines*), *Symbols*, polygons (*Poly_lines*), and solid shapes (*B-shapes*). These are specialized by the FEs that use them. Other geometric properties are defined as needed by the various FEs. Means to coordinate these descriptions, through constraints and accumulations, are also presented.

Here, we assume that all coordinates are located relative to a single global coordinate system. This is appropriate for archiving and translation. A transform and parent reference, on the other hand, may be desired for a design database.

Walls The top level of wall geometry treats the wall as a single entity. Four different geometry FEs are commonly used to describe a wall and are embedded within it as specializations, as diagrammed in Figure 8.

The first description (*shape1*) simply defines a wall by its centerline. This description is used for feasibility analysis and circulation studies. These highly abstracted walls allow space sizes and relations to be defined, without consideration of more detailed issues. (For examples of this type of wall description, look in the real estate section of a Sunday newspaper for apartment layouts.)

The second and third definitions of wall geometry are for plan (*shape2*) and elevation (*shape3*). In plan, the wall geometry is a set of polygons of the wall outline, with inserted symbols showing the location of doors and windows. In elevation, it is a set of vertical polygons of the wall, with symbols showing the location of doors and windows. These descriptions are used for schematic drawings of plans and elevations and are typically supported by architectural CAD systems. They rely on the pre-defined *Poly_line* and *Symbol* FEs, which they both specialize. Editing operations manage the treatment of wall joining conditions, so they may appear continuous.

The fourth description (*shape4*) defines the wall as a set of solid model polyhedra. Here, geometry is represented in boundary representation format, already predefined as an FE. This description is the most detailed for the wall geometry as a whole; further details are represented only for its components. This description is used for 3-D visualizations, spatial conflict evaluation, and various evaluations that will be described later. This level of definition is usually omitted in conventional architectural practice because of the cost of its manual generation; its realization is sometimes as wood or plastic models.

These geometry definitions are packaged into a single geometry FE and carried by the wall FE. They are sufficient for layout, drawing and most other high level uses. The wall geometry carries constraints needed to maintain the consistency of the various definitions: that the polygons and symbol are mutually non-overlapping and that the polygons are consistently oriented. If any one definition changes, they allow checking and possibly updating of the others (constraints are defined schematically in the Appendix).

Accumulations define the relations between the geometric description of the wall and the geometric description of the components of the wall. The first checks that the boundary rep description of the wall is consistent with the part descriptions below. Through the *wall_geom* constraints, it maintains consistency with the centerline, plan and elevation representations. Two other constraints maintain the equivalence of opening symbols, one for plan and the other for elevation symbols. Notice that all geometric FEs at this level are related by constraints, either from *wall_geom* or the accumulations.

Core Wall Components At the second level, a wall is broken into a specific composition, here consisting of three FEs: *pass-thrus*, *openings* and *segments*. These units of composition would change if another wall composition were used; (Eastman, Bond and Chase, 1991b).

Pass-thrus are objects routed through a wall to serve functions different from the wall. They are defined at two levels of detail. The first defines the routing as a centerline. This level definition allows identification of major conflicts, such as pass-thrus conflicting with openings. The next level definition provides a complete B-rep geometric description of the pass-thru, if needed. This FE would be carried by the object level FEs such as plumbing, electrical or heating and ventilation objects. The views of pass-thrus must be maintained consistently by constraints residing in the top level *pass-thru* FE.

Openings are another FE component of a wall. They consist of regions of wall, without a core, that serve special functions, such as allowing human passage or light. Like pass-thrus, they are defined at two levels of detail. The first level definition supports a wall's composition in terms of its vertical and horizontal extent. However, the detail definition of the filler of the opening is described separately, as a 3-D shape. This description allows complete detailing, if necessary.

Wall segments define methods of construction and correspond to homogeneous vertical regions of core, insulation and surfaces. At the top level, the geometries of these components are accumulated together. The wall segment accumulation consists of the derivation of the total wall segment shape. Detailed thicknesses are built up for wall segments from their core, then from the surfaces added to each side. The next level geometry description defines the composition of wall segments, in terms of *core*, *insulation* and *surfaces*, their thickness and, for surfaces, their side and order.

This structure suggests the general configurational rules of the EDM data model. Some FEs require high level packaging because they need external constraints for their integrity maintenance. Some FEs include both specialized FEs and aggregations in their definition, such as *open_geom.hole* and *segment.geom*.

Discussion The different structures presented should be adequate for representing geometry for most uses of a product model for buildings. These FEs correspond to views in database terms and are required by the different types of drawings typically generated. Additional forms of functional evaluation could necessitate additional geometric FEs and these would have to be added to this structure. This many levels is assumed to be typical of the different views of geometry found in other areas of design and engineering.

We assume that the geometry FE structure would continue to be accumulated upward beyond the wall, providing the equivalent of a regionalization scheme, supporting efficient spatial queries (Bentley and Friedman, 1979), (Tammisen, 1981) for uses in spatial conflict testing, visualization studies and so forth.

Structures for Deriving Thermal Performance

Another illustration of the structure of FEs involves the thermal analysis of buildings. Here we consider steady state thermal flow through the wall and use as our guide, Kinzey and Sharp (1963).

The basic calculation for energy flow through a material is:

$$H = U * A * (t1-t2) \quad (8)$$

where

- H = rate of heat flow in btu/hr
- U = overall coefficient of transmission, in BTU/hr/ft²/°F
- A = area of construction having coefficient U , in ft²
- t_1 = temperature, warmer side in °F
- t_2 = temperature, colder side in °F

The coefficient U is derived from combining the behavior of several phenomena, including the surface air barrier on both sides of the partition, the thermal properties of the individual layers of material and/or the effect of interstitial air spaces.

In general, these take the form:

$$U = \frac{I}{(r_1 + r_2 + r_3 + \dots + r_n)} \quad (9)$$

where r is the thermal resistance of each component of the partition, including surface and interstitial air. The r values of all common building materials have been calculated and are available in tabular form.

Our interest is in the thermal performance of a wall between two spaces, usually one inside and the other being the outside. From eq. 8 it is seen that the wall's performance is a function of the temperature in the two spaces; these must be accessed in doing the accumulation. The performance of the wall is an accumulation of properties of the materials in each wall segment or opening multiplied by the area of the wall segment. For aggregation of intermediate results, we compute:

$HS = U * A$, for each opening and wall segment, and eq. 8 becomes

$$H = \text{sum } (HS * (t_1 - t_2)) \text{ for each space.} \quad (10)$$

The needed accumulations are shown diagrammatically in Figure 9. Notice that they are specific to the core wall composition. The FEs carried by the wall FEs are presented in the Appendix and described below.

Each space must carry its ambient (desired) temperature amb_T and the summation of all heat exchanges. Its values are derived from an accumulation of the energy flows through all bounding walls and surfaces. Each wall carries the summation of all heat exchanges resulting from its component wall segments and openings (pass thrus are ignored). Each wall's performance is an accumulation of its wall segments and openings. These each carry the total heat flow through themselves. This heat flow is the product of the coefficient of conduction across all its component materials and its area. The area of the wall segment or opening is carried in the geometry of the wall segment or opening and is used in the summation. The HS value for both opening and segment is managed by a FE constraint, as the relations include both geometry and thermal properties.

The wall segment HS is derived from the accumulation $to_segment_therm$. It accesses various FEs in the composition of the segment. Each of the wall segment components, i.e., core, insulation and surface, carries its resistance r . In addition the surfaces carry a film resistance for the air film. In most cases, the derivation of the thermal resistance of a

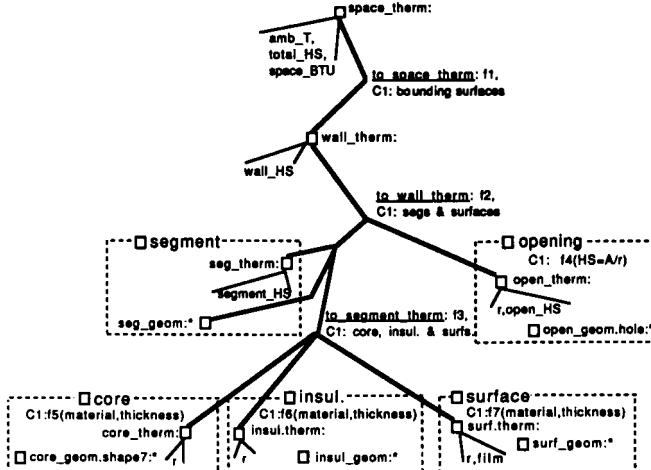


Figure 9. The thermal FEs and their accumulations for the wall example. The accumulations use both thermal and shape variables.

material is dependent upon its thickness, accessed from its geometry. The resistance is managed by a constraint carried in the core, insulation or surface FE, as it must access different subFEs within these objects.

We have focused on thermal transmission. A complete analysis would also require derivation of the effects of thermal infiltration and ventilation.

Integration of Structures

The two FE, constraint and accumulation structures: geometry and thermal, would in practice all be integrated into a single FE structure of the wall. They were separated here for expository uses. Each of the structures can be evaluated separately and be shown to roughly function as desired. In fact, Figure 7 is based on their integration and defines the subFEs carried by each of the higher level objects.

Figure 7 shows a single composition between the wall structure and its components. It generally groups the relevant components in a manner allowing them to be easily selected and manipulated. This composition also carries the common accumulations. Various other accumulations might be added as design proceeded. It should be noted that some accumulations over the same wall elements might not be integrated with the wall composition. Only those accumulations that operate between the same two levels of description as the composition could be integrated. Some accumulations, for example, might not need the wall as a structure and accumulate instead directly to the space FE. In fact both of the analyses presented here could be organized that way. By breaking accumulations at the wall, the intermediate results of analyses to this level are recorded. Some benefits of carrying such intermediate results are discussed elsewhere (Eastman, Bond

and Chase, 1991b).

APPLICATIONS OF EDM

Some of the uses of EDM data models for various products are discussed below. In previous papers, consideration was given to the criteria that a product model should satisfy (Eastman, Bond and Chase, 1991a and 1991b). These included extensibility, in terms of adding new methods of evaluation and new forms of composition, support for multiple levels of abstraction, representation of functional criteria as well as the product's form, and support for inference making and integrity checking. We refer the reader to these earlier papers for discussion of these criteria.

Database Definition Within a CAD Environment

The general use of EDM for product oriented design databases can be considered from inspecting the example. EDM allows developers to lay out a product model and to investigate its extension to support new functional criteria or new objects and composition schemes. EDM includes information beyond what is typically included in a CAD database today, such as structures for abstract data types, multiple hierarchical relations, inheritance and data objects. These structures are becoming important for capturing the semantics needed in advanced CAD databases as well as expert systems. Where data will come from can be explicitly mapped and integrity conditions defined. Issues of functionality can be readily addressed.

EDM allows defining the semantics of a particular application domain, for a given set of technologies and set of functional considerations. The structures are also useful in allowing extension to the application domain, by allowing addition of new technologies and/or new functional evaluations.

EDM offers another capability that warrants its consideration for more integrated use. As multiple applications become capable of adding to and modifying a CAD database, integrity testing of the database becomes mandatory. An important evolution of data independence is to move integrity management from the application programs (in which it is impossible to embed all database semantics) to the database. Thus an additional use of the database is in communicating the integrity status of the data it receives from and sends to applications.

Within a larger environment, it is assumed that the constraints could be evaluated as to their truth-value on the demand of applications. The pathnames define the specific variables needed by the constraint expressions. Each constraint is embedded into a structure with a well-defined scope: an FE, accumulation or composition. Thus each FE, accumulation and composition could be evaluated for its truth-value and the truth-values of its constituents.

This structure of integrity values, some of which are likely to be false, poses a challenge for proper evaluation, in light of the different needs of applications that will use the data. A simple integrity check for an application is to evaluate constraint truth-values and accepting as valid data that which is referenced only by constraints that evaluate to true. Various applications, however, do not require global integrity, but only that specific preconditions be

met (Kutay and Eastman, 1983). Finer grain integrity checking, that can respond to specific demands, is called for, but remains for further development. Another unresolved issue is the propagation of one constraint violation to others that may be invalid because the prior one is.

Several other extensions to EDM would be required. Specific output and file formatting capabilities would be required for defining operational interfaces to CAD applications. In addition, a database would need to record transaction-based data, such as a version identification, source of data, and owner.

Use in Translation

An alternative use of EDM is in the translation of engineering data among applications. Here, the product model should include the integrity checking outlined above. Again, the possible use of EDM is only outlined.

The problem of translation involves the following steps:

- (a) extraction of the different information from the source database or file.
- (b) making necessary inferences (accumulations, compositions, generalizations) to restructure the data into the forms needed by the other system. This step may also include making inferences from the constraints, after asserting that they are true.
- (c) formatting the data so that it can be accessed by the receiving application.

FEs define the possible views of data. In our example, the views were for drawings and thermal analysis. For extraction, these views could be classified by the type of dataset in which they reside. For example, the wall polygon views in plan and elevation drawings could be extracted for use in inferences and accumulations. After data is extracted for one FE, the constraints in the enclosing FE define how such information recognized in one view may be translated to other views, and the accumulations define how they may be restructured for use in other applications. For example, a wall section may be input, indicating the various materials and thicknesses for the core, insulation and surfaces. The R-value for the wall sections could then be accumulated from the constituent values for energy analysis.

CONCLUSION

EDM responds to the needs of a data model for engineering product databases, as discussed in the introduction. Partial integrity is supported, by providing explicit constraints and evaluation of their truth value. Some data can be stored that is known to be possibly incorrect. Incomplete data is also supported, in the same way. The full ramifications of these capabilities are still being explored.

EDM was defined so as to allow dynamic and incremental schema definition, as the design

evolves. In order to fully support this capability, the authors are working on a language for defining the EDM structures.

Until now, there has not been a means to map out, plan or design new CAD applications. Our limited experience with EDM suggests that it is useful for designing product models. It allows definition of composition structures and the rules for their application. It allows easy identification of the interfaces and changes one must make to integrate a new application. It makes clearer the limitations of a particular conceptual design. Further work is required, however, before it can be applied widely.

Acknowledgements The work presented here has been supported by the National Science Foundation, grant number DDM-8915665.

REFERENCES

- Atwood, T. M. (1985). An object-oriented DBMS for design support applications, *Proc. IEEE COMPINT '85*, Montreal, Canada, pp. 299-307.
- Baer, A., Eastman, C. and Henrion, M. (1979). Geometrical modeling: A survey, *Computer-Aided Design* 11(5): 253-272.
- Batory, D. S. and Kim, W. (1985). Modeling concepts for VLSI CAD Objects, *ACM Transactions on Database Systems* 10(3): 322-346.
- Bentley, J. and Friedman, J. (1979). A survey of algorithms and data structures for range searching, *ACM Computing Surveys* 11(4): 397-409.
- Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks, *Communications of the ACM* 13(6): 377-381.
- Codd, E. F. (1979). Extending the database relational model to capture more meaning, *ACM Transactions on Database Systems* 4(4): 397-434.
- Cornelio, A. and Navathe, S. B. (1989). *Modeling the structure and function of engineering designs*, unpublished paper, Department of Computer Science, University of Florida, 20 pgs.
- Eastman, C. M., Bond, A. H. and Chase, S. C. (1991). A formal approach for product model information, *Research in Engineering Design* (in process).
- Eastman, C. M., Bond, A. H. and Chase, S. C. (1991). Application and evaluation of an engineering data model, *Research in Engineering Design* (in process).
- Geilingh, W. (1988). *General AEC Reference Model (GARM)*, ISO TC184/SC4 Document 3.2.2.1 (Draft), October, TNO-IBBC.
- Kemper, A., Wallrath, M. and Lockemann, P. C. (1987). An object-oriented system for engineering applications, *Proc. SIGMOD 1987 Annual Conference*, pp. 299-310.
- Kinzey, B. Y. and Sharp, H. (1963). *Environmental Technologies in Architecture*, Prentice-Hall, N. J.

- Kutay, A. and Eastman, C. M. (1983). Transaction management in engineering databases, *SIGMOD Conference on Engineering Databases*, IEEE, paper 31.3.
- Kwok, P. C. (1988). A thinning algorithm by contour generation, *Communic. ACM* 31(11): 1314-1324.
- Miller, R., Southall, J. W. and Wahlstrom, S. O. (1979). Requirements for management of aerospace engineering data, *Computers and Structures* 10: 45-52.
- Peckham, J. and Maryanski, F. (1988). Semantic data models, *ACM Computing Surveys* 20(3): 153-190.
- Requicha, A. (1980). Representations of rigid solids: theory, methods and systems, *ACM Computing Surveys* 12(4): 437-466.
- Shaw, N. K., Bloor, M. S. and de Pennington, A. (1989). Product data models, *Research in Engineering Design* 1(1): 43-50.
- Sibley, E. H. (1974). Data Management Systems - User Requirements, in D. Jardine (ed), *Data Base Management Systems*, 1973 SHARE Working Conference, North-Holland press, N. Y., pp.83-104.
- Smith, B. (1986). *A Report on the PDES initiation effort*, National Bureau of Standards, (May 20), Gaithersburg, MD.
- Tamminen, M. (1981). The EXCELL method for efficient geometric access to data, *Acta Polytechnica Scandinavica*, UDC 681.3.025, Helsinki.
- Ulfssby, S., Meen, S. and Oian, J. (1982). Tornado: a data-base management system for graphics applications, *IEEE Computer Graphics and Applications*, pp. 71-79.
- Van Griethuysen (ed) (1983). *On Conceptual Schema*, ISO Report TC97/SC5/WG3.

APPENDIX

Geometry FEs

In order to define geometry succinctly, it is necessary to first introduce four aggregations used repeatedly in geometric representations: *2point*, *3point*, *2line*, *3line*, and six FEs, used repeatedly:

```

aggregation(2point, {xcoord:real, ycoord:real})
aggregation(3point, {xcoord:real, ycoord:real, zcoord:real})
aggregation(2line, {pt1:2point, pt2:2point})
aggregation(3line, {pt1:3point, pt2:3point})
FE(Multi_line, {}, {C1}, {[line(i):3line]}) where
  C1: 3lines are end-connected in a chain
FE(Poly_line, {Multi_line}, {C1}, {}) where
  C1: Multi-line is well-formed planar polygon

```

Aggregations are used until constraints are needed, creating the *Poly_line* FE. Some of these are then used to define a boundary representation polyhedron FE and an arbitrary symbol:

$FE(Loop, \{Poly_line\}, \{C1\}, \{\})$ where
C1: Poly_lines are oriented
 $FE(Face, \{Loop\}, \{C1\}, \{\})$ where
C1: nested loops must be consistently oriented
 $FE(B\text{-shape}, \{Face\}, \{C1\}, \{\})$ where
C1: all 3lines must be pairwise matching
 $FE(Symbol, \{\}, \{\}, [line(i):3line])$

The *B-shape* FE here is a simplified boundary representation consisting of a hierarchical set of *faces*, *loops*, *edges* and *3point* (Baer et al., 1979), (Requicha, 1980). These FEs are used in the following example for illustration purposes; the geometric definitions could instead follow the more general (and verbose) definitions in IGES, Version 4.0 (Smith et al., 1988). *Symbol* is an arbitrary set of lines.

Walls The top level of wall geometry consists of a single FE. It is made up of four sub-FEs and three constraints to manage their consistency, as diagrammed in Figure 8.

$FE(shape1, \{Multi_line\}, \{\}, \{\})$: a chainline representing the centerline of the pass-thru item. Here, the geometry and constraints are inherited, without extension.

$FE(shape2, \{Poly_line, Symbol\}, \{C1,C2\}, \{\})$ where

C1: nested loops must be consistently oriented

C2: Poly-lines and Symbols must not overlap

a set of *polygons* and a set of *symbols*; *polygons* describe the wall in plan; a *symbol* is a set of line segments that may be inserted into spaces between wall polygons. The constraints guarantee that the wall plan is well-formed.

$FE(wall_geom3, \{Poly_line, Symbol\}, \{C1,C2\}, \{\})$ where

C1: nested loops must be consistently oriented

C2: Poly-lines and Symbols must not overlap

poly_line is a set of polygons, describing the wall in elevation; *symbol* is a set of line segments defining window or door symbols in elevation.

$FE(wall_geom4, \{B\text{-shape}\}, \{\}, \{\})$: a set of boundary representation shapes, representing the walls and doors and segments of the wall.

These geometry FEs are packaged up into an overall wall geometry FE:

$FE(wall_geom, \{shape1, shape2, shape3, shape4\}, \{C1, C2, C3\}, \{\})$ where

C1: constraint(wall_geom1_con, f1(skeleton of polygon),

{wall_geom.shape1.Multi_line, wall_geom.shape2.Poly_line}): maintains the equivalence between the centerline of the wall and the skeleton on the wall plan polygon (Kwok (1988) gives an algorithm for generating the skeleton).

C2: constraint(wall_geom2_con, f2(horizontal section of polyhedron),

{wall_geom.shape2.Poly_line, wall_geom.shape4.B-shape}): the floorplan wall polygon is maintained geometrically equivalent to a horizontal section cut of the wall polyhedron.

C3: constraint(wall_geom3_con, f3(vertical section of polyhedron),

{wall_geom.shape3.Poly_line, wall_geom.shape4.B-shape}): the wall elevation polygon is maintained geometrically equivalent to a vertical section cut of the polyhedron.

Three accumulations relate this level description to the more detailed ones of the wall's components:

*accumulation(to_wall_geom4, wall_geom.shape4.B-shape,
f4(shape union(all pass-thrus, segments, openings)),
{pass-thru.shape5, seg_geom, open_geom.filler.B-shape}), {C1, C2, C3}):* the vertical polygons of openings and segments are each made into a 2-1/2D polyhedron and unioned together. Pass-thrus are located within the wall.
C1: segment_geom and open_geom cover a single connected region
C2: all seg_geom and open_geom are mutually disjoint
C3: all pass-thru_geom subjacent to seg_geom

*accumulation(to_wall_geom3, wall_geom.shape3.Symbol,
f44(equivalence of symbol extents), {open_geom.hole.Symbol}, {})*
*accumulation(to_wall_geom2, wall_geom.shape2.Symbol,
f45(equivalence of symbol extents), {open_geom.hole.Symbol}, {})*

Core Wall Components Pass-thru geometry is defined with two component FEs:

FE(pass-thru_geom.shape5, {Multi_line}, {}, {}): a polyline, defining the centerline of the utility or other FE passing through the wall.
FE(pass-thru_geom.shape6, {B-shape}, {}, {}): a set of B-rep polyhedrons defining the FEs passing thru the wall.

The pass-thru geometry FE carries the two FEs above. It also must maintain their consistency. One implementation of the consistency maintenance constraint is as a comparison of the skeleton of the polyhedral shape, using a thinning algorithm, to the centerline (Kwok, 1988):

*FE(pass-thru_geom, {shape5,shape6},{C1},{}) where
C1: constraint(pass-thru_geom_con,f5(skeleton of polyhedron),
{pass-thru.shape5.Multi_line, pass-thru.shape6.B-shape})*

The *pass-thru* FEs would also reside in hierarchies associated with the function of the object, such as the electrical or water subsystems.

The opening geometry descriptions are:

*FE(hole,{Symbol, Poly_line}, {C1, C2}, hole_agg) where
aggregation(hole_agg, {area:real}):* a vertical polygon of the opening and a symbol of its filler are defined using the pre-defined FEs. In addition, the area of the polygon is carried in an aggregation defined at this level.

C1: symbol non-overlapping with polygon.

C2: area = polygon area.

FE(filler, {B-shape}, {}, {}): a B-rep polyhedron defining the geometry of the filler, e.g., window or door, for the wall opening.

Two constraints at the opening FE level maintain the consistency of these descriptions:

FE(open_geom, {hole,filler}, {C1, C2}, {}) where

C1: constraint(open_geom1_con, f6(symbol equivalent to polyhedron),

{open_geom.hole.Symbol, open_geom.filler.B-shape})

C2: constraint(open_geom2_con,

f7(vertical polygon equivalent to silhouette of polyhedron),

{open_geom.hole.Poly_line, open_geom.filler.B-shape})

These constraints deal with two equivalences: a symbol equivalence with the geometry of the opening filler and the equivalence of a polygon with the perimeter of the wall filler shape.

The segment geometry is described with both specialized FEs and a new aggregation:

FE(seg_geom, {Poly_line}, {C1, C2}, seg_geom_agg) where

aggregation(segment_geom_agg, l_offset:real, r_offset:real, area:real)): a vertical polygon and two offsets from the centerline to the sides of the wall; area of the polygon.

C1: l_offset < r_offset

C2: area = polygon area

In *seg_geom*, *l_offset* and *r_offset* are measured relative to the polygon orientation. Segment and opening geometry FEs are defined at this level as vertical polygons, so that the general composition of the wall can be consistently defined:

accumulation(to_seg_geom, seg_geom.Poly_line, f8(sum thickness from centerline),
{all core_geom.shape7.offsets, all surf_geom.thickness, all insul_geom.thickness},

{C1, C2, C3}): where

C1: width = sum of thicknesses

C2: insul. thickness =< core thickness

C3: surfaces uniquely ordered.

Segment Components The next level geometry description defines the composition of wall segments, in terms of core, insulation and surfaces, their thickness and, for surfaces, their side and order:

FE(insul_geom, {}, insul_geom_agg) where

aggregation(insul_geom_agg, {thickness:real}): the thickness of the insulation going into the interstices of the core.

FE(surf_geom, {}, {C1}, surf_geom_agg) where

aggregation(surf_geom_agg, {thickness:real, side:{left,right}, order:integer}): the

thickness, side and ordering of various surface finishes of the wall. (Wall surfaces have partial orders and possibly constraints; these are not detailed for this example.)

C1: order values positive ascending for each side.

Wall cores can be defined at two levels of detail:

*FE(core_geom.shape7, {}, {C1}, core_geom1_agg) where
aggregation(core_geom1_agg, {l_offset:real, r_offset:real}): the offsets to the sides of
the wall core from the centerline*

C1: l_offset < r_offset

*FE(core_geom.shape8., {B-shape}, {C1}, {}): a B-rep polyhedron defining the shapes of
core components of the wall
C1: polyhedra are disjoint.*

The *core* FE maintains the consistency between the polyhedral representation of the core components and the core thickness through the constraint:

*FE(core_geom, {shape7, shape8}, {C1}, {}) where
C1: constraint(core_geom_con,
f9(vertical silhouette of polyhedron equivalent to thickness),
{core_geom.shape7, core_geom.shape8.B-shape})*

Thermal FEs

Space The space thermal FE carries its ambient temperature, and total energy flow, calculated by an accumulation from all bounding surfaces:

*FE(space_therm, {}, {}, space_therm_agg) where
aggregation(space_therm_agg, {amb_T:deg.F, space_BTU:BTU/hr., total_HS:HS})
accumulation(to_space_therm, space_therm, f1(sum(wall_HS * temperature differential),
{wall_therm.wall_HS}, {constraints defining bounding surfaces of space}))*

Wall The wall thermal FE stores its total heat exchange:

*FE(wall_therm, {}, {}, wall_therm_agg) where
aggregation(wall_therm_agg, {wall_HS:HS})*

The wall accumulation is over its component wall surfaces and openings:

*accumulation(to_wall_therm, wall_therm, f2(sum(segment_HS, open_HS)),
{seg_therm.segment_HS, open_therm.open_HS},
{constraints defining relevant segments and openings})*

Wall Segments and Openings The FEs and aggregations for the wall segment and opening are:

```
FE(seg_therm, {}, {}, segment_therm_agg) where
    aggregation(segment_therm_agg, {seg_HS:HS})
FE(open_therm, {}, {}, open_therm_agg) where
    aggregation(open_therm_agg, {open_HS:HS, r:resistance})
```

The HS value for openings is maintained by an FE constraint:

```
constraint(open_therm_cons, f4(HS = area * sum(l/resistance)),
    {open_therm, open_geom.hole})
```

The accumulation for a wall segment is over its component core, insulation and surfaces:

```
accumulation(to_segment_therm, segment_therm, f3(area * sum(l/resistance)),
    {core_therm, insul_therm, surf_therm},
    {constraints defining relevant cores, insulation and surfaces})
```

Cores, Insulation and Surfaces Each component FE carries its resistance. In addition, surfaces also carry an air film resistance:

```
FE(core_therm, {}, {}, core_therm_agg) where
    aggregation(core_therm_agg, {r:resistance})
FE(insul_therm, {}, {}, insul_therm_agg) where
    aggregation(insul_therm_agg, {r:resistance})
FE(surf_therm, {}, {}, surf_therm_agg) where
    aggregation(surf_therm_agg, {r:resistance, film:resistance})
```

Constraints maintain the integrity of the resistance value in relation to geometry. The surface constraint is shown below:

```
constraint(surf_therm_cons, f7(r = f(material, thickness)), {surf_therm, surf_geom})
```

The core and insulation constraints would be similar.

Representing design objects

G. T. Nguyen[†] and D. Rieu[§]

[†]INRIA

[§]IMAG

Laboratoire de Génie Informatique
BP 53 X
38041 Grenoble Cedex
France

Abstract Engineering design applications have always been challenging to system designers. Their requirements are complex and there seem to be very few systems able to handle them globally so far. Among the requirements are : the definition and manipulation of composite objects, the management and control of their evolution. A detailed analysis enlightens many other issues : the definition and manipulation of semantic relationships that relate objects together, and the management of multiple representations that are simultaneously required by the designers. A proposal is made to define a data and knowledge representation model dedicated to design applications. It elaborates on object-oriented programming and knowledge representation languages. The emphasis is on the semantic relationships, the evolution and the multiple representations of design objects.

1. INTRODUCTION

Engineering design appears as one of the most challenging application area for the designers of operating systems, data management systems and user interfaces. The reasons are both inherent to that particular field and historical.

They are historical because the amazing evolution of computer hardware and software technologies dramatically increase the obsolescence of existing tools, and severely questions the users' ability to develop adequate tools in time to benefit from a given state of the art technique.

Next they are inherent to the application field for an apparently contradictory reason. For a long time, existing software techniques have been unable to take into account the complexity of the design applications. They manipulate composite objects which are related by complex semantic relationships. Further, engineering design requires that the objects and their relationships be able to evolve over time. Meanwhile, software tools are being developed

with expensive human, computer and financial investments. There is therefore a dramatic discrepancy between the particular evolutions of :

- the hardware and software,
- the available design tools,
- the design data.

It is our opinion that existing software tools, although extensively used, do not meet the needs of current design applications when the available software technology is considered. For example, object evolution is usually taken into account by the notion of version. But the data models used to design the objects do not by themselves support that notion. Stated otherwise, version servers could and in fact are indeed used independently of any data model (Katz, 1986). This is because the data models do not support the evolution of the data. Moreover, the particular semantics of versions is left to the designer. He has to define when to create, delete or merge versions, because there is no automated software available with a clear semantics attached.

New approaches in the field of data modeling , programming languages and database systems appear however (Goldberg, 1983, Stroustrup, 1986, Keens, 1988, Kim, 1990). This paper considers specifically the object-oriented approach. Some of the advantages and shortcomings that are of interest to design applications are detailed (Section 3). The requirements that must be supported for engineering design are briefly sketched in Section 2). Section 3 is a proposal for a data model intended to support the flexibility required by the design process :

- it uses object-oriented concepts,
- it also draws on knowledge representation languages,
- it stresses the dynamic aspects of data definition and values, i.e object evolution,
- it also supports semantic relationships between composite objects,
- it finally provides multiple representations for objects.

All these aspects are integrated in a powerful and flexible model called SHOOD. Experiments are currently underway in mechanical CAD/CAM to test the adequacy of the model and refine its functionalities.

2. DESIGN OBJECTS

Design objects are characterized by both structural and semantic complexities. Structural complexity is often amenable to part/sub-parts hierarchies that model composite objects (Blake, 1987, Kim, 1989). Semantic complexity is however not amenable to semantic relationships. The semantics of the objects is usually disseminated in ad-hoc design rules and constraints verification programs, because the modeling paradigms available do not

usually permit all the semantics to be implemented by a single framework (Rieu, 1986). The emphasis is here on the structural complexity i.e, on composite objects (Section 2.1), on the evolution of the design objects (Section 2.2) and multiple representations (Section 2.3). Semantic complexity e.g, dependency relationships, is addressed in Section 4.

2.1 Composite objects

Composite objects definition is the rationale for design applications (Giacometti, 1990). In contrast with business applications, the incremental nature of the design process involves the integration of existing parts or the design of new components that will eventually form the required objects (Figure 1).

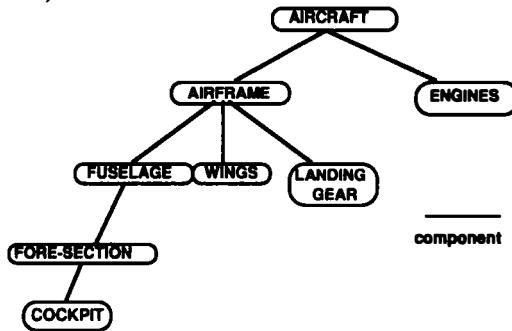


Figure 1. A composite object "aircraft".

New artifacts may involve only specific modifications to previous designs. Reuse and evolution of earlier designs is therefore a common practice.

Designs may also share subparts. This might put a real challenge on the designers because the available paradigms do not usually allow high-level component sharing. The goal here is therefore to define a model able to support :

- part/sub-part aggregates i.e, composite objects,
- sub-part sharing,
- object reuse,
- object evolution,
- semantic relationships between objects and/or sub-parts (dependencies, etc...),
- multiple representations.

Existing modeling paradigms do not support simultaneously all these points, if at all.

As described in Section 3, the object-oriented paradigm provides a sophisticated notion of composite objects. However, "*the basic object-oriented concepts are not sufficient for modeling design objects*" (Kim, 1990). For example, the semantics of composite objects is hard-wired and cannot be altered. There is a confusion between structural aspects e.g, the

sub-part hierarchies, and the semantic links that relate them e.g., the dependencies between sub-parts. This severely limits the ability of these languages to perform design representation adequately.

2.2 Object evolution

Existing designs can be altered to produce new objects or to produce more satisfactory results. Also, new customized components can be designed by trial and error cycles. This implies that they are not stabilized for long. That is, the definition of the design objects is not completed until some constraints and verification or simulation programs have been completed. Meanwhile, they can be inconsistent and only partially complete.

Most database systems do not support adequately unknown and incomplete data. Systems providing integrity checking may in fact prohibit inconsistent data. This is in contradiction with the basic nature of the design process which requires a more sophisticated (and probably tunable) consistency control on the objects (Borning, 1987). In particular, constraint violations should be controlled by the user (Kotz, 1988).

In contrast, knowledge base systems often allow inconsistent data without user control. They may even lack any control on the absence or inconsistency of the data : information may be untyped until it is actually created (Stefik, 1986, Unland ,1990). They are in severe contradiction with usual database systems which are generally too restrictive concerning completeness and consistency issues. A compromise must therefore be found to allow a user-controlled form of data incompleteness and inconsistency in order to cope with the evolution of the design objects. A proposal is made in Section 4 concerning the model SHOOD.

Integrated CAD/CAM systems also handle design variants which may ultimately be chosen for the final design or which may be reworked for further testing and modifications. This is the price to pay for object reuse and evolution. The basic variants must therefore support alterations in their definitions i.e, in their structural and semantic definitions. This has also been recognized in CASE applications (Duc, 1990).

Clearly, the dynamic evolution of the data requires sophisticated mechanisms, alleviating the task of what the software engineering community call "configuration management" (Duc, 1990). This involves the definition, modification, documentation and retrieval of pieces of designs - all varying in time, and all related by semantic links and impacting on each other (Anderson, 1990). A first step towards the assistance to the user in managing object evolution is described in Section 4.2.

2.3 Multiple representations

Multiple representations allow to merge in a unified framework the various requirements emanating from cooperating designers. They are commonly called "perspectives" in knowledge representation languages (Stefik, 86), "views" in the database world (Schek, 1988) and "representations" in the design world (Carre, 1989).

For example, besides the engines and load carrying aspects, commercial aircraft exhibit today complex electric and hydraulic apparatus, pressurization mechanisms, electronic monitoring and navigation equipment, large computerized flight and engine management systems, video displays in the cockpit, etc. Each can be modelled as a particular representation that designers and maintenance staff can work with (Figure 2).

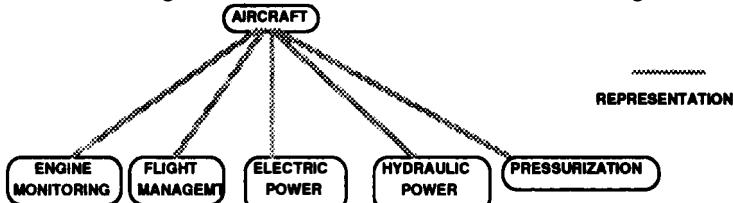


Figure 2. Aircraft representations.

Representations in object-oriented and knowledge representation languages allow the coexistence of several descriptions of the same real world entities. Several questions therefore arise in an object-based approach (Rieu, 1991):

- how are the various representations connected, if at all ?
- how is object identity preserved ?
- are derived values permitted across the various representations ?
- how is consistency preserved between them ?
- how is object evolution taken into account - within and across representations ?

We elaborate here on these characteristics to support simultaneous representations for the objects. Our model is called SHOOD (Escamilla, 1990, Nguyen, 1991). It provides multiple inheritance, relationships, composite objects as detailed in Section 4, and multiple instantiation. It is reflexive and meta-circular, i.e it is defined and implemented in terms of its own concepts (Cointe, 1987).

3. OBJECT-ORIENTED ISSUES

The object-oriented paradigm is often advertised as one of the most sophisticated and flexible approach for a large variety of applications, ranging from software engineering to office automation and database applications (Giacometti, 1990). Various features in object-

oriented languages are discussed in this section to point out the potential advantages and shortcomings that are of interest for design applications. It appears that some basic functionalities are in contradiction with specific requirements of design applications, including semantic aspects and object evolution. Some advantages are detailed first.

3.1 Abstraction

In the following, we shall distinguish between the notion of abstraction, which hides the implementation of the objects to the user, and the notion of encapsulation found in the object-oriented paradigm. Encapsulation provides data abstraction and includes also methods in object definitions to form their interface (Goldberg, 1983). Abstraction is of interest in design application because it allows specific details to be hidden from the user. It allows also different variants to be used successively in composite objects without changing their definitions. For example, designers may be working on components which are not dependent on other sub-parts e.g the airframe and the engines of an aircraft. Abstraction is also a means to implement the notion of "grey boxes" to provide various levels of details to the designers (Blake, 1987).

3.2 Reusability

Reusability is another important feature in object-oriented languages which is of great interest for design applications. This has been recognized in CASE applications also and the notion of "reusable element" is explicitly defined by some advanced software engineering environments (Duc, 1990). Together with the notion of components in sub-part hierarchies, it provides the ability to abstract the objects at various levels of details. It also provides the opportunity to define clean interfaces and thus improve modularity and reusability. Clearly, abstraction also favors the reuse of components since the redesign of existing parts is no longer necessary.

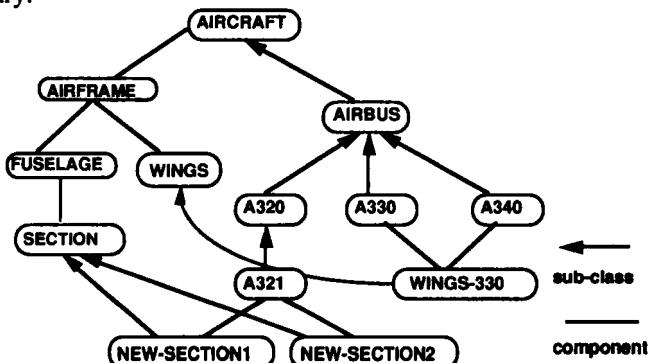


Figure 3. Sharing and adding components.

For example, the upcoming Airbus A330 and A340 share the same wing design. They may refer to the same class Wings-330 - a sub-class of Wings, itself a component of the class Airframe, which is in turn a component of the class Aircraft (Figure 3). Similarly, the A321 is a stretched version of the earlier A320 model fitted with two new fuselage sections fore and aft of the wing (Figure 3). These two sections can be created as new sub-classes of the class Section - itself a component of the class Fuselage. A shorter version of the A320 is being designed and currently known as the A319.

3.3 Specialization

Inheritance is also used in object-oriented models to implement a specialization relationship between objects. Objects can therefore be refined and defined incrementally using more specific constraints and new attributes. This can be straightforwardly adapted to design applications.

A systematic use of built-in inheritance imposes some constraints however, because it is never clear where the specialization process must stop. For example, each particular aircraft delivered is indeed a most specific specialization of some standard model. Within each specific airline, dozens of A320 aircraft may only differ by their serial number. From a strict object-oriented perspective, they should be modeled by instances of a generic A320 model, implemented itself by an aircraft class. This is fundamentally a design decision which is not facilitated by the object-oriented paradigm. It is the result of the dichotomy existing between classes and their instances. It does not exist in paradigms based on prototypes (Stefik, 1986). This is one of its main drawbacks and probably the main challenge that object-oriented fanatics have to face. Some other problems follow.

3.4 Encapsulation

A systematic use of the abstraction and encapsulation mechanisms, although adequate for a clean implementation, can be cumbersome for one single reason : access to the values of attributes requires the definition of many specific read and write operations. This is of course burdening, although it can be generated automatically for atomic data types (integer, strings, etc) (DeMichiel, 1987). For example the "age" of an aircraft can be accessed by sending a message to the Aircraft class, using the selector argument "age", the argument being the object identifier (oid) of the aircraft, and the receiver of the message the Aircraft class.

However calling methods to access attribute values is very time-consuming. Specific mechanisms have been proposed to extract attribute values elsewhere e.g, by a particular "**"

prefix to attribute names (Bancilhon, 1988). This syntactic sugar seems quite unnatural. Record-based management systems do not require such artificial mechanism : invocation of the attribute's name is all what is needed. This is common sense, and do we really want such peculiarities in object-oriented systems ?

Another problem specifically related to encapsulation is the definition of methods within classes. Most programs in engineering design applications are external simulation, verification and 3D display programs. Most of them already exist and need only be invoked where and when appropriate on the argument objects. Mixing code and object definitions is in contradiction with design processes where much code operates on few object definitions. This point is of course irrelevant in CASE applications - where it all comes from - because the objects and the code are used together to produce large software environments (Duc, 1990), whereas CAD/CAM applications only use the code to produce new design objects. In our opinion, this discrepancy is the reason for the first important mismatch between engineering applications and object-oriented concepts. Our conclusion is that programs or methods should be separated from object definitions (Keens, 1988).

3.5 Relationships

Another disadvantage of the object-oriented approach for modeling design objects is to support only generic relationships. They can only be defined between classes of objects, thus reducing the ability to implement specific relationships depending on the values of the instances being related. For example, dependency relationships can be defined in ORION between composite objects and their components (Kim, 1989). But the underlying semantics for this dependency is exclusive and existential : the deletion of the object implies the deletion of all its components. This is of course very restrictive for design applications where, for example, engine deletion on some aircraft produces lots of spare parts that can be used to retrofit other engines. Instances should therefore be allowed to migrate from one class to another - say from the class Engine to the class SpareEngine - prior to the deletion of an aircraft. This in turn implies that the dependency between the aircraft and its engines be changed. Similarly, the semantics of composite objects in LOOPS implies that they can only be created if all their components are simultaneously created (Stefik, 1986). This is in contradiction with the design process where partially unknown objects must be worked out and later assembled together to form larger designs. The conclusion here is that dependency relationships should be decoupled from the part/sub-part graphs.

4. A DESIGN REPRESENTATION MODEL

The model SHOOD presented in this section is based on three principles :

- every information is represented by an object,
- every object is an instance of some other object,
- any information concerning an object is stored as the value of an attribute.

The model is reflexive and meta-circular i.e, it has access to its description and it is implemented using its own concepts (Escamilla, 1990). This approach provides the basis for extensibility. The model is implemented using two layers :

- the knowledge representation layer (KRL),
- the knowledge manipulation layer (KML).

A storage layer will also be provided to support object persistency. This will be achieved by interfacing with an existing object server (Figure 4).

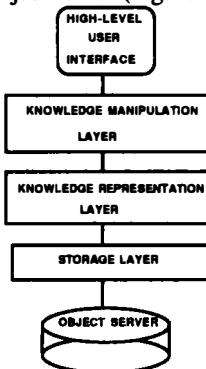


Figure 4. Architecture.

The KRL provides data abstraction, but does not provide encapsulation. Interface with existing programs should therefore be made at this level because object structures are made visible. The KML provides encapsulation and method invocation based on message passing techniques. The interface with design programs can therefore be implemented at both layers, providing the opportunity to access the data from both the usual attribute level of the objects, resembling the record-field access methods, and the encapsulated - method based - layer for sophisticated programming paradigms.

The rationale behind this approach is to provide the designer with the usual attribute-based representation of data which is in our opinion mandatory for CAD/CAM applications. It also provides higher level abstraction mechanisms in the KML, should the applications require casual user query facilities and graphics or other high level design facilities.

4.1 Concepts

In the following, the term object stands for object instance. An object instance belongs to one or more classes that define its structure and behavior.

The KRL includes the fundamental characteristics of frame-based knowledge representation languages, combined with concepts found originally in object-oriented languages. As such, the KRL merges functionalities usually not provided simultaneously in either paradigms.

Among them stand the usual notions of class, meta-class, instance, multiple inheritance and method (Section 4.1.1 and 4.1.2). Outstanding features include a disjunction relationship between classes and semantic relationships which are either dependency relationships between objects (Section 4.1.3.1) or attribute propagation among composite objects (Section 4.1.3.2).

Further, the systematic use of the principles mentioned previously for the design of SHOOD has given rise to a powerful, reflexive and extensible model. Its implementation is based on a kernel of approximately 10 meta-classes which form the bootstrap of the model. The basic concepts are then implemented as instances of these meta-classes (Escamilla, 1990).

The implementation is in Lisp. The extensive use of the kernel and of its meta-classes support a powerful representation language where the methods, attributes, inferences and constraints are themselves classes which are instances of specific meta-classes in the kernel. The invocation of a particular method or inference and the value of an attribute are instances of the classes which model methods, inferences and attributes. Besides its very condensed and easy to develop kernel, this provides the basis for extensions and evolution of the model itself (Section 4.3).

The actual concepts implemented by the model are generated from the bootstrap to provide the user with a basic set of functionalities e.g, set, list and bag manipulations together with the usual notions of class, inheritance, instance and composite objects. An emphasis is put on object evolution and its control by the designers (Section 4.2).

4.1.1 Object classes

Objects are modeled in SHOOD by aggregations of attributes e.g the length, wingspan, range and weight of an aircraft. They model either atomic values (integer, strings, etc) e.g range is 2,000 nautical miles, or complex components e.g the landing gear which includes a nose gear and two main gears (Figure 5). The scope of attributes is the class where they are first defined plus all their sub-classes. A sub-class may refine inherited attributes. It may not redefine them entirely, e.g give them a new domain class. Attribute refinement means for

example more stringent constraints on the attribute. Attributes are defined by descriptors. There are three different descriptors attached to attribute definitions:

- type descriptors,
- constraint descriptors,
- inference descriptors.

Type descriptors define the domain of an attribute i.e the class where it takes on its values. Constraint descriptors define restrictions on the attribute values. Inference descriptors define the different ways an attribute value can be obtained. A partial order between the different inferences is required. Should an inference fail to provide the attribute value, the next inference in the list is activated. The conflict between different candidate inferences is solved by the method selection. This out of the scope of this paper. The user is always a default option i.e, if no other inference can provide the attribute value then the user is requested to give one.



Figure 5. Attributes of the class Aircraft.

The complex components are themselves objects. This implies that components are referred to by identifiers. The identifiers, or oids, are unique within the system. Objects having the same attribute structure are grouped into classes e.g, the class Aircraft.

4.1.2 Inheritance

As provided by frame-based representation and object-oriented languages, the classes are structured in SHOOD along a super-class/sub-class relationship. It implements an inheritance relationship, by which sub-classes inherit all the attributes and methods of their super-classes. The semantics of inheritance in SHOOD is the strict set inclusion : a sub-class groups a set of instances which is a strict subset of the instances in the set represented by each of its super-classes.

SHOOD also features a disjunction relationship between classes. In contrast with the previous one, which is usually provided by object-oriented languages with various semantics, the disjunction relationship is genuinely provided in SHOOD to support generic concepts which are known to be distinct. This accelerates the classification mechanisms. It also inhibits name conflicts and several other features e.g, multiple inheritance and multiple instantiations. For example, aircraft and submarines are different and so far incompatible

vehicles. They can therefore be defined by two sub-classes of the class Vehicle, namely Aircraft and Submarine which are related by a disjunction relationship (Figure 6). A side-effect of the disjunction relationship is that attributes defined in disjoint classes with the same name e.g "age", are considered different.

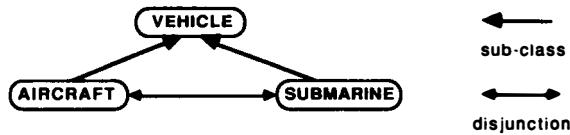


Figure 6. Inheritance and disjunction relationships.

A class may have several sub-classes which inherit its definition e.g, the class Aircraft has the sub-classes Long-Range, Short-Range which have the same attributes and which may refine them e.g the range attribute may only have specific values depending on the type of the aircraft. Subclasses may also add new attributes : Amphibian aircraft must have specific floating equipment (Figure 7).



Figure 7. Sub-class/super-class example.

A class may have several super-classes : multiple inheritance is supported. For example, the class **Amphibian** inherits the attributes of the classes **Aircraft** and **Vessel**. Because the attributes are inherited by all the sub-classes of a given class, name conflicts may appear in the attributes of a class having several super-classes. In SHOOD, such name conflicts are solved by an explicit inheritance of all conflicting attributes : they are prefixed by the name of the class which define them. For example in Figure 8, the class **Amphibian** inherits both the attribute "engine" of the class **Aircraft** and of the class **Vessel**. But it happens that naval engines have little to do with airborne engines, as well as the corresponding propellers. They are both inherited and defined in the class **Amphibian** with their full name i.e their class name dot attribute name : "Aircraft.engine" and "Vessel.engine" respectively.

The semantics of inheritance is a strict set inclusion. This excludes exceptional instances i.e., objects which do not conform to the class definitions. This also implies that a class which has several super-classes models the intersection of the sets defined by the super-classes : an amphibian aircraft is both an aircraft and - a limited form of - a vessel.

4.1.3 Name conflicts

Due to the potential name conflicts involving inherited attributes, SHOOD avoids the duplication of attributes when they represent the same concept. For example the classes

Aircraft and Vessel depicted in Figure 14 have both an attribute "length" with the same semantics. The designer can define in a common super-class e.g Vehicle a so-called "deferred attribute". It defines a virtual attribute which cannot be instantiated in the class where it is defined, here Vehicle. It simply states that all the sub-classes from there down the inheritance graph will define this particular attribute "length" with the same semantics attached - up to disjunction of classes. The "length" attribute therefore need not be prefixed by any class name. It is implicitly here the Vehicle class where it appeared as deferred (Figure 8). It bears the same semantics wherever it appears in the sub-classes of Vehicle : the classes Aircraft, Vessel and Amphibian. In contrast, the attributes "engine" are different. They are inherited with their full name : "Aircraft.engine" and "Vessel.engine".

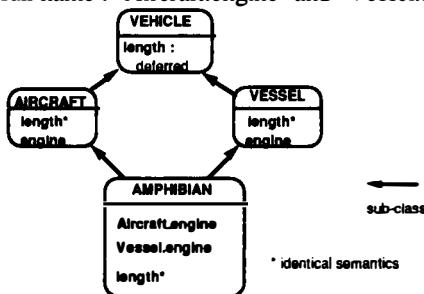


Figure 8. Full names for inherited attributes and deferred attributes.

4.1.4 Multiple instantiation

Multiple instantiation allows instances to belong simultaneously to different classes. This departs from the usual notion of instance in object-oriented approaches (Goldberg, 1983). It bears several advantages. First, it allows hollow classes to be avoided i.e classes with very few instances. This is a common problem in object-oriented modeling. Second it supports different simultaneous representations on the objects (Rieu, 1991). Instances related to a particular class are seen by the user with the corresponding definition, and they can be seen simultaneously by other users as instances of other classes. We believe that it is easier to manage the "instance_of" relationship between object instances and their classes than to define new sub-classes modeling various points of view involving multiple inheritance, thus complicating the inheritance graph.

4.1.5 Semantic relationships

Semantic relationships are the most powerful and original feature in SHOOD. They elaborate on the relationships found in SRL where the user can define application dependent links, with their particular operations and attributes (Fox, 1986). Inheritance and disjunction are treated in SHOOD as particular cases of relationships. There is therefore no distinction

between system defined and user-defined relationships in SHOOD. Two kinds of user-defined relationships are provided : dependency relationships and attribute propagation.

4.1.5.1 Object dependencies

The dependency relationships are complementary to composite objects. They define the semantics of the part-component relationships which would otherwise be mere structural aggregations of sub-parts. Recall that it is the case of all frame-based languages and that object-oriented languages impose an implicit and generic semantics to the composite objects. Semantic relationships are there mixed with the structural definitions of the composite objects. This appears inadequate in many applications. As noted elsewhere (Diaz, 1990), various relationships must be available to the user. One of the most flexible system which offers user definable relationships is SRL. The goal in SHOOD is to provide a more limited although powerful set of links that the designers can use to model usual inter-object dependencies (Escamilla, 1990). There are four different kinds of such links : the exclusive, shared, existential and specific dependencies.

The most simple and stringent form of dependency is the exclusive dependency. It allows a part to be the single owner of a component. In SHOOD, it allows an instance of the owner class to hold a privilege over an instance of the component class. The exclusive dependency can be defined between the corresponding classes i.e, in generic terms. It will then apply to all their instances. For example, an instance of the class Aircraft owns exclusively the instances of the class Engine which are fitted on its airframe. It should be noted however that no existential dependency is associated here. This is in contrast with most proposals in the literature (Kim, 1989, Stefik, 1986). It is our opinion that the notion of existence should not be mixed with other notions of dependency as is usually the case : an aircraft may be equipped with a particular engine. The engine may be overhauled and later fitted on another aircraft. Or the previous owner aircraft may be scrapped and its engines reused on another. This is how it really happens. So there seem to be no reason to limit the flexibility of the dependencies by inappropriate confusion.

The notion of dependency is not necessarily linked to the notion of composite object either. A dependency may exist between two classes which are not linked by any component relationship. An aircraft belongs to a specific company, but it is not necessarily defined as a component of the company.

The second form of dependency supported in SHOOD is the shared dependency. It is used to define a link between various parts and a particular object. Various airline carriers may for example share a particular hub in an airport. A particular side-effect is that the shared object cannot be deleted as long as any other object holds a dependency on it. This is a particular form of implicit existential dependency.

The existential dependency is the most commonly used and provided in object-oriented languages. The main problem however is that it is always mixed with other kinds of relationships e.g composite objects. This confuses the design and the designers. In SHOOD, it can be used together with other relationships but it has to be explicitly defined by the user. As noted before, the existence of aircraft engines do not depend on the existence of aircraft, and vice-versa. But the existence of an aircraft is dependent on the existence of its airframe. Conversely, the airframe exists prior to the aircraft which is built around it. These differences have to be taken into account and are expressible in SHOOD. They were not in previous languages and representation models.

A degenerated form of exception is supported concerning the dependencies. It is the specific dependency, which outrules the generic dependencies defined by classes. It states that between any particular instances of objects exist a dependency that does not follow the same pattern as the generic form. For example, an exclusive existential dependency may exist between the instances of helicopters - which can be otherwise defined as particular aircraft - and their rotor. These cannot be serviced and must be replaced periodically, as specified by the manufacturer. There is therefore a specific existential dependency between each particular rotor blade and the helicopter using it.

4.1.5.2 Attribute propagation

Attribute propagation is a feature allowing values from component attributes to be visible from various levels in the part hierarchy of composite objects. It is sometimes called horizontal or selective inheritance (Carre, 1989). It is emphasized here that inheritance is a generic relationship in object-oriented languages, relating classes in the inheritance graph. It implements the super-class/sub-class relationship between classes. In contrast, the propagation of attribute values among objects is a instance specific relationships. It broadcasts attribute values among sub-parts of composite objects. It is therefore a relationships between instances, hence the "horizontal" qualifier, to differentiate from the "vertical" and generic inheritance relationship which is orthogonal.

Assuming that an airframe has an attribute "age", and that an aircraft is composed of an airframe and engines, the age of the aircraft may be inferred from its airframe's age. Due the fact that aircraft engines have to be overhauled periodically - typically every 2,000 hours - and that an airframe can be used 20 years or more, it is clear that selective propagation must be defined between components.

In the example, the aircraft age cannot be inferred from the engines' age. The designer must therefore propagate the airframe's "age" to the aircraft, excluding the "age" of the engines (Figure 9). Note that with attribute propagation, the class Aircraft need not have an attribute "age" defined. Again, this is an instance specific relationship, and the propagation must be

done for each particular aircraft. From the designers point of view, the "age" of an aircraft will appear as if it had been defined for the class Aircraft.

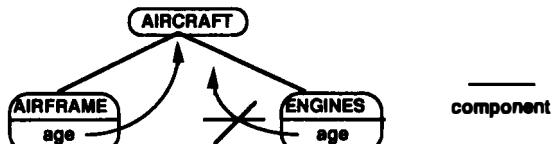


Figure 9. Attribute propagation.

4.2 Modeling design objects

The modeling paradigm offered by SHOOD is threefold. It gives the ability to represent objects using three combined perspectives (Figure 10) :

- a semantic perspective, implementing the semantic relationships between object components,
- an structural perspective where objects can be altered, refined and reuse existing definitions, implementing the structural definitions of composite objects,
- a variant perspective where the evolution of the objects is modeled. It copes with the structural, semantic and content evolution of the objects using object versions.

While the inheritance perspective does not introduce new concepts with respect to existing object-oriented and frame-based languages, it is treated like other relationship in SHOOD. The semantic perspective includes the notions of dependency relationships and attribute propagation. The semantic perspective includes all semantic relationships because the composition relationship is decoupled from the notion of dependency between components. The variant perspective includes the notion of object evolution using the concepts of design improvement and degradation. It automatically generates versions of the successive designs produced.

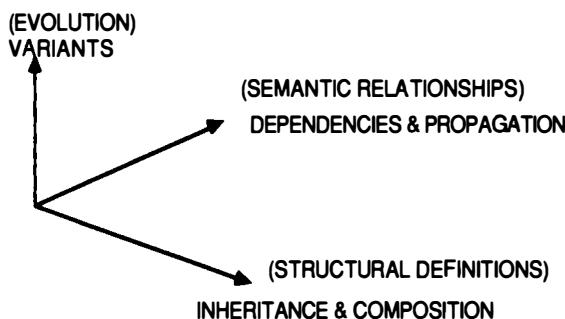


Figure 10. The three modeling perspectives in SHOOD.

Managing the various changes in the design objects, classifying them in the appropriate variants, finding the adequate variants to perform a specific design task and documenting them is called "configuration management" in CASE applications. While the model SHOOD does not provide extensive functionalities dedicated to this problem, some interesting features are available to handle the evolving nature of the designs. They are described in the following sections.

4.2.1 Evolving definitions

Evolving structures may result from two main causes. The first one is the implementation of inappropriate class definitions. This is out of the scope of this paper. It is the "object-oriented design" problem. The second reason is reusability. It is of first importance here because it may result in the reuse of previous definitions for the design of new objects.

Evolving structures concern two different aspects. The first one concerns the operations available on the class definitions e.g adding an attribute. The second is the impact of these modifications on the corresponding instances. While the first aspect requires only appropriate code to implement the modifications of the class graph and is usually provided in some limited form by existing languages, the second requires carefull attention and specific mechanisms. It is usually very limited and unflexible (Rieu, 1988).

The operations concerning class modifications can be classified in two categories : those modifying a single class e.g delete an attribute from a class, and those involving several classes e.g make a class a component or a subclass of another. Operations in SHOOD allow the modification of any part of an inheritance and composition graphs. Among the operations detailed elsewhere (Nguyen, 1989) are the addition and deletion of super-class/sub-class and part/sub-part relationships.

Most important is the impact of the operations on the existing instances. A designer may want to leave the existing instances unchanged, and thus provoke the creation of a new version of the class. He may also wish to test the effect of the modifications performed on a class definition to all or part of the instances. A original feature is provided by SHOOD to support this form of evolution. It is based on the notion of relevant structure which corresponds to the potential combinations of attributes that can be generated from a class definition. Any modification performed on the original class definitions will be traced and the instances - whether incomplete and inconsistent - migrate and are attached to exactly one relevant structure. This allows the instances to be modified automatically according to the modifications performed on their class.

4.2.2 Evolving instances

Instance evolution is the most common aspect in data-intensive applications. In object-oriented paradigms, modification of attribute values may force instances to migrate. Available languages are usually very limited concerning this aspect because it requires a classification mechanism. In SHOOD, modification of the instances values triggers their propagation in the class graph.

The qualifying classes are all the super-classes of its current class, plus all their subclasses of which the modified instance meets the constraints. This is called instance propagation. It uses multiple instantiation, by which an instance may implicitly or explicitly belong simultaneously to several classes. This can be used as an assistance to the designer who may occasionally wonder to which classes the object he is working on resembles most.

The ultimate goal for all design process is to define objects that meet a given set of specifications, should this concern a space shuttle or a bicycle. The design evolves incrementally in stepwise manners but not linearly. Various alternatives may be explored in parallel. Older designs may also be deleted or reused and modified. Today, the control of this task is entirely left to the designers. They have to define with the appropriate tools and expertise what defines good or better designs : greater speed, smaller space, lighter weight, etc.

An assistance is given by SHOOD concerning the evolution of the artifacts towards more complete and more consistent designs, with respect to the design specifications. Using the notion of relevant structure mentioned above (Section 4.2.1) the system generates automatically different versions of an object when its completeness or consistency improves. This is called an improvement cycle. Whenever the completeness and the consistency of the object does not improve, a degradation cycle is automatically created and hooked on the current improvement cycle. It can be deleted whenever the object reaches it back again after the appropriate modifications.

5. CONCLUSION

The inadequacy of object-oriented languages to support flexible semantic relationships between objects leads to severe restrictions concerning their usability by design applications. The model SHOOD is a compromise between object-oriented modeling paradigms and knowledge representation languages found in Artificial Intelligence. Its goal is to implement a unified model to support :

- composite objects,
- object reuse,

- object evolution,
- semantic relationships (dependencies, attribute propagation, etc),
- multiple representations.

It includes features usually not provided simultaneously by existing knowledge representation and object-oriented languages e.g. meta-classes, disjunction, composite objects, multiple representations and semantic relationships. Other features include :

- the ability to support evolving object structures,
- incomplete and inconsistent objects.

The model overcomes current limitations in object-oriented languages e.g. generic inheritance or composition hierarchies with hard-wired semantics. It also overcomes limitations found in existing knowledge representation languages by supporting object evolution. Experiments are underway in mechanical CAD/CAM for the project CIM-ONE, in cooperation with INSA, Ecole Centrale and Université Claude Bernard in Lyon (France).

Acknowledgements

The authors wish to thank Annie CULET, Chabane DJERABA and José ESCAMILA for their contributions to the design and implementation of SHOOD. They also thank Profs. B. DAVID, C. MARTY and D. VANDORPE for many invaluable discussions. This work is supported by INRIA and IMAG for the project SHERPA and by Région Rhône-Alpes for the project CIM-ONE.

REFERENCES

- Anderson B., Gossain S. (1990). *Hierarchy evolution and the software lifecycle*. Proc. 2nd Intl. Conf. Technology of Object-oriented Languages & Syst. Paris (F).
- Bancilhon F. (1988). *Object-oriented database systems*. Proc. 7th ACM Symposium on principles of database systems. Austin (Texas).
- Blake E., Cook S. (1987). *On including part hierarchies in object-oriented languages, with an implementation in Smalltalk*. Proc. ECOOP '87. Paris (F).
- Borning A. & al. (1987). *Constraint hierarchies*. Proc. ECOOP '87. Paris (F).
- Carre B. (1989). *Méthodologie orientée objet pour la représentation des connaissances*. PhD Thesis. Université de Lille Flandres-Artois (F).
- Cointe P. (1987). *Metaclasses are first class : the ObjVlisp model*. Proc. OOPSLA '87 Conf. Orlando (Fla).
- De Michiel L., Gabriel R.P (1987). *The Common Lisp Object System : an overview*. Proc. ECOOP '87. Paris (F).
- Diaz O. & Gray P.M. (1990). *Semantic-rich user-defined relationship as a main constructor in object-oriented databases*. Proc. IFIP Conf. on Database Semantics. North-Holland.

- P. Duc. (1990). *ROSE/ADELE data models comparison*. Eureka Software Factory.
- Escamilla J., Jean P. (1990). *Relationships in an object knowledge representation model*. Proc. 2nd Intl. Conf. Tools for Artificial Intelligence. Washington D.C (USA).
- Fox M.S et al. (1986). *Experiences with SRL : an analysis of a frame-based knowledge representations*. Proc. 1st Int. Workshop on Expert Database Systems. Kiawah Island (South Carolina).
- Giacometti F., Chang T.C (1990). *Object-oriented design for modelling parts, assemblies and tolerances*. Proc. 2nd Intl. Conf. Technology of Object-oriented Languages & Syst. Paris (F).
- Goldberg A., Robson D. (1983). *Smalltalk 80 : the language and its implementation*. Addison-Wesley Publ. Co.
- Katz R. & al. (1986). *Version modeling concepts for computer-aided design databases*. Proc. ACM SIGMOD Conf.
- Keens S.E (1988). *Object-oriented programming in Common Lisp. A programmer's guide to CLOS*. Addison-Wesley.
- Kim W. & al. (1989). *Composite objects revisited*. Proc. ACM SIGMOD Conf.
- Kim W. (1990). *Object-oriented databases : definition and research directions*. IEEE Trans. on Knowledge and Data Engineering. Vol. 2, n° 3.
- Kotz H. (1988). *Supporting semantic rules by a generalized event/trigger mechanism*. Proc. Intl. Conf. Extending Database Technology. Venice (I).
- Nguyen G.T, Rieu D. (1989). *Schema evolution in object-oriented database systems*. Data & Knowledge Engineering, North-Holland. Vol. 4, n° 1.
- Nguyen G.T, Rieu D. (1991). *Database issues in Object-Oriented Design*. Proc. 4th Intl. Conf. Technology of Object-oriented Languages & Syst. TOOLS '91. Paris (F).
- Rieu D., Nguyen G.T. (1986). *Semantics of CAD Objects for Generalized Databases*. Proc. 23rd Design Automation Conference, Las Vegas (USA).
- Rieu D., Nguyen G.T (1988). *Dynamic schemas for engineering databases*. 4th Intl Conf. on Systems Research, Informatics and Cybernetics. Baden-Baden (FRG).
- Rieu D., Nguyen G.T (1991). *Multiple instantiation for object representations*. Submitted for publication.
- Schek H. (1988). *Nested relations, a step forward or backward?* IEEE Data Engineering. Vol. 11, n° 3.
- Stefik M., Bobrow D.G (1986). *Object-oriented programming: themes and variations*. The AI magazine.
- Stroustrup B. (1986). *The C++ programming language*. Addison-Wesley.
- Unland R., Schlageter G. (1990) *Object-oriented database systems : concepts and perspectives*. Lecture Notes in Computer Science. Springer-Verlag.

Protein modelling: a design application of an object-oriented database

G. J. L. Kemp

Departments of Computing Science and
Molecular and Cell Biology
University of Aberdeen
Aberdeen AB9 2UB Scotland

Abstract. Protein modelling is a design activity in which protein chemists generate hypothetical arrangements of molecular structures which satisfy geometric and chemical constraints. Models of proteins can be built from molecular fragments taken from an object-oriented database. The data generated in the process of constructing a protein model is itself stored in the database and can be accessed using the logic programming language Prolog, or the query language Daplex. Representing different versions as objects provides a suitable version management mechanism for this application. Extensions to Daplex permit new action methods to be defined.

INTRODUCTION

Knowledge of the three-dimensional structure of a protein is vital to a full understanding of its function. Experimental determination of a protein's structure is a slow and difficult process, so there is a demand to be able to generate hypothetical computer models of proteins. An approach to this is to construct models based on known related structures by assembling molecular fragments taken from known proteins. These models can be tested by experiments, and refined in the light of experimental results.

When modelling proteins we are performing an assembly task. Assembling molecular fragments into protein structures is similar, for example, to assembling parts of a car. As with conventional design applications, a protein modelling system has to permit alternative versions of models to be generated.

We are trying to produce a tool to assist protein chemists in constructing models of proteins. A modelling system has been built as an application of the object-oriented database system (OODBS), P/FDM, which is itself an extension of the functional data model (Shipman, 1981). P/FDM is being used to store approximately 40Mb of

protein structure data (Gray *et al.*, 1990) from the BIPED project (Islam and Sternberg, 1989). The database can be accessed using both the logic programming language Prolog (Clocksin and Mellish, 1984) and the query language Daplex (Shipman, 1981). Complicated search strategies are possible, which combine data retrieval with calculation, including calls to arithmetic-intensive routines written in C and FORTRAN. The architecture of P/FDM is discussed by Gray (1989).

This paper begins with a brief explanation of protein structure and what is meant by homology modelling. This is followed by a discussion of how an OODBS can be used to support a protein modelling system, with particular attention to the organisation of modelling data and how version objects provide a suitable version management mechanism. Then we describe extensions to the Daplex language which make it easier to access and manipulate protein models under construction. Finally some approaches used in refining "first-approximations" to molecular structures are described.

Protein structure

Proteins are molecules which are found in all living organisms and perform a variety of biochemical functions. Each protein contains at least one polypeptide chain, which is made up of hundreds of amino-acid residues. Twenty different kinds of amino-acid residue are found in proteins. All have a common backbone configuration of atoms, but each has a different side chain attached to the backbone at an atom called the alpha-carbon ($C\alpha$). The twenty different side chains differ in size, charge, and various other properties. The order in which the side chains occur in the sequence determines how the protein chain will fold in three dimensions, and it is the resulting three dimensional shape which gives a protein its particular biochemical activity.

Protein structure can be viewed at different levels. The *primary structure* is the sequence of amino-acid residues in a protein chain, and is often represented as a string of characters with different letters standing for different residue kinds.

Sections of protein chains often fold into regular shapes which are stabilised by hydrogen bonds between backbone atoms. Such a structure is a helix. Occasionally, extended sections of chains lie against each other to form sheets. These extended sections are called strands. Other sections of protein chain which are neither helices nor strands are referred to as loops or random coils. The *secondary structure* of a protein is its organisation in terms of a sequence of loops, strands and helices.

The *tertiary structure* of a protein is a description of its conformation in terms of the coordinates of its individual atoms.

Our database contains information on primary, secondary and tertiary structures of over 80 proteins. Additional information stored includes the positions of hydrogen bonds, salt bridges and disulphide bridges which are important in stabilising protein structures. The schema diagram for the protein structure database is shown in Figure 1.

Homology modelling

If a protein's sequence is unlike any other then it is necessary to use biochemical and biophysical evidence from a variety of sources in attempting to construct a model of

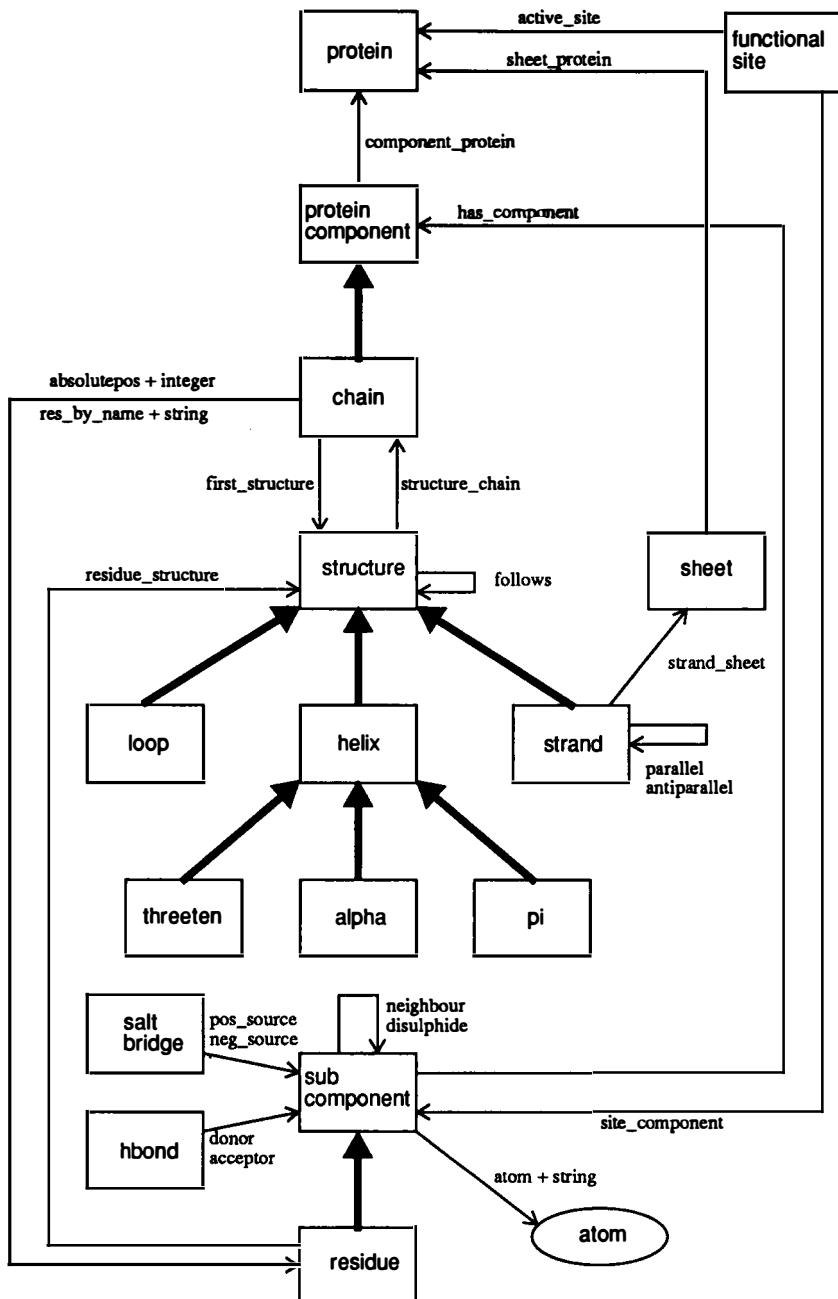


Figure 1. Protein structure database schema diagram. Object classes are represented by rectangular boxes. Labelled arrows represent relationships between objects. Thick arrows point from subclasses to their superclasses.



Figure 2. A $\text{C}\alpha$ -trace of the crystallographically determined structure of γ -chymotrypsin is shown by the thick line. The thin lines show where the backbone of the model of the serine protease domain of C1s is expected to adopt a different conformation from the known structure of γ -chymotrypsin.

its structure (Clark *et al.*, 1990). However, when there is over 30% sequence identity with a known protein, then a technique known as *homology modelling* can be used.

Homology modelling is the process of constructing a model of a protein based on predicted similarity to a protein with a known structure. The underlying assumption in homology modelling is that if two proteins have related primary structures, then they will have similar tertiary structures. Residues common to both sequences are usually expected to occupy the same positions in the model and the known structure. Also, regularly repeating conformations of protein backbone such as helices are often found to be conserved over a family of related proteins. Differences in two protein chains with closely related sequences mostly occur in surface loops.

As an example of homology modelling, the thick line in Figure 2 shows a $\text{C}\alpha$ -trace of the crystallographically determined structure of γ -chymotrypsin (Cohen *et al.*, 1981). A model of a related molecule (the serine protease domain of C1s) was constructed based on the known structure of γ -chymotrypsin. The thin lines show where the backbone of the model is expected to adopt a different conformation from the known structure.

The first step in constructing a model based on homology to some known structure is to align the sequences of the two proteins. From the sequence alignment, it is possible

to predict which parts of the molecule are most likely to be conserved in the model. Regions of high sequence identity are likely to be structurally conserved, as are regions of regular secondary structure in the known protein.

Homology modelling can be performed interactively using molecular graphics systems, which typically offer facilities for substituting one side chain for another (mutation), and rotating around covalent bonds (changing torsion angles). Conserved regions can be identified and mutations performed in these, where required. However, variable regions corresponding to insertions and deletions prove much more difficult to model interactively, requiring coherent rotations about backbone bonds to close up the chain after a deletion, or to create room for an insertion.

We use a fragment-fitting approach to remodelling variable loops following Jones and Thirup (1986). In this approach, variable regions are modelled by substituting parts of the chain of the starting structure by fragments taken from other known structures. The criteria used in selecting fragments to be incorporated in the model are that the fragments must have the correct number of residues and that the positions of C α atoms at the ends of fragments should match those of anchoring C α atoms in the known structure. A fragment-fitting approach is also employed in the automatic modelling system COMPOSER (Blundell *et al.*, 1988) and in work by Claessens *et al.* (1989). In their work, tables of inter-C α distances are precalculated, and these alone are used to identify fragments. In contrast, we search for fragments in a database containing additional structural information organised in a way which provides efficient access and selection of fragments. Further calculations and tests can be performed on these fragments when making a selection, referring back to stored structural data. Therefore, it is not necessary to inspect visually many of the fragments selected according to distance criteria alone which are unsuitable for other reasons.

Homology modelling with an OODBS

Since a modelling exercise is typically continued over a number of sessions, perhaps by different people, it is important that working data generated and *ad hoc* functions introduced during one session can be retained for use in subsequent sessions. This persistence can be achieved by extending the database to include objects representing modelling concepts and molecular fragments. Therefore, in this application the database is used not just for relatively static protein structure data, but also for storage of dynamic working data.

The implementation language for P/FDM is Prolog and originally all methods were written in Prolog (with calls to C). However, there is a requirement for a high level language to enable protein chemists to play as active a role in the modelling process as they wish. Users should not have to be experienced Prolog programmers in order to modify the modelling strategy. Therefore, the system must provide a way for routines expressing users' preferred criteria and personal strategies to be composed easily. This is achieved by using the Daplex language of P/FDM. Daplex queries can also be optimised (Paton and Gray, 1990) which gives similar advantages to the use of SQL with a relational database. For example, users can express queries in the most natural way without having to consider how the query will be executed, and the optimiser will reorder the query if a more efficient form is found. We have also extended

Daplex to include *actions* (see later).

REPRESENTING WORKING DATA IN AN OODBS

The basic concepts in the P/FDM database are *entities* and *functions*. Entities are used to model conceptual objects (corresponding to “nouns”). Functions represent the properties of an entity, which can be either scalar attributes of entities or relationships between entities. With this model there is a similarity between a database schema and a semantic net (Gray, 1984) — entity classes correspond to nodes and relationship functions correspond to arcs in a semantic net. Functions may be single or multi-valued. Multi-valued functions enable us to represent sets of related objects.

The schema for this working data is shown in Figure 3. Most of this schema is shown in the diagram in Figure 4. Entity classes are represented by rectangles and relationship functions are represented by the arrows. For example, *protein_model* and *chain* are entity classes, and *based_on_chain* is a relationship. Each protein model is based on a chain from a protein in the main protein structure database and is uniquely identified by its name, which is used as a key. Keys are an important feature of our database (Paton and Gray, 1988) since they provide fast access to data, facilitate loading large quantities of data from files and provide useful integrity constraints.

Functions and relationships can represent the semantics of the modelling activity and can represent both molecular fragments and design concepts. The database provides persistence by entity (Gray and Kemp, 1990). That is, we store the state of an entity, and not the state of arbitrarily complex computations, as in persistent systems such as PS-Algol (Atkinson *et al.*, 1983).

The first step in generating a protein model based on the known structure of a molecule with a similar sequence is to align the two sequences, introducing gaps where appropriate into either sequence in order to improve the alignment. A fictional sequence alignment is shown in Figure 5. The position of a gap in the sequence of the known protein corresponds to a position in the chain of the known structure where an insertion of one or more residues will have to be made when constructing the model, and a gap into the new sequence indicates where a deletion will have to be made. We are interested in the residue names and numbers at each alignment position in both the old and new sequences. Since both sequences may contain gaps, a third number, *position* is introduced which uniquely identifies an alignment position within a protein model. We model the concept of an alignment position as an object class in our database and store residue names and numbers as scalar attributes of this class. Prolog rules are used to predict whether particular alignment positions are expected to be structurally conserved, using information in the alignment about sequence identity at particular positions and structural information retrieved from the database.

Consider a surface loop containing an insertion. This variable region can be defined by the range of alignment positions which it spans and can be represented within our database as a *backbone_range* object. A backbone range is defined by the alignment positions at which it starts and ends. In fact the entire model can be partitioned into insertions, deletions and other ranges which are conserved in length. This division can be done interactively by the user, or automatically by invoking Prolog rules which additionally create the necessary *backbone_range* objects if these do not already exist.

```

create private module protmod

declare modeller ->> entity
declare user_name(modeller) -> string
key.of modeller is user.name

declare protein.model ->> entity
declare model.name(protein.model) -> string
declare based_on_chain(protein.model) -> chain
declare chief(protein.model) -> modeller
declare old_sequence(protein.model) -> string
declare new_sequence(protein.model) -> string
declare nuro_alignment_positions(protein.model) ->
    integer
key.of protein.model is model.name

declare alignment_position ->> entity
declare in_model(alignment_position) -> protein.model
declare position(alignment_position) -> integer
declare old_name(alignment_position) -> string
declare new_name(alignment_position) -> string
declare old_number(alignment_position) -> integer
declare new_number(alignment_position) -> integer
key.of alignment_position is key.of(in_model), position

declare point ->> value.entity
declare x(point) -> float
declare y(point) -> float
declare z(point) -> float

declare rotation ->> value.entity
declare r1c1(rotation) -> float
declare r1c2(rotation) -> float
declare r1c3(rotation) -> float
declare r2c1(rotation) -> float
declare r2c2(rotation) -> float
declare r2c3(rotation) -> float
declare r3c1(rotation) -> float
declare r3c2(rotation) -> float
declare r3c3(rotation) -> float

declare backbone.range ->> entity
declare start_position(backbone.range) ->
    alignment_position
declare end_position(backbone.range) ->
    alignment_position
declare range_type(backbone.range) -> string
declare frame_centre(backbone.range) -> point
key.of backbone.range is key.of(start_position),
    key.of(end_position)

declare candidate_atom ->> value.entity
declare x(candidate_atom) -> float
declare y(candidate_atom) -> float
declare z(candidate_atom) -> float
declare radius(candidate_atom) -> float

declare candidate_fragment ->> entity
declare for_range(candidate_fragment) -> backbone.range
declare from_residue(candidate_fragment) -> residue
declare rms_error(candidate.fragment) -> float
declare main_chain_atom(candidate.fragment,
    integer, string) -> candidate_atom
declare frame_centre(candidate.fragment) -> point
declare rotation(candidate.fragment) -> rotation
key.of candidate.fragment is key.of(for_range),
    key.of(from_residue)

declare candidate.side_chain ->> entity
declare side_chain_number(candidate.side_chain) ->
    integer
declare for_candidate_fragment(candidate.side_chain) ->
    candidate.fragment
declare for_position(candidate.side_chain) ->
    alignment_position
declare side_chain_atom(candidate.side_chain, string) ->
    candidate_atom
key.of candidate.side_chain is
key.of(for_candidate_fragment),
    key.of(for_position), side_chain_number

declare version ->> entity
declare version_name(version) -> string
declare version_of(version) -> protein.model
declare workers(version) ->> modeller
declare selected_ranges(version) ->> backbone.range
declare selected_fragments(version) ->>
    candidate.fragment
declare selected_side_chains(version) ->>
    candidate.side_chain
key.of version is key.of(version_of), version.name

declare comment ->> entity
declare comment_js(comment) -> string
declare made_by(comment) -> modeller
declare on_protein_model(comment) ->> protein.model
declare on_position(comment) ->> alignment_position
declare on_range(comment) ->> backbone.range
declare on_fragment(comment) ->> candidate.fragment
declare on_side_chain(comment) ->>
    candidate.side_chain
key.of comment is key.of(made_by), comment_js;

```

Figure 3. Daplex declaration of working data.

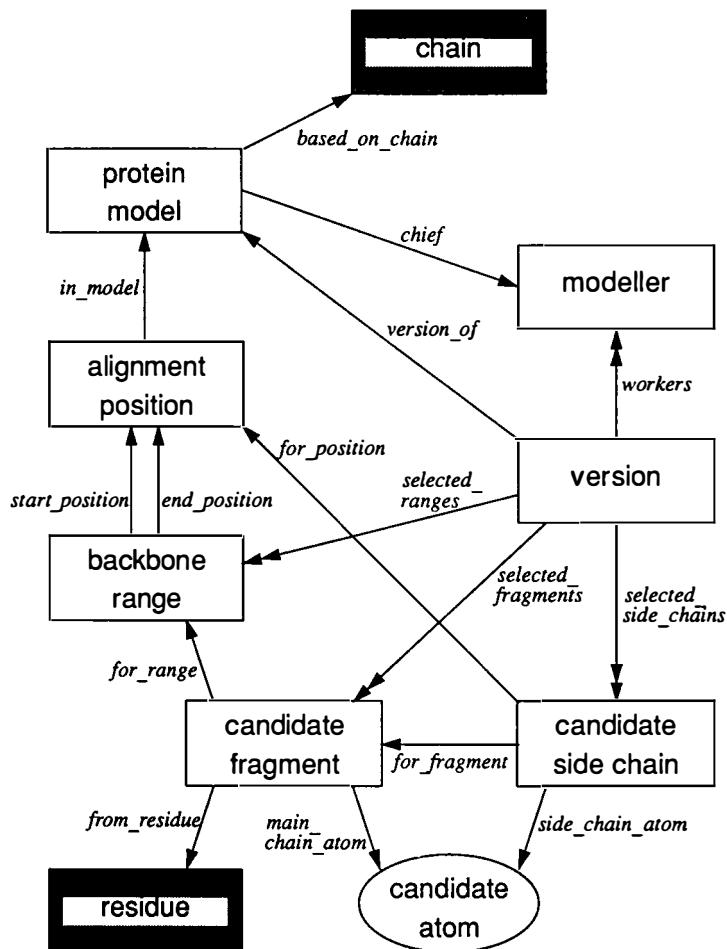


Figure 4. Schema diagram showing working data objects. The shaded rectangles represent entity classes in the main protein structure database (see Figure 1).

Sequence 1: AGYNWTRPEHEAAGY--VVLLTTWGPSAKCMQEYQ-DGNKKDACAPGD

|| ||||||| |||| |||| || | | || |||| |

Sequence 2: AGP--SRPEHEAAATSGLVLLSTWGGH-LCMNEWQGALPLKDSCAPAD

Figure 5. An example of sequence alignment. Each letter represents an amino-acid residue. Vertical bars show identities between the two sequences. Hyphens represent gaps introduced to the the sequences to improve the alignment.

Range_type can be one of ‘insertion’, ‘deletion’ or ‘conserved’.

For insertions and deletions, the database of known structures will be searched for polypeptide fragments which satisfy some set of criteria. Our protein structure database contains coordinate information for a number of experimentally determined protein structures. Since Prolog allows calculation and data retrieval to be combined, our system can support the kind of fragment searches proposed by Jones and Thirup. A Prolog predicate searches the OODBS for fragments of specified lengths which match some inter-C α distance requirements, calling out to fitting code written in FORTRAN (McLachlan, 1979) to calculate the RMS fit for fragments satisfying the distance constraints. Searching the database for replacement loops for remodelling variable regions typically identifies a number of candidate fragments for each variable region. These are represented by *candidate_fragment* objects which are related to the corresponding backbone range by the function *for_range*. Candidate fragments can be thought of as a *hit-list*, that is they are the results of a database search which are stored in the database as objects for subsequent processing. A candidate fragment for a ‘conserved’ backbone range will be taken from the chain on which the protein model is based. The value of *from_residue* is the residue object in the native structure, stored in the main database, from which a particular fragment starts.

Fragment selection

The next step in constructing a protein model is to select from the candidate fragments identified for a particular backbone range the one which should be included in the model. Users are able to specify their own criteria for high level strategic modelling decisions such as this, for example, by introducing a derived function *best_fragment* which given a *backbone_range* identifies one of the *candidate_fragments* for that range which is most suitable according to some criteria. Alternative criteria can be used by redefining this function. Selection criteria of arbitrary complexity can be expressed in Prolog, however users can express simple selection criteria in Daplex.

One possible selection criterion is to use the fragment which has the smallest value for *rms_error* of all the candidate fragments identified for that range. This value can be derived using the following Daplex function :-

```
define smallest_rms_error(r in backbone_range) -> float in protmod
    minimum(rms_error(for_range_inv(r)));
```

Daplex code is optimised and translated to Prolog for subsequent execution :-

```
smallest_rms_error([backbone_range],get,[A],B) :-
    findall(C,
        (getfnval(for_range_inv,[A],D),
         getfnval(rms_error,[D],C)),
        E),
    minimum(E,B).
```

The resulting Prolog predicate has four parameters — a list of the function’s input parameter types; a keyword indicating whether the method is for retrieving, adding, deleting or updating the function value; a list of the function’s parameters; a variable for

returning the function's result. The Prolog goal *getfnval* ("get-function-value") invokes the function given by the first parameter with the arguments given by the second, returning the result in the variable which is the third. The function *for_range_inv* is the automatically declared inverse of the function *for_range*. Since all inverse functions are multi-valued in our system, alternative values for these functions are found by backtracking. In this example, the rms error is retrieved for each fragment identified for the given range. These values are gathered in the list, *E*. The smallest value in this list is then found and returned as the value of the function. The Prolog code generated for this function definition will be stored on disc in the database module called *protmod*.

The function *best_fragment* can then be defined as follows :-

```
define best_fragment(r in backbone_range) -> candidate_fragment
    in protmod
    the c in for_range_inv(r) such that rms_error(c) = smallest_rms_error(r);
```

Again the code for this function will be stored in the database and can be shared by all users who open the *protmod* module.

Other fragment selection criteria may make use of information about the sequence of residues of the fragments in their native structures. For example, the following alternative definition of *best_fragment* selects the fragment whose sequence in its native structure has the most residues in common with the target sequence of the given backbone range :-

```
define best_fragment(r in backbone_range) -> candidate_fragment
    in protmod
    the c in for_range_inv(r) such that
        percentage_identity(sequence(c), sequence(r)) = greatest_identity(r);
```

where the function *greatest_identity* returns the percentage identity which the most identical candidate fragment has with the target sequence. *Greatest_identity* can be defined in Daplex as follows :-

```
define greatest_identity(r in backbone_range) -> float in protmod
    maximum(percentage_identity(sequence(r), sequence(for_range_inv(r))));
```

In the definition of *greatest_identity*, the call to *sequence(r)* returns a single string and that to *sequence(for_range_inv(r))* returns a set of strings — one for each fragment in the set of candidate fragments identified for range *r*. The function *percentage_identity* is evaluated for each pair of parameters in turn, to give a set of real numbers. The built-in function *maximum* is then used to find the largest number in that set. The function *percentage_identity* is defined in Prolog. It takes two strings as parameters, and calculates the percentage of their characters which are identical. Clearly, this function has to be derived rather than stored, since we cannot enumerate all possible values for the parameters.

Neither the sequences for backbone ranges nor those for candidate fragments are stored in the database since they can be calculated when required using methods defined in Prolog. However, if it is found later that these sequences have to be calculated

frequently, then we can arrange that the values of these functions should be stored, rather than derived.

While many methods can be written in Daplex, some are coded directly in Prolog. Typically, these are methods which need to call out to C or FORTRAN, or require Prolog's list-processing features, or perform a complicated search which cannot be expressed in Daplex. An example of a database search that includes some list-processing is shown in Figure 6.

Other modelling objects

Each *candidate_side_chain* object contains information about a side chain conformation for a residue at a given position in the sequence, and within a particular candidate fragment.

A *candidate_atom* contains the *x*, *y* and *z* coordinates, and the *radius* of an atom belonging to the backbone of a candidate fragment or to a *candidate_side_chain*.

The object class *comment* enables annotations to be added to a model. Each comment has a text string value, a reference to the modeller making the comment, and references to those object instances to which the comment applies. One use of comments is to record which computationally expensive modelling tasks have already been performed on which objects. If re-invoked, then the presence of the comment is detected and the action is not executed for a second time. The requirement that particular actions should not be performed on the same object more than once can be made a precondition of that action.

In addition to these objects, users share objects in the main database. In applications such as modelling, these shared objects are referenced and accessed, but the shared objects are not updated.

The user is not restricted to the entity classes and properties described here. New classes can be declared and populated "on-the-fly" (Gray and Kemp, 1990), and new attributes and relationships can be declared on existing classes. If additional data introduced by the user is to be persistent, then these must satisfy type declarations and referential integrity. This is not like storing arbitrary pieces of LISP or Prolog. This may sound restrictive but we have yet to find an example of data which we have wished to make persistent which could not be modelled in this way which gives the advantages of being able to enumerate these objects and of fast access by key value or object identifier to large quantities of data, and other advantages generally provided by a database system.

VERSION MANAGEMENT

Version management is an important requirement of a modelling system. The main issues in version management are outlined by Kent (1989).

In the protein modelling application, users should be able to construct alternative versions of the same protein model. If several modelled structures are identical except for a few conformational changes, then it should not be necessary to store the entire structure of each model, since this would result in much duplication of data. Each user

```

%-----%
% NEIGHBOURS_WITHIN(+Distance, +StartList, +Visited,
% -ResultList)
%
% Find all sub-components which might have atoms
% within the given distance of the any atoms of
% those sub-components in the start list.
%
% The list "Visited" contains those sub-components
% which have been found so far
% (and should be empty on the first call).
%
% The function "neighbour" returns those sub-components
% which have atoms within 5 Angstroms of those in the
% given sub-component.
%
% More distant neighbours can be found by applying
% the function "neighbour" recursively.
%
% It is assumed that if two sub-components, A and B,
% have atoms within 10 Angstroms, but none within 5,
% then there will be some third sub-component which
% has atoms within 5 Angstroms of both A and B.
%-----%

```

```

neighbours_within(_, [], Visited, Visited) :-  

    % No sub-components in the start list, so stop.  

    !.  

neighbours_within(Distance, StartList, Visited, ResultList) :-  

    Distance =< 0,  

    % Gone far enough, so stop.  

    append(StartList, Visited, ResultList), !.  

neighbours_within(Distance, StartList, Visited, ResultList) :-  

    % Find sub-components at the next layer out, and call recursively.  

    findall(Neighbour,  

        (member(S, StartList),  

        getfnval(neighbour, [S], Neighbour),  

        \+ member(Neighbour, StartList),  

        \+ member(Neighbour, Visited)),  

        NextStartList),  

    remove_dups(NextStartList, UniqStartList),  

    append(StartList, Visited, NextVisited),  

    NextDistance is Distance - 5,  

    neighbours_within(NextDistance, UniqStartList, NextVisited, ResultList).

```

Figure 6. An example of a database search expressed in Prolog.

should be able to access and make use of the work of others, however a particular user should not be able to alter versions belonging to other individuals, or groups.

Instead of implementing a general management scheme, a suitable version management mechanism is provided by using objects to represent different versions of protein models. A *version* consists of a set of *selected_ranges* spanning the length of the chain, a set of *selected_fragments* (one for each *selected_range*) and a set of *selected_side_chains*. Sharing of working data between different versions is achieved by different versions referring to the same working objects.

Each *version_of* a protein model can be worked on by a number of *modellers*. By registering a set of modellers for each version we can impose control on who may amend each version, thus the work of individuals or groups is "protected" from others. The login name of the current user of the system is used to check whether the current user has permission to make changes to particular versions of protein models.

The version management mechanism used here is much simpler than that provided by the Version Server of Katz and Chang (1987), which can deal with modifications to sub-components in component/sub-component hierarchies. The kind of version management which we use is satisfactory for our application since users observe the convention that backbone ranges, candidate fragments and candidate side chains are neither updated nor deleted. As modelling proceeds, new instances of these classes are created, but these instances are not subsequently changed or removed. Therefore, the problems of "change propagation", described by Katz and Chang do not arise.

Versions of protein models are independent of each other in our system — we do not maintain version hierarchies or graphs. A new version may be based on an existing version, in which case references from the existing version are copied to the new one. Thereafter, the two versions are completely independent, and modifications to one do not affect the other.

MODULES IN P/FDM

Data in P/FDM is partitioned into modules. Individual users can create new modules and modules of data may be shared by all users.

Locking in P/FDM is by module. A module can either be open for writing by one user, or open for reading by several users. Thus, by partitioning data into modules one can limit the amount of data which must be locked off from other users while updates are performed. This kind of locking is cost-effective for design applications which proceed slowly over long time periods before committing changes.

In the protein structure application the data from the BIPED project is divided into two modules, with a third module introduced to store reference data on individual atom and residue types. Users developing applications which require new entity classes and functions to be introduced are encouraged to add these to separate modules so that the main corpus of protein structure data remains available to other users. Should these new classes and function prove to be of general value, then they may be added later to modules shared by all users.

Originally, the values of relationship functions were stored in the same module as the entity instances on which they were declared. However, this can be inconvenient when relationship functions are declared between entities in one module and those

in others. Values for such functions were stored in one module, while those of the automatically declared inverse function were stored in the module containing the result class. Therefore, both modules have to be open for writing. This was undesirable for users writing applications using the protein structure database. Often, private applications would need to refer directly to objects in the main modules of structure data which were used by all application developers. In order to do this it was necessary to open the main database for writing, locking out all other potential users. Also, since the main database modules were open for writing, and being updated in the course of the application, these reference modules could become cluttered with individual users' *ad hoc* data, or even corrupted through a careless update. Therefore, there is a requirement to be able to declare relationship functions from a class in one module to a class in another without it being necessary for that other module to be open for writing.

In the protein modelling application we want to have relationship functions from private data to shared objects, for example, to relate each protein model instance to the chain on which the model is to be based, and each candidate fragment to the residue, in its native chain, at which the fragment starts.

Relationships from objects in an individual user's module to those in the main database can be implemented by storing the values of the key attributes of objects in the main database with objects in the application module, and then defining a derived relationship function using these values. Thus, a relationship between a candidate fragment and a residue could be achieved by storing the position, chain identifier and protein code of the related residue with the candidate fragment object and then defining a derived function which will rebuild the key and access the required residue directly. However, it is more natural and convenient to have just a stored relationship function, rather than having to store values of key attributes of some other object and then derive the relationship from these.

A satisfactory solution to the problem of permitting stored relationship functions from objects in private application modules to those in modules of data shared with other users is to support different kinds of modules and to allow relationships without inverses to be declared from modules with low status to modules of higher status. Not having inverses stored is not a major limitation since, in applications, users normally use the "forward" relationship from the application object to objects in the main database and are not interested in the inverses of these functions.

The current implementation of P/FDM supports three kinds of module — shared, private and temporary (Jiao, 1990).

Temporary modules have the lowest status. These modules are stored in main memory and, consequently, access to data in these modules is relatively fast. However, these modules do not persist when the module is closed or the database session ends.

Private modules are used when persistence is required by user applications.

The main use of *shared modules* is for storing reference data, such as data from the Protein Data Bank. A characteristic of this data is that it is seldom updated, and when they do occur updates usually involve adding new items of data (e.g. new entity classes and functions) or adding new data values (e.g. data for a new protein structure), rather than altering or deleting existing data from these modules.

Relationship functions are permitted from entities of a class in one module to those in a second module, provided that the status of the second module is at least as high as that of the first. This is because data in lower status modules will normally be more dynamic, and there is a risk of the integrity of the database being lost if an entity in a private module, referenced by one in a shared module, is deleted. This problem is demonstrated more clearly with temporary modules. All entity instances in a temporary module are lost when that module is closed. Any references to temporary entities from persistent ones will be left and referential integrity lost.

An inverse function will be declared automatically on declaring a relationship function between classes in different modules if the two modules have the same status. If they have different status then no inverse is allowed.

The ability to declare relationship functions between modules of the same status permits data from a large application to be partitioned. It also enables data from independent applications to be combined. For example, data on hydrophobic microdomains (Kemp and Gray, 1990) could be combined with that for a homology modelling exercise, enabling information on hydrophobic clusters to be used in the prediction of structurally conserved regions.

ACCESSING AND MANIPULATING WORKING DATA

P/FDM provides facilities for printing results and performing simple updates in both Prolog and Daplex. It has become necessary to make extensions to P/FDM which permit user-definable *action* methods which allow protein modelling tasks to be invoked from Daplex and frequently even defined in Daplex. Disciplined use of working data is encouraged through the use of high level routines to perform modelling tasks, accessible from Prolog and Daplex.

Action definitions in Daplex have the following form :-

```
define <action_name> (<arg_list>) in <module_name>
    <imperative>
```

The header for each action contains the name of the action, a parameter list, and the name of the module in which the optimised Prolog code generated for the action will be stored.

Imperatives in Daplex consist of (optional) loops followed by print statements, database update commands and other actions. For example, the following imperative prints the length and type of each range selected for the version "V1" :-

```
for the v in version such that version_name(v) = "V1"
    for each r in selected_ranges(v)
        print(range_type(r), range_length(r));
```

Actions can be used to print information about working data. For example, the following definition is for a method to print information about backbone ranges :-

```
define print_range(r in backbone_range) in protmod
    print("Type is", range_type(r),
        "length is", range_length(r),
```

```
"between alignment positions", position(start_position(r)),
"and", position(end_position(r)));
```

This action provides a concise way of printing all the salient information about backbone ranges. As an example of its use, we can request information on all ranges selected for the version "V1" :-

```
print_range(selected_ranges(the v in version such that
version_name(v) = "V1"))
```

This example shows a very simple action. Arbitrarily complex actions can be written in Prolog which contain intricate database searches and calculations, and can call out to C or FORTRAN passing parameters (integers, floats, strings and pointers to C structures).

As well as printing information about working data, actions have been defined to display fragments of models, or entire versions, using the molecular graphics package *HYDRA*, written by R. Hubbard.

Action methods can also be used for updates. *Select_fragment* is an action which selects the given fragment for use in the given version, and can itself be defined in Daplex :-

```
define select_fragment(v in version, c in candidate_fragment) in protmod
  include {c} into selected_fragments(v);
```

To automatically select the best fragment for each selected range of a given version :-

```
define auto_select_fragments(v in version) in protmod
  select_fragment(v, best_fragment(selected_ranges(v))));
```

However, the above action definition of *select_fragment* is not sufficient in itself since there are restrictions on which users may modify each version, and on which ranges each version should have fragments for. Rather than trusting that these restrictions will be checked by the user, it is better to make them preconditions of the action. Preconditions can be placed in any action definition between the header and the imperative, and have the syntax :-

ensuring <predicate>

The definition of *select_fragment* can be rewritten as follows :-

```
define select_fragment(v in version, c in candidate_fragment) in protmod
  ensuring current_user() in workers(v) and for_range(c) in selected_ranges(v)

  include {c} into selected_fragments(v);
```

Preconditions can be used to help maintain the consistency of the database. However, they can be used for more than expressing integrity constraints. In this example the precondition is used to provide access control by checking that the current user of

the system has permission to modify a particular version of a protein model. It is not that individual methods are accessible only to particular users — in the case of *select_fragment*, the method can be used by any recognised modeller. The restriction is that each user can only use the method with a subset of the instances of the parameter types.

Information about steric overlap between fragments is calculated when selecting which fragments should be incorporated in the model. Another action is used to print occurrences of steric overlap between atoms in a pair of fragments. Instead of measuring all inter-atomic distances, space is partitioned into cubes, the sides of which are the length of the diameter of the largest atom in the structure, and each atom in the fragments is allocated to the cube in which its centre lies. This enables efficient searching for possible steric overlap, since only distances between atoms in the same or neighbouring cubes need to be checked.

DISCUSSION

The invocation of each modelling step can be expressed concisely in Daplex, and quickly compiled to Prolog with some optimisation which determines the most promising order for goals. By calling each of these goals in turn, a first approximation to the protein model can be generated in a deterministic way. The sequential control structure for doing this is explicit in the order in which the goals are called.

Our experience has shown that models constructed in this deterministic manner are unlikely to be satisfactory for a variety of reasons, e.g. steric overlap of atoms, poor packing, or no suitable fragments identified by the database search for a particular range. Therefore, it is important that the system should allow subsequent refinement of the structure. Improvements to structures are performed by users of our system through high level Daplex calls to routines for rejecting and selecting alternative fragments for variable regions. Users may also re-select anchor points for variable regions and hence define new backbone ranges which may enable better fragments to be identified.

Since the best way to construct models from homologous proteins is not yet known, and models will have to be modified to satisfy any experimental information, it is important that a modelling system should allow users to modify or improve a first approximation to the structure.

Once we know more about the criteria to be used in modifying model structures, it may be possible to automate some of the refinement strategies employed using Prolog. This could be done easily since primitives allow the database to be accessed and updated directly from Prolog, and calculation can be combined with data retrieval. Prolog's control structure enables backtracking and re-binding variables with alternative values, allowing alternative models to be explored. A software architecture combining Prolog with an OODBS enables rapid prototyping of applications which requires flexible access to data, combined with calculation (Kemp and Gray, 1990).

```

create temporary module tempmod
declare part ->> entity
declare part_name(part) -> string
declare subpart(part) ->> part
declare quantity(part, part) -> integer
declare incremental_cost(part) -> integer
key_of part is part_name;

define total_cost(p in part) -> integer in tempmod
  incremental_cost(p) +
  total(over sp in subpart(p) of quantity(p, sp) * total_cost(sp));

total_cost([part],get,[A],B) :-
  (findall(C,
    (getfnval(subpart,[A],D),
     getfnval(total_cost,[D],E),
     getfnval(quantity,[A,D],F),
     C is F*E),
    G),
   total(G,H)),
  getfnval(incremental_cost,[A],I),
  B is I+H.

```

Figure 7. Example of a part/sub-part hierarchy. The schema for the hierarchy is shown at the top, followed by the definition, in Daplex, of a method for calculating the cost of a composite part, and the Prolog code to which this query is translated.

CONCLUSIONS

The modelling system described here was motivated by limitations of entirely interactive modelling systems and automatic model building programs. Tasks such as re-modelling loops were extremely difficult using an interactive modelling system alone. Automatic programs which build entire structures are not suitable since the process of model building is not well enough understood for their results to be satisfactory, and they do not allow individual users' preferences and strategies to be incorporated into the modelling process.

This application is one which makes use of large quantities of data stored on disc. Prolog is used to reason with data stored in a large database, not just data held in memory. The database is tightly coupled, with data brought in on demand, using indexes on disk. It is not just copied into memory from disk like a saved state.

P/FDM provides entity-based persistence for entity classes which are used to store design concepts as well as physical entities such as molecular fragments. The semantics of the protein modelling application can be represented using these entity classes and

relationships between them. Representing versions of protein models using objects gives a suitable version management mechanism for this application.

Daplex provides a high level interface for accessing working data and for defining and invoking modelling tasks. Preconditions on Daplex actions can be used to provide for access control.

The software architecture described is based on Prolog. The database system is itself written in Prolog, with calls to C. The Daplex parser is written in Prolog, as is the optimiser which searches for efficient strategies for answering queries. Arbitrarily complex database searches can be expressed in Prolog.

While this paper concentrates on the protein modelling application, the P/FDM database is general purpose. For example, part/sub-part hierarchies present in many design applications can be represented in P/FDM, and recursive methods (e.g. to find the total cost of a composite part) can be expressed in Daplex (Figure 7).

The characteristics of the protein modelling application are of an assembly problem. There is a need for efficient search, using *ad hoc* criteria, through large quantities of structured data, and for data retrieval to be combined with calculation. This software architecture, which combines an object-oriented database with Prolog, should be of use in other design and assembly applications with similar requirements.

ACKNOWLEDGEMENTS

I would like to thank Peter Gray for useful comments on drafts of this paper, Norman Paton who implemented the original P/FDM system, Zhuoan Jiao who, with myself, has continued the development of P/FDM, Suzanne Embury for producing Figure 7 and for help with POSTSCRIPT. This work was supported by a grant from the SERC.

REFERENCES

- Atkinson, M.P., Bailey, P.J., Chisholm, K.C., Cockshott, P.W. and Morrison, R. (1983) An Approach to Persistent Programming, *The Computer Journal* 26 :360-365.
- Blundell, T., Carney, D., Gardner, S., Hayes, F., Howlin, B., Hubbard, T., Overington, J., Singh, D.A., Sibanda, B.L. and Sutcliffe, M. (1988) Knowledge-based protein modelling and design, *Eur. J. Biochem* 172:513-520.
- Claessens, M., Van Custem, E., Lasters, I. and Wodak, S. (1989) Modelling the polypeptide backbone with "spare parts" from known protein structures, *Protein Engineering* 2:335-345.
- Clark, D.A., Barton, G.J. and Rawlings, C.J. (1990) A knowledge-based architecture for protein sequence analysis and structure prediction, *J. Mol. Graphics* 8:94-107.
- Clocksin, W.F. and Mellish, C.S. (1984) *Programming in Prolog*, Springer-Verlag.
- Cohen, G.H., Silverton, E.W. and Davies, D.R. (1981) Refined Crystal Structure of -Chymotrypsin at 1.9A Resolution, *J. Mol. Biol.* 148:449-479.

- Gray, P.M.D. (1984) *Logic, algebra and databases*, Ellis Horwood.
- Gray, P.M.D. (1989) Expert Systems and Object-Oriented Databases: Evolving a new Software Architecture, in Kelly, B. and Rector, A. (eds.), *Research and Development in Expert Systems V*, Cambridge University Press, pp 284–295.
- Gray, P.M.D. and Kemp, G.J.L. (1990) An OODB with entity-based persistence: a protein modelling application, in Addis, T.R. and Muir, R.M. (eds.), *Research and Development in Expert Systems VII*, Cambridge University Press, pp 203–214.
- Gray, P.M.D., Paton, N.W., Kemp, G.J.L. and Fothergill, J.E. (1990) An object-oriented database for protein structure analysis, *Protein Engineering* **3**:235–243.
- Islam, S.A. and Sternberg, M.J.E. (1989) A relational database of protein structures designed for flexible enquiries about conformation, *Protein Engineering* **2**:431–442.
- Jiao, Z. (1990) Modules and Temporary Data in P/FDM, *Technical Report AUCS/TR9016*, Department of Computing Science, University of Aberdeen.
- Jones, T.A. and Thirup, S. (1986) Using known substructures in protein model building and crystallography, *The EMBO Journal* **5**:819–822.
- Katz, R.H. and Chang, E. (1987) Managing Change in a Computer-Aided Design Database, in Stocker, P.M. and Kent, W. (eds), *Proceedings of the 13th VLDB Conference*. Brighton, pp 455–462.
- Kemp, G.J.L. and Gray, P.M.D. (1990) Finding hydrophobic microdomains using an object-oriented database, *CABIOS* **6**:357–363.
- Kent, W. (1989) An Overview of the Versioning Problem, in Clifford, J., Lindsay, B. and Maier, D. (eds.), *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, Portland, Oregon, pp 5–7.
- McLachlan, A.D. (1979) Gene Duplication in the Structural Evolution of Chymotrypsin, *J. Mol. Biol.* **128**:49–79.
- Paton, N.W. and Gray, P.M.D. (1988) Identification of Database Objects by Key, in Dittrich K.R. (ed.) *Advances in Object-Oriented Database Systems: Proc. OODBS-II*, Springer-Verlag, pp. 280–285.
- Paton, N.W. and Gray, P.M.D. (1990) Optimising and Executing Daplex Queries using Prolog, *Computer J.*, **33**:547–555.
- Shipman D.W. (1981) The Functional Data Model and the Data Language DAPLEX, *ACM Transactions on Database Systems* **6**:140–173.

Task-driven descriptions of mechanical parts in a CAD system

U. Cugini,^{*} B. Falcidieno,[†] F. Giannini,[†] P. Mussio[§] and M. Protti[§]

***Istituto Macchine Utensili—CNR
via Ampere 56 20138 Milano Italy**

**†Istituto per la Matematica Applicata—CNR
via L. B. Alberti 4 16132 Genova Italy**

**§Universita' degli Studi di Milano—Dip. di Fisica
via Viotti 5 20100 Milano Italy**

Abstract. A feature recognition system is described for the detection of shape features which allow the performance of a task in a given working environment. The features are recognized in a technical drawing or in a 3D CAD model describing the part to be managed. The output of the recognition processes represents the recognized features as volumes in a hybrid geometric model. The system is designed by using Conditional Attributed IL Systems, that allow the representation of both boundary models and technical drawings describing the solid as string of attributed symbols.

These strings are rewritten by the use of rules describing features according to a given working context.

The recognized features are in any case denoted by sets of symbols corresponding to volumetric representations.

1. INTRODUCTION

Mechanical experts describe a mechanical part in several different ways, depending on the task to be performed on it and the working context. This characterization is obtained in terms of the features relevant to the task. These features are related to specific shapes in the object and are therefore named shape features.

It must be noted that the same entity can be described in terms of different sets of features, depending on the task to be performed (Cugini, Falcidieno, Mussio 1989). For example, a mechanical part (see fig. 1) can be described in terms of pairs of parallel planes if it has to be handled or in terms of through holes for the manufacturing or assembling tasks.

Thus, each different expert in a specific environment describes the same mechanical part by a specialized set of features and associate relationships.

In the traditional industrial environments, the experts describes the mechanical parts by the use of technical drawings. The conventional rules used allow to draw documents which are unambiguous for any reader. In this way, these documents collect all the data necessary for the workpiece production and are interpreted by each expert within a specific technical environment, on the basis of personal knowledge.

The use of geometric modellers poses the problem of this integration in a new perspective. The workpiece model is generated from the point of view of the mechanical designer, and it is difficult to restructure it from other points of view. However, a computerized document (here called c-document) can be proposed, which collects all the data necessary and sufficient to the production process and plays the role of technical drawings in the traditional design and communication process.

The c-document is stored as a CAD model which can be specialized to each different context by an interpreter exploiting a knowledge base in which the descriptions of features used in that specific working environment are collected. The result of the interpretation is a specialized description, which permits to associate to the geometric model of a part its functionality in a context. This representation can be used by the experts in each environment for their activity to browse the document, retrieve the data necessary to their activity, update it always exploiting their own set of specialized form features.

When necessary, they can even invoke the same interpreter to update the general data structure (the c-document), thus becoming able to communicate to the other environments their requests, as it happens in the traditional environment when one actor modifies the technical drawing and sends the modified copies around.

To show the feasibility of a system based on these ideas, in this paper, we describe how data can be captured from traditional archives or from a 3D description which constitute the c-document.

As to the first point, (data capture) mechanical parts are described in term of 3D features obtaining a partial 3D description of the part itself, from which a simplified engineering drawing is derived. Starting from this simplified drawing the c-document describing the mechanical part can be obtained using existing reconstruction algorithms (Sakurai and Gossard 1983, Preiss 1984, Aldefeld 1984). This procedure shows some advantages on the use of existing 3-D reconstruction algorithms which are commented later.

Then, the second point (c-document management) is illustrated, showing how the c-document can be specialized by the recognition of form features, typical of a given environment.

The two points are illustrated on the basis of experiences gained by the use of two existing modules, that execute these tasks independently, described respectively in Cugini, Mussio, Protti and Rossello 1989 and in Falcidiено and Giannini 1989.

The novelty in this work is the introduction of a common notation to describe in uniform way the two approaches. This notation is derived from conditional attributed L-systems (CAIL), which extend to parallel rewriting the attributed tools introduced by Knuth 1968 (see also Prusinkiewicz and Hanan 1988) as discussed in Mussio, Padula and Protti 1988 and allows the design of a unique interpreter to manage both the activities and hence the design of the sought system.

2. CONDITIONAL ATTRIBUTED IL-SYSTEMS

All the documents (the technical drawing, the c-document, the specialized descriptions) managed by a system like the one outlined in the introduction, can be described by strings (words) of attributed symbols.

An attributed symbol x is a symbol to which a finite set of attributes $A(x)$ is associated, each attribute $\alpha \in A(x)$ having a set $D_x(\alpha)$ of finite or infinite possible values, the associated properties. Words over a set of attributed symbols are formed by concatenation of the attributed symbols from the set

Each entity forming a form feature as well as the form feature itself can be denoted by an attributed symbol, in which the properties (i.e. the values of the attributes in the specific instance) denote its mechanical, geometric and topological characteristics.

A mechanical part can be associated to several attributed words , each one describing it at a different level of abstraction and/or in a different context, and formed by the attributed symbols denoting the entities or the shape features characterizing the part in that context at that level of abstraction. The transformation of a string into another one (i.e. of a description into a different one) is defined exploiting Conditional Attributed Lyndenmayer Systems with Interaction (CAILs), which are parallel rewriting devices, see Merelli, Mussio and Padula 1985, used to define formal languages in an evolutionary way, defining how a string of the language evolves into another of the same language, rather than describing its deep hierarchical structure as grammars do.

Formally, a conditional attributed IL-system is a 4-tuple $LS = \langle V, Ax, R, I \rangle$ where: V is an attributed alphabet: a finite set of attributed symbols. Ax is a set of words over V , called the set of axioms. R is a set of contextual conditional attributed rewriting rules. I is a set of metarules, called interpreter.

A CAIL rule is divided in two parts: a syntactic part - a pair $\langle A, C \rangle$ - and a semantic part - a pair $\langle Co, F \rangle$.

In the syntactic part: $A = z^0 r^0 t$ is a string of symbols (first element of an attributed string) called the antecedent; where z, t are strings (including the empty one) representing the context within which the string r (the rewriting part) has to appear in order to be rewritten as the string C .

In the semantic part:

- Co is a predicate whose variables are attributes of the symbols of the antecedent.
- F is a set of functions specifying how to compute the properties of the symbols in C starting from those of the symbols in A .

Given a string of attributed symbols, an interpreter rewrites it according to the following procedure, Mussio, Padula and Protti 1988:

- 1) verification of the occurrences of the antecedent;
- 2) evaluation of the predicate Co for each instance of the antecedent (and therefore if the properties of the structures denoted by the attributed symbols of the specific antecedent satisfy certain conditions);
- 3) substitution of each instance of the rewriting part of the antecedent for which Co is satisfied, by the consequent and calculation of the properties of the structure denoted by the consequent through the functions of the semantic rule.

Rule $r: \langle \langle A, C \rangle, \langle Co, F \rangle \rangle$ can either be expressed in the form:

$$A \dashv Co \rightarrow C \quad (a)$$

or in the verbal form for the sake of clarity:

$$r: \text{if } A \text{ has meaning } Co \text{ then } C \quad (b)$$

avoiding to specify function F when it can be taken as given.

3. FEATURE RECOGNITION FROM 2D DESCRIPTIONS

In traditional industrial environments, technical drawings in orthographic projections are iconic messages, drawn to communicate in a complete and non ambiguous way the specification of a solid object, (see an example in figure 1). They constitute a communication language based on well-defined orthographic conventions, although not formalized in the sense of computer science. These conventions even specify the set of tokens which can be used in a drawing and are exploited by the draftsmen when they define and trace a drawing or when they read it in a traditional non-automated environment.

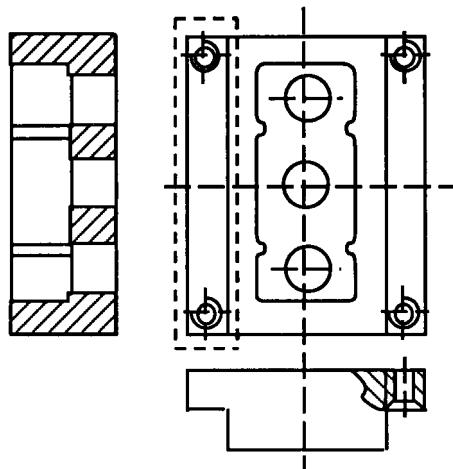


Figure 1. An example of technical drawing.

Of course, the use of this kind of communication language requires that the overall message is complete, and non ambiguous. Given a 3D object, there are several ways to represent it univocally by 2D documents. The chosen way depends on the point of view and the purpose of the representation, but it can be always assumed that a well done technical drawing will never be ambiguous, if interpreted by a person who understands the informal rules by which the document is constructed.

In general, the generation and interpretation of the drawing are always done on the basis of what we could define a "default principle". For instance, if we refer to the drawing in fig.1 the cross-section in the top view shows explicitly only one threaded hole and in principle we could imagine that the definition of the other three holes is undefined and ambiguous. But every reader, aware of the rules by which an engineering drawing is built (Jensen and Hines 1979), applies the "default principle", and extends the use of the information, expressed explicitly by the cross-section, to all the four holes.

In fact, if one or more holes have a different shape, they should be described by an explicit different cross-section. The lack of this form feature forces the interpreter to use the only one present.

The orthographic conventions, as expressed informally for example in Jensen and Hines 1979, and the default principles are codified into attributed conditional rules, and are exploited by the system to automatically analyze the input drawing so that, a c-document is obtained.

The technical drawing is first digitized into a black and white image (called binary digital image, BDI). A first set of rules are exploited in order to recognize the tokens conventionally used by the draftsman to draw the 2D document, Cugini, Mussio, Protti and Rossello 1989.

Once restored a description, based on those tokens, a 3D description of the form features relevant to a given working environment can be derived by the same technique, Mussio Protti and Rossello 1991. The tokens associated to these form features are then sieved out from the description (Bottoni, Protti, Mussio and Schettini 1989). The resulting c-document describes a set of tokens which form a simplified technical drawing which can be translated by an easier process in a 3D description using an existing algorithm (Preiss 1984, Sakuray and Gossard 1983, Aldefeld and Richter 1984).

This feature-based process allows the introduction of some novelty. First the automatic recognition and management of iconic symbols, like the top view of the threaded

hole, whose interpretation requires the knowledge of conventions of technical drawings language.

Second the presence of non completely described structures which can be interpreted only expanding their definition on the basis of the unique completely described similar structure and of the conventions of the language, as shown for the threaded hole in fig.1. This procedure simulates the application of the "default principles" by the human interpreter.

This means that, being the technical drawing a conventional message, the knowledge of the exploited rules and not only a simple geometric reconstruction permits the 3D description of the mechanical part.

The recognition and description of all these not well defined 3D structures allow to partially describe the mechanical part and to obtain a simplified drawing which can be reconstructed using other algorithms which manage only the geometric part of the drawing.

Only this last step can be performed by other similar techniques based on structural descriptions (Aldefeld and Richter 1984). However, the use of a feature based method avoids the combinatorial explosion typical of these approaches.

Note that the feature based method requires the coding of the draftsman's knowledge in an homogeneous way by means of rules. In our system these rules are managed within Conditional Attributed IL systems which act on strings. To use them therefore, the BDI must be translated into a description in the form of a string, that is obtained with the method discussed in Merelli Mussio and Padula 1985, based on the coding of each pixel of the BDI into a decimal number. Each pixel can be described by a symbol (the number) and two attributes, its coordinates. It is there demonstrated that the original image can be reconstructed from only a subset of these attributed symbols, the subset of codes denoting some special elements of its contours, called Multiple Elements (ME).

The set of attributed symbols associated to the set of the MEs forms a complete description of the image which is organized as a string, that is as a linear representation and which is called the "rough description" of the image.

An algorithm is presented in Mussio Padula and Protti 1988, which partitions the rough description into substrings, each one describing a simple contour of a connected set. This new description is called the primal one, because it is the input for the CAIIL-driven system which produces the c-document. This process can be simplified limiting the number of combinations of attributed symbols to be examined, by the intermediate recognition of some form features typical of the language of the draftsman. Once recognized, the data related to them can be subtracted by the description by a procedure which is identical to that used later to sieve out the 3D form features.

This set of procedures, which have been informally introduced, are related by the manipulation of description, which are strings of attributed symbols.

For each structure, a name v , the finite set of its attributes, $A(v)$, and the range of the admitted values $Dv(\alpha)$ for each attribute α in $A(v)$ are established. The set of all names of structures of interest (including the codes describing the MEs) constitutes the alphabet V of the conditional attributed IL system.

The subset V_1 of the alphabet V , required for the recognition of the 2-D top view of a threaded hole, is shown in Tab 2. Strings on V , which describe all or part of a technical drawing, can be defined. For example, the string a) of tab. 1 is a substring of the description of the top view of fig. 1 at the stage in which elementary form features like arcs, circumferences and dashed lines have been recognized and denoted, as shown in tab. 2.

The knowledge, used to steer the recognition is exploited to code the rules in R, which were studied so that the system simulates the draftsman activity.

For example reading a technical drawing, the rules of interpretation allow the combination of draftsman strokes into composed graphical structures as when two dashed lines are understood to form a cross, signalling the centre of a circle. (see fig. 1).

In an analogue way a CAIIL rule looks for the cross: let a dashed line be denoted by the symbol "d" with attributes x_i, y_i, x_f, y_f (the coordinates of the initial and end point respectively) whose values are real numbers and a cross be denoted by the symbol s with attributes the coordinates of the centre x_c, y_c which again range on reals.

a)	h	a	a	h	l	l	d	h	d	d	h	d
	$x_i: 50$	$x_c: 100$	100	$x_i: 50$	$x_c: 100$	100	$x_i: 30$	50	100	100	150	30
	$y_i: 200$	$y_c: 250$	950	$y_i: 200$	$y_c: 950$	250	$y_i: 250$	1000	180	880	200	950
	$x_f: 150$	$r: 25$	25	$x_f: 50$	$r: 20$	20	$x_f: 170$	150	100	100	150	170
	$y_f: 200$	$\alpha_i: \pi$	π	$y_f: 1000$			$y_f: 250$	1000	330	830	1000	950
	$\alpha_c: \pi/2$	$\alpha_d: \pi/2$	$\pi/2$	$\alpha_s: \pi/2$			$\alpha_d: 0$	0	$\pi/2$	$\pi/2$	$\pi/2$	0
b)	h	a	a	h	l	l	s	h	s	h	h	
	$x_i: 50$	$x_c: 100$	100	$x_i: 50$	$x_c: 100$	100	$x_c: 100$		$x_i: 50$	$x_c: 100$	$x_i: 150$	
	$y_i: 200$	$y_c: 250$	950	$y_i: 200$	$y_c: 950$	250	$y_c: 250$		$y_i: 1000$	$y_c: 950$	$y_i: 200$	
	$x_f: 150$	$r: 25$	25	$x_f: 50$	$r: 20$	20			$x_f: 150$		$x_f: 150$	
	$y_f: 200$	$\alpha_i: \pi$	π	$y_f: 1000$					$y_f: 1000$		$y_f: 1000$	
	$\alpha_c: \pi/2$	$\alpha_d: \pi/2$	$\pi/2$	$\alpha_s: \pi/2$					$\alpha_d: 0$		$\alpha_s: \pi/2$	
c)	h			Ft	h		Ft	h		h	h	
	$x_i: 50$	$x_c: 100$			$x_i: 50$	$x_c: 100$		$x_i: 50$		$x_i: 150$		
	$y_i: 200$	$y_c: 250$			$y_i: 200$	$y_c: 950$		$y_i: 1000$		$y_i: 200$		
	$x_f: 150$	$r_e: 30$			$x_f: 50$	$r_e: 30$		$x_f: 150$		$x_f: 150$		
	$y_f: 200$	$r_n: 25$			$y_f: 1000$	$r_n: 25$		$y_f: 1000$		$y_f: 1000$		
	$\alpha_c: \pi/2$							$\alpha_d: 0$		$\alpha_s: \pi/2$		

Table 1. The rewriting of the substring describing the particular of figure 1, from the stage in which elementary form features are recognized to the identification of top view of the threatened hole.

Rule 1 in tab.2 describes how and when a cross is identified.

Its application is shown in tab. 1 in the first step of rewriting. In this substring, two instances of the antecedent of the mentioned rule - i.e. two couples of symbols 'd' interleaved by a string- exist and whose attributes match the condition CROSSING $d_1 d_2$. Therefore they are simultaneously substituted by two occurrences of the symbol 's', to which different properties are associated by the semantic rules: in this way the string b) of tab. 1 is generated.

The string c) of tab. 1 is obtained by the application of rule 2 in tab. 3. The same rule is represented in an iconic form, more suitable for human communication, in fig. 2.

$V_1 = \{a, l, d, Ft, s\}$	$A(s) = \{X_c, Y_c\}$	$A(a) = \{X_o, Y_o, r, \alpha_l, \alpha_s\}$	
$A(l) = \{X_c, Y_c\}$	$A(d) = \{X_l, Y_l, X_f, Y_f, \alpha\}$	$A(Ft) = \{X_o, Y_c, r_n, r_e\}$	
$D_a(X_c) = D_a(Y_c) = D_l(X_c) = D_l(Y_c) = D_s(X_c) = D_s(Y_c) = D_{Ft}(X_c) = D_{Ft}(Y_c) = [N_{min}, N_{max}]$			
$D_d(r) = D_l(r) = D_{Ft}(r_n) = D_{Ft}(r_e) = [N_{min}, N_{max}]$	$D_a(\alpha_l) = D_a(\alpha_s) = D_d(\alpha) = [0, 2\pi]$		
$D_d(X_l) = D_d(X_f) = D_h(X_l) = D_h(X_f) = [1, N]$	$D_d(Y_l) = D_d(Y_f) = D_h(Y_l) = D_h(Y_f) = [1, M]$		
Symbol a arc l circumference d dashed line Ft top view of threatened hole s cross x x-coordinate of a point y y-coordinate of a point r radius α angle h thick line	stands for a arc l circumference d dashed line Ft top view of threatened hole s cross x x-coordinate of a point y y-coordinate of a point r radius α angle h thick line	Subscript i initial n internal f final e external c centre	stands for i initial n internal f final e external c centre

Table 2. A subset of an alphabet used for the recognition of a threatened hole, with its interpretation.

1) Syntactic Rule

$d_1 \beta d_2 \longrightarrow \text{CROSSING } d_1 d_2 \rightarrow s_-$

1) Semantic Rule

$(X_{c_s}, Y_{c_s}) \leftarrow (-\alpha_{d_1}) \text{ROTATE}(\alpha_{d_1}) \text{ROTATE } d_1 \text{CROSS}(d_1) \text{ROTATE } d_2$

2) Syntactic Rule

$a \gamma \delta s \longrightarrow \text{IF CONCENTRIC } a \mid s \rightarrow Ft_-_-$

2) Semantic Rule

$X_{c_{Ft}} \leftarrow X_{c_l} \quad Y_{c_{Ft}} \leftarrow Y_{c_l} \quad r_{e_{Ft}} \leftarrow r_a \quad r_{n_{Ft}} \leftarrow r_i$

Where:

- β denotes any string on V_1 ,
- IF_CONCENTRIC is a conditional function whose value is true if the center of arc a , circumference l and cross s coincide
- CROSSING is a conditional function which is true if d_1 cross d_2
- ROTATE and CROSS are functions such that: Let b, c, d quadruples of numbers, each one associated to the coordinates of pixels which are end elements of a straight segment and α a number ($0, 2\pi$): α ROTATE b rotates the coordinate in b of α ; c CROSS d finds the coordinates of the crossing element of the dashed lines described by c and d .

Table 3. The two rules required for the recognition of the threatened hole

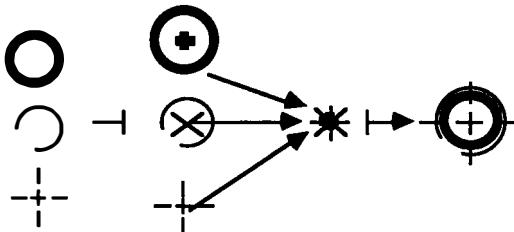


Figure 2. Iconic representation of the rule for the recognition of the top view of a threaded hole

In the second rule of tab3, the antecedent is formed by a string of attributed symbols, those of an arc a, a circle c and a cross s (g and d represent any other set of attributed symbols), which must be concentric, as stated by the condition. If this is the case, the three icons are grouped into a complex one.

One hundred and sixteen rules of this type have been used to define the elementary icons like arcs and six classes of complex form features, Cugini, Mussio, Protti and Rossello 1989. The set AX of axioms, completing the definition of the CAIL, is the set of admissible "primal descriptions" obtained by the use of the quoted algorithm on legal technical drawings.

4. THE COMPUTERIZED DOCUMENT

Another possible way of designing mechanical parts consists in using directly a CAD system that automatically produces a c-document in terms of data structure. In our system the data structure is a relational boundary model called Attributed Face Adjacency Hypergraph, Falcidieno and Giannini 1989.

The *Attributed Face Adjacency Hypergraph* of an object S is a labelled hypergraph defined as a quadruple $G = (N, A, H, T)$ such that:

- (i) Let F_S be the set of the faces of S, for every face f in F_S there exists a unique node in N corresponding to f and labelled f .
- (ii) Let E_S be the set of the edges of S, for every edge e in E_S , shared by two faces f_1 and f_2 , there exists a unique arc in A joining the two nodes f_1 and f_2 corresponding to faces f_1 and f_2 which is labelled e .
- (iii) Let V_S be the set of the vertices of S, F_v the set of the faces of S incident into vertex v of V_S corresponding to N_v in N. Then for every v in V_S , there exists a unique hyperarc in H, which connects the nodes of N_v , and is labelled v .
- (iv) To every arc e in A an attribute t in T is assigned, where $t=1$ if the edge associated to the arc e is a convex edge, and $t=2$ if the edge is concave.

Thus, the three primitive topological entities describing the boundary of an object (faces, edges and vertices) are explicitly represented together with their mutual adjacency relationships in the AFAH model. Moreover, for each face its bounding edges are grouped in loops, that are closed ordered cycles.

Also in the case of c-documents the identification of the features relative to the part is a fundamental problem for the automation of the processes involved in the part production, see Shah 1988, Henderson 1984, Kyprianou 1980, Joshi and Chang 1988, Sakyrai and Gossard 1988.

According to the shape feature definition previously introduced, each feature in a given context is coded as a set of entities which obey precise topological and geometric constraints.

Thus, the AFAH model, which keeps in a complete way the topological and geometric aspects of a solid object, is examined by checking for the presence or the absence of this set of entities.

To do this, a method has been devised which is able to automatically identify and extract feature information from the model of a part and organize that feature information into a higher level data structure, the Structured Face Adjacency Hypergraph, suited for explicitly representing the specialized description in a specific context.

This feature model is still a CAD model, which is feature-based, suitable for feature representation, easy to modify, and coherent with the input model of the examined part. In order to also express the feature information in terms of geometric and topological entities it encodes the basic relationships among features: hierarchy and adjacency.

These high-level features are described by means of intermediate features of a lower-level, in turn defined in terms of their intermediate features lower than themselves, and so on down to intermediate level features that are defined in terms of the input entities of the system.

The structure called Structured Face Adjacency Hypergraph (SFAH) can be formally defined as a pair $g^* = (H, G)$ where G is the acyclic oriented graph describing its hierarchical structure and H is a family of AFAHs, each of which, called component of g^* , is associated with a distinct node in H .

Any non-root component of g^* is the FAH representation of a feature in its parent graphs. The parent-child relation between any pair of components H_i and H_j of g^* is defined by a set of nodes which belong to both H_i and H_j .

These nodes are called *connection_nodes* in the parent component H_i and *dummy_nodes* in the child component H_j . The connection nodes of H_j in H_i correspond to those faces in the object to which the feature is attached, while the dummy nodes correspond to those faces that are added to the feature component in order to have a solid closed compact volume representing the feature.

5. 3D FEATURE RECOGNITION SYSTEM

The system for the 3D feature recognition and extraction maps an AFAH model into a specialized description and acts in three step: at first the faces constituting the feature are identified starting from those faces that certainly are interested by features, i.e. the so called primary faces that contain inner loops or that have concave edges in their external loops, Kyprianou 1980. Then in the second step, the identified feature faces are completed by adding so called dummy entities (faces, edges and vertices) in order to create a volume corresponding to the feature object. In the third step, the created feature volumes are inserted in the hierarchical structure previously described.

Also this activity can be described by means of a CAIL which is different according to the considered application context and store the knowledge of the expert in that context. In fact, the AFAH description of an object can be represented as a string of attributed symbols, that encode the relational information present in the boundary model.

In this case, the alphabet V is given by the union of two different alphabets: $V = V_1 \cup V_2$. V_1 is used to describe the input FAH model and is formed by three different symbols: $V_1 = \{L, L_E, L_F\}$, where L represents a loop on a face, and L_E and L_F are the labelled loops indicating if they belong to an explicit protrusion (L_E) or to any other kind of features (generic depressions or implicit protrusions, see fig. 3). These symbols have a set of attributes: $A(L) = A(L_E) = A(L_F) = \{Id, T_1, T_2, L_f, Ne, Me\}$, where Id is the loop identifier, T_1 specifies if L is an external or an internal loop for the face to which it belongs. T_2 , with domain $D_L(T_2) = [\text{conc}, \text{conv}, \text{hybr}]$ denotes if the loop is formed by only

concave, convex or both kinds of edges. L_f is the identifier of the loop containing L , if L is an external loop then $L=L_f$. N_e is the number of edges forming L . M_e is the matrix specifying the information for the loop edges: each row corresponds to an edge of the loop and columns contain the edge identifier, the type (concave or convex) and the name of the adjacent loop.

In figure 4 the string corresponding to a box with a pocket is depicted together with the attributes associated to each symbol of the string.

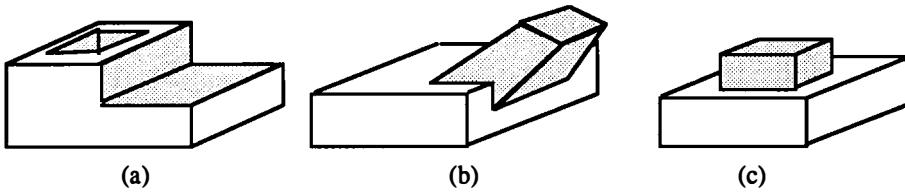


Figure 3. Form feature examples: (a) generic depression, (b) implicit protrusion, (c) explicit protrusion

V_2 is constituted by the symbols corresponding to the shape features present in the SFAH structure: $V_2=\{D, P\}$, where D corresponds to a Depression feature and P to a Protrusion feature, both the symbols are the same attributes: $A(P)=A(D)=\{N_{df}, A_{df}, N_{cf}, A_{cf}, N_e, M_e\}$, where N_{df} is the number of the dummy faces added to the feature in order to create the feature volume. A_{df} is the array containing the loop identifiers corresponding to the dummy faces. N_{cf} denotes the number of the connection faces for the feature associated to D , i.e. the faces of the father components to which the feature is attached, the list of the connection faces is specified by the array A_{cf} containing the identifiers of their corresponding external loops. N_e is the number of edges of the feature D (P) and M_e is the matrix describing them, where each row corresponds to an edge and columns contain respectively the identifier of the two loops sharing the edge and its type (concave or convex).

Also the set of the rewriting rules can be seen as formed by two sets of rules: the first set uses only symbols of V_1 and corresponds to the first step of the system: the feature loops identification, while the rules of the second set rewrite symbols of the V_1 with symbols of V_2 and correspond to the extraction and organization steps of the implemented system. For the sake of simplicity, here we describe only the rules for the identification of the feature loops. These rules reflect the specification of the feature loops sets:

Let L_S be the set of the loops in the FAH model, L be a concave or hybrid external loop, $L_{FF} = \{L_i \in L_S / \exists \text{ a concave edge } e = L_i \cap L_k \text{ with } L_k = L \text{ or } L_k \in L_{FF}\}$ be the set of the loops forming a depression or an implicit protrusion.

While the set L_E of the loops forming an explicit protrusion is given starting from an internal concave loop L : $L_E = \{L_i \in L_S / \exists \text{ an edge } e = L_i \cap L_k, \text{ with } L_k = L \text{ or } L_k \in L_{FF}\}$.

Thus, according to the two sets definitions we have the two rules shown in table 4 and table 5.

The completing rules specify how the feature loops must be completed they depend on the considered kind of feature and consequently on the considered context, and correspond to the acceptance of the feature made by the user of the system.

In figure 4, it is shown how the system acts on a simple object, a box with a simple pocket. The first string corresponds to the FAH representation of the object, for simplicity the matrices for the edge information are omitted. The second string corresponds to the rewritten one after the application of the identification rule, where symbols L corresponding to the loops belonging to the pocket ($L_8, L_9, L_{10}, L_{11}, L_{12}$), have been replaced by the symbol L_F . Finally, the third string represents the final structure where all the loops of the pocket have been substituted by the symbol D that indicates the presence of a depression feature with only one dummy loop (in figure labelled -7) attached to the father component through the loop 1.

Syntactic Rule

$$L_1 \partial L_2 \dashv C_1 L_1 L_2 \dashv \rightarrow \dots L_F$$

Semantic Rule

$$Id_{LF} <-- Id_{L2}$$

...

$$Me_{LF} <-- Me_{L2}$$

$$C_1 \quad T_1(L_1) = \text{external}$$

$$T_2(L_1) = \text{concave or hybrid}$$

$$\exists i / Me(L_1)_{i,2} = \text{concave}$$

$$Me(L_1)_{i,3} = L_2$$

$$T_1(L_2) = \text{external}$$

Table 4. Rule for the loops belonging to a depression or to an implicit protrusion

Syntactic Rule

$$L_E \partial L_1 \dashv C_3 L_E L_1 \dashv \rightarrow \dots L_E$$

$$L_1 \partial L_2 \dashv C_2 L_1 L_2 \dashv \rightarrow \dots L_E$$

Semantic Rule

$$Id_{L1} <-- Id_{L2}$$

...

$$Me_{L1} <-- Me_{L2}$$

$$C_2 \quad T_1(L_1) = \text{internal}$$

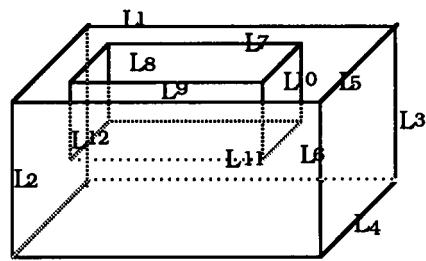
$$T_2(L_1) = \text{concave}$$

$$\exists i / Me(L_1)_{i,3} = L_2$$

$$C_3 \quad \exists i / Me(L_E)_{i,3} = L_1$$

$$T_1(L_1) = \text{external}$$

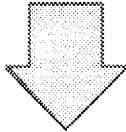
Table 5. Rules for the identification of the loops belonging to an explicit protrusion



	L	L	L	L	L	L	L	L	L	L	L	L	L
Id:	1	2	3	4	5	6	7	8	9	10	11	12	
T1:	Ext	Ext	Ext	Ext	Ext	Ext	Int	Ext	Ext	Ext	Ext	Ext	Ext
T2:	Conv	Hybr	Hybr	Hybr	Conc	Conc	Hybr						
Lf :	1	2	3	4	5	6	1	8	9	10	11	12	
Ne:	4	4	4	4	4	4	4	4	4	4	4	4	4
M e:



	L	L	L	L	L	L	L	L _F					
Id:	1	2	3	4	5	6	7	L _F					
T1:	Ext	Ext	Ext	Ext	Ext	Ext	Int	Ext	Ext	Ext	Ext	Ext	
T2:	Conv	Hybr	Hybr	Hybr	Conc	Conc	Hybr						
Lf :	1	2	3	4	5	6	1	8	9	10	11	12	
Ne:	4	4	4	4	4	4	4	4	4	4	4	4	4
M e:



	L	L	L	L	L	L		D	
Id:	1	2	3	4	5	6		Nfd:	1
T1:	Ext	Ext	Ext	Ext	Ext	Ext		Afd:	- 7
T2:	Conv	Conv	Conv	Conv	Conv	Conv		Nfc:	1
Lf :	1	2	3	4	5	6		Afc:	1
Ne:	4	4	4	8	8	4		Ne:	12

Figure 4. A box with a pocket with its string representation, the rewritten string deriving from the rule for the feature loops identification.

ACKNOWLEDGMENT

This work was supported by the Progetto Finalizzato Robotica sottoprogetto PRORA and agreement n. 89.00537.67.115.17487.

REFERENCES

- Aldefeld B., Richter H. (1984) Semiautomatic three-dimensional interpretation of line drawings, *Comp. and Graph.* Vol. 8, N. 4 pp. 371-380.
- Bottino P., Protti M., Mussio P., Schettini R. (1989) Knowledge-based contextual recognition and sieving of digital images, *Pattern Recognition Letters* vol. 10, n.2, pp.101-110, Amsterdam, North Holland, 1989
- Cugini U., Falcidieno B., Mussio P. (1989). Exploiting Knowledge in a CAD/CAM architecture to be published in *Intelligent Cad II*- H. Yoshikawa, T Holden eds. Elsevier.
- Cugini U., Mussio P., Protti M., Rossello A. (1989). From technical drawings to 3-D objects descriptions, *Tech. Report n. 5*, University of Milan, Department of Physics, System theory Chair.
- Falcidieno B., Giannini F. (1989). Automatic Recognition and Representation of Shape based Feature in a Geometric Modeling System, *Computer Vision, Graphics and Image Processing*, 48 pp. 93-123.
- Henderson M.R., Anderson D.C. (1984). Computer Recognition and Extraction of Form Feature in a CAD/CAM link. *Computer in Industry* 5.
- Jensen C.K., Hines R.D. (1979). *Interpreting engineering drawings*, Van Nostrand Reinholds Co. ed.
- Joshi S., Chang T.C.(1988). Graph-based heuristics for Recognition of Machined Features from 3D Solid Model. *Computer Aided Design*, March 1988
- Knuth D. (1968). Semantics of context-free languages, in *J. Math Syst. Theory*, vol. 2, pp. 127-145.
- Kyprianou L.K., (1980) Shape classification in Computer Aided Design, *Ph.D. Dissertation*, Computer Laboratory, University of Cambridge, England.
- Merelli, Mussio, Padula (1985). An approach to the definition, description and extraction of structures in Binary Digital images, *Computer Vision, Graphics and Image Processing*, Vol 31, pg. 19-49.
- Mussio P., Padula M., Protti M. (1988). Attributed conditional L-systems: a tool for image description, in *Proc. 9th Int. Conf. Pattern Recognition*, vol. I, pp. 607-609
- Mussio P., Protti M., Rossello A. (1991), Task-driven form features recognition: a tool for robotic activity management, accepted for ICAR91.
- Negoita C.V., Ralescu D. (1987). *Simulation, knowledge-based Computing and statistics*, Van Nostrand Reinhold, New York.
- Preiss K. (1984). Constructing the solid Representation from engineering Proiections. *Comp. and Graph.* Vol. 8, N. 4 pp. 381-389, 1984.
- Prusinkiewicz P., Hanan J., (1988). Lindenmayer Systems, Fractals and Plants, *ACM Siggraph* 1988, April.
- Sakuray H., Gossard D. C. (1983). Solid model input through orthographic views, *Siggraph*. pg 242-252.
- Sakurai H., Gossard D. C. (1988). Shape Features Information from 3D Solid Models. *ASME Computers in Engineering Conference*.
- Shah J.J., Sreevalsan P., Rogers M., Bathia (1988). Current status of feature technology. *CAM-I Report*.

Being economical with the truth: assumption-based context management in the Edinburgh Designer System

B. Logan, K. Millington and T. Smithers

Department of Artificial Intelligence
University of Edinburgh
Edinburgh EH1 2QL UK

Abstract. The problem of controlling inference is a common one in AI systems. It is particularly acute in AI-based design support systems, which to be effective must not only produce the right answer, but must also produce it at the right time. In this paper we discuss these problems in the context of a particular implementation, the Edinburgh Designer System (EDS), a design support system for mechanical design problems. We present an approach to the control problem which utilises the *context* of a task to determine which inferences the system should perform. We describe a hybrid architecture, an assumption-based truth maintained blackboard system and its associated context management system which partitions the blackboard into a series of *views* containing the information relating to a particular design solution or task. The resulting system integrates consistency maintenance and the control of inference within the framework of an assumption-based truth maintenance system (ATMS).

INTRODUCTION

The problem of controlling inference is a common one in AI systems. It is particularly acute in AI-based design support systems, which to be effective must not only produce the right answer, but must also produce it at the right time. In this paper we discuss these problems in the context of a particular implementation, the Edinburgh Designer System (EDS), a design support system for mechanical design problems. We present an approach to the control problem which utilises the *context* of a task to determine which inferences the system should perform. We describe a hybrid architecture, an assumption-based truth maintained blackboard system and its associated context management system which partitions the blackboard into a series of *views* containing the information relating to a particular design solution or task. The resulting system integrates consistency maintenance and the control of inference within the framework of an assumption-based truth maintenance system (ATMS).

In the following sections we briefly outline the major characteristics of the design process and how these have influenced the architecture of EDS. The design and implementation of the assumption-based truth maintained blackboard system (ATMB) is then outlined

and the problem of controlling inference in multiple inconsistent contexts described. An assumption-based context management system which overcomes some of these difficulties is then described and an example of its use presented. Finally, we describe some of the limitations of this approach and argue that further progress will require significant extensions to the architecture of the ATMB.

THE NATURE OF DESIGN

At its most general level, a design problem arises from a desire to change the world we live in in some way. The desired change is expressed in terms of a set of objectives or requirements which the designed artefact must meet. These initial requirements are often only poorly defined and may be in conflict. They may be incomplete and/or inconsistent and they may be expressed in a way which suggests what turns out to be an inappropriate solution to the problem. As a result, a large part of the design process is devoted to discovering the nature and scope of the task set by the requirement description. Particular aspects of the problem may suggest certain features of solutions, but these solutions in turn create new and different problems. As a result design problems are full of uncertainties both about objectives and their relative priorities, and both priorities and objectives are likely to change as solutions emerge.

It is rare for any part of a designed artefact to serve only one purpose, and it is frequently necessary to devise a solution which satisfies a whole range of different requirements. In many cases the stated objectives are in direct conflict with one another and the designer cannot simply optimise one requirement without suffering losses elsewhere. Different trade-offs between the criteria result in a whole range of acceptable solutions, each likely to prove more or less satisfactory in different ways to different clients and users. It is the very inter-relatedness of all these factors which is the essence of design problems rather than the isolated factors themselves, and it is the structuring of relationships between these criteria that forms the basis for the design process (Lawson, 1980). The fundamental objective is therefore that of understanding the structure of the problem and analysing the inter-relationships between criteria to gain some insight into the relationship between any individual design decision and all of the other decisions which together define the solution.

Simon (1973) calls such problems 'ill-structured' and argues that any problem with a large base of potentially relevant knowledge falls into this category. Design problems are ill-structured in this sense in a number of respects. There is initially no definite criteria to test a solution, much less a formal process to apply the criteria. In addition the problem space cannot be completely defined due to a radical lack of knowledge. Also, while the set of alternative solutions may be given in a certain abstract sense, it is not given in the only sense that is practically relevant. As a result there can never be an exhaustive list of all the possible solutions to such problems. The design process cannot therefore be adequately characterised as a *search* process in which the task is essentially one of selection or optimisation over a completely defined space of alternatives.¹ This is not to deny that search forms part of the design process. However, it is not what characterises design as a distinct kind of intelligent

¹The term *search* is here used in the sense of (Charniak & McDermott, 1985). A search problem is characterised by an initial state and a goal-state description, which though necessarily complete (no further useful information can be added) and consistent (it does not contain contradictory information) does not necessarily define a state within the bounds of a particular search space.

behaviour.²

The designer's exploration of this structure begins with the initial formulation of the problem. To a large extent, a design problem has no inherent structure; it acquires structure as solutions are proposed and problems are reduced to sub-problems. In a very real sense the relationships between criteria can be seen as a function of the approach to design embodied in the investigation of possible solutions rather than as inherent in the problem itself. This initial formulation forms the basis of subsequent exploration of the problem. As possible solutions are constructed and developed they provide an increasingly detailed context against which to test the designer's hypotheses, and the evaluation of a proposal can result in the discovery of previously unrecognised relationships and criteria. In a sense later decisions are constrained by earlier decisions in that they are taken within the context of an existing partial solution, and each decision further limits the range of possible alternatives. Solutions to particular sub-problems are apt to be disturbed or undone at a later stage when new aspects are attended to and the considerations leading to the original solution are forgotten or not noticed. Such side effects accompany all complex design processes. As a result, while the final solution may satisfy all the requirements that are evoked when it is tested, it may violate some of the requirements that were imposed (and temporarily satisfied) at an earlier stage in the design. The designer may or may not be aware of these violations. Other appropriate design criteria may simply remain dormant, never having been evoked during the design process. The development of a design is thus constrained by what best fits the knowledge the designer has at that time.

The formulation of the problem at any stage is not final; rather it reflects the designer's current understanding of the problem. As the design progresses the designer learns more about possible problem and solution structures as new aspects of the solution become apparent and the inconsistencies inherent in the formulation of the problem are revealed. As a result, designers gain new insights into the problem (and the solution) which ultimately result in the formation of a new view; the problem and the solution are redefined. This process of exploration and redefinition continues until one or more of the following conditions is met (Bazjanac, 1974):

- (a) the incremental gain in knowledge has become insignificant and the understanding of the problem (and the solution) cannot change enough to warrant further redefinition (i.e. the designer has reached the limits of his or her understanding); or
- (b) the available resources have become exhausted.

There is no meaningful distinction between the analysis of problems and the synthesis of solutions in this process; problems and solutions are seen as emerging together rather than one logically following from the other. The problem is explored through a series of attempts to create solutions and understand their implications in terms of other criteria. Simon (1973) argues that this methodology of 'analysis-through-synthesis' is inherent in the limitations of the cognitive processes underlying the design activity. The designer comes to understand the critical relationships and possible forms as a solution evolves. Between generic solutions design is less a search for the best solution than an exploration of the compromises that give sufficient solutions. These explorations help the designer appreciate which requirements

²In making this distinction, we do not wish to distinguish between problems in which the search space (the space of possible designs) is incompletely defined and those in which the search space, although theoretically finite is sufficiently large as to make search impractical.

may be most readily achieved and those that may be neglected without loss. As part of this process, the designer learns which criterion values will achieve the design goals and how much variation of these values can be tolerated while still achieving acceptable performance. The designer also discovers the implications of achieving the current goal, and any other decisions required to make the attainment of these goals consistent with the existing solution (Logan & Smithers, 1989; Smithers et al., 1990).

Discovering more about the nature of the problem and the structure of the space of possible solutions is the most important part of this process. The fundamental objective becomes one of understanding the structure of the problem, with a major part of the effort in design being directed towards structuring problems and only a fraction of it devoted to solving them once they have been structured (Simon, 1970). The design process can be viewed more generally as a process of discovering information about problem structures that will ultimately be valuable in developing possible solutions.

AN ARCHITECTURE FOR DESIGN SUPPORT

In this section we describe the architecture of the Edinburgh Designer System (EDS), a design support system for mechanical engineering design developed as part of the Alvey large scale demonstrator 'Design to Product' (Smithers, 1987). A *design support system*, unlike a CAD system, attempts to actively assist the designer throughout the design process. Such systems are to be distinguished from conventional CAD systems which aid the execution of particular activities within the design process and automated design systems which attempt to replicate design behaviour in fully autonomous systems. EDS doesn't actually design anything — rather it attempts to support the designer in exploring the space of possible designs. This support is manifest at two distinct levels:

- (a) support for the overall control of the design process; and
- (b) support for the production of alternative design solutions.

These might loosely be described as *strategic* and *tactical* support. Strategic support is primarily concerned with the overall organisation of design process: the generation and maintenance of alternatives or solutions to particular sub-problems; the comparison of alternatives; the selection of the most promising candidates for further development; and the revision and refinement of the requirement description. Tactical support is concerned with the solution of particular problems: the selection of possible solution schemes and configurations; the assignment of values to design parameters; and the evaluation of the resulting partial design solutions. To the extent that the exploration of the space of possible designs is carried out through the development of possible solutions, the overall control of the design process is dependent on the production of alternative solutions. For example, the comparison of alternative proposals relies on there being some means of generating and evaluating alternative solutions.

The design process outlined above typically results in the generation of a large number of alternative designs or 'variants'. These alternatives may embody radically different approaches to the problem or they may be variations on a common theme or both. Each of these solutions may in turn be broken down into a set of simpler sub-problems, (for example, the design of an assembly may be reduced to the design of its constituent components) together with the problem of integrating the resulting part solutions, or it may be broken

down into a set of associated functions. A major task of any design support system is therefore one of complexity management — keeping track of the various alternative design proposals and the information required to solve particular problems, both as a record of the design process (which alternatives have been tried and what was learned) and to allow the designer to partition the problem appropriately (Logan, 1989).

At the level of individual design proposals the designer requires both information about the consequences of design decisions and assistance with the solution of problems. Central to the exploration process is the making of assumptions about design solutions or parts of design solutions and the derivation of the consequences of these assumptions in an attempt to understand their implications for other design criteria. Of particular concern are inconsistencies which arise in attempting to satisfy multiple goals, for example when the derived performance fails to meet the design requirements or when the proposed solution implies two or more values for a given design parameter. The system should therefore provide support for context specific inference based on design decisions, constraints and the requirements defining the design problem, together with physical laws, heuristics and other domain knowledge relating the parameters of the design. In addition the system should, where possible, provide assistance in solving particular design problems, drawing on the large amounts of knowledge encoded in design handbooks, codes of practice and in the expertise of individual designers.

In the remainder of this paper we describe the design and implementation of (part of) EDS (Smithers et al., 1990). EDS consists of four main functional elements which together embody the model of design support outlined above:

- (a) Consistency Maintenance
- (b) Context Management
- (c) Knowledge Representation
- (d) Inference

In EDS, the consistency maintenance and context management sub-systems provide strategic support, and the knowledge representation and inference sub-systems provide tactical support.

In what follows we shall concentrate primarily on the consistency maintenance, context management and inference sub-systems of EDS, which are implemented as an assumption-based truth maintained blackboard system (consistency maintenance and inference) described in the next section, and an assumption-based context management system (context management) described in section 6. Knowledge representation, the user interface and the overall design of the system are discussed in more detail elsewhere (Smithers et al., 1990).

THE ASSUMPTION-BASED TRUTH MAINTAINED BLACKBOARD

The production of a large number of alternative design solutions results in a major consistency maintenance problem. If the system is to effectively support the exploration activities of the designer, the various incompatible design alternatives must be considered in isolation. This, together with the need to retain alternatives for comparison and/or possible later development, led to the adoption of an assumption-based truth maintenance system (ATMS) as the basis of EDS (de Kleer, 1986). A truth (or reason) maintenance system records the dependencies

between sets of facts or data. In an assumption-based truth maintenance system a subset of these facts, called *assumptions*, are taken as primitive and everything else is derived from them. Each datum is associated with an ATMS *node*. A datum p is said to have a *justification* if there exists a set q_1, \dots, q_n of nodes (assumptions or data) from which it can be derived. The set q_1, \dots, q_n are termed the *antecedents* of the justification and p is termed the consequent. By tracing backwards through the supporting justifications, the ATMS identifies the set(s) of assumptions on which a datum ultimately depends. Such a set of assumptions is called an *environment*. The set of environments in which a datum is known to be derivable is called its *label*. A datum which has a non-empty label is said to have support, i.e. it can be consistently derived from the assumptions forming each of the environment(s) in its label. An environment in which an inconsistency can be derived is called a *nogood*. When an environment is discovered to be inconsistent, the ATMS restores consistency by deleting all environments which subsume the nogood from the labels of all datum nodes. Data which are only derivable in such an environment (i.e. they depend on an inconsistent set of assumptions) become unsupported. This process is known as truth maintenance or belief revision. Note that the ATMS simply maintains dependencies between data items — it has no knowledge of the contents of the nodes or assumptions. Each node is assigned a unique integer identifier called a *node-tag* which is used by the ATMS to refer to the node in environments and justifications. This results in a significant user interface problem, as there is no easy way for the user to identify an assumption or datum node without reference to its node-tag. It does not make sense to talk about “*the value of parameter p of instance i*” as p may have several values.

The ATMS forms the core of the system and all of the other system components are implemented using the facilities it provides. The ATMS builds and maintains a dynamic datastructure, the Design Description Document (DDD), and provides an interface between the contents of the DDD and the other sub-systems, passing out relevant pieces of information to them as required and incorporating new information which it receives from them into the dependency structure. Its central role in the system architecture has resulted in a number of extensions to the ‘conventional’ ATMS model which are discussed in more detail below.

At the tactical level a number of particular tasks within the overall exploration process can be identified for which partial support is possible. In our work on mechanical design, for example, we have identified algebraic equation solving, constraint manipulation, reasoning about spatial relationships, shape and space occupancy, and reasoning about relational information as common to several different activities in the design process. Support for particular design tasks is provided by a series of general purpose support systems — forward chaining or data-driven inference engines which attempt to infer by means of constraint satisfaction the consequences of the designer’s decisions. In general the inference engines derive necessary consequences of what is currently known about the design and can be allowed to proceed relatively unhampered by the designer, with the sole constraint that they should refrain from re-deriving information which is already present in the DDD. Insofar as the logic implemented by the system’s inferential capabilities is sound, such inferences cannot be wrong. The system may draw such conclusions at the wrong time or draw so many irrelevant conclusions it is effectively unusable, but this is a different problem.³

³For example, the editor used to prepare this document interprets “ as either ‘ ‘ or ‘ ’ depending on context. However this is only a plausible inference; I may, as above have wanted a “. The editor also counts the number of words in the text; this is a necessary consequence of the particular arrangement of characters and spaces forming the text, even if, as in this case, it is irrelevant. Furthermore I have to ask for it.

Control of the interactions between these inference engines is in the style of a blackboard system (Hayes-Roth, 1985). The blackboard model of problem solving is a highly structured case of opportunistic problem solving. In this case the 'problem' is one of deriving all possible consequences of a set of design decisions. A blackboard system consists of three major components: a set of *knowledge sources*; a global data structure called the *blackboard*; and a *control strategy*. The problem solving strategy employed is to divide the problem into a number of loosely coupled sub-tasks, which roughly correspond to areas of specialisation within the task. The knowledge needed to solve the problem is partitioned into a series of knowledge sources (KSs) each of which performs a particular sub-task. The knowledge sources are kept separate and independent. The current state of the task and the results produced by the knowledge sources are recorded in a global database or blackboard. The knowledge sources use information on the blackboard to derive new information using algorithmic procedures or heuristic rules. Communication and interaction between the knowledge sources take place solely through the blackboard. The decision as to which knowledge source to apply in any particular situation is determined dynamically based on the current solution state (and in particular the latest additions to the blackboard) and on the ability of the knowledge sources to contribute to the developing solution. In EDS the inference engines act as knowledge sources deriving consequences of the current design description represented in the DDD. The system maintains an agenda of Knowledge Source Activation Records (KSARs) which mark an intent to carry out a particular inference. The ATMS truth maintains the KSARs within the DDD on an equal footing with existing data and its justifications. This benefits the system by automatically removing flawed KSARs (KSARs which are no longer valid due to recent changes in the DDD) as they come to light as a result of other inferences and the operation of the ATMS. The agenda control mechanism therefore does not need to be vigilant on this matter. A similar arrangement can be found in (Jones & Millington, 1986).

Blackboard systems are usually designed and implemented as 'single context' problem solving systems in which the knowledge sources work together to construct one consistent solution. Truth maintaining the blackboard has therefore resulted in a number of departures from the conventional practice in blackboard systems, notably the absence of deletions (except for KSARs) or amendments as it is unclear how these fit into an assumption-based truth maintenance scheme. The implications of this decision for the interaction of the ATMS and the knowledge sources is discussed in more detail in the next section.

EDS represents domain knowledge in a structured knowledge-based called the Domain Knowledge Base (DKB). This contains definitions of domain objects, called *module classes*, related by *part_of* and *kind_of* relations. Each module class declares a set of parameters, variables and constraints which define a particular class or type of object. The creation of an instance of a module class results in datum nodes being created in the ATMS for each of the module's parameters, variables and constraints. Justifications link new data introduced into the DDD as a consequence of applying a knowledge source to some particular set of domain data already present in the DDD (in this case the assumption of an instance). This existing data is also pointed to by the justification. Note that a particular datum can be supported by more than one justification — for example if it can be derived in different ways or from different sets of assumptions.

Design proceeds by creating instances of module classes and assigning values to their

parameters to define one or more possible solutions.⁴ When the user makes an assumption one (or more) datum nodes are created in the DDD to hold the new information. As new information becomes available, it is examined by the knowledge sources to see if it, together with any information already in the DDD, can be used to make further inferences. If a knowledge source thinks it can make an inference, it generates a bid in the form of a Knowledge Source Activation Record which is scored and merged into the agenda. When a KSAR reaches the front of the agenda, it is executed and the results are claimed into the ATMS. This information may in turn form the basis for a new round of bids and this cycle continues until no executable KSARs remain in the agenda. As new values for parameters or bounds on them are assumed or derived, consistency checks are performed between constraints and values by the *valueConflict* KS. Conflicts result in the creation of a justification for the distinguished node (*false*) recording the fact that the assumptions involved are mutually inconsistent and causes the ATMS to partition the assumptions into mutually consistent sets. If there is no conflict, EDS marks this by justifying the datum (*consistent*) and proceeding as usual. The user is viewed as a knowledge source whose 'bids' are always processed first. This allows the system to follow several lines of reasoning as it attempts to infer the consequences of the user's design decisions, while still giving priority to user input. From this brief description, it is clear that a single assumption can result in the system doing a considerable amount of work. This approach, which derives from the idea that to infer all possible consequences of the user's assumptions and in particular any inconsistencies implicit in the design description is a way to actively support exploration, has led to a number of problems which are described in more detail in the next section.

The ATMS-Blackboard (ATMB) is implemented as a 'shell' and is capable of maintaining multiple blackboards. The current state of each blackboard is represented as an instance of a blackboard record. User-assignable fields within the blackboard record allow the behaviour of the system to be tailored to particular applications, for example through the addition of new knowledge sources and their associated classes. EDS is simply a set of assignments to the fields of a single blackboard which is used to maintain the DDD. The implementation of the ATMB as a shell has a number of implications for the design and operation of the system. For example, checking for ill-formed bids is the responsibility of the ATMB rather than the knowledge sources, allowing the rapid development and integration of new knowledge sources. Similarly, the user-extensible classification system facilitates the integration of knowledge sources based on 'new' classes of information.⁵ The system is implemented in a mixture of Pop11 and Prolog in the Poplog multi-language programming environment (Sloman & Hardy, 1983).

BEING ECONOMICAL WITH THE TRUTH

There are, however, a number of problems with the approach outlined above. Perhaps the most obvious is the computational cost involved. The desire to derive all of the necessary consequences of a design description results in the combinatorial explosion common to many forward chaining or data-driven systems: all logically valid inferences which can be made will ultimately be made. While the ATMS is effective in restricting inferences to those based on consistent premises, in general the consistency of a set of assumptions is too weak

⁴This is an oversimplification—the user can also define new parameters and constraints and assemble novel designs from existing modules.

⁵The design and implementation of the ATMB will be described in more detail in a future paper.

a criterion to determine the relevance of a possible inference and typically results in the derivation of a large amount of redundant information.

For example, assuming the values a and a' and b and b' for the parameters x and y ⁶

$$\begin{array}{ll} x = a & [1] \\ y = b & [2] \end{array} \quad \begin{array}{ll} x = a' & [3] \\ y = b' & [4] \end{array}$$

together with the constraint

$$(\forall \Phi) (\forall u v w) (\Phi(u, v) \wedge \Phi(u, w) \supset v = w)$$

results in four environments [1 2],[1 4],[3 2] and [3 4].⁷ Given a function $u = f(x, y)$, for example a constraint or knowledge source, it is possible to derive four values for u , one in each environment. Assuming an additional value for each of the parameters x and y , (e.g. $x = a''$ and $y = b''$) gives nine environments and therefore nine values for u . Alternatively, adding another two values for a new parameter to the description (e.g. $z = c$ and $z = c'$) results in the formation of eight environments and given a function $v = g(x, y, z)$ it is possible to derive eight values for v . The number of possible bindings is limited by the ATMS — KSSs can't fire on inconsistent antecedents. However, in the worst case where all values are inconsistent or (as above) only pairwise inconsistent, the number of environments is given by $\prod_{i=1}^m n_i$; where m is the number of parameters and n_i is the number of values for parameter i . This is exponential in the number of parameters and polynomial in the number of values.⁸

For a typical design problem many parameters will be required to describe a design proposal, each of which will typically take a number of different values as the designer explores the space of possible designs. While the knowledge sources and constraints are selected with a view to the production of relevant information, the large number of environments generated by the ATMS means many of the inferences produced by the system are redundant in the sense that they are irrelevant to the task in hand. Put another way, the system lacks any understanding of what the designer is trying to do or of the information relevant to a particular design task; it has no concept of the *context* of a particular design task or operation. The notion of context is implicitly equated with that of a ‘maximal’ environment.⁹ All such contexts are equivalent, both in the sense that all possible contexts will eventually be produced by the ATMS and in that the system processes all the resulting contexts in parallel, deriving all of the possible inferences in each context.

Returning again to the example above, we can re-interpret the assumptions 1–4 as two alternative solutions to the same design problem. The inconsistencies between the assignments are in a sense irrelevant. The designer does not care that they are inconsistent (if in fact they are) and the system should not pursue inferences based on their union. Whether two sets of assumptions should be considered disjoint in this sense is itself dependent on the current context. At some point in the future the designer may wish to consider amalgamating these two alternatives, at which point their consistency or otherwise does become an issue.

⁶In reality this is a binary relation $p(i, v)$ — the parameter p of instance i has value v , where i is an instance of a module class and v is a constant of the appropriate type.

⁷Numbers in square brackets, e.g. [1], are the node-tags of the assumptions and denote the environment in which the datum has support, in this case the assumption itself.

⁸This is something of an oversimplification. ATMS label computation is also worst case exponential in the number of assumptions even though a satisfying assignment can be found in linear time (Provan, 1990). However we shall only consider the operation of the blackboard in the remainder of this paper.

⁹Two designs are considered to be alternatives just in case it is possible to derive an inconsistency between them.

Distinguishing between those assumptions which should be considered part of the current proposal and those which mark the creation of a new proposal is often difficult. Clearly two proposals based on radically different approaches constitute distinct alternatives. However attempting to formalise this intuition within the framework of the ATMB is not straightforward. While it seems unproblematic to interpret multiple inconsistent assignments to a parameter as denoting alternative proposals, there is an important sense in which they may still be seen as the ‘same’ proposal. For example, when attempting to repair an existing design which has failed to meet some problem requirement, the introduction of new values for parameters can best be understood as replacements for the existing values. While this may require consistency maintenance, it does not necessarily imply a change of context. In general, consistency cannot be interpreted as determining context — a design proposal will be inconsistent for much of its history as the designer attempts with varying success to reconcile the various conflicting requirements. As a result, a particular partial description will frequently imply two or more different values for the same parameter. We therefore cannot safely distinguish these on the basis of derived or assumed values. Often whether a value is derived or assumed is a consequence of a particular problem solving strategy or of the design problem itself.

This problem arises, at least in part, because the basic mechanism — the addition of assumptions to the DDD — is used to model two conceptually distinct operations: the replacement of one value with another value (in attempting to repair a design); and the introduction of a new value (in creating a new design proposal). While replacement preserves the existing context, introduction implicitly creates a new context. However, as noted in the previous section, the introduction of deletion or amendment of assumptions in an attempt to model the revision of an existing context, results in the DDD no longer accurately reflecting the history of the design process. Moreover it fails to address the problem of context definition and consequently does nothing to control the interaction of the inconsistent contexts that remain. The other obvious approach involves the creation of a new instance of the module class to represent each of the alternatives under consideration (see footnote 6). This prevents knowledge sources reasoning across contexts, but the overhead of duplicating all the parameter values of an instance simply to allow the designer to investigate the implications of changing the value of a single parameter is considerable. In addition, the resulting proliferation of instances tends to obscure the differences between variants. Minor variants and radically different ways of solving the problem are represented in the same way — as (collections of) instances of module classes.

However, even if it were possible to eliminate those environments which do not correspond to design solutions from consideration, the system would still waste the greater part of its resources. The blackboard control strategy attempts to pursue all environments in parallel, scheduling a knowledge source for execution whenever its antecedents jointly hold in some context. This leads to much wasted effort, as at any given time it is likely that many of the alternative designs represented within the Design Description Document have been abandoned by the designer — either they suffer from some fatal flaw or inconsistency; or although they violate no constraints they are inferior to the available alternatives; or they were generated simply to better understand the the structure of the problem and having served their purpose are no longer of interest. Yet the system pursues inferences based on all these alternatives in parallel.

Forbus and de Kleer (1988) have argued that the basic ATMS is ill-equipped to cope with domains such as design where the ‘search space’ is vastly larger then the subset which needs

to be explored to find an acceptable solution. In (Forbus & Kleer, 1988) they propose a new strategy, the *implied-by* strategy, for building ATMS-based problem-solvers which can efficiently explore large search spaces.¹⁰ With each problem or goal they associate a *focus environment* containing the assumptions relevant to solving the problem. When the problem solver begins work on a particular problem it notifies the ATMS that the environment associated with that problem is now in focus. Only inferences which are *implied-by* the current focus, i.e. which are based on information implied by the focus environment, are performed. All new information is therefore a consequence of the focus and hence likely to be relevant. Every time the problem solver switches to a new problem, the focus environment is changed accordingly. In this way, the system concentrates on one problem at a time.

Forbus and de Kleer present experimental results from a number of example problems which demonstrate the advantages of the implied-by strategy. However while the use of a focus environment can be effective in directing the efforts of an automated problem solving system, it is inappropriate in the context of a support system where the user is responsible for determining the context of the current task. In an automated problem solving system, the problem solver determines the order in which problems are to be addressed and hence what the current focus should be on the basis of its control strategy. When a problem is solved or abandoned, the problem solver must notify the ATMS of the assumptions which form the basis of the new focus. While this is plausible for a problem solver searching an and-or tree where the relationship between the operators and the assumptions are well defined, human designers often work on several sub-problems or alternative solutions ‘in parallel’ with no obvious task ordering. It is unrealistic to expect the designer to specify those assumptions relevant to a task every time they switch tasks. Moreover, it is only possible to focus on an environment, i.e. on some consistent *part* of what is typically an inconsistent design solution, effectively precluding any consideration of a solution as a whole. These difficulties suggest that the notion of a focus environment cannot serve as the basis of a design context. While the focus mechanism is extremely flexible, the control it provides is at too low level for use by a human designer.

ASSUMPTION-BASED CONTEXT MANAGEMENT

We have therefore adopted a different approach which generalises that proposed by Forbus and de Kleer in two ways:

- (a) it provides support for multiple contexts;¹¹ and
- (b) contexts need not be consistent.

In this approach the Design Description Document is partitioned into a number of contexts or *views*.¹² These contexts need not be alternative designs, rather they can be any collection of assumptions, for example those resulting from a particular problem decomposition or the

¹⁰A similar approach can be found the ‘worlds’ mechanism of the KEE knowledge-based system building tool (Filman, 1988) and elsewhere.

¹¹An earlier attempt at context management in EDS — the *focus set* — concentrated on the control of inference and did not address the problem of multiple disjoint contexts.

¹²Note that this differs from de Kleer’s terminology (de Kleer, 1984), where *context* is taken to mean the set of data derivable from an environment. Our use of the term is closer to that of Martins and Shapiro (1986). In their terminology, a view is a *belief set*.

information required for a particular design task. In the remainder of this paper we describe the implementation of an Assumption-Based Context Management System (ACMS) based on these ideas. In this and the following sections we describe the operation of the system and its relationship to the ATMB. In the conclusion we discuss some of the problems of the current implementation and identify areas for future research.

We begin by defining the notion of *context* in more detail and outlining the relationship between contexts and the ATMS. The Design Description Document consists of a monotonically increasing collection of assumptions and inferences from these assumptions stored as a series of nodes and their associated justifications. The set of all items in the DDD which have support is termed the ‘supported set’. A ‘view’ is an abstraction of the supported set — a collection of supported nodes together with their associated justifications and KSARs which form the context for a particular task or tasks. A view is defined by specifying the assumptions it contains. These assumption are termed the ‘assumption base’ of the view. The *extension* of a view is everything which can be derived from its assumption base. A datum, justification, KSAR or subsumption link is said to be a member of a view if it is either a member of the assumption base or a member of the view’s extension. More precisely, an item is a member of a view if the assumption base of the view subsumes one or more of the environments in the item’s label.

Views are defined relative to the the set of currently supported assumptions or ‘universal assumption base’, \mathcal{V} . The members of a view’s assumption base are specified by its *defining abstraction*. The definition of a view can be either intentional or extensional depending on whether its defining abstraction is open or closed. An intentional definition is a predication P defining a set of assumptions $\{x : Px\}$ where x ranges over \mathcal{V} and P is a boolean test defined on node labels or their contents. For example, the abstract

`consistentWith(1)`

is true for all assumptions which are consistent with assumption 1. An extensional definition is simply a list of assumptions $\{a_1, \dots, a_n\}$. For example the abstract

`assumps(1, 2, 42)`

defines the set consisting of of assumptions 1, 2 and 42. Unlike an open abstract which produces different results at different times, a closed abstract always returns the same set of assumptions. A small number of system-defined tests are provided by the ACMS. More complex view definitions can be constructed from these primitive definitions using the set operators union (\cup), intersection (\cap) and set-difference (\setminus). For example the abstract

`assumps(1, 2, 42) \cup consistentWith(101)`

defines the set consisting of assumptions 1, 2 and 42 together with those assumptions which are consistent with assumption 101. Since the contents of a node never change, the content of a view in terms of its assumptions and inferences is completely determined by its definition.

The *current* assumption base of a view is found by evaluating its defining abstraction relative to the universal assumption base. The evaluation of an (open) abstract α is written $*\alpha$ and returns the assumptions for which it is currently true, i.e. $*\alpha = \{x : \alpha(x)\}$. For example

`* consistentWith(1)`

will return those assumptions which are currently consistent with assumption 1. If there are no assumptions which are consistent with assumption 1 this is the empty set \emptyset . Evaluating a closed abstract simply returns the corresponding list of assumptions, i.e. $*\beta = \beta$ for any closed abstract β . Evaluation effectively freezes the definition of a view relative to the current state of the DDD. Assumptions or inferences introduced into the DDD after the evaluation of the view's defining abstraction do not form part of the view's definition. For example, those assumptions added to the DDD after evaluating the abstract $*\text{consistentWith}(1)$ which are consistent with assumption 1 will not be included in the view's assumption base. Similarly (since the logic implemented by the knowledge sources is not complete), assumptions which are subsequently discovered to be inconsistent with assumption 1 will not be removed from the assumption base, although the ATMS will partition environments to maintain consistency.¹³

The set of views forms a tree. Each view is assigned a name corresponding to a unique node in the tree. A 'view-name' is a sequence of the form

name · view-name

Within this structure there are two distinguished views called 'univ' (the universal or 'root' view) and 'nil' (the empty view) corresponding to the set of all supported assumptions \mathcal{V} and the empty set \emptyset respectively. The name of a view specifies its location within the tree relative to the root node. For example

univ · widget

denotes the view *widget* which is a child of the universal view. As a syntactic convenience, any view name beginning with the view name separator '·' is assumed to be an absolute view name. For example

·assembly · sub-assembly · component-1

names the view *component-1* which is a child of the view *·assembly · sub-assembly*. All assumptions are by definition members of the universal view and all views are defined relative to it.

The organisation of views into a tree allows 'relative naming' of views. For example, the components of an assembly can have the same name in different assemblies. To refer to the same part in *assembly-1* and *assembly-2* we could write

·assembly-1 · part

and

·assembly-2 · part

The ACMS provides three basic operations for creating and manipulating views:

- (a) *define <view>(<abstract>):* (re)define the view *view* to be an abstraction of the view(s) named in *abstract* or the universal view.

¹³This leads to counterintuitive behaviour in some circumstances when the view is defined by an open abstract which is subsequently discovered to be inconsistent. Datum nodes which are justified in the view only by the inconsistent assumption(s) will continue to be members of the view even though these assumptions should be excluded from the view's assumption base. This is rectified the next time the abstract is evaluated.

- (b) `copy <view1><view2>`: copy the sub-tree of views rooted at the view `view1` to `view2`, resolving all references to views below `view1`.
- (c) `delete <view>`: delete the sub-tree of views rooted at the view `view`. Note that any assumptions which were members of the view are not deleted.

View definition is ‘permissive’. To define a view it is only necessary that the name of the view and of any views referenced in its defining abstraction be ‘well-formed’, (i.e. they do not attempt to reference beyond the root view). However they need not be defined or be a child of an existing view: any nonexistent views referenced in the defining abstraction will be automatically created and marked as undefined. Note that the definition of a view is independent of its position in the view structure. The name of a view simply provides a convenient way to refer to its defining abstraction. Views are defined as abstractions of other views. The argument to a primitive abstract is an assumption or a set of assumptions. In particular, it can be the name of a view. Since a view name ultimately reduces to the abstraction defining its assumption base, it is straightforward to define a view in terms of another view. Consequently, views *inherit* from any views which form part of their definition. As the contents of a view change so too do those views defined in terms of it. By defining views in terms of other views it is possible to build complex inheritance structures. While the view structure is a tree, the inheritance structure is a directed acyclic graph.

A problem with defining views in this way is that the abstractions are not guaranteed to uniquely define the extension of a view in the face of future additions to the universal assumption base. This could result in extraneous assumptions becoming members of the assumption base of a view after the view has been defined. This is often undesirable, for example in creating variants of a particular design, or when one wishes to define a view in terms of those assumptions which currently match a particular abstraction.¹⁴ To avoid this, the user can force the evaluation of part or all of a view’s defining abstraction to obtain its current extension which is stored as the view’s definition. This has the effect of ‘freezing’ the definition of the view and breaking any inheritance links it has to other views. For example

```
define ·component-1a *·component-1
```

has the effect of copying the definition of the view `component-1` so that subsequent changes to the view do not result in changes to the new view. In a similar way, the expression

```
define ·view *α
```

evaluates the abstraction α and uses the resulting extension as the definition of the new view.

The `copy` operation is a specialisation of `define` which simplifies the creation of alternative designs. `copy <view1><view2>` copies the sub-tree of views rooted at the view `view1` to `view2`, resolving all references to views below `view1`. References to views outwith the tree rooted at `view1` are preserved; references to views below `view1` (and the corresponding definitions) are copied to the new tree below `view2`. For example if we have a view `·view-1` defined in terms of (and therefore inheriting from) its children `·view-1·view-2` and `·view-1·view-3` and in terms of some standard component `·component`, then copying `·view-1` to `·view-1a` creates two new views `·view-1a·view-2` and `·view-1a·view-3` with the same definitions as `·view-1·view-2` and `·view-1·view-3`. The link to `·component` is preserved. This facilitates

¹⁴On the other hand, this may be useful in allowing the user to collect all nodes matching a particular abstraction in a view for comparison purposes.

the creation of ‘variants’, and when combined with evaluation and/or redefinition can be used to efficiently explore the implications of alternative parameter values.

The third operation provided by the ACMS is deletion. Although assumptions are never deleted it can occasionally be useful to delete parts of the view structure. Deleting a view deletes the view and its children (if any) from the view structure. Any view whose defining abstraction references a deleted view is marked as undefined but is otherwise unchanged. In particular, it continues to reference the now non-existent view. Any view defined in terms of an undefined view is also marked as undefined. Together with copy, evaluation and redefinition, this provides a means of rapidly modifying the view structure. For example, the designer can delete the subtree of views representing part of a solution and replace it with an alternative solution by substituting a copy of another part.

At any given time exactly one view is selected as the *current* view. Only those items which are members of the current view are ‘visible’ to the knowledge sources and hence can form the basis of further inferences. Potential inferences which could be performed in other views — for example in views which inherit from the the current view — are ignored. Assumptions made within the current view are automatically added to its defining abstraction and hence to its assumption base. This may in turn result in new inferences being made or previously derived inferences gaining support within the view. A view’s defining abstraction (and hence the assumption base it specifies) can be thought of as consisting of two parts, an evaluated part and an unevaluated part. The evaluated part denotes the ‘direct’ members of the view, i.e. those assumptions assumed to be members of the view either directly (their tags form part of the view definition) or as a consequence of the evaluation of another abstract, or assumptions made while in the current view. The unevaluated part can be thought of as representing that part of the assumption base which is inherited from other views, and includes unevaluated references to view names, compound expressions constructed using such names and any open abstracts involving datum, node-class or label expressions.

When the user changes view, the system must bring the new current view up to date by performing any inferences made possible by information (assumptions or inferences) added to the DDD since the view was last visited. The defining abstraction of the view is re-evaluated and the resulting assumption base is used to rebuild the view’s extension. If the view is defined in terms of other views, their defining abstractions must first be evaluated and so on recursively. This effectively recomputes the inheritance relation between the current view and the views forming its definition. New bids are posted for any work based either partially or wholly on ‘new’ information. Any pending KSARs which were not processed the last time the view was visited (and which have not subsequently been executed in other views) are also added to the agenda. Switching views can therefore be used as a simple but effective means of process control. By changing context, the designer can force the system to redirect its efforts; as different assumptions become visible, so different knowledge sources are activated and different kinds of inferences are performed.¹⁵

The system goes to some lengths to minimise the cost of switching views by maintaining a record of when each view’s defining abstraction was last evaluated, and by attempting to ensure that the KSARs produced on switching views have not been processed before. This recognises that out of any collection of contexts or variants, many are effectively ‘dead’ in that they will never be considered again. Even in the worst case, in which the designer elects

¹⁵Note that while inference is confined to the current view, the operation of the ATMS is not. It would be possible to modify the ATMS to limit both inference and label propagation to the current view, see for example (Dressler & Farquhar, 1990).

to visit all the views, the total amount of processing is still likely to be less than if the same assertions were made in the universal view, as the interactions between the assumptions are controlled. As the extension of a view is rebuilt only when the view is visited, the overhead of a view is limited to the small amount of memory required to store the view's definition. Given the relatively low cost of creating views and the fact that each assumption is stored only once there is little penalty in creating as many views as necessary to solve a particular problem.

USING VIEWS TO MODEL PROBLEM STRUCTURE

The facilities provided by the ACMS can be used to model the structure of the design problem. There are many ways in which this might be done. In this section we describe one of the simplest, an assembly or *part-of* hierarchy, which decomposes the problem of designing an object into a series of sub-problems corresponding to the design of its parts. This type of design is common in reasonably well understood domains, and is typical of the kind of 'variant design' EDS was developed to support. However, as we shall see, the views system can also be used to support exploration in less well understood domains.

For the sake of concreteness, we will assume that the task is to design an assembly consisting of two components, each of which comprises a number of parts. The exact nature of the assembly is not important, for example it could be the barrel and plunger assembly of a fuel injection pump. We further assume that the DKB contains an appropriate module class describing the assembly to be designed in terms of its parameters, variables and constraints etc. and that the designer has created an instance (assumption 1) of this module class, say *a1*. The designer can now proceed to create a view for each of the parts and components which form the assembly. For example, the designer might define the views *part-1* and *part-2* as

```
define ·assembly·component-1·part-1 assumps (1)
define ·assembly·component-1·part-2 assumps (1)
```

which defines the assumption base of the views *part-1* and *part-2* to be assumption 1. This also results in the creation of two undefined views, *assembly* and *component-1*. Such views can be used as placeholders for future work. The designer can now define *component-1* as the union of *part-1* and *part-2*. The views *component-2*, *assembly* etc. can be defined in a similar way to give the assembly hierarchy depicted in Figure 1.

The intention is that each view should contain only those parameter values which are appropriate to the design of the part or component. For example, the view *part-1* should contain all assumptions relating to the design of part 1, *part-2* should contain all the assumptions relating to the design of part 2 etc. The view *component-1* inherits all assumptions (and any inferences based on these assumptions) from the views *part-1* and *part-2*. Similarly, *assembly* inherits all assumptions made in the views *component-1* and *component-2* (and any inferences based on them) and hence from the 'part' views. Values of parameters and variables are propagated to higher levels whenever a higher-level view becomes the current view. This organisation is appropriate to bottom up design. For top down design, the dependency relationships between views can be reversed, so that assumptions are inherited down the tree, i.e. parameter values relating to the assembly are inherited by the components and component values are inherited by the parts.¹⁶ By making one of the part views the current

¹⁶Note that it is not possible to inherit in both directions, as view definitions must be acyclic. The same effect can be obtained by working in the universal view, but the advantages of the views system are lost.

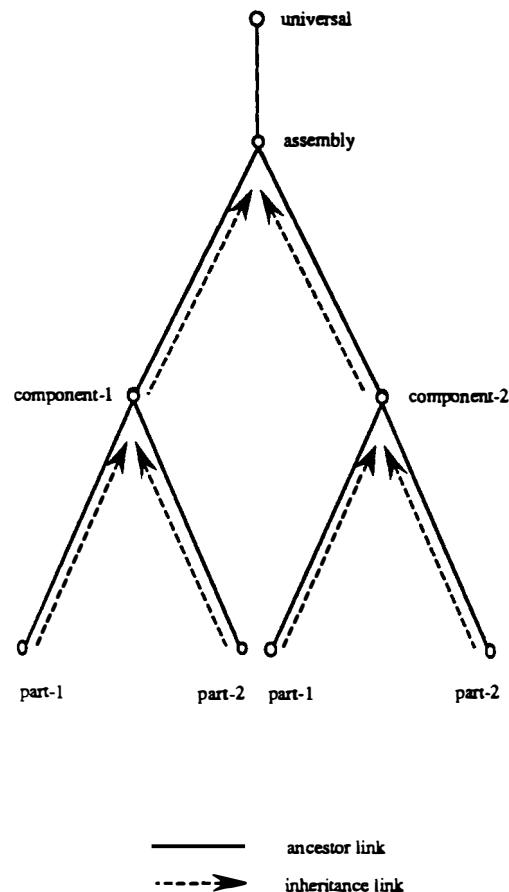


Figure 1: An assembly hierarchy

view, the designer can pursue the problems of designing each of the parts in isolation without worrying about their interactions.

As the design proceeds the consequences of the designer's assumptions are derived by the support systems. Such derived information typically relates to the predicted performance of the designed artefact (including any constraints violated by the proposed design) and any parameter values which can be inferred from the designer's assumptions or their consequences and the constraints linking these values. If inconsistencies are discovered or the design is found to be unsatisfactory (as is typically the case), the designer can attempt to modify the offending parameter values using information on which assumptions are jointly inconsistent provided by the ATMS. Even if the design fails to violate any constraints the designer may elect to pursue several different designs in parallel in an attempt to determine which gives the best overall performance, or to determine the sensitivity of the derived performance to the values of the design parameters. This will typically involve trying several alternative values for parameters until the constraints are satisfied or the relative performance of the various alternatives is understood. Multiple assignments to parameters (giving rise to multiple alternative solutions) and their interactions are handled automatically by the ATMS as are inconsistencies between parameter values and any assumed constraints.

When an inconsistency arises the designer has several options. The designer can ignore the inconsistency and continue to pursue the development of the inconsistent design based on what can be coherently derived. The opportunistic nature of the blackboard control strategy means that whatever can be consistently derived within the current view will be derived. This approach may be appropriate when, for example, the inconsistency is considered minor or peripheral and the main interest lies with the consequences of some (consistent) set of assumptions which are considered central to the proposed solution. Alternatively, the designer can continue to work in the current view until the inconsistency is resolved (which may involve assigning new values to parameters or relaxing constraints, i.e. changing the requirements or adopting a different approach) and then redefine the view to filter out unwanted assumptions. However, if there are several different ways of 'patching' the design to restore consistency which are to be investigated in parallel, it may be more appropriate to create a view for each of the alternative solutions. For example, a copy of the view *part-1* omitting, say, assumption 42 which has been identified as giving rise to an inconsistency, can be defined as

```
define ·component-1·part-1a *·(·component-1·part-1 \ assumps(42))
```

Similarly, to retain assumption 42 and omit those assumptions with which it is inconsistent we write

```
define ·component-1·part-1b *·(·component-1·part-1 ∩ consistentWith(42))
```

This effectively partitions the original inconsistent view, *part-1* about the inconsistent assumption into two new views, *part-1a* and *part-1b* both of which are consistent. Such variant views can either be used as a 'scratchpad' for rough working, merging the results back into the main view structure by redefining the part view as the chosen variant (*part-1* = * *part-1x*), or they can form part of the final design record, in which case the component view can be redefined to inherit from the appropriate part view (*component-1* = *part-1x* ∪ *part-2*).¹⁷

¹⁷The problem of when a modification or revision should entail the creation of a new view is left to the designer. The views system is intended to facilitate the exploration of alternative solutions and any predefined strategy would simply constrain the exploration process. This is discussed in more detail in the next section.

At any time the designer is free to move to the component level to explore the implications of the current part design by allowing the knowledge sources to 'see' the definition of the parts and any parameter values associated with the component. Note that the design of the part need not be complete for the system to be able to derive useful consequences. As with inconsistent views, whatever can be derived in any particular context will be derived. Any inconsistencies are again handled by the ATMS. If an inconsistency arises either between the parts or between the parts and values assumed for component parameters the designer again has the option of creating additional views. The designer could create a new part view as described above and redefine the component view to inherit from the new part. However in this case the situation is complicated by the inheritance relationship between the part level and the component level. If the problem is at the part level but the inconsistency is only apparent at the component level (for example if the parts are inconsistent with each other), the designer may elect to work at either the component or part level, although the problem is perhaps more naturally resolved at the component level as there is no way to make an assumption in a view other than the current view. The designer can create a new component either by copying the definition of the current view

```
define ·component-1a *(·component-1 \ assumps(42))
```

where, as before, assumption 42 has been identified as inconsistent, or by *copying* the subtree of views below *component-1*

```
copy ·component-1 ·component-1a
```

which results in the creation of a new component view *·component-1a* together with two new part views *·component-1a ·part-1* and *·component-1a ·part-2* corresponding to *·component-1 ·part-1* and *·component-1 ·part-2* respectively. In the former case the designer works directly at the component level, effectively 'masking' the parameter values inherited from the part level (in this case assumption 42). In the latter the designer creates a new component structure. Which is more appropriate depends on the nature of the problem. If the designer is pursuing alternative component designs (perhaps using different types of component or part) then creating several component views each with its own parts would appear to be more natural. On the other hand if the designer is simply interested in 'patching' an unsatisfactory design by modifying its parts, then working in the current view or a copy of it (possibly leading to a redefinition of the relevant part view when the problem is resolved) would seem more appropriate.

The view structure described above can be elaborated in a number of ways. There is no necessary connection between the definition of a view and its position in the view structure. For example, to study the interaction of various design criteria we can redefine the views shown in Figure 1 in terms of design *tasks* or the kinds of inferences appropriate to some task, decomposing each of the views in Figure 1 into a number of task-based views as shown in Figure 2. In this new structure assumptions about particular aspects of an instance are made in the appropriate view. For example, *component-1* inherits its shape (and hence the shapes of its parts) from the view *component-1-shape* and its cost from the view *component-1-cost*. The designer can concentrate on a particular aspect of the design by switching to the appropriate task hierarchy. The implications of decisions about the shape of the parts or components for other criteria can be investigated by switching to the assembly hierarchy or in various other views created for the purpose (such as *shape-and-cost*). Alternatively, the designer can build several different part-of hierarchies each corresponding to a different way

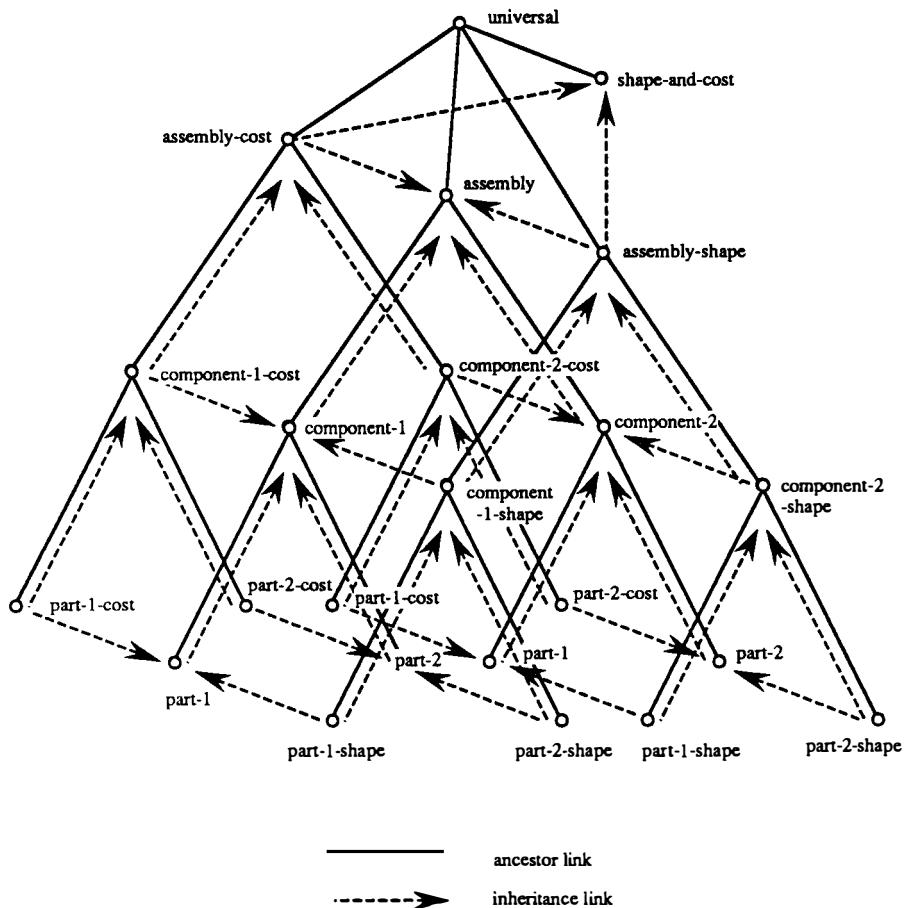


Figure 2: A task hierarchy

of conceptualising the design problem. Alternative conceptualisations can be tied together at key points by constraints equating parameter values in one conceptualisation with the corresponding parameter values in another conceptualisation.

The example described above presents an oversimplified view of the design process.¹⁸ However it serves to illustrate the the flexibility of the system. The designer can pursue the development of the design without worrying about inconsistencies except insofar as they indicate the shortcomings of a proposed solution. The ATMS ensures that only valid inferences are drawn from the current design description by automatically detecting inconsistent sets of assumptions and violated constraints and partitioning the design description to restore consistency. The views system provides a level of organisation above that of the dependencies maintained by the ATMS which reflects the designer's interests and goals. Whereas the ATMS dependency structure models the 'fine' structure of the problem, views capture the 'broad' structure. They provide a simple but effective means of process control, allowing the designer to focus the system's attention on a particular part of the problem or a particular kind of inference. The designer is free to create as many views as are required to represent and explore the structure of the space of possible designs. However, it must be emphasised that while this is the intended interpretation of the view structure, the designer is free to pursue the design in any order and assign values to any parameter which has support in the current view. For example, no attempt is made to ensure that only assumptions relating to the parameters of part 1 are made in the view *part-1* (a fact which was exploited in resolving the inconsistencies at the component level as described above) or that assumptions relating to other parts are not made in the current view.

CONCLUSION

The combined ATMB and assumption-based context management system described above forms the basis of the current version of the Edinburgh Designer System and has proved to be a useful tool.¹⁹ However the experience of building EDS has highlighted some of the shortcomings of the model of design support which the system embodies and we briefly summarise these below. While in the main these are not problems with the ATMB as such, it seems likely that significant revisions to the architecture of the ATMB will be necessary if they are to be overcome.

The idea that the designer is best served by deriving all the consequences of the design description begins to break down for design problems with large numbers of parameters. The original motivation for this approach was the desire to derive all the inconsistencies inherent in the design description. Designs are typically inconsistent for much of their history. However, the system lacks any concept of the importance of a failure to achieve a particular design goal. There is no way in the current scheme to distinguish between what might be termed trivial inconsistencies due to minor conflicts between design parameters and the 'radical inconsistencies' which are, at least initially, of greater interest to the designer, and which may completely invalidate an approach to the problem. At present if a set of assumptions prove to be inconsistent no further inferences are possible in any environment which subsumes these

¹⁸No pun intended.

¹⁹EDS was used extensively by several of the collaborators in the Alvey large scale demonstrator project 'Design to Product' (DtP) and formed part of the final DtP demonstration (The DtP Consortium, 1991). It has also been used internally in studies of nuclear power systems design and option trading.

assumptions. This is untenable if the knowledge sources are interpreted as implementing different logics.

One solution to this problem is to include (a name for) the knowledge source itself in the antecedents of a justification. This information is currently available in the method 'slot' of the justification in the ATMS. However making explicit the dependency of a derivation on the knowledge source responsible for the inference simplifies the subsequent truth maintenance and allows the system to distinguish between different kinds of inconsistencies. For example, it becomes possible to distinguish between 'rounding errors' in the calculation of the same parameter by different methods and 'semantic' inconsistencies arising as a result of conflicting design decisions.²⁰

The notion of 'current context' has proved elusive. We do not fully understand the relationship between the designer's notion of context and the environments of the ATMS. The ACMS has proved partially successful in providing a framework for the exploration of the design problem. However the designer needs to exercise close supervision to ensure that the system's support is well directed. The indirect control of inference through context, while desirable in principle, is limited in the current implementation and it seems likely that further progress will require explicit meta-level reasoning about which inferences to perform. This in turn involves reasoning about knowledge sources as first-class objects (Smithers, 1989).

Both of these problems suggests that the knowledge sources themselves may profitably be viewed as first-class objects maintained by the ATMS. This facilitates reasoning about the knowledge sources for control purposes and renders them subject to both consistency maintenance and context management allowing further (indirect) controls on their use in inference. For example, the system can identify those knowledge sources which are mutually 'inconsistent', (in that they will always derive inconsistent results from the same assumptions) or those which are considered appropriate in a particular context. A logical extension of this would be to allow the system to reason about contexts as first-class objects — for example to infer from the current task what the current context should be. The detailed development of this idea is hampered by our current lack of understanding of how task and context are related in design and also by the technical difficulties associated with making views or other collections of assumptions first-class objects within the ATMS. However, such a self-referential system offers a significant increase in expressiveness, in allowing statements (and hence reasoning about) other statements or collections of statements, such as their utility in particular contexts or the completeness and consistency of design descriptions. It also provides a natural interpretation of knowledge sources as procedurally interpreted views. This area is currently the subject of active research.

ACKNOWLEDGMENTS

The development of the Edinburgh Designer System was partially funded by the UK Science and Engineering Research Council as part of the Alvey large scale demonstrator project Design to Product (DtoP), other parts of which are funded by GEC plc and Lucas Diesel Systems (a division of Lucas Automotive Ltd.) whose active involvement in our research through the DtoP project we gratefully acknowledge, together with that of the other DtoP collaborators at Leeds University Department of Mechanical Engineering and at Loughborough University

²⁰At present EDS handles rounding errors using an 'epsilon' value which is the same in all contexts, as without knowledge of the justification structure, the knowledge source responsible for detecting value conflicts cannot determine how the parameters were derived.

Department of Computing Studies, Department of Engineering Production, and the Human Factors in Advanced Technology research centre. Work on EDS is currently supported under SERC grant No. GR/F/6200.1.

We are grateful to Dave Corne and Nils Tomes who read an earlier draft of this paper and made many helpful comments, and to the other members of the AI in Design Research Programme at Edinburgh for many discussions during the development of these ideas. We are also grateful to Nils Tomes for invaluable assistance in producing the diagrams.

REFERENCES

- Bazjanac, V. (1974). "Architectural Design Theory: Models of the Design Process," in *Basic Questions of Design Theory*, W. R. Spillers, ed., North Holland, Amsterdam, 2-19.
- Charniak, E. & McDermott, D. (1985). *Introduction to Artificial Intelligence*, Addison-Wesley.
- Dressler, O. & Farquhar, A. (1990). "Putting the Problem Solver Back in the Driver's Seat: Contextual Control of the ATMS," in *ECAI-90 Workshop on Truth Maintenance Systems*.
- Filman, R. E. (1988). "Reasoning with Worlds and Truth Maintenance in a Knowledge-Based System Shell," *Communications of the ACM* 31, 382-401.
- Forbus, K. D. & Kleer, J. de (1988). "Focusing the ATMS," in *Proceedings of the Seventh National Conference on Artificial Intelligence*, American Association for Artificial Intelligence, 193-198.
- Hayes-Roth, B. (1985). "Blackboard architectures for control," *Journal of Artificial Intelligence* 26, 251-321.
- Jones, J. & Millington, M. (1986). "An Edinburgh Prolog Blackboard Shell," Edinburgh University, Department of Artificial Intelligence, DAI Research Paper No. 281.
- de Kleer, J. (1984). "Choices without backtracking," in *Proceedings of the National Conference on Artificial Intelligence*, Austin, Texas, 79-85.
- de Kleer, J. (1986). "An Assumption-based TMS," *Artificial Intelligence* 28, 127-162.
- Lawson, B. (1980). *How Designers Think*, Architectural Press, London.
- Logan, B. S. (1989). "Conceptualizing Design Knowledge," *Design Studies* 10, 188-195.
- Logan, B. S. & Smithers, T. (1989). "The Role of Prototypes in Creative Design," in *Preprints of the International Round-Table Conference: Modelling Creativity and Knowledge Based Creative Design*, Department of Architectural and Design Science, University of Sydney, Sydney, 233-248.
- Martins, J. P. & Shapiro, S. C. (1986). "Theoretical Foundations for Belief Revision," in *Theoretical Aspects of Reasoning about Knowledge*, Joseph Y. Halpern, ed., Proceedings of the 1986 Conference, Morgan Kaufmann, Los Altos, 383-398, Monterey, March 1986.
- Provan, G. M. (1990). "The Computational Complexity of Multiple-Context Truth Maintenance Systems," in *Proceedings of the 9th European Conference on Artificial Intelligence*, L. C. Aiello, ed., Pitman Publishing, London, England, 522-527.
- Simon, H. A. (1970). *The Sciences of the Artificial*, MIT Press.

- Simon, H. A. (1973). "The Structure of Ill Structured Problems," *Artificial Intelligence* 4, 181–201.
- Sloman, A. & Hardy, S. (1983). "Poplog: A Multi-Purpose Multi-Language Program Development Environment," *AISBQ* 47, 26–34.
- Smithers, T. (1987). "The Alvey Large Scale Demonstrator Project Design to Product," in *Artificial Intelligence in Manufacturing, Key to Integration*, T. Bernhold, ed., North Holland, Amsterdam, 251–261.
- Smithers, T. (1989). "Intelligent Control in AI-Based Design Support Systems," Department of Artificial Intelligence, Edinburgh University, Technical Report No. 423, Prepared for the Workshop on Research Directions for Artificial Intelligence in Design, Stanford University, March 1989..
- Smithers, T., Conkie, A., Doheny, J., Logan, B., Millington, K. & Tang, M. X. (1990). "Design as Intelligent Behaviour: An AI in Design Research Programme," *Artificial Intelligence in Engineering* 5, 78–109.
- The DtoP Consortium (1991). '*Design to Product' An Alvey Programme Large-Scale Demonstrator Project: Final Report to the Department of Trade and Industry and the Science and Engineering Research Council on Contract Number LD/004*', (Commercial in Confidence).

Automated belief revision in plausibility-driven design processes

S. Patel[†] and S. Dasgupta

Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette LA 70504 USA

Abstract *Truth Maintenance Systems* (TMSs) offer the most elaborate approach to the *belief revision* problem and are thus ideal candidates for automating belief revision during the course of *plausibility-driven* design evolution. However, the *theory of plausible design* (TPD) necessitates multi-valued belief propagation while conventional TMSs are two valued (IN/OUT) systems. Due to this mis-match, encoding *constraint dependency graphs* (CDGs) in conventional TMSs requires an exponential number of *justifications*. In this paper we propose a belief revision system for TPD called the theory of plausible design, belief revision system (TPD-BRS). TPD-BRS uses a modified version of the assumption based TMS label propagation algorithm in conjunction with a “label-interpreter” to efficiently support automated belief revision in TPD.

INTRODUCTION

Improving the means of designing artifacts is an important focus of Artificial Intelligence (AI) research (Simon, 82).¹ Typically, design processes involve making decisions that are of the following form:

- (a) Given a set of interacting goals or objectives, where the nature of the interactions is only known qualitatively or imprecisely, how should these goals be prioritized?
- (b) Given a number of alternative choices (concerning a component of the target system, say) that are approximately equivalent in terms of some characteristic function, performance, cost reliability, modifiability, etc., but exhibit different trade-offs, which of these choices should be made and how does one justify the choice?

Because of *bounded rationality* (Simon, 82) the ramifications of such decisions or a chain of such decisions cannot always be comprehended at the time the decisions are

[†] This research was performed at the Center for Advanced Computer Studies. First Authors Current Address: Lockheed Software Technology Center, Lockheed Palo Alto Research Laboratories, 3251 Hanover Street, Palo Alto, CA 94304-1191.

1. In this paper when we use “design” as a noun, it denotes an explicit description or representation of the target artifact in some symbolic language. This representation serves as a blueprint for either the physical implementation of the artifact or as an abstract implementation in some other, lower level symbolic language. When we use “design” as a verb, it refers to the intellectual processes involved in the development of such representations.

made. Thus a design at any stage of its development must necessarily be regarded as a *tentative proposal* parts of which may, at any later stage, have to be *retracted* or *revised*.

At any stage of its development, the design is a tentative or conjectural solution to the problem posed. The designer's task in each cycle of the evolutionary (design) process is to elaborate or revise the design or the requirements so as to establish or converge to a *fit* between the two. Since the sufficiency of a design at any stage can be determined solely with respect to the requirements prevailing at that stage, an integral part of each cycle of design evolution is the *critical testing* (for fit or misfit) of design against requirements and the subsequent elimination of a *misfit* by revising the design, the requirements, or both (Figure 1).²

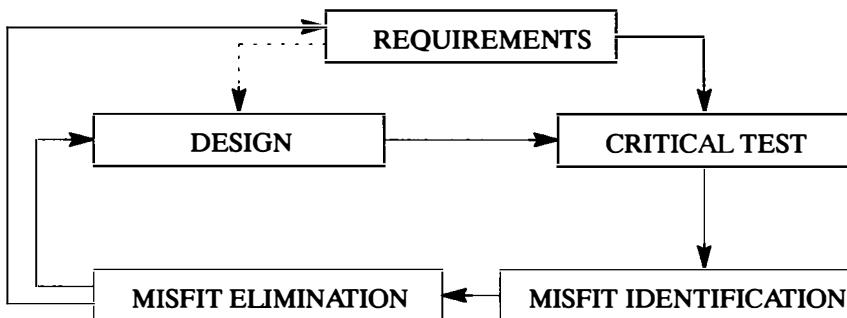


Figure 1: The Evolutionary Structure of Design Processes

If one accepts this evolutionary model, one must also agree that the believability or plausibility of a design depends on:

- (a) The *history* of the design — that is, to the recorded sequences of design decisions and the cause–effect relationships between design decisions; and
- (b) The nature of the *evidence* invoked by the designer to justify design decisions and/or establish the fact that a design feature satisfies a particular set of requirements.

The *theory of plausible design* (TPD), a relatively recent *design paradigm*, has been proposed in response to these issues (Aguero, 87; Aguero and Dasgupta, 87; Hooton, 87; Landry *et al.*, 88; Pedovsky *et al.*, 89; Patel, 90).³ A design method based on TPD — i.e., a *plausibility-driven* design method — (to be explained shortly) can not only be used to develop a design; it will also (1) generate and document explicitly, the cause–effect relationships between design decisions and thus provide an explanation of why a particular design evolved in the way it did; and (2) document explicitly, the nature and source of the evidence used by the designer in acquiring confidence in the design, thus establishing its plausibility.

This paper utilizes the principles of *truth maintenance systems* (TMSs) to facilitate the automation of belief revision. It shall be shown that a belief revision system for TPD

2. This is a highly abbreviated account of our evolutionary model of design. For a more detailed discussion see (Dasgupta, 89) which discusses in particular, the sources and nature of misfits that might emerge between designs and requirements, and provides empirical instances of evolutionary design.

3. We use the term “design paradigm” to denote an abstract prescriptive model of the design process that serves as a useful schema for constructing practical design methods, tools and procedures.

using conventional TMSs is very inefficient. As an alternative we shall present a system called TPD–BRS which is a novel belief revision system (based on a conventional TMS) that is particularly efficient for maintaining and revising beliefs about a plausibility–driven design. TPD–BRS, then, constitutes the main result of the present paper.

THE THEORY OF PLAUSIBLE DESIGNS

The central notion in TPD is that the design (of some system) consists of a specification of interacting *constraints* that must be satisfied by an implementation of the system. Note the particular sense in which the word “constraint” is used here: a constraint is any property or feature that the implementation must strive to meet — such properties are constraints on the implementation, not on the design itself. In TPD, the constraints collectively are the design. Constraints are packaged in the form of *plausibility statements*. A plausibility statement is a 5-tuple:

$$S = \langle C, A, R, V, P \rangle$$

where, C is the non-null set of constraints the plausibility of which is being established; A is the knowledge base which is used to assist in establishing C 's plausibility (this may include facts, theories, principles, heuristics as well as other constraints belonging to the current design); R denotes the relationships between constraints appearing in A that must be verified in order to establish the plausibility of C (R is expressed as a well-formed formula (wff) in the predicate calculus); V is the means employed to verify S ; and P is the *plausibility state* in which C will be placed upon successful verification of S .

Thus, S is to be interpreted as stating that C can be placed in the plausibility state P if it can be verified using the means cited in V that R is in plausibility state P .

Plausibility States

Intuitively, a plausible constraint is a constraint which we believe in because we have evidence that it exhibits some desirable characteristics. During the course of design, however, the evidence at our disposal may change. This change in evidence may, in turn, affect the plausibility of the constraint(s). At any stage k of the design, a constraint (say C) may be in one of four plausibility states: *assumed* when there is no evidence against C 's plausibility; *validated* when there is significant evidence in favor of and no evidence against the plausibility of C ; *refuted* when there is evidence against the plausibility of C ; and *undetermined* when it is not known what counts as evidence for or against the plausibility of C .

The plausibility state of C is determined by, and is identical to, the plausibility state of R . Thus,

$$\forall S \in \{ass, val, ref, und\} \quad S(k, C) = S(k, R) \quad (1)$$

A plausibility statement $S = \langle C, A, R, P, V \rangle$, is such that C may be placed in state $P \in \{val, ass, ref, und\}$ if R is shown to be *val*, *ass*, *ref*, or *und*.

Means of Verification

In TPD, the means of verification (cited in the V field of a plausibility statement S) that may be used to gather evidence in order to assign a constraint C to a plausibility state P may be any combination of the following:

- (i) Precise: the use of formal deductive logic
- (ii) Heuristic: approximate reasoning involving the use of heuristics
- (iii) Experimental: the use of experimental methods such as simulation, emulation, or the operation of prototypes
- (iv) Knowledge Bases: reference to the data, analyses, theories or results reported in the literature.

In essence, TPD allows a continuum of verification means as a basis for gathering evidence. Clearly, the stronger the evidence the greater the designer's confidence in the plausibility assigned to a constraint.

Belief Revision In TPD

The role of belief revision in TPD can be best understood by considering the general structure of a plausibility driven design process. This can be characterized as follows:

- (a) A design D_k at any stage k of its evolution is an organized set of constraints. Each constraint may be stated formally (in a design language or as a set of logical propositions) or in the semi-formal language of scientific discourse.
- (b) The plausibility of D_k is captured by a collection of plausibility statements. Since a design should not include any constraints that have been *refuted*, the plausibility state of each constraint in D_k will be in the *assumed*, *validated*, or *undetermined* state.
- (c) The constraints in D_k may have dependencies among them. Let $S_i = \langle C_i, A_i, R_i, V_i, P_i \rangle$ be the plausibility statement for C_i and let C_j be a constraint appearing in R_i . This means that the plausibility state of C_i depends (at least in part) on the plausibility state of C_j . Dependencies between constraints within a design D_k is explicitly depicted by a directed acyclic graph termed a *constraint dependency graph* (CDG) (see Example 1). The nodes of a CDG denotes constraints C_i, C_j , etc. A directed edge from C_i to C_j indicates that the plausibility of C_i depends on the plausibility of C_j . We shall say that the node corresponding to C_i is a *consequent node* and that the node corresponding to C_j is an *antecedent node*. We shall also refer to nodes without dependencies as *unsupported nodes*.
- (d) Given D_k , the design further evolves by the designer's attempt to (i) change an assumed constraint to a *validated* one, or (ii) place an *undetermined* constraint in the *assumed* or *validated* state. In order to do either, the original constraint C may have to be refined into a set of new constraints C_i, C_j, \dots , such that some relationship between these constraints can be used to demonstrate the desired plausibility state for C .⁴
- (e) A design may evolve in other ways also. For instance, in the process of attempting to change the plausibility state of a constraint C_j from *assumed* to *validated* it may happen that the evidence E_k is such that C_j has to be *refuted*. In that case, C_j must be eliminated from the design and, if some other constraint C_i is dependent on C_j then C_i 's plausibility state will have to be revised in the light of the new state of affairs.

In other words, the plausibility states of constraints in a design will have to be constantly revised when new constraints are introduced, the states of one or more constraints are reassigned or when new evidence is at hand.

Figure 2 provides an overview of a *plausibility-driven design environment* (PDDE). At the beginning of design step k , the current *plausible design* (PD) is represented by the pair (Aguero, 87; Aguero and Dasgupta, 87):

4. TPD includes rules such that when a constraint C is refined into constraints C_i, C_j, \dots , the designer can infer the plausibility state of C from the plausibility state of C_i, C_j, \dots (Aguero, 87).

<specification, warranty >

where *specification* is a set of interrelated constraints that are part of the current design (the relationships being established by the CDG), and *warranty* is the evidence supporting the plausibility of the design and consists of a set of verified plausibility statements⁵.

During step *k*, the PD may be altered by *generating* a new constraint, *modifying* an existing constraint, or *destroying* an existing constraint. In the first two cases, a new plausibility statement or an altered plausibility statement would be produced by the designer. In the case of a destroyed constraint the corresponding plausibility statement would have to be deleted.

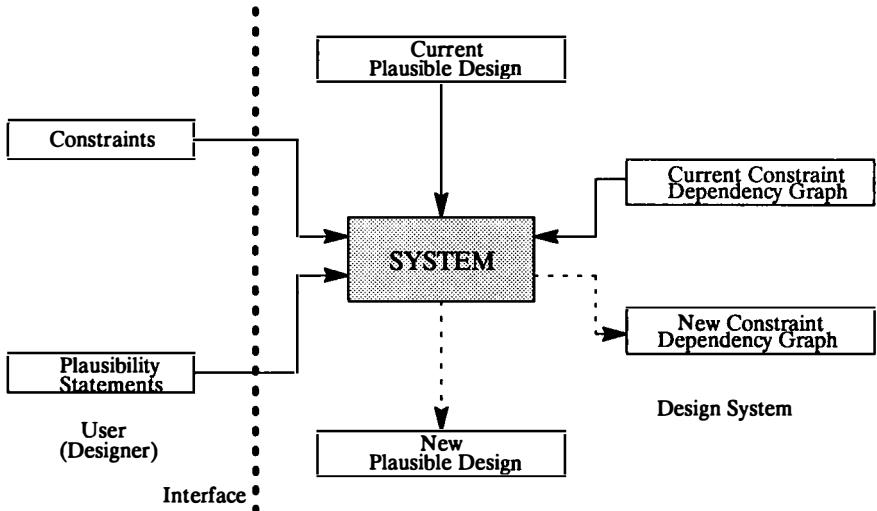


Figure 2: A Plausibility-driven Design Environment (PDDE).

TRUTH MAINTENANCE SYSTEMS

Based on studies of design problem solving processes we may also conclude that (Patel 90; Patel and Dasgupta, 89):

- (1) Design problem solving is nonmonotonic in nature and is viewed as a search through some problem space.
- (2) Designers explicitly represent alternative solutions to subproblems and problem solving proceeds by heuristically selecting the most plausible alternative.
- (3) Design problem solving may reveal that certain alternatives are undesirable, forcing the designer to concentrate problem solving efforts on more promising paths.
- (4) Alternatives may be added incrementally (i.e., problem solving may reveal alternatives that were not initially considered).

5. We have simplified slightly, the description of a PD in this discussion. Actually, a PD may also include the *history* of the design which would include previous (but now discarded) specifications and warranties (see (Aguero, 87; Aguero and Dasgupta, 87)).

In view of these circumstances, TMSs can help designers by offering support for:

- (a) Efficient revision of a designer's beliefs. Helps to determine the impact of revised plausibility states.
- (b) Focus of reasoning. Helps the designer decide where further problem solving efforts should be concentrated.
- (c) Recording the reasons behind the current beliefs. Helps to (i) record and explain the evolution of the design, and (ii) prevent re-visiting failed search paths.

Detailed examples illustrating these benefits are cited in (Patel 90; Patel and Dasgupta 89). Here, we shall illustrate aspects (a) and (b) with the help of the following example.

Example 1

The overall requirement driving the Computer Architecture Simulation Engine (CASE) multiprocessor design experiment (Hooton, 87; Hooton, Aguero, and Dasgupta 88) was expressed in the root constraint C_1 :

“ C_1 : CASE efficiently supports the execution of architectural specifications written in S*M for the purpose of simulation.”

The driving design methodology behind the CASE effort was partitioning and it was decided that C_1 would be satisfied if $R_1 = C_2 \wedge C_3$ would be satisfied.

“ C_2 : CASE exploits possibilities for concurrent activity during the execution of specifications.”

“ C_3 : CASE provides specialized architectural support for the constructs in S*M that could not be efficiently supported using general-purpose approaches”

Subsequently C_2 was refined into $R_2 = C_4 \wedge C_5 \wedge C_{10}$:

“ C_4 : CASE allows for module-grain concurrency to occur where possible.”

“ C_5 : CASE allows for statement-grain concurrency to occur”

“ C_{10} : Some processing of temporal information is done concurrently with the (possible) execution of modules.”

The design of the CASE architecture continued in this manner and evolved into a MIMD (multiple-instruction stream, multiple data stream) multiprocessor. Figure 3 displays the status of the CASE CDG with the plausibility states of the constraints at an appropriate intermediate point (For simplicity many constraints have been omitted). Note that arcs originating from (approximately) the same point on a constraint represent a conjunction of constraints, whereas arcs originating from different points represent a disjunction of constraints. Thus the R field of constraint C_2 should read $R_2 = (C_4 \wedge C_5 \wedge C_{10}) \vee (C_4 \wedge C_5 \wedge C_{16})$.

Now with reference to Figure 3, consider the problem of determining the set of constraints that would be affected if the plausibility of C_7 was changed to the refuted state. In Figure 3, constraints C_1, C_2, C_3 , and C_5 are affected and the updated plausibility states of these constraints are displayed within parenthesis in italics.

Assuming C_7 to be still in the assumed plausibility, where should the designer concentrate further problem solving efforts? Closer analysis reveals that efforts must be concentrated on constraint C_{16} (or C_2 to be safe) because as long as C_{10} is in the refuted plausibility state, the product term $C_4 \wedge C_5 \wedge C_{10}$ will also be in the refuted plausibility state. Hence the product term $C_4 \wedge C_5 \wedge C_{16}$ is the only product term that satisfies the (requirements of) R field of C_2 and which is currently not in the refuted plausibility state. ■

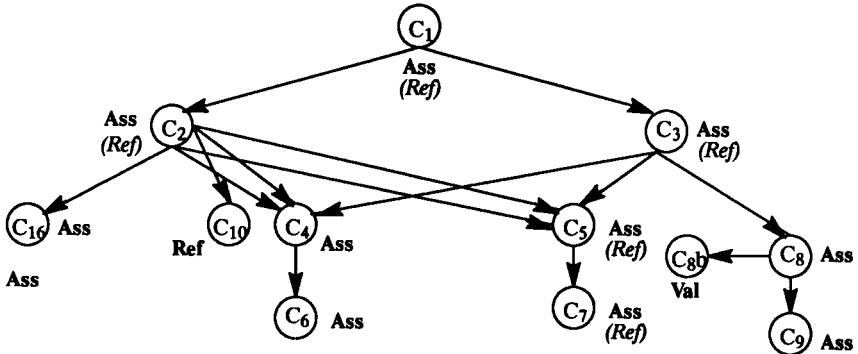


Figure 3: CDG for the CASE design experiment.

Doyle's TMS

Based on the preceding example it should be clear how TMSs can help designers cope with the complexities associated with nonmonotonic design processes. Space limitations preclude a complete or thorough discussion of TMSs in this paper. The reader is referred to (Doyle, 79; de Kleer, 86; McAllester, 80; McDermott, 83; Stallman and Sussman, 79) for detailed discussions. In the following we consider two particular TMSs due to Doyle (79) and de Kleer (86) respectively, that have been widely documented in the literature and that also represent two general categories of TMSs, viz., those that are *justification based* and *assumption based*, respectively.

Doyle's TMS is called a *justification based* TMS because the truth maintenance algorithm is based on manipulating justifications. Intuitively a justification records the reasons behind problem solver beliefs. Doyle's TMS supports two kinds of justifications, viz., *support list* (SL-) justifications and *conditional-proof* (CP-) justifications⁶. SL-Justifications have the general form:

$$N : SL \langle inlist \rangle \langle outlist \rangle$$

where N is the consequent node. An SL-Justification denotes the condition

$$i_1 \wedge i_2 \wedge \dots \wedge \neg o_1 \wedge \neg o_2 \wedge \dots \Rightarrow N \quad (2)$$

where, $i_i \in inlist$ and $\neg o_i \in outlist$, for all $i \geq 0$, and is said to be *valid* if and only if each node of the *inlist* is IN and each node of the *outlist* is OUT. In (2) $\neg o_i$ is to be read as disbelief in o_i . A node may have several SL-justifications:

$$J_1 \vee J_2 \vee \dots \vee J_m \Rightarrow N \quad \text{for some } m \geq 0 \quad (3)$$

In order for the node in question to be IN, it must have at least one valid justification. A node with no justification is called a *premise node*.

de Kleer's ATMS

An ATMS node is a triple:

6. In the remainder of this paper we shall primarily focus on SL-Justifications. CP-justifications have rarely been used or discussed in the open literature. Hence, we shall not discuss CP-justifications further; the interested reader is referred to (Doyle, 79) for detailed discussions on CP-Justifications.

Node : < datum, label, justification >

The datum has internal structure and meaning to the problem solver but is ignored by ATMS. The justification field contains problem solver created justification(s). This field is analyzed, but never modified, by ATMS. The label is computed by ATMS on the basis of the justifications. Justifications are interpreted in accordance with equation (3). Let J denote the current set of justifications. An ATMS environment is a set (a conjunction) of *assumptions*⁷. A node N holds (is IN) in an environment E if any environment E' of N 's label is a subset of E .

ATMS labels must be *consistent*, *sound*, *complete* and *minimal*. An environment is consistent if it is not true that $E, J \vdash \perp$ (false). N 's label is sound if for every E in the label we have $E, J \vdash N$. A label is complete if every consistent environment E' for which $E, J \vdash N$ is such that $E \subseteq E'$ for some E in N 's label. A label is minimal if no environment of the label is a superset of any other environment in the label.

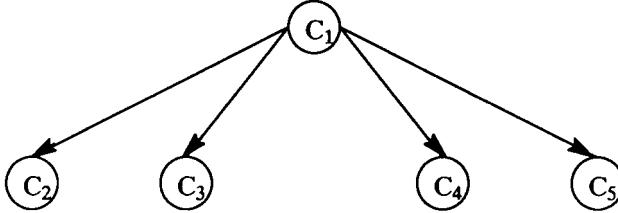


Figure 4: Initial Constraint Dependency Graph

The ATMS has a distinguished node for \perp (representing a contradiction). We shall denote this node as NF (F for false). The justifications for NF denote an inconsistent set or sets of beliefs. As inconsistent environments are deleted from labels, NF always has an empty label. The ultimate assumptions of the inconsistent beliefs are stored in a *nogood* data base (see (de Kleer, 86) for details).

ENCODING CONSTRAINT DEPENDENCY GRAPHS

Here, we encode the simple CDG as shown in Figure 4 according to the two types of TMSs. The R field in the plausibility statement of C_1 should read $(C_2 \wedge C_3) \vee (C_4 \wedge C_5)$.

Because conventional TMS only assign a status of IN/OUT to TMS nodes, a CDG with n constraints must have $4n$ nodes. Thus, NiA (NiV) (NiR) (NiU) should be IN if C_1 is assumed (validated) (refuted) (undetermined).

Table 1 shows an encoding (1.A) for the CDG of Figure 4 in Doyle's TMS. The entry corresponding to justification 1J1V (1J2V) encodes the fact that N1V is IN (i.e., C_1 is in the validated state) when, {N2V,N3V} ({N4V,N5V}) are IN. Similarly 1J1A encodes the fact that N1A is IN (i.e., C_1 is assumed) if N1V is OUT and {N2A,N3A} are IN, i.e., $N2A \wedge N3A \wedge \neg N1V \Rightarrow N1A$

7. In ATMS, assumptions are distinguished nodes that form the foundations to which every beliefs support can be traced (de Kleer, 86). To avoid confusion with our use of the term "assumed" to represent a particular plausibility state - the assumed plausibility state - we shall henceforth refer to assumptions as unsupported nodes.

Node	Number	Justification	Node	Number	Justification
N1A	1J1A	SL < N2A,N3A > < N1V >	N2A		
	2J1A	SL < N2A,N3V > < N1V >	N2V	1J2V	SL < > < >
	3J1A	SL < N2V,N3A > < N1V >	N2R		
	4J1A	SL < N4A,N5A > < N1V >	N2U		
	5J1A	SL < N4A,N5V > < N1V >	N3A	1J3A	SL < > < >
	6J1A	SL < N4V,N5A > < N1V >	N3V		
N1V	1J1V	SL < N2V,N3V >	N3R		
	2J1V	SL < N4V,N5V >	N3U		
N1R	1J1R	SL < N2R,N4R >	N4A	1J4A	SL < > < >
	2J1R	SL < N2R,N5R >	N4V		
	3J1R	SL < N3R,N4R >	N4R		
	4J1R	SL < N3R,N5R >	N4U		
N1U	1J1U	SL < > < N1A,N1V,N1R >	N5A N5V N5R N45	1J5A	SL < > < >

Legend: N_{ix} = Node i in state $x \in \{A, V, R, U\}$
 j_{ix} = jth justification for Node i in state $x \in \{A, V, R, U\}$

Table 1: Encoding 1.A under Doyle's TMS

Consider a *product term* p_i involving constraints:

$$p_i = C_1 \wedge C_2 \wedge \dots \wedge C_n \quad (4)$$

Using the laws of TPD, it can be easily proved that (Patel, 90; Patel and Dasgupta, 89):

Lemmas

- (a) There is exactly 1 assignment of plausibility states to the C_j s for p_i to be validated.
- (b) There are $2^n - 1$ assignments of plausibility states to the C_j s for p_i to be assumed.
- (c) There are $4^n - 3^n$ assignments of plausibility states to the C_j s for p_i to be refuted.
- (d) There are $3^n - 2^n$ assignments of plausibility states to the C_j s for p_i to be undetermined.

Now consider a *sum of products* expression P where

$$P = p_1 \vee p_2 \vee \dots \vee p_t \quad (5)$$

where the p_i s are product terms. Without loss of generality we assume that each p_i to be a conjunction of n variables (i.e., constraints).

Using (a) — (d) and Theorems T1 — T4 (see Section on TPD-BRS), it is easy to show that under encoding 1.A:

NPV requires t SL-justifications (6)

NPA requires $t \times (2^n - 1)$ SL-justifications (7)

NPR requires n^t SL-justifications (8)

NPU requires 1 SL-justification (9)

Corollary: From (4.3)—(4.6), Encoding 1.A requires a total of $t \times 2^n + n^t + 1$ justifications.

Arguments supporting the optimality of encoding 1.A are given in (Patel, 90; Patel and Dasgupta, 89).

Encoding Under de Kleer's ATMS

Table 2 shows an encoding (1.B) for the CDG of Figure 4 under ATMS. Recall that in ATMS, a node is denoted by a triplet $\langle \text{datum}, \text{label}, \text{justification} \rangle$. Each entry in Table 2 follows this notation. One may then verify that when the environment is:

$$E_I = \{N2V, N3A, N4A, N5A\}$$

$NIA \in \text{IN}$. Note that under 1.B it is possible for both $N1A$ and $N1V$ to be IN. This happens, for example, when

$$E_I = \{N2A, N3V, N4V, N5V\}$$

Node	Node Representation
$N1A$	$\langle N1A, \{\dots\}, \{(N1A)(N2A, N3A)(N2A, N3V)(N2V, N3A)(N4A, N5A)(N4V, N5A)(N4A, N5V)\} \rangle$
$N1V$	$\langle N1V, \{\dots\}, \{(N1V)(N2V, N3V), (N4V, N5V)\} \rangle$
$N1R$	$\langle N1R, \{\dots\}, \{(N1R)(N2R, N4R)(N2R, N5R)(N3R, N4R)(N3R, N5R)\} \rangle$
$N2A$	$\langle N2A, \{\{N2A\}\}, \{(N2A)\} \rangle$
$N2V$	$\langle N2V, \{\{N2V\}\}, \{(N2V)\} \rangle$
$N2R$	$\langle N2R, \{\{N2R\}\}, \{(N2R)\} \rangle$
$N3A$	$\langle N3A, \{\{N3A\}\}, \{(N3A)\} \rangle$
$N3V$	$\langle N3V, \{\{N3V\}\}, \{(N3V)\} \rangle$
$N3R$	$\langle N3R, \{\{N3R\}\}, \{(N3R)\} \rangle$
$N4A$	$\langle N4A, \{\{N4A\}\}, \{(N4A)\} \rangle$
$N4V$	$\langle N4V, \{\{N4V\}\}, \{(N4V)\} \rangle$
$N4R$	$\langle N4R, \{\{N4R\}\}, \{(N4R)\} \rangle$
$N5A$	$\langle N5A, \{\{N5A\}\}, \{(N5A)\} \rangle$
$N5V$	$\langle N5V, \{\{N5V\}\}, \{(N5V)\} \rangle$
$N5R$	$\langle N5R, \{\{N5R\}\}, \{(N5R)\} \rangle$
$N6A$	$\langle N6A, \{\{N6A\}\}, \{(N6A)\} \rangle$
$N6V$	$\langle N6V, \{\{N6V\}\}, \{(N6V)\} \rangle$
$N6R$	$\langle N6R, \{\{N6R\}\}, \{(N6R)\} \rangle$

Labels: $N1A \quad \{\{N1A\}(N2A, N3A)\{N2A, N3V\}\{N2V, N3A\}\{N4A, N5A\}\{N4V, N5A\}\{N4V, N5V\}\}$
 $N1V \quad \{\{N1V\}(N2V, N3V)\{N4V, N5V\}\}$
 $N1R \quad \{\{N1R\}, \{N2R, N4R\}\{N2R, N5R\}\{N3R, N4R\}\{N3R, N5R\}\}$

Table 2: Encoding 1.B under ATMS

This is possible for ATMS because multiple contexts can simultaneously coexist. To avoid this situation, we can encode $N1A \wedge N1V \Rightarrow \perp$ (a contradiction). However, this technique will cause ATMS to create an exponential number of *nogoods*. Since multiple contexts are possible we may allow both $N1A$ and $N1V$ to be IN. The problem solver may first inquire if $N1V$ is IN (we want the strongest state). If $N1V$ is IN we know that C_1 is validated. If $N1V$ is OUT and $N1A$ is IN we know that C_1 must be assumed.

It has been shown that encoding 1.B requires one justification less (1.B does not have a justification for NiU) than that required for Encoding 1.A. It may therefore be concluded that representation of CDGs in conventional TMSs such as those of Doyle's and de Kleer's, turn out to be extremely inefficient.

The ATMS Labeling Algorithm

Figure 5 shows the the label update algorithm (based on (de Kleer, 86)). To maintain consistency with subsequent discussions, we shall refer to the justification field of a node as a *sum of products* (sop) field. In a sop expression we use the notation:

```

label-update(N:node,l:label,p:sop)
begin
    L := {};
    for j := 1 to |p| do
        begin
            x := {};
            for i := 1 to |p[j]| do
                x := x ∪ lij
            end;
            L := L ∪ x;
        end;
    L' := delete-ng-environments(L);
    L'' := delete-supersets(L);
    if l = L'' then return;
    if N is nogood
    then
        update-ng-data-base(N);
        eliminate-ng's-from-labels();
    end;
    for all N' such that N in SOP-of-N' do
        label-update(N',label-of-N',SOP-of-N');
    end;
end label-update;

```

Figure 5: ATMS Label Update Algorithm

$$\{\{a_{11}, a_{12}, \dots, a_{1p}\}, \{a_{21}, a_{22}, \dots, a_{2q}\}, \dots, \{a_{r1}, a_{r2}, \dots, a_{rs}\}\} \quad (10)$$

to mean,

$$(a_{11} \wedge a_{12} \wedge \dots \wedge a_{1p}) \vee (a_{21} \wedge a_{22} \wedge \dots \wedge a_{2q}) \vee \dots \vee (a_{r1} \wedge a_{r2} \wedge \dots \wedge a_{rs}) \quad (11)$$

Let the sop field (P) of the new node contain t product terms $\{p_1, p_2, \dots, p_t\}$. Let l_{ij} denote the label for the i^{th} node of the j^{th} product term (p_j). The label update algorithm requires three input parameters, a node N , the label of N , and the sop field of N . If the sop field of N contains t (i.e., $|p| = t$) product terms, then the outer for loop (j) is executed t times; once for each product term. For each iteration of the j loop, the i loop is executed $|p[j]|$ times. Here, $|p[j]|$ is the number of literals in the j^{th} product term of P . Each literal of $p[j]$ is assumed to have a sound, consistent, complete, and minimal label. The i loop computes the union of all these labels for a particular product term $p[j]$. The result of the union operation (within the i loop) is stored in the set x . Thus, x contains the “partial label” of each product term. Subsequently, these partial labels are collected in L within the j loop. On termination of the j loop L is sound and complete label for N .

The next step is to remove environments, from node labels, which are supersets of nogoods (‘delete-ng-environments’). Following this, the label is made minimal (‘delete-supersets’); environments which are supersets of other environments are eliminated. Now, the label for the nodes is sound, consistent, complete, and minimal. If the new label (L'') is the same as the old label (l), then, further processing is not required and the algorithm terminates. If the node is a nogood, then all the environments of the label (L'') are recorded in the nogood data base (‘update-ng-data_base’). Since the nogood data base has been updated, the environments of all the labels must be checked. Environments which are supersets of these new nogoods must be eliminated. Finally since the label of node N has been updated, the new label must be propagated, recursively, to all nodes (N') which mention N in their sop fields. The correctness of ATMS label propagation has been proved in (Fujiwara, Mizushima, and Honiden, 90).

The following issues must be resolved before ATMS label propagation can be used within TPD-BRS:

- (1) What modifications (if any) must be made to the algorithm?
- (2) Given the labels, how can we efficiently determine the plausibility states of dependent nodes?
- (3) How should *contradictions* be recorded in TPD-BRS?

Issues (2) and (3) are directly related to the semantics of TPD. On the other hand, (1) is a little more subtle as it deals with the dynamic nature of CDGs.

Example 2

Consider an evolving CDG. Figure 6 shows the states of the graph at distinct design steps. For simplicity, let us concentrate on the label of N1 only. This label is computed in accordance with the update algorithm for ATMS. During step 1 there is only one unsupported node — N1. During step 2 there are two unsupported nodes N2 and N3; during step 3 there are three unsupported nodes. Note that, even though during step 2 N1 is not an unsupported node it is still present in the label of N1. Similarly, N1, N2, and N3 are still present in the label of N1 during step 3. ■

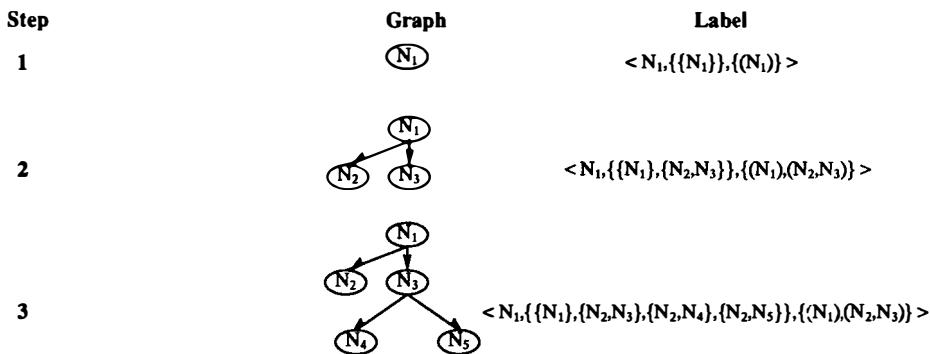


Figure 6: Evolving CDG

In CDGs unsupported nodes quickly become dependent nodes and it is necessary to handle the above mentioned situation without introducing undue implementation overheads.

INTRODUCING TPD-BRS

A node in TPD-BRS has a structure similar to that of a node in ATMS:

$$N_i: \langle C_i, \{P'\}, \{label\}, \{sop-field\} \rangle$$

Here N_i is the TPD-BRS node. C_i is the constraint associated with N_i and P' is the postulated plausibility of C_i . The *label* field is computed by TPD-BRS based on the (designer supplied) *sop-field*. Thus, the *wff* corresponding to the *R* field of the plausibility statement associated with C_i is expressed in the *sop-field*. For the sake of efficient revision all *wff's* are required to be expressed in sum of product forms (*sop*). If the *sop-field* is empty the node has no current dependencies. Nodes with an empty *sop-field* will be

called unsupported nodes. For all practical purposes the P' field may be regarded as a comment. It serves only to record the plausibility state of the constraint at its inception.

The label field represents the ultimate unsupported nodes on which the node depends. Labels in TPD-BRS are similar to the labels in ATMS. However, because nodes can have different degrees of beliefs, our notion of an environment is somewhat different from that in ATMS.

TPD-BRS supports three kinds of nodes; (1) Unsupported nodes; (2) Dependent nodes; and (3) Nogood nodes (NGs). An unsupported node has an empty sop-field. By convention, the label computed by TPD-BRS for such a node consists of the singleton set mentioning the node itself. Labels for other nodes are computed in accordance with the ATMS label update algorithm (suitably modified as described in the sub-section dealing with modifications to the label-update algorithm). *Nogood* nodes will be explained shortly.

Example 3

Consider the following nodes:

$$\begin{array}{ll} N1 : < C1, \{\{N1\}\}, \{\} \rangle & N2 : < C2, \{\dots\}, \{\{N1\}\} \rangle \\ N3 : < C3, \{\{N3\}\}, \{\} \rangle & N4 : < C4, \{\dots\}, \{\{N2, N3\}\} \rangle \end{array}$$

Here, N1 and N3 are unsupported nodes. N2 is a dependent node because it depends on the unsupported node N1. That is, the sop field of N2 mentions the unsupported node N1. N4 depends on two nodes N2 and N3. Thus, N4 depends on an unsupported node and a dependent node. If we trace through the dependencies of dependent nodes, we always terminate at some unsupported nodes. In the example above, N4 depends on two unsupported nodes, N1 and N3. ■

The Environment

An *environment* is a collection of unsupported nodes. For a given CDG, a *current environment* is a 4-tuple:

$$CE \equiv < V, A, R, U >$$

where, *V* is the set of unsupported nodes in the CDG which are currently in the *validated* plausibility state; *A* is the set of unsupported nodes in the CDG which are currently in the *assumed* plausibility state; *R* is the set of unsupported nodes in the CDG which are currently in the *refuted* plausibility state; and *U* is the set of unsupported nodes in the CDG which are currently in the *undetermined* plausibility state.

The set of all unsupported nodes *E* in the CDG is given by $E = V \cup A \cup R \cup U$. Thus, *E* contains all the unsupported nodes of the CDG and *CE* is a partitioning of *E* into four mutually exclusive sets to each of which an assignment of one of the plausibility states has been made.

Plausible Environments

Given a node *N*, let *val(N)*, *ass(N)*, *ref(N)* and *und(N)* denote that *N* is in the validated, assumed, refuted, and undetermined plausibility state respectively. Let *J* denote a conjunction of nodes. For convenience, we can view *J* as a set of nodes. *J* is a *validated* (*as-*

sumed) (refuted) (undetermined) environment – VE (AE) (RE) (UE) – iff J satisfies Property 1.v (1.a) (1.r) (1.u).

Property 1.v: $\forall N(N \in J \Rightarrow val(N))$

Property 1.a: $\forall N(N \in J \Rightarrow ass(N)) \wedge (\exists N'(N' \in J \wedge \sim val(N')))$

Property 1.r: $\exists N(N \in J \wedge ref(N))$

Property 1.u: $\exists N(N \in J \wedge und(N)) \wedge \sim \exists N'(N' \in J \wedge ref(N'))$

Every node of a VE is in the validated plausibility state. Every node of an AE is in the assumed plausibility state and at least one member of an AE is not in the validated plausibility state. A RE has at least one node in the refuted plausibility state. A UE has at least one node in the undetermined plausibility state and no node in the refuted plausibility state. Applying the laws of TPD it may be easily shown, under the assumption of non-conflicting evidence⁸, that (see (Patel 90; Patel and Dasgupta, 89) for proofs):

Theorems

(T1) A sop expression P (equation 10) can be placed in the validated plausibility state iff there exists at least one p_i that constitutes a VE.

(T2) A sop expression P can be placed in the assumed plausibility state iff there exists at least one p_i that constitutes a AE and no p_j constitutes a VE.

(T3) A sop expression P can be placed in the refuted plausibility state iff each p_i constitutes a RE.

(T4) A sop expression P can be placed in the undetermined plausibility state iff there exists at least one p_i that constitutes a UE and no p_j constitutes a VE or a AE.

Contradictions and Competing Constraints

Competing constraints in TPD are similar to the commonly encountered contradictions of conventional IN/OUT based TMSs. In TPD, a *competing* set of constraints is such that complete satisfaction of one member of the set precludes the satisfaction of one or more other members. We shall refer to sets of competing nodes as *nogoods*. And the third type of node supported by TPD–BRS allows such sets to be represented. A *nogood* node in TPD–BRS is a distinguished node and we shall denote these nodes by the symbol NG. By definition, a nogood node has an empty label; however, a nogood node must have a nonempty sop field.

As done in the ATMS, all nogood sets are stored in a separate data base. To enhance search efficiency, unsatisfiable product terms (and their supersets) are deleted from the labels of nodes. Till now, our treatment of nogood sets has been similar to the treatment of contradictions in ATMS. However, TPD–BRS also allows nogood node processing in a different manner from that of ATMS. In ATMS a label is first created for the nogood (on the basis of the justification – the sop-field). On the other hand TPD–BRS directly stores each product term of a NG as nogood sets in the nogood data base. The reasons for treating nogoods in this manner are as follows:

- (i) To give the designer flexibility in recording nogoods. In doing so, the designer may record nogoods in accordance with his/her goals and intentions. Further-

8. Let $evid_ag(C_i) = 1$ and $sig_evid_fav(C_i) = 1$. Here, the designer (apparently) has significant evidence in favor of C_i . At the same time, however, the designer claims that there exists evidence against C_i 's plausibility. In such circumstances, it is up to the designer to re-evaluate the evidence and resolve the conflict. TPD–BRS assumes conflict free plausibility state assignments.

more, this gives the designer the flexibility to backtrack to any previous node of a failed path.

- (ii) To prevent *unintended* processing during nogood recording.

Example 4

During the CASE design experiment (Hooton, 87; Hooton, Aguero, and Dasgupta, 88), the following constraint was generated during step 1:

C_{10} : Some processing of temporal information is done concurrently with the (possible) execution of modules.

The R field of the plausibility statement for C_{10} is defined as $R_{10} \equiv C_{11}$, where

C_{11} : A control processor, CP, exists; it supports all I/O with the host, and global clock precessing.

Here N11 is unsupported and N10 depends on N1 1. In an attempt to justify the introduction of C_{10} the designer discovered that C_{10} is not desirable. Here, the designer wishes to refute C_{10} and not C_{11} ; in fact C_{11} was successfully verified to be in the assumed plausibility state. In TPD-BRS, the designer can accomplish the refutation of N10 by declaring:

NG2 : <nogood, {},{{N10}}>

This leads TPD-BRS to add nogood{N10} to the nogood data base. Note that if we processed nogoods as done in ATMS, nogood{N11} would be added to the nogood data base ({{N11}} corresponds to the label for N10). Recording nogood{N11} is not what the designer intended and tracing the dependencies to ultimate unsupported nodes leads to unintended processing. ■

Modifications to the ATMS Label Update Algorithm

In order to minimize implementation costs, sets are represented as bit-vectors. Each ATMS assumption occupies a unique bit-vector position. A '1' indicates that the corresponding assumption is in the particular set (de Kleer, 86). Thus, ATMS labels are collections of bit-vectors (environments). Bit-vectors facilitate efficient synthesis of set theoretic operations. Given the preceding discussions, the following modification must be made to "label-update":

```
L' := delete-ng-environments(L);
L' := delete-supersets(L');
```

of Figure 5 must be replaced with (see Figure 7 for complete revised algorithm):

```
l := eliminate-dependent-unsupported-nodes(l);
L := eliminate-dependent-unsupported-nodes(L);
L' := delete-ng-environments(L);
L' := delete-supersets(L');
```

The preceding modification ensures that dependent unsupported nodes are never allowed to persist in label environments. This can be achieved by "anding" E with the environments of the label. Let E' be any environment of the label. If $E' = E' \wedge E$ then E' is retained in the label. On the other hand if $E' \neq E' \wedge E$ then E' must be discarded from the label.

Given the representation of the environment both these modifications can be easily implemented. In order for this technique to work the following protocol should be fol-

lowed;(i) first, allocate bit-positions for unsupported nodes; then (ii), delete unsupported node(s) which just became dependent. This protocol also solves the garbage collection problem associated with vacant bit-vector positions.

Example 5

With reference to Figure 6, let $E_0 = \{0,0,0,0,0,0,0\}$ (i.e., the environment during step 0). After introducing N1 in step 1, we have $E_1 = \{1,0,0,0,0,0,0\}$ (N1 has been allocated to position 1). L of N1 (i.e., label of N1) is $\{\{1,0,0,0,0,0,0\}\}$. $E_1 \wedge \{1,0,0,0,0,0,0\} = \{1,0,0,0,0,0,0\}$. Since the environment did not change due to the “and” operation, this environment is retained. Hence,

$$< N1, \{\{1,0,0,0,0,0,0\}\}, \{(N1)\} > \text{ (step 1)}$$

After introduction of N2 and N3 in step 2, we have $E_2 = \{1,1,1,0,0,0,0\}$ (N2 at position 2 and N3 at position 3). Now, we are required to remove N1 from the environment. Doing so gives $E_2 = \{0,1,1,0,0,0,0\}$. Now L of N1 becomes $\{\{1,0,0,0,0,0,0\}, \{0,1,1,0,0,0,0\}\}$.

$$\begin{aligned} (1) E_2 \wedge \{1,0,0,0,0,0,0\} &= \{0,0,0,0,0,0,0\} \\ (2) E_2 \wedge \{0,1,1,0,0,0,0\} &= \{0,1,1,0,0,0,0\} \end{aligned}$$

In (1) the environment changed, and hence, must be discarded. In (2) the environment did not change and is retained. Hence,

$$< N1, \{\{0,1,1,0,0,0,0\}\}, \{(N1), (N2, N3)\} > \text{ (step 2)}$$

In step 3 N4, N5 are introduced. Let $E_3 = \{1,1,1,1,0,0,0\}$ (N4 at position 1 and N5 at position 4). After deleting N3 we have $E_3 = \{1,1,0,1,0,0,0\}$. L of N1 is now $\{\{0,1,1,0,0,0,0\}, \{1,1,0,0,0,0,0\}, \{0,1,0,1,0,0,0\}\}$

$$\begin{aligned} (1') E_3 \wedge \{0,1,1,0,0,0,0\} &= \{0,1,0,0,0,0,0\} \\ (2') E_3 \wedge \{1,1,0,0,0,0,0\} &= \{1,1,0,0,0,0,0\} \\ (3') E_3 \wedge \{0,1,0,1,0,0,0\} &= \{0,1,0,1,0,0,0\} \end{aligned}$$

Only (1') has changed. Hence,

$$< N1, \{\{1,1,0,0,0,0,0\}, \{0,1,0,1,0,0,0\}\}, \{(N1), (N2, N3)\} > \text{ (step 3)}$$

Note that the label of N1 during step 3 is $\{\{N2, N4\}, \{N2, N5\}\}$ as required. Also note that garbage collection is unnecessary — even though N4 was allocated to N1s bit position, the updated labels are as required. One may also wish to verify that all nodes of Figure 6 have correct labels which mention only unsupported nodes. ■

The next modification deals with nogood node processing. The following segment if n is *nogood* ...

```
then update-ng-data_base();
      eliminate-ngs-from-labels();
```

in the label update algorithm of Figure 5 should be replaced by the first *if*-statement shown at the top of Figure 7. This statement considers two cases. If the sop-field of the NG contains only unsupported nodes then each and every product term of the sop-field is converted into a nogoodset. After the conversion, label environments which are supersets of nogood sets are deleted from node labels. If, on the other hand, the NG contains dependent and unsupported nodes, the dependent nodes are saved in list1. After this, the first node (say N_i) is selected from list1. First, N_i is converted into an unsupported

node. This involves deleting the sop-field of N_i . Then all labels of the consequents of N_i are invalidated (recursively). This procedure is carried out for every node in list1 and all the consequents are collected during the invalidation process. Once the labels have been invalidated, the product terms of the sop-field of the NG are converted into nogood sets. Finally, after updating the NG data base, the labels for the nodes are recomputed. It should be noted that the increased flexibility for recording nogoods has a penalty in terms of computational effort. The other option is to eliminate this flexibility and mandate that only unsupported nodes be mentioned in the sop-field of NGs; in this case only the first part of the *if*-statement (Figure 7) is needed.

Computation of the Current Plausibility State

The algorithm for computing the plausibility state of a node is based on the properties stated in Theorems T1 — T4. The algorithm consists of four sequential steps. Let l be the label of node N. Suppose l consist of t environments, $\{E_1, E_2, \dots, E_t\}$. And, finally, let $CE = <V, A, R, U>$ be the current environment. The algorithm is shown in Figure 8.

The function first checks if the label is empty. If so, the function terminates by returning ‘Ref’. An empty label indicates that all the environments of the label have been deleted; this happens when environments are supersets of nogoods.

The first ‘for’ loop checks for the existence of a VE in the label of N. If, $E_i \subseteq V$, then, E_i is a VE. Here, E_i denotes the i^{th} environment of the label (l). If a VE is found, the function terminates by returning ‘Val’ (representing the validated plausibility state, see Theorem T1).

The second ‘for’ loop checks for the existence of a AE. Because, $val(N) \Rightarrow ass(N)$, the union of A and V is stored in AV — an AE can contain a node in the validated plausibility state, but a VE cannot contain a node in the assumed plausibility state. If $E_i \subseteq AV$, then E_i is an AE. If an AE is found, the function terminates by returning ‘Ass’ (the assumed plausibility state). Note that, since the first loop checked for the existence of a VE, this check need not be repeated during the second loop (see Theorem T2).

At the beginning of the third loop, we know that N belongs in the undetermined plausibility state or in the refuted plausibility state. Hence, the third loop checks for the existence of an UE. If $E_i \cap R = \{\}$, then E_i cannot be a RE. Since we know E_i is not a VE, nor a AE, nor a RE, E_i must be a UE. If an UE is found then the function terminates by returning ‘Und’ (the undetermined state), else the function terminates by returning ‘Ref’ (the refuted plausibility state) (see Theorems T3 and T4).

In the worst case, the algorithm performs $2t$ subset operations, and t intersection operations, giving a complexity of $O(t \times |E|)$; here t is the number of environments in the label of the node and E denotes the number of nodes in the current environment. It has been shown that, in the worst case, t may be exponential in the number of nodes (Provan, 88) (see also (Elkan, 90) for additional TMS/ATMS complexity results). Consequently, recent research has focused on techniques to minimize label sizes by controlling the search process. Control strategies for ATMS are discussed in (de Kleer and Williams, 89; Forbes and de Kleer, 88; Provan, 88) and shall not be pursued further in this paper.

```

TPD-BRS-LABLE-UPDATE(N:node,l:label,p:sop)
begin
  if Nogood(N)
    then if All-Unsupported-Nodes
      then
        x := product-terms(p)
        update-ng-data-base(x)
        eliminate/ngs-from-labels();
        return;
    end;
    else
      list1 := dependent-nodes ∈ p
      for i ← 1 to |list1| do
        make-unsupported(Ni);
        invalidate-consequents(Ni);
        consequents := consequents ∪ Ni;
      end;
      x := product-terms(p);
      update-ng-data-base(x);
      recompute-labels(consequents);
      return;
    end;
  else
    L := {};
    for j ← 1 to | p | do
      x := {};
      for i ← 1 to | p[j] | do
        x := x ∪ lij;
      end;
      L := L ∪ x;
    end;
    l := eliminate-dependent-unsupported-nodes(l);
    L := eliminate-dependent-unsupported-nodes(L);
    L' := delete-ng-environments(L);
    L' := delete-supersets(L');
    if l = L then return;
    forall N' such that sop-N' contains N do
      TPD-BRS-LABEL-UPDATE(N',Label(N'),SOP(N'));
    end;
  end;
end;

```

Figure 7: TPD-BRS Update Algorithm

```

determine-plausibility(l:label,CE:current-environment)
begin
  if l = {} then return(Ref);
  for i := 1 to |l| do
    if Ei ⊆ V
      then return (Val);
    end if;
  end;
  AV := A ∪ V;
  for i := 1 to |l| do
    if Ei ⊆ AV
      then return (Ass);
    end if;
  end;
  for i := 1 to |l| do
    if Ei ∩ R = R
      then return (Und);
      else return (Ref);
    end if;
  end;
end determine-plausibility;

```

Figure 8: Algorithm for Computing Current Plausibility State

We are now ready to encode the graph of Figure 4. This be concisely encoded by 5 nodes:

N1 < C1,{ {N2,N3},{N4,N5} }>
N3 < C3,{ {N3} },{}>
N5 < C5,{ {N5} },{}>

N2 < C2,{ {N2} },{}>
N4 < C4,{ {N4} },{}>

One may verify that if the $CE = \langle \{N4\}, \{N5\}, \{N2\}, \{N3\} \rangle$, N1 (i.e., C_1) is in the assumed plausibility state. It should be clear from this encoding that

- (1) If a dependency graph contains n constraints, then n TPD-BRS nodes are required.
- (2) TPD-BRS requires t justifications for encoding a SOP expression with t product terms.

Contrast the encoding above, with those of Tables 1 and 2. The succinct encoding is possible because TPD-BRS explicitly supports four plausibility states.

CONCLUSIONS

The advent of ATMS has triggered considerable research interest in *problem solving architectures*. In fact, ATMS may indeed be regarded as a new paradigm for symbolic truth maintenance systems. In this paper, we have extended the scope of this paradigm to include the multi-valued logic of TPD. We have also incorporated changes to the basic algorithm to account for the idiosyncrasies of CDGs and plausibility driven designs. In particular, implementation ideas for ATMS have been extended to; (1) provide an elegant solution to the garbage collection problem; and (2) develop an efficient procedure for plausibility state determination.

Currently, we are in the process of developing interactive computer aided design tools for use in plausibility-driven designs. In particular, we are interested in providing mechanical support for “solving” so called *ill-defined* (Reitman, 64) (see also (Brown and Chandrasekaran, 86; Finger and Dixon, 89; Goel and Pirolli, 89)) or *ill-structured* (design) problems (Simon, 73). We seek design tools that can; (a) aid designers during design development; and (b) provide an environment for further experimentation and exploration of design in general and TPD in particular — our proposed system allows for the structured representation and/or acquisition of design information

REFERENCES

- U. Aguero, “A Theory of Plausibility for Computer Architecture Design”, Ph.D. dissertation, Center for Advanced Computer Studies, University of Southwestern Louisiana, 1987.
- U. Aguero & S. Dasgupta, “A Plausibility-Driven Approach to Computer Architecture Design”, *Communications of the ACM*, 30, 11, Nov. 1987, 922–932.
- D.C. Brown, and B. Chandrasekaran, (1986), “Knowledge and Control for a Mechanical Design Expert System”, *IEEE Computer*, July 1986, 92–100.
- S. Dasgupta, “The Structure of Design Processes”, *Advances in Computers*, Vol. 28, Yovitis, M.C. (Ed), Academic Press, New York, 1989.

- S. Dasgupta & U. Aguero, "On the Plausibility of Architectural Designs", *Proc. 8th International Symposium on Computer Hardware Description Languages and their Applications* (CHDL-87), M. Barbacci & C.J. Koomen (Eds), North-Holland, Amsterdam, 1987, 177-194.
- J. de Kleer, "An Assumption-based TMS", *Artificial Intelligence*, 28, 1986, 127-162.
- J. de Kleer and B. Williams, "Diagnosis with Behavioral Modes", *Proc. IJCAI*, 1989, 1324-1330.
- J. Doyle, "A Truth Maintenance System", *Artificial Intelligence*, 12, 1979, 231-272.
- C. Elkan, "A Rational Reconstruction of Nonmonotonic Truth Maintenance", *Artificial Intelligence*, 43, 1990, 219-234.
- S. Finger, and J.R. Dixon, (1989), "A Review of Research in Mechanical Engineering Design. Part 1: Descriptive, Prescriptive and Computer-Based Models of the Design Process", *Research in Engineering Design*, (1) 1, 51-67.
- K.D. Forbes, & J. de Kleer, "Focusing the ATMS", *AAAI-88*, 1988, 193-198.
- Y. Fujiwara, Y. Mizushima, and S. Honiden, "On Logical Foundations of ATMS", *Proc. Conf. on Nonmonotonic Reasoning*, 1990.
- V. Goel, and P. Pirolli, (1989), "Motivating the Notion of Generic Design within Information-Processing Theory: The Design Problem Space", *AI Magazine*, Spring 1989, 19-36.
- Hooton A., "The Plausible Design of CASE: A Computer Architecture Simulation Engine", M.S Thesis, Center for Advanced Computer Studies, University of Southwestern Louisiana, April 1987.
- A. Hooton, U. Aguero, and S. Dasgupta, "An Exercise in Plausibility Driven Design", *IEEE Computer*, 21, 7, July 1988, 21-31.
- S. Landry, *et al*, "PDI: A Plausibility-Driven User Interface for UNIX", Center for Advanced Computer Studies, University of Southwestern Louisiana, 1988.
- D. McAllester, "An Outlook on Truth Maintenance", Artificial Intelligence Laboratory, AIM-551, MIT, Cambridge, MA, 1980.
- D. McDermott, "Contexts and Data Dependencies: A Synthesis", *IEEE Trans. Pattern Matching and Machine Intelligence*, Vol. 5, No. 3, May 1983, 237-246.
- Patel, S., "Computer-Aided Support for Plausibility-Driven Designs", Ph.D. Thesis, Center for Advanced Computer Studies, University of Southwestern Louisiana, 1990.
- S. Patel, and S. Dasgupta, "Automated Belief Revision in Plausibility Driven Designs", Center for Advanced Computer Studies, Tech. Report TR-89-2-1, University of Southwestern Louisiana, September,
- H. Pedovsky, W. Rogers, L. Delcambre, & S. Landry, "Reduction of Space Systems Cost Via Use of Expert Systems", SBIR Phase I Final Report, Orbital Systems Limited and the University of Southwestern Louisiana, March 1989.
- G. Provan, "A Complexity Analysis of Assumption-Based Truth Maintenance", In *Reason Maintenance Systems and their Applications*, B. Smith and G. Kelleher (Eds.), Ellis Horwood Limited, Chichester, England, 1988, 98-113.
- W.R. Reitman, (1964), "Heuristic Decision Procedures, Open Constraints, and the Structure of Ill-Defined Problems", In *Human Judgements and Optimality*, M. W. Shelly II and G. L. Bryan (Eds.), Wiley, New York, 1964.
- H.A. Simon, (1973), "The Structure of Ill-Structured Problems", *Artificial Intelligence*, (4) 1973, 181-201.
- Simon, H.A., *Sciences of the Artificial*, 2nd Edition, MIT Press, Cambridge, MA, 1981.
- Simon, H.A., *Models of Bounded Rationality*, Vol. 2, MIT Press, Cambridge, MA, 1982.
- R.M. Stallman, and G.J. Sussman, "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis", *Artificial Intelligence*, 9 (2), Oct. 1977, 135-196.

Methods for improving the performance of design systems

I. F. C. Smith,[†] D. Haroud[§] and B. Faltings[§]

[†]ICOM (Steel Structures)

[§]LIA (Artificial Intelligence)

Swiss Federal Institute of Technology (EPFL)

CH-1015 Lausanne

Switzerland

Abstract. When projects are carried out in fragmented domains, such as within the AEC (architecture, engineering and construction) industry, design tasks are usually complicated. Typically, these tasks are performed over months or even years by many people - all of whom employ different criteria for reaching decisions. In this paper, a system which simulates aspects of such design is described by means of modelling knowledge associated with preliminary bridge-design. The system employs non-monotonic reasoning methods such as truth maintenance, default reasoning, dynamic constraint satisfaction and conflict resolution in order to represent the design process.

INTRODUCTION

The AEC industry contributes to at least ten percent of the gross national product of most industrialized nations. Design activities within this industry usually present problems which are not easily modelled. Nevertheless, efforts to rationalize tasks in computer-aided systems could lead to substantial savings. The most costly problems encountered during construction projects are often due to difficulties associated with knowledge management. This is in contrast to problems which can be traced to a lack of fundamental scientific knowledge - a very rare occurrence.

Researchers in artificial intelligence recognise that modelling design activities represents an important study area (Forbus, 1988). Although some work in artificial intelligence has treated design, for example Mittal and Araya (1986), most attempts to model structural design has thus far come from engineers and architects, for example IABSE (1989) and Gero (1990). Computer-aided design (CAD) systems offer the possibility of attaching information about design decisions to the geometric information traditionally expressed in drawings. Such additional information could be exploited in order to improve communication between designers. Efficient capture of this information requires that the CAD system follows design decisions from the early phases of conceptual design.

In spite of growing activity, artificial intelligence research has resulted in tools that have a limited scope for design applications. Most commercial development tools are only capable of modelling process simulation and analytical tasks such as classification, selection, identification, fault-model diagnosis and some aspects of routine design. Design tasks proceed using non-monotonic processes and therefore, design methodology can not be modelled using current development tools.

In this paper, we describe how a non-monotonic model of the design process could improve CAD systems. Firstly, characteristics of complex design tasks and corresponding needs for knowledge modelling and inference are identified. A prototype system for preliminary bridge design is then presented to examine implementation of new methods for design. Finally, these methods are examined for their applicability to large design systems.

CHARACTERISTICS OF COMPLEX DESIGN TASKS

It is of interest to examine characteristics of complex design tasks in order to identify appropriate representations, strategies and methods for computer modelling. In most cases, vast amounts of knowledge are used. Therefore, it is practically impossible to ensure that all knowledge is consistent. Conflict resolution must be treated explicitly. Furthermore, the project usually begins with only approximate and incomplete information, thereby requiring capabilities such as flexible input specifications and adaptable inferencing.

Development of systems for complex design tasks requires prior consideration of knowledge characteristics and task scheduling. Since knowledge is unavoidably contradictory, techniques such as those which organise knowledge into modules and hierarchies are necessary. Tasks may be performed simultaneously. Also, many knowledge modules must be combined with databases and traditional software. This leads to a need for conflict resolution strategies. System architectures which employ blackboard driven control strategies have successfully managed competing knowledge modules, for example, Fenves et al (1989). Nevertheless, conflict resolution is most often carried out through use of domain-specific heuristic knowledge.

If a design task is likely to begin with approximate and incomplete information, knowledge modelling must include a provision for useful assumptions (defaults) in order to stimulate and guide inference. Situations where information is incomplete usually result in greatly increased solution spaces. When defaults are added to a knowledge base, inference must accommodate reasoning which is capable of retracting and updating information, and resolution strategies may be necessary if initial assumptions conflict.

Knowledge modelling and inference should accommodate change as efficiently as possible. Change may originate from events such as revision of design criteria, reception of more accurate information and retraction of information. Large systems, especially those which require some tasks to be completed by hand and those which exploit peripheral systems, cannot be rerun easily. Knowledge bases need to be configured so that new knowledge can be added easily and there should be some mechanism for automatic updating of deductions (truth maintenance). Truth maintenance mechanisms are also useful for conflict resolution - this will be discussed in a later section.

A summary of characteristics of complex design tasks and their effects upon system development is given in Table 1. The success of most large design projects is determined by the ability of designers to manage change. A system which satisfies the requirements mentioned above could improve performance and prevent mistakes, thereby avoiding delays, accidents and unnecessary costs.

Characteristics of complex design tasks	Requirements for system development	
	Knowledge representation	Inference needs
Vast amounts of knowledge	Knowledge modules Inconsistent knowledge must be accommodated	Integration Conflict resolution strategies
Project begins with approximate and incomplete information Usually, many possible solutions	Provision for useful assumptions	Default reasoning
Project cycle completed over several years	Allow for changes in design criteria	Rerun of complete system not practical
Accurate information received late in design phase	Knowledge bases need to accept new information	Automatic updating of deductions Design decision monitoring
Success of project determined, in part, by ability to manage change	Configuration of system that can efficiently satisfy above requirements	

Table 1. Characteristics of complex design tasks and their effects upon system development

SELECTION OF TOPIC FOR DEVELOPMENT

Although we focus on methods for large systems, they are impractical for experimentation. Testing new approaches is most efficiently performed on small systems which are capable of being extended, particularly when there is a need to model the complex design tasks described in Table 1. Extensible systems are those having knowledge which requires most of the representation and inference attributes necessary for large systems. Knowledge related to preliminary bridge design, particularly aesthetic knowledge, has these characteristics.

During the preliminary design of bridges, effort is often concentrated upon factors which determine safety, economical and serviceability aspects. Aesthetic considerations may be overlooked completely by some designers. Interestingly, there is no evidence to suggest that

good aesthetics are expensive. In fact, an aesthetic design feature is often a simple one, resulting in structural efficiency and easier construction and maintenance. Hence, low cost and aesthetic appeal can be complementary design goals.

Since the end of the last century, various authors have attempted to formalize knowledge of bridge aesthetics. Many researchers have concentrated on the psychological and philosophical aspects; see for example, Arnhiem (1954), Wittkower (1952) and Venturi (1966). Unfortunately, much of this work is of little direct use to bridge designers. Publications of more practical use to engineers include an exceptional contribution by Leonhardt (1982). Recently, awareness of aesthetics has increased; for example in Japan, engineers have developed the Manual of Bridge Design Practice (Tahara, 1982) and in the USA, a bibliography (Burke, 1989) has been compiled.

Explicit knowledge of aesthetics remains badly organised and poorly distributed. Understandably, knowledge is often contradictory. Also, much heuristic information is used in situations where information is incomplete. Therefore, special reasoning techniques are particularly useful for managing such knowledge within computer systems.

PRELIMINARY BRIDGE DESIGN

Four aspects of preliminary bridge design are shown in Figure 1. Designers consider the environment of the proposed bridge in order to select potential bridge types. Environmental factors include the characteristics of the obstacle to be spanned, the surrounding topography and soil conditions. Once bridge types are selected, the designer proceeds with decisions relating to form for individual bridge types. This is where much of the aesthetic quality of the bridge is determined. After this, an overall evaluation of engineering feasibility, aesthetics, economical and political factors is performed. If the evaluation is not satisfactory, changes may be made or an alternative bridge type may be examined. Note that these considerations are performed iteratively and furthermore, they are largely qualitative; no detailed numerical analysis is carried out.

A prototype system called Aesthetic Bridge Consultant (ABC) is under development in order to model primarily the second and third aspects of preliminary bridge design shown in Figure 1. Firstly, details which describe the context of the design are placed in the knowledge base. This data is used to make an initial choice of bridge type. In the next part of the system, a bridge design is configured using rules which are specific to its type. No stress analysis is performed; calculations are only employed to apply general rules of thumb. Default values are used in situations where information is lacking.

In the last part of the system, an overall consideration of aesthetics and engineering feasibility is carried out by the user through consultation of a graphical representation of the design solution. New values may amend design values offered by the system, thereby enabling the user to adapt results to specific requirements. In addition, another bridge type may be investigated manually through a direct request from the user, or automatically if the user modifies information required by the part of the system which selects bridge types.

ABC employs frame representations and rules in hierarchies. Design knowledge is divided into parts; each part corresponds to a possible bridge type. Sub-parts of each type are possible designs. Possible designs are shown in Figure 2 for the cable-stayed bridge type.

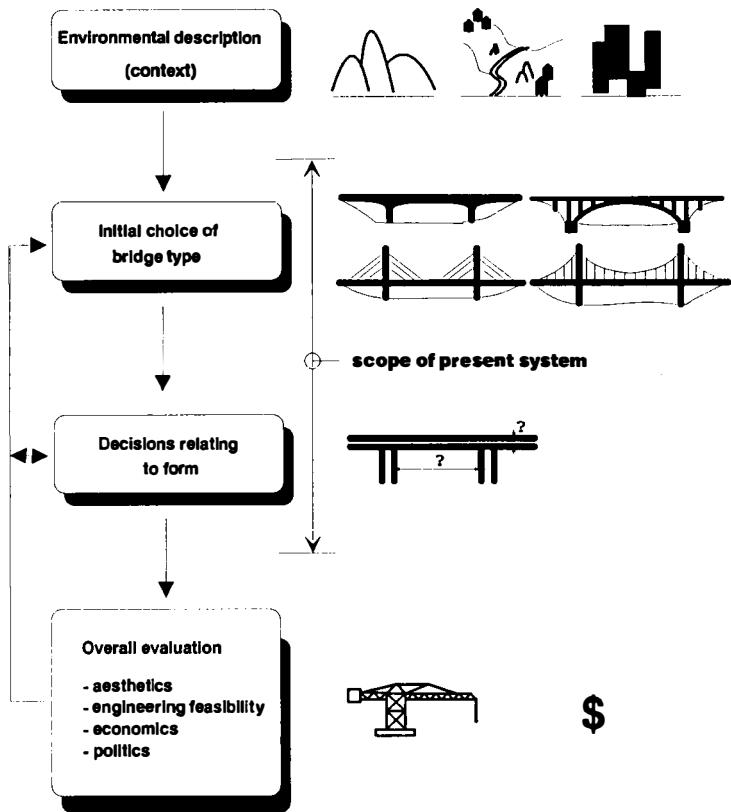


Figure 1. Aspects of preliminary bridge design

There are two types of knowledge base, an active knowledge base (AKB) and a basic knowledge base (BKB), details of which are given next.

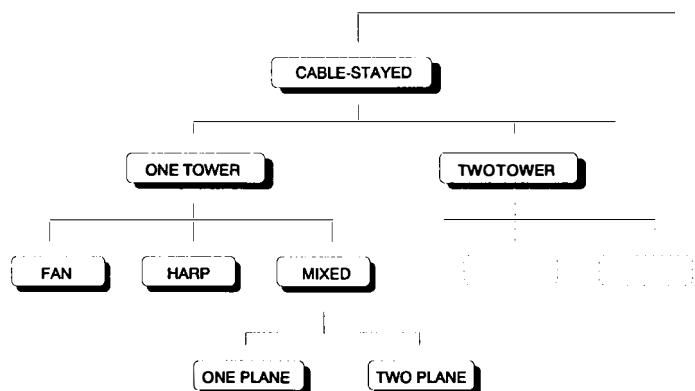


Figure 2. Possible bridge types for cable-stayed bridges

The AKB contains domain-based inference rules. These rules are divided into groups, and cable-stayed bridges, Figure 2, is one such group. Many rules represent knowledge based on experience, or heuristics. As an example of rule development, Figure 3 shows the transition from an "expert statement" to a Lisp rule; clearly, some transformation is required. Furthermore, rules are classified into two types, default and fixed, thereby enabling some rules to represent an initial approximation or default value when information is incomplete while other rules provide values for parameters which cannot be changed.

FROM "Expert statement" [4] :

"for large spans and high level decks (of cable-stayed bridges), A-shaped towers are most suitable"

LISP RULE :

```
CS11
(rule((:in (level 5) :var ?f)
      (:in (status-of-cable-stayed-bridge t) :var ?f1)
      (:in (main-span-for-cable-stayed-bridge ?x)
            :test (> ?x 200) :var ?f2)
      (:in (min-clearance-due-to-obstacles ?y)
            :test (> ?y 10) :var ?f3))
      (rassert-b! (shape-of-towers-for-cable-stayed-bridge "A")
      (CS11 ?f ?f1 ?f2 ?f3)))
```

Figure 3. Example of rule in the Active Knowledge Base (AKB)

Rules are structured in a hierarchy of layers in order to model the way designers employ knowledge. Also, hierarchies accommodate knowledge that otherwise, would be conflicting. Rules in lower levels are always examined before those in higher levels. Values fixed by rules in lower levels may be changed later by rules examined in higher levels. Only rules within the same layer need to be consistent. Such structuring simplifies knowledge acquisition because generally, experts find it easier to give a priority to two contradictory rules rather than try to think of all conditions necessary for application of rules at the same level of importance.

In addition, a layered rule structure is useful insofar as that without such a strategy, rules would be much more complicated. Complicated rules create an unfortunate abstraction from the design process because they rarely correspond to an expert's description of the solution to the problem. On the other hand, simple rules facilitate knowledge modelling and subsequent management.

The example of Figure 4 shows that layers provide a means of representing conflicting rules. Four out of seven levels employed in the complete knowledge base are shown. Rules which determine the tower height of cable-stayed bridges are employed in order to demonstrate two cases. In Case 1, the height of the tower is not fixed until Level 4, whereas in Case 2, tower height is determined at Level 3. These rules could all be considered to be default rules. Other types of rules, those providing values which cannot be changed are discussed further in the next section.

Example : Cable stayed bridge;			CASE 1	main span = 200 m side span = 50 m cable spacing = 30 m
			CASE 2	main span = 150 m side span = 50 m cable spacing = 4 m
LEVEL	RULES		TOWER HEIGHT, m	
1	IF & THEN	number of towers = 2 number of cable planes = 2 height of towers = 0.3 * main span	60	45
2	IF & THEN	classification = symmetrical shape of cables = harp height of towers = $\tan 25^\circ \times$ side span	23	23
3	IF THEN	classification = symmetrical internal position of longest cable = 0.5 (main span - 2 * cable spacing) [case 1 = 70] [case 2 = 71]		
3	IF & & THEN	classification = symmetrical shape of cables = harp internal position of longest cable < (0.5 * main span) height of towers = $\tan 25^\circ \times$ internal position of longest cable	33	33
4	IF & THEN	internal position of longest cable < (0.5 * main span) external position of longest cable = < side span height of tower < main span * 0.5 & height of tower >= main span * 0.2	ok 40	ok ok
			TOWER HEIGHT ADOPTED FOR DESIGN, m	40 33

Figure 4. An example of rules in levels and two cases showing evolution of tower height through several levels of default rules.

The BKB contains structural information organised in frames. Figure 5 shows an example of the information needed to consider a bridge parapet of a cable-stayed bridge. Common-sense knowledge is included, such as the number of parapets (set to two). Much of this information is required for graphic representation.

The default value of the shape of parapet, used when no default rule has been fired, is set to OPEN, thereby describing a railing type of parapet (as opposed to a solid type which would be CLOSED). These two values are restricted INSTANCES of the slot, 'shape of

```

(fget-frame 'parapet-for-cable-stayed-bridge)
(put-nt 'parapet-for-cable-stayed-bridge 'cable-stayed-bridge
  'number-of-parapet      'VALUE '(2)
  'shape-of-parapet       'VALUE ()
  'shape-of-parapet       'DEFAULT 'open
  'shape-of-parapet       'INSTANCE '(open closed)
  'shape-of-parapet       'RESTRICT '(((INSTANCE)))
  'height-of-parapet     'VALUE ()
  'height-of-parapet     'DEFAULT 1.2
  'height-of-parapet     'RESTRICT '((number 0 2))
  .......)

```

Figure 5. Frame structure for knowledge in the Basic Knowledge Base (BKB)

'parapet'. Hence, no value other than the instances would be allowed. For numerical values, RESTRICT helps avoid illogical numbers. For example, parapet height is restricted to two metres. This height would not be exceeded considering that sound barriers and other such features of bridges in urban areas are included as separate features. More importantly, the restrict values provide absolute bounds for the constraint satisfaction strategy explained later in the paper. Also, the default height is set to 1.2 metres, the most common parapet height. The VALUE slots of the frame are used to store values determined by rules; example values are shown.

METHODS USED FOR IMPLEMENTATION

Four methods are described : truth maintenance, default reasoning, dynamic constraint satisfaction and conflict resolution. Methodologies were chosen according to their capacity to accommodate changing, incomplete, inexact and conflicting information. To a large extent, such capacity defines necessary characteristics of an experienced and competent designer.

Figure 6 shows connections between top level control and other parts of the system through the AKB, BKB, inference engine and the *justification-based truth maintenance system*, JTMS. When a JTMS is used, parameters are not true or false, but current beliefs are represented as 'in' or 'out'. The function of the JTMS (Doyle, 1979) is to maintain consistency. This is performed by labelling datums, justifications, consequences and contradictions. Through examination of the rule set, the inference engine conveys this information to the JTMS where it is stored in nodes; every instantiation of a parameter has a node. Once stored in this way the JTMS activates when information is added and when a rule is fired by the inference engine.

The JTMS strategy is more efficient than search strategies which rely on seeking information when required. Also, temporarily irrelevant information is not necessarily discarded. Through storage of all links, regardless of immediate relevance, the inference engine can retrieve information quickly. When backtracking is required, the JTMS guides inference to relevant relationships, thus eliminating fruitless search. Also, the JTMS representation is used to manage change even after the design is completed.

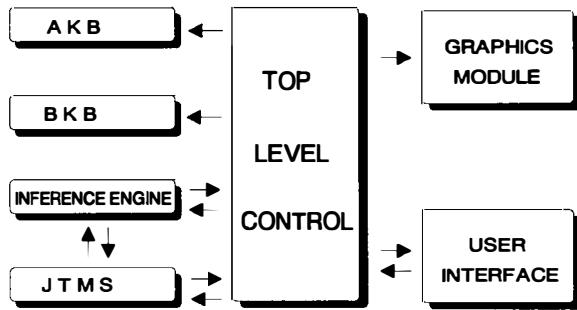
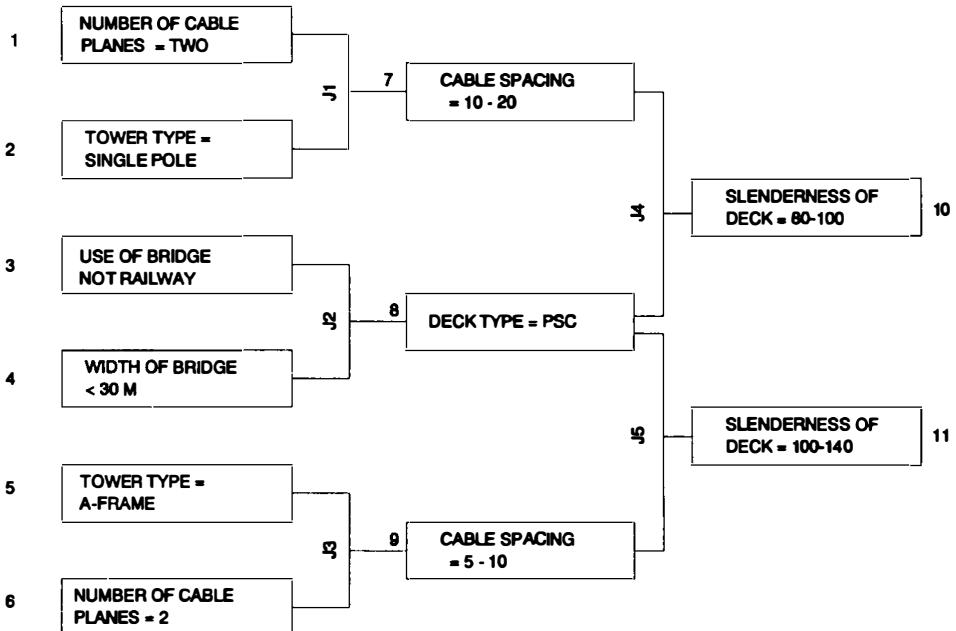


Figure 6. Organisation of the system

The JTMS guides propagation of beliefs when they are changed. In the case of an added assumption, forward propagation occurs towards the goals, to determine new consequences. Conversely, in the case of a retracted assumption, backward propagation traces alternative support. Clearly, a JTMS is well suited to the rule hierarchy described in Section 4.1 because it helps maintain consistency when a higher level rule changes assertions made by rules in lower levels.



J1 INDICATES JUSTIFICATION 1, etc
NODE NUMBERS ARE SHOWN NEXT TO BOXES

Figure 7. An example of nodes and justifications used by the JTMS

An example of JTMS operation is given in Figures 7 and 8. In Figure 7, conditions on the left, when satisfied, lead to conclusions to their right. Each group of conditions and conclusion represents one rule and one justification set. Information stored by the JTMS includes a representation by nodal reference numbers for each fact asserted as shown in Figure 8. As explained earlier, the status of an asserted fact can be IN or OUT. For this example, assume Nodes 1-4 and 6 to be IN, whereas Node 5 is OUT.

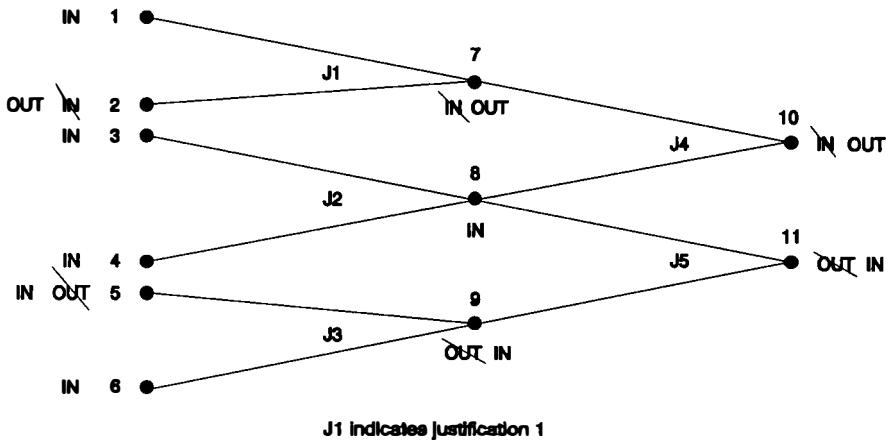


Figure 8. Nodal representation of the example in Figure 7

Node 7 represents 'cable spacing = 10-20' and Figure 8 shows the justification (JUST 1) supporting the status of this parameter which is derived from the rule relating 'tower type' and 'number of cable planes'. The consequence of this node having status IN is Justification 4. Should the value of 'tower type' change from 'single pole' to 'A-frame' the JTMS will change Node 2 to OUT and propagation occurs such that the values of 'cable spacing = 10-20' and 'slenderness of deck = 80-100' become OUT as shown in Figure 8. Equally, the status of Node 5 may change to IN and propagation occurs such that Nodes 9 and 11 become IN. Such changes in status are performed automatically by the JTMS outside of the inference engine. An example of information stored by the JTMS for a node is given in Figure 9 for deck type, Node 8. Information includes node number, the current status, support for the current status of the node and consequences of this status.

A capacity for *default reasoning* (Reiter, 1980) is an important feature since engineers have similar design strategies when information is lacking. Systems without such capabilities would halt if knowledge was missing. Single value defaults are stored in the slots of frames, such as those in Figure 5. These default values are immediately withdrawn if a value is found during inference. Another type of default is created by attaching values to parameters in default-type rules, for example see Figure 4.

TMS data for NODE 8	Explanation
TMS node 8	The node reference
status : IN	IN = currently true OUT = currently not true
support: JUST2	the support for the status is Justification 2
justifications : JUST 2	justifications supporting the status are shown
consequences : JUST 4, JUST 5	relevant justifications are shown

Figure 9. Explanation of TMS information for Node 8

In this system, the JTMS is an important prerequisite for default reasoning. When default values are found to be no longer necessary due to existence of other information, their belief values are changed and the JTMS automatically updates the values. A default reasoning mechanism without a truth maintenance system would be very cumbersome.

Since design criteria is often formulated in terms of constraints, design systems must include algorithms for *constraint satisfaction*. This system employs a novel approach, called dynamic constraint satisfaction (Hua et al, 1990) which is intended to accommodate

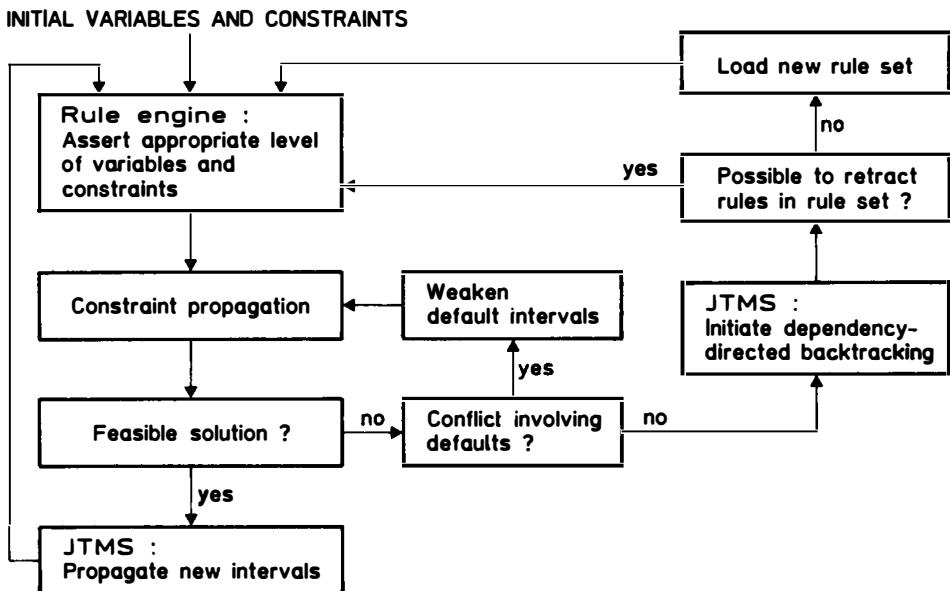


Figure 10. The components of the dynamic constraint satisfaction algorithm

multiple sets of constraints for continuous parameters and multiple constraint networks relating to different designs. Design decisions can create situations where the design process investigates more than one variable set before an acceptable solution is found.

Mittal and Falkenhainer (1990) studied dynamic constraint satisfaction for symbolic constraints through proposing the use of constraints for variables in left hand sides (IF ...) of rules. This study extends this work to include numerical values of constraints, default limits as well as maximum bounds and conflict resolution strategies. Process flow between the components of the dynamic constraint satisfaction algorithm are shown in Figure 10.

Initial information is used to activate the rule engine for the first level of variables and constraints. Typically, this is the point where specific initial conditions trigger rules which attach default intervals to variables. Once the rule engine has fired all rules relevant to a particular level of hierarchy, the intervals are applied to all related parameters using the constraint propagation algorithm of Davis (1987). Constraint propagation is carried out on both physically feasible intervals (provided in the frames in the BKB) and the current design interval (given by the rules in the AKB) for each variable. However, constraint propagation may provide intervals for variables which cannot lead to a feasible solution (Davis, 1987). The Fourier-Motzkin algorithm (Schrijver 1986) is employed to detect contradictions between algebraic constraints. When a feasible solution is possible, the JTMS is activated in order to propagate new intervals to all current consequences and control returns to the rule engine for the next level of rules. The process described thus far corresponds to the left vertical line of actions shown in Figure 10.

When a conflict occurs, that is when no feasible solution is possible for the current design intervals, the system attempts to find a solution firstly through weakening default

Partial list of user input

length-of-bridge	400
position-of-obstacle	70
obstacle	river
size-of-obstacle	260
positions-where-piers-not-possible	(30,70) (330,370)

BKB knowledge

FRAME: deck-for-constant-depth-beam-bridge
 SLOT: main-span-for-constant-depth-beam-bridge
 PHYSICAL limits 0 & 300

FRAME: environment
 SLOT: positions-where-piers-not-possible
 USER INPUTS PAIRS (POS1, POS2)
 SLOT: ranges-where-piers-not-possible
 CALCULATE RANGE 1 (POS1-POS2)
 & RANGE 2, etc

Figure 11a. An example of constraint satisfaction and conflict resolution applied to the main span of a constant-depth beam bridge
 Input data and BKB knowledge

intervals within the current constraint network. This is possible for two cases. The first case is when two default intervals are in conflict for a given variable. In such situations, the interval fixed by the higher level rule is given priority through weakening the interval fixed by the lower level rule to the physical limits defined for the variable. The second case is encountered when a default interval is in conflict with an interval given by a fixed rule. Here,

From initial choice module

Order of preference :	Main span values after rule fires
Inference begins with first choice	
1. Constant depth beam bridge	
2. Arch bridge	
3. Cable-stayed bridge	
4. Suspension bridge	
5. Truss bridge	
RULE 1 (Level 1) Type : default IF status of constant depth beam bridge = t & number of spans = 3	
THEN length of bridge / 1.8 = < main span = < length of bridge / 1.6	222-250
Note : default interval	
RULE 2 (Level 2) Type : fixed IF status of constant depth beam bridge = t & obstacle = river & size of obstacle > 0.2 * length of bridge - position of obstacle	
THEN main span > size of obstacle	260-300
Note : default interval from Rule 1 weakened	
RULE 3 (Level 3) Type : fixed IF status of piers = t & position where piers not possible NOT EQUAL nil & interior positions where piers not possible > = edges of obstacle	
THEN main span > (span of obstacle + 1st range where piers not possible + 2nd range where piers not possible)	340

Note : 340 exceeds physical limits (0-300 m) fixed in Figure 11a

No opportunity to retract rules. Therefore explore conditions for second choice, arch bridge.

Figure 11b. An example of constraint satisfaction and conflict resolution applied to the main span of a constant-depth beam bridge
AKB knowledge

the complete interval given by firing the fixed rule is adopted. The default interval is not used for further propagation. It is however, weakened to a single value corresponding to the nearest limit of the interval given by the fixed rule in order to guide instantiation of the variable in the final stages of design. These processes are shown in Figure 10 by the internal loop in the middle of the figure.

When a conflict occurs and no default values are involved, resolution is performed through investigation of other constraint networks by means of dependency-directed backtracking. When the conditions for triggering a rule contain variable values which do not correspond to the boundaries of its current design value, a case split is introduced between the part of the interval satisfying the trigger and the part which does not. The rule engine first explores the constraint network containing the default value and for situations when a default interval is split, the rule is fired. The first priority of dependency-directed backtracking is to follow the network formed by the case not previously investigated. If no rule can be retracted in this way, a new rule set, corresponding to a completely different design is loaded, see the right side of Figure 10.

Figure 11 demonstrates an example of dynamic constraint satisfaction for one variable according to this algorithm. Figure 11a provides a partial list of user input and frame information and Figure 11b shows the beginning order of preference determined by a separate module for bridge choice, as well as three rules which change the value of the variable, main span. Note that the values for the physical limits are 0 and 300 meters.

Once Rule 1 has fired, a default interval for main span, becomes 222 and 250 m. Rule 2 takes into account the fact that an obstacle (river) 260 m wide is under the bridge. This rule causes default weakening since the lower bound for this constraint (260 m) is greater than the upper limit set previously for main span (250 m). The physical limit for the variable is now adopted as its upper bound.

Rule 3 accounts for the presence of an area where no piers are possible due to factors such as scouring from the river, bad foundation conditions, etc. When Rule 3 is processed conflict occurs again because the value for the main span (340 m) is greater than the current upper limit for the variable (300 m). However, weakening is not possible since the conflict involves intervals set by two fixed-type rules. Note that in this case, the conflict was created by the system attempting to assign an interval which exceeded the physical limits of the variable. Since there is no opportunity to retract rules, the system explores the network corresponding to the second choice named in Figure 11a, the arch bridge. The evolution of the design is shown on Figure 12. Note that the third sketch from the top on this Figure is never adopted - this would have been the design proposed in the absence of dependency-directed backtracking.

Currently, this algorithm resolves conflicts using the importance of constraints implied in the rule hierarchy through constraint weakening and through rule retraction when weakening is not possible. Resolution strategies could be focused further to account for other criteria such as cost and security in relation to specific rule sets and depending on the type of resolution. Therefore, this algorithm could accommodate a range of resolution strategies to be applicable for different cases of weakening and retraction, thereby increasing overall flexibility.

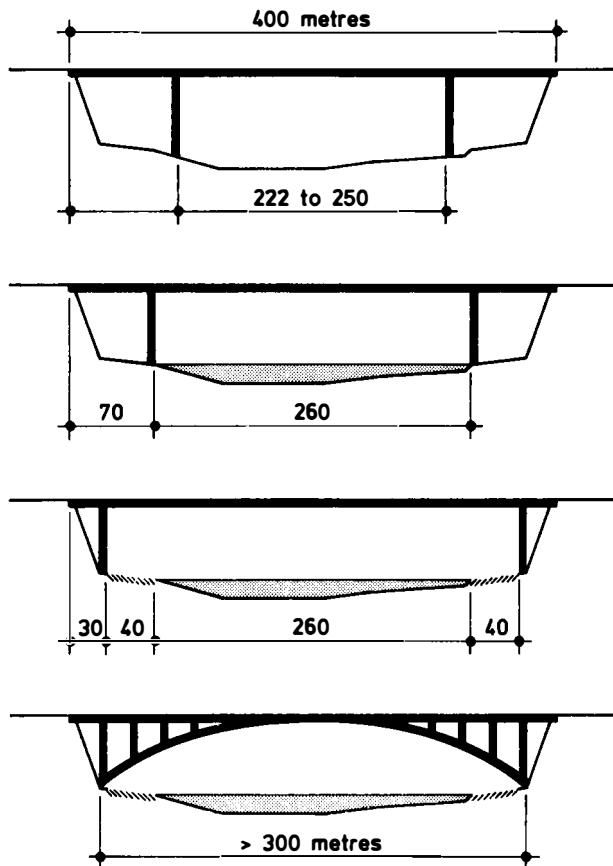


Figure 12. Evolution of the design example in FIGURE 11

When this algorithm is implemented, the system can emulate a situation where a designer renounces development of a design solution and investigates another design possibility. Such activity is common practice in the design of complex artifacts. Sophisticated reasoning methods as well as the organization of rules in a hierarchy has simplified knowledge acquisition. Determining the knowledge necessary for default reasoning has allowed an explicit study of a kind of engineering knowledge which is rarely formalised. Finally, constraint satisfaction and conflict resolution strategies are essential ingredients in any large design system and this prototype is well suited to examine them in more detail.

CONCLUSIONS

1. Simulation of difficult design tasks is improved through use of advanced methods in knowledge modelling and reasoning.

2. Organization of rules into hierarchies enables a simple and easily modifiable representation of rules which otherwise, would be contradictory.
3. Methods such as truth maintenance, default reasoning, constraint satisfaction and conflict-resolution strategies improve models of complex design tasks and facilitate knowledge acquisition.
4. Dynamic constraint satisfaction is useful for accommodating conflicting design information and for focusing resolution strategies.

ACKNOWLEDGEMENTS

The authors would like to thank G. Kimberley and F. Donzé of ICOM as well as K. Hua of LIA for their contribution to the development of this research. Additionally, the staff of ICOM are recognised for their help during knowledge acquisition and for their contribution to the production of this paper. This research is funded by the Swiss National Science Foundation.

REFERENCES

- Amheim, R. (1954). *Art and visual perception*, University of California, Berkeley, California.
- Burke, M.P. (1989). *Bridge aesthetics bibliography*, Transportation Research Board, National Research Council. Washington.
- Davis, E. (1987). Constraint propagation with interval labels, *Artificial Intelligence*, 32, pp.281-331.
- Doyle, J. (1979). A Truth maintenance system, *Artificial Intelligence*, 12, pp.127 -162.
- Fenves, S.J., Fleming, U., Hendrickson, C. Maher, M.L. and Schmitt, G. (1989). An integrated software environment for building design and construction, *CIFE Symposium*, Stanford.
- Forbus, K. (1988). Intelligent computer-aided engineering, *AI Magazine*, 9, pp.23-36
- Gero, J.S. (1990). *Applications of artificial intelligence in engineering V*, Vol. 1 Design, Springer Verlag.
- Hua, K., Faltings, B., Haroud, D. Kimberley, G. and Smith, I. (1990). Dynamic constraint satisfaction in a bridge design system, *Expert systems in engineering*, Lecture notes in artificial intelligence, 462, Springer Verlag, pp.217-232.
- IABSE (1989). *Expert systems in civil engineering*, Proceedings of colloquium in Bergamo, Italy, The International Association for Bridge and Structural Engineering, Zurich.
- Leonhardt, F. (1982). *Bridges, aesthetics and design*, Verlags-Anstalt, Stuttgart.
- Mittal, S. and Araya, A. (1986). A knowledge-based framework for design, *AAAI/86*, pp.856-865.
- Mittal, S. and Falkenhainer, B. (1990). Dynamic constraint satisfaction problems *AAAI/90*, pp.25-32.
- Reiter, R. (1980). A logic for default reasoning, *Artificial intelligence*, 13, pp.81-132.
- Schrijver, A. (1986). *Theory of linear programming*, Wiley, New York.

- Tahara, Y. (1982). *Manual for aesthetic design of bridges*, Japanese group of IABSE, Tokyo.
- Venturi, R. (1966). *Complexity and contradiction in architecture*, Musueum of Modern Art, New York.
- Wittkower, R. (1952). *Architectural principles in the age of humanism*, Tiranti, London.

A combined generative and patching approach to automate design by assembly

J. P. Tsang

Alcatel Alsthom Recherche
Route de Nozay
91460 Marcoussis
France

Abstract. In this paper, design is regarded as the process of determining concurrently the functional structure (goal organization) and the physical structure (physical composition) of an artifact given an end-user assessment of it called its specifications. Major benefits of such an approach are that goal satisfaction interferences (positive and negative) may be explicitly reasoned about and may therefore promote design of more principled artifacts. Major difficulties of such approach stem from the evolutive nature of the solution space such as (1) static knowledge must dynamically reference constituents of an ever-changing representation and (2) decision-making may not be based on absence of constituents since any absent constituent may become present at a later stage of the process (contingent nature of absence). We address these issues and present a combined generative and patching approach in which patching is achieved through "supplantation", a computationally and heuristically more attractive mechanism than backtracking in many cases. Experimentation carried out on SMACK, an implementation of these principles with regard to a case-study of electronic equipments for satellites, is also briefly described.

1. Introduction

The long-term objectives of our design automation enterprise are basically two-fold: first, reduce design time cycle and second, enhance mastery of design know-how to cope with the scaling up of clients' demands and complexity of new merging technologies. The case-study we worked on consists of designing block-schema diagrams of electronic equipments for satellites to answer to calls for proposals. Our scientific endeavour in this undertaking is primarily to understand and reproduce the mental process of design that takes place in the designer's mind before and during the act of drawing the artifact itself. Partly for that reason, we laid more emphasis on the high-level aspects of design as opposed to the low-level aspects which are more of the realm of traditional CAD systems.

As a first approximation, we may distinguish two ways of automating the design process: generatively or analogically. The generative approach starts from scratch and builds the solution from first principles (without any prejudice as to what the artifact will look like).

The analogical approach¹, on the contrary, draws directly upon past experience: it adapts the solution of a "similar" solved problem to current needs. Note that the distinction between the two approaches is not clear-cut: generative is akin to analogical since generative knowledge encodes in fact problem-solution patterns. This takes us to the core of the problem: grain-size. A generative approach is fine-grained (tiny chunks of knowledge) while an analogical approach is coarse-grained (the artifact is encoded more or less as such). There are advantages of each approach: the first one may, to some extent, address "unthought of" problems while the second helps reduce or avoid altogether the ordinary process of design. Though it is clear that what is needed is a combination of the two, we will limit ourselves in this paper to the generative approach.

We identified and elaborated two distinct reasoning paradigms (Tsang, 1990): a functional reasoning for generating from scratch a first-shot version of the artifact (Tsang *et al*, 1990) and a delta reasoning for fine-tuning it. This paper presents an extended version of our functional reasoning framework by encompassing, in addition to its generative capabilities, patching functionalities that may be deployed during the elaboration process of a first-shot version of the artifact. Note that delta reasoning is still relevant to fine-tune the completely generated artifact. We should point out that the idea of functional reasoning is not new: Freeman and Newell reported in the early 70's the application of such a reasoning to the design of symbol tables (Freeman & Newell, 1971). Since then, there have been proposals of functional reasonings applied to the design of different types of artifacts: electromechanical assemblies (TROPIC (Latombe, 1977)), electronic circuits (PEACE (Dincbas, 1983)), paper handling systems (PRIDE (Mittal *et al*, 1986)), configuration of microcomputer hardware (MAPLE (Bowen, 1986)), fixture set-up (SERF (Ingrand, 1987)). Some related research tends to emphasize the language aspect of functional representations (Goel & Chandrasekaran, 1989) while others the qualitative nature of the reasoning (Williams, 1990).

The rest of the paper is organized as follows. Section 2 presents basic principles of the functional reasoning framework. Section 3 discusses major related issues and reports an enhancement of the constraint propagation process. Section 4 addresses supplantation, a mechanism we propose for incorporating patching capabilities. Section 5 reports a brief discussion of SMACK, an implementation of described principles, and discusses limitations and enhancements.

¹ The interested reader may refer to (Hall, 89) for a recent survey of works in analogy. Of special relevance are principles advocated by Carbonell: transformational (Carbonell, 83) and derivational analogy (Carbonell, 86).

2. Functional Reasoning Framework

2.1 Background

Designing an artifact is regarded as the process of concurrently determining the functional structure (F) and the physical structure (P) of an artifact given an end-user assessment of it called its specifications. The functional structure expresses goals the artifact has to meet. In our framework, this is encoded as a tree of goals. The physical structure describes how the artifact is organized physically: it gives an account of its constituents and specifies relationships among them. In our framework, this is encoded as a graph. At this stage, we may not be more specific about this graph unless additional assumptions regarding the artifact are made. Design by assembly constitutes a subclass of design problems where constituents of the artifact have to be selected from a predefined library of building blocks.

F and P are non-isomorphic *a priori*². This allows goal satisfaction interferences (positive and negative) to be explicitly reasoned about. Positive interferences (opportunism) occur when some goal G' turns out to be fulfilled by some component C , already in place, meant to fulfill some other goal G . In this case, no additional component is needed. Negative interferences occur when the installation of component C to fulfill goal G disturbs the fulfillment of goal G' by the component C' already in place (because of the physical relationship between C and C'). This may be the case, for instance, when choosing an amplifier to fulfill some gain function because the heat it liberates may perturb components in its neighbourhood.

Specifications are not synonymous with functional structure. In effect, specifications only express a very particular viewpoint of the artifact: this viewpoint is intrinsically approximate and non definitive since it is based on expectations of what the artifact will look like. Evidence for this is the well-known fact that errorless specifications may only be articulated when the artifact has been designed. The functional structure is quite different: it encodes goals to be met by the artifact.

2.2 Artifact Representation

A. Functional Representation

The functional representation is a tree of goals where nodes are multi-attribute objects. A node represents a goal to fulfill (i.e. function) and its attributes characteristics of the goal. Arcs are oriented and represent set inclusion relations. The fact that N has children N_1 through

² This is not the case of systems such as PEACE (Dincbas, 83) where the physical organization of the artifact is isomorphic to its functional structure (assimilated here to the expression of the impedance of the electronic equipment: "*" corresponds to "in series" and "+" to "in parallel").

N_m means that to achieve the goal designated by N , one may achieve goals designated by N_1 through N_m .

B. Physical Representation

Two reasons motivate our hypothesis (see below) regarding the structure of the artifact: first, it fits our case-study; second, it makes plan representation techniques directly relevant to our problem (total order of precedence relation).

Hypothesis on the structure of the Artifact

The artifact is strongly linear³, i.e., its components are all in series. Put differently, the "upstream" relation among components of the artifact defines a total order.

A powerful technique for handling plans consists of employing a non-linear representation like the ones epitomized in NOAH (Sacerdoti, 1977) and DCOMP (Nilsson, 1980, pp 333-341). The leverage of such a technique stems from the conceptual dissociation between existence and organization (in the overall structure). The strong similarity between temporal precedence among actions and the upstream relation among components led us to build upon this non-linear technique for representing the physical structure of the artifact. This implicitly defines two types of operations: (1) component linearization that effects transformation of a partial order into a total order and (2) component specification that permits progressive detailing of characteristics (attributes) of the components. Note that for convenience our non-linear representation contains, in addition to standard components, two fictive components *start* and *end* which respectively correspond to the most upstream and the most downstream constituents. Now, it is not always possible to assign explicitly a component to a goal (non-isomorphism). Two additional operations are introduced to handle this problem: (3) component dissociation that allows splitting of an already in place component into a collection of components and (4) component grouping that effects the inverse operation. As a result, the goal/component relation is slightly more complex. Three cases are possible: (1) component C fulfills goal G , (2) component C' fulfills goal G and C' is an ancestor of C (case when C results from a dissociation), (3) component C' fulfills goal G and C' is some descendant of C (case when C results from a grouping).

The physical representation is best thought of as made up of three subgraphs. The first subgraph is a non-linear graph of components related by "upstream" relations (this subgraph contains in addition two fictive *start* and *end* components). Nodes of the second subgraph represent remnants of components that have been dissociated: they are related to components of

³ At the end of the paper (section 5), we will see how the artifact representation may be enhanced to accomodate a softened "weakly linear" version of the hypothesis whereby components of the artifact are either in series or in parallel, i.e., the "upstream" relation defines a partial order.

the first subgraph by "dissociation" relations. Nodes of the third subgraph represent remnants of components that have been grouped: they are related to components of the first subgraph by "grouping" relations.

It is worth pointing out that nodes of the physical representation encode in fact constraints that have to be met by candidate building blocks. They therefore amount to variables that have to be instantiated with building blocks from the library, hence the need for some matching process between the physical representation and the library. Note that no matching process is required for the functional representation, since the latter's nodes correspond to instantiated goals.

2.3 Basic Tasks

Our functional reasoning comprises five operational tasks: functional decomposition, materialization, physical organization, functional-physical verification and constraint propagation. There exists an additional control task which decides which task (among the five) to execute based on the current problem-solving context. In our current implementation, the control task is overly simple: it picks at random any executable task (we nonetheless expect overall performance may be significantly enhanced by endowing the control task with the capability of building and following lines of reasoning).

A. Functional decomposition

Overall goals are high-level in general. For this reason, there exists no building blocks in the library that fulfill them directly. As a result, they have to be broken down into lower-level goals. (This is an example of classical problem reduction (Nilsson, 1971)). This type of decomposition is *functional-dependent*., since the decomposition process is directly motivated by goal achievement. There also exists a physical-dependent decomposition whose motivations are either repair of negative interferences (e.g. components C and C' have to be isolated from one another) or explicitation of residual goals (e.g. partial fulfillment of goal G by component C leaves us with residual goal G'). Functional decomposition has recourse to knowledge of the form:

If G [Context] then decompose G into G₁ ... G_n

Note : The context encompasses current functional and physical structures as well as specifications and library of building blocks.

B. Materialization

Materialization is the process that takes place when decomposed goals are sufficiently detailed. We identify two types of materialization: *physical* and *non-physical*. Physical materialization consists of assigning a building block to a goal or a collection of goals. Note that this may either involve creation of a new building block or use of a building block already

in place. The case where a collection of building blocks is to be assigned to a goal or a collection of goals is handled using dissociation (see § 3.1.A for underlying rationale). Non-physical materialization consists of posting global constraints on the physical representation (e.g. the total mass of the artifact should be less than 800 g). Note that local constraints are handled by refinement primitives. Materialization has recourse to knowledge of the form:

If G [Context] then materialize G by C
If G [Context] then materialize G by R (att_{φ(1)} of C₁) ... (att_{φ(n)} of C_n)

C. Physical Organization

Activities of physical organization include linearization, refinement, dissociation and grouping (as explained in §2.2.C). The need for linearization stems from the partial nature of the "upstream" relation among components of the physical representation. Linearization constraints express how components are positioned with respect to one another (at least partially). Refinement details the artifact progressively during the process: it specifies the type and characteristics (attributes) of components. Dissociation splits up a component into a collection of components. This is particularly useful, for instance, when an amplifier in an early stage of the process takes the form of two separate amplifiers in the final artifact. Last but not least, grouping allows a collection of components to be regarded as a single component (e.g. grouping of a collection of electric modules on a special circuit board). Physical organization has recourse to knowledge of the form:

If C_i C_j [Context] then Linearize C_i C_j
If C [Context] then Refine (att_k of C) to Dom_k
If C [Context] then Split C into C₁ ... C_n
If C₁ ... C_n [Context] then Group C₁ ... C_n into C

D. Functional-Physical Verification

Design knowledge is typically unsound in the sense that even its scrupulous application may lead to incorrect artifacts. This means that inconsistencies must be checked for during the design process. We distinguish two types of inconsistencies. The first one may be detected in the representation, for instance, cycle formation for an antisymmetric relation, empty domain for an attribute, etc. The second type is more semantic and includes domain-specific problems that may only be spotted by specific simulation packages. Functional-physical verification knowledge corresponds to so called integrity constraints and is encoded in the form:

If Condition₁ ... Condition_n then inconsistency

E. Constraint Propagation

This task includes two types of constraint propagation.

The first type of constraint propagation ensures that consequences of constraints specified in a constraints list are enforced. This constraints list comprises physical laws the physical representation has to comply with (e.g. algebraic sum of losses and gains must be equal to overall specified gain) as well as wishes expressed by the user (e.g. avoid "mmic" technology) along with artifact specifications properly speaking. One particularity of this constraint propagation is that its activation may become relevant upon completion of practically any task. There are two reasons for this: first, the constraints list evolves with time due to non-physical materialization discussed in §2.3.B and second, consequences of global constraints of the constraints list (due to their global nature) are strongly dependent on the current context.

The second type of constraint propagation is related to the library of building blocks: it exploits inherent constraints of building blocks to prune the search space associated with the physical representation. This constraint propagation is relevant whenever the set of candidate building blocks that match a component is directly or indirectly updated. It is based on the boomerang effect discussed in §3.3.

3. Issues and Enhancements

3.1 Dynamic Referencing Issue

One central problem of our functional reasoning framework is the impedance mismatch between knowledge and representation. In effect, design knowledge is static (defined lexically) whereas the entities they refer to (goals of the functional structure and components of the physical structure) evolve dynamically with the design process. Put differently, the problem consists of finding a method which allows a piece of knowledge to refer non-ambiguously to any dynamic entity of the artifact representation (goal or component).

Our idea consists of injecting some asymmetry between the two structures so that the more dynamic structure may be accessed via the less dynamic structure. The latter would then be accessed directly by static knowledge.

Hypothesis

The functional structure is less dynamic than the physical structure. Supporting evidence: goals are more fundamental than matter (i.e. physical components) and are therefore less subject to change.

As a result of this hypothesis, goals of the functional structure may be mentioned and referred to directly in expressions of knowledge. Non-ambiguous designation of goals is guaranteed based on the tree organization of the functional structure by imposing explicitation of the complete path (from root goal to designated goal) in case two goals bear the same name (analogous to file access). The problem now is accessing a component from a goal. There is no

reason *a priori* that the relation between F and P be a function (one goal may be satisfied by a collection of components). One solution consists of defining a function in related spaces. The relation between F and 2^P is indeed a function but this does not seem to simplify the problem. An alternative solution consists of forcing the relation between F and P to be a function, but this has the disadvantage of prohibiting a goal from materializing a collection of components. In fact, this is not really a problem since dissociation primitives offer a nice detour to express the same thing.

This discussion establishes the fact that components are not referenced as such but as objects that materialize some goal. Non-ambiguity is achieved by having recourse to grouping and dissociation relations (e.g. first child of that dissociated object which materialized goal G) since the component may have resulted from a grouping or a dissociation. It is also handy to have recourse to "upstream" relations to access a component (e.g. the object that is upstream of the object that materializes G) but this is not foolproof (components that meet this condition may not be unique) and should therefore be used in situations in which unicity is guaranteed by some other means.

3.2 Contingent Nature of Absence Issue

Another issue of particular relevance to our functional reasoning framework is guarantee that knowledge is not abusively applied. Consider a piece of knowledge recommending certain actions to be taken if some component C is absent from the final artifact. If no special precaution is taken, that piece of knowledge may be applied to an artifact containing C since the absence of C is true all the way up to the moment when C is introduced. Before going further, we should distinguish two cases. In case 1, knowledge pertains to the final artifact as in the example above. In case 2, knowledge pertains to any state in the design process. It is clear that only case 1 is problematic.

Let us make a few observations regarding presence and absence in our framework. If an entity (goal or component) is absent from the final state of the artifact representation (say state n) then it is absent from any state i of the artifact representation ($i \leq n$). Conversely, if an entity is present in the artifact representation at some state i then it is present for any subsequent state j of the artifact representation ($j \geq i$). This is because no retraction is effected: grouping and dissociation preserve components on which they act (see § 2.2.B). Summarising, presence is sufficient for effective presence, while absence is necessary for effective absence.

The solution we propose (and adopt) is rather simple and a little drastic: it consists of regarding knowledge that comprises absence conditions as pertaining to any state of the artifact (case 2). As a result, knowledge pertaining to the final artifact should comprise only presence conditions.

3.3 Boomerang Effect

As explained earlier (see § 2.2.B), components of the physical representation are not physical entities but multi-attribute variables awaiting instantiation from building blocks of the library. Now this instantiation may be carried out in such a way that the search space is reduced more than what we might expect at first sight based on what we call the boomerang effect. We explain this phenomenon below.

Consider the following example. Suppose I go to a store to buy a gift for a friend. Suppose I do not wish to spend more than 50\$, carry more than 1000 g and the capacity of my bag does not exceed 3000 cc. Suppose 5 articles are available on the shelf and upon query about the price, only 3 of them are eligible. The boomerang effect is the fact that the answer to the query not only constrains the price (called the key here) but also the weight and the volume (we coined this expression to vehicle the metaphor of throwing a query and receiving back an answer not only to the query but also to other attributes of the variable (Tsang, 1990b)). In the example below, if articles a1 through a5 take on values indicated below, variable v which evaluated to (price [0 50] weight [0 1000] volume [0 3000]) reduces to (price {25 50} weight {400 500 1000} volume {900 2000}) upon query.

a1	a2	a3	a4	a5
price 25	price 75	price 150	price 50	price 50
weight 400	weight 300	weight 2000	weight 1000	weight 500
vol 2000	vol 500	vol 500	vol 900	vol 900

Let $P_{att_k}^{bb_i}(v)$ represent the fact that attribute att_k of variable v takes on the value of the attribute att_k of building block bb_i (i.e. $v.att_k = bb_i.att_k$). Let $P_{att_k}^\theta(v)$ represent the overall effect of previous constraints regarding attribute att_k of variable v at state θ of the graph.

A simple query regarding the attribute key enforces the constraint below. Note that L is the set of indices of all available building blocks. We will refer to this approach as the simple key querying method:

$$P_{key}^\theta(v) \wedge \bigvee_{i \in L} P_{att_k}^{bb_i}(v)$$

The boomerang effect of the query enforces additional constraints, one per attribute, which for some att_k other than the key may be expressed as:

$$P_{att_k}^\theta(v) \wedge \bigvee_{i \in I} P_{att_k}^{bb_i}(v) \text{ where } I = \{ i \mid P_{key}^{bb_i} \Rightarrow P_{att_k}^\theta \}$$

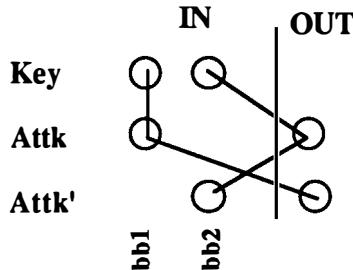
Note that I is simply the set of indices of those building blocks compatible with the key attribute of variable v at state θ . The extra constraining power of the boomerang effect is now explicit. If

$C_{att_k}^\theta(v, key)$ denotes the above constraint, the constraint enforced by the boomerang effect is exactly given by the expression below. We will refer to this approach as partial boomerang.

$$\wedge_{1 \leq i \leq m \& att_i \neq key} C_{att_i}^\theta(v, key)$$

Note that in general, the above expression is weaker than what is actually the case.
Explanation : among constraints impinging on some attribute att_k there may exist some

predicate $P_{att_i}^{bb_j}$ for which building block bb_h fails to satisfy some other attribute $att_k' \neq att_k$. From a building blocks perspective, this means there exists a building block bb_h which satisfies the key attribute, some attribute att_k but not some other attribute att_k' as depicted in the schema below.



In the gift example, if the volume of article a4 was 4000 instead of 2000, then the weight of 1000 (due to a4) would be retained though not supported by any article. To get around this problem, one should ensure that all considered building blocks have acceptable values for every attribute of the variable. The constraint relative to attribute att_k is given by:

$$P_{att_k}^\theta(v) \wedge \vee_{i \in J} P_{att_k}^{bb_j}(v) \text{ where } J = \{ j \mid \{ \wedge_{1 \leq k \leq m} P_{att_k}^{bb_j} \} \Rightarrow P_{att_k}^\theta \}$$

Simply note that J precisely achieves the aforementioned condition: J contains those building blocks that are compatible with all attributes of v . If $D_{att_k}^\theta(v, key)$ denotes the above constraint, then enforced constraint is exactly given by:

$$\wedge_{1 \leq i \leq m \& att_i \neq key} D_{att_i}^\theta(v, key)$$

We call this approach full boomerang or a full-fledged multi-attribute query answering method.

Quiescent states (those in which no current constraints may be propagated) avoided by boomerang procedures are those states in which imposing the only constraint regarding the key attribute maintains quiescence which would be destroyed by enforcement of constraints resulting from partial or full boomerang. Remark that quiescent states avoided by partial boomerang are subsumed by those avoided by full boomerang since additional constraints entailed by full boomerang imply those entailed by partial boomerang (see implication below).

$$\wedge_{1 \leq i \leq m \& att_i \neq key} D_{att_i}^\theta(v, key) \Rightarrow \wedge_{1 \leq i \leq m \& att_i \neq key} C_{att_i}^\theta(v, key)$$

We give below a simple version of the boomerang procedure. The interested reader may refer to (Tsang & Wrobel, 1991) for more powerful boomerang procedures as well as their analyses.

```

procedure partial-boomerang (v key  $\hat{D}$ )
  L := if v.bb defined then v.bb else library
  L' := {l  $\in$  L | l.key  $\in$  v.key}
  forall att  $\in$  ATT(v) do
    w.att :=  $\cup_{l \in L} l.att$ 
    v := v  $\cap$  w
    v.bb := L'
    v.key :=  $\hat{D}$ 

```

We make some remarks on the algorithm. In addition to regular attributes $ATT(v)$ of variable v , we consider a distinguished attribute bb (for building blocks) which records the set of candidate building blocks at the last evaluation (it is undefined otherwise). The "library" designates the whole set of building blocks available in the library. For clarity reasons, conflict handling (e.g. ensure that $D \cap \hat{D} \neq \emptyset$ and that $L' \neq \emptyset$) is not inserted though quite simple to take into account.

4. Patching through Supplantation

4.1 Drawbacks of Backtracking

Backtracking is not entirely satisfactory when modeling patching capabilities. In addition, it has shortcomings regarding efficiency and retraction of inferences in some cases.

Inadequacy to Handle Patching. We may distinguish two kinds of primitives in a generative process: refinement and rearrangement. Refinement primitives basically effect solution space reduction, whilst rearrangement primitives effect non reduction modification of the artifact (e.g. addition of a component). Backtracking is fine when handling refinement primitives but runs into severe problems when tackling rearrangement primitives. Consider, for example, a situation where an object A is created (consequence: A is present) and is then split into A_1 and A_2 (consequence : A is absent). This provokes a conflict which backtracking attempts to fix by retracting the two decisions "create A " and "split A ". This is indeed quite inelegant since decision 2 may not be applied (it depends on decision 1). This problem is well-known in planning and gave rise to situational calculus. We propose a related solution which in this case amounts to supplanting " A is present" by " A is absent" to solve the conflict.

Gross Efficiency: Consider the following situation where decisions D_1 through D_n have been applied using rules of the form $D_k \implies D_{k+1}$. Suppose the culprit is D_1 (e.g. amplification = 60 dB) and its substitute D_1' (e.g. amplification = 61 dB). To accomplish this substitution, backtracking will undo all decisions D_1 through D_n and apply D_1' through $D_{n'}$. This is clearly grossly inefficient when many D_k' amount to D_k (often the case in practice) and the substitute decision is close to the culprit decision considering that the knowledge base does not exhibit chaotic behaviour.

Retraction of Useful Inferences. Consider the following situation where D_1 asks for insertion of A, D_2 for insertion of B on the left of A, and D_3 for insertion of C on the left of B. If A, B and C are collinear, one may deduce that C is on the left of A by transitivity. Suppose now that B is to be removed. This is effected by retracting D_2 and its consequences, i.e., B left of A and C left of A. In the collinear case, C is still on the left of A, but this fact has been retracted. Explanation: the intended real world interpretation exists independently of the formalization that is made. In this example, that C is on the left of A does not depend on C left of B and B left of A though it is convenient to "deduce" it this way. Furthermore, facts deduced by the logical apparatus give a glimpse of what the artifact will look like and for this reason discarding logically unsupported facts that do not provoke failures is clumsy though correct.

4.2 Supplantation

Supplanting a fact F by another fact F' amounts to undoing F, applying F' instead and getting rid of proper consequences⁴ of F incompatible with F' . In case F' still leads to a conflict, the failure may either be handled by backtracking (the meta-decision of supplanting is undone) or by an additional supplantation whereby a culprit has to be identified and supplanted by a substitute to be determined. The major difference with backtracking is that proper explicit consequences of F compatible with F' are preserved (they have the status of hypotheses though).

Let us first present essential elements of our ontology. A fact is either a decision or a logical consequence of decisions. A major decision is a piece of advice recommended by the inference engine. A minor decision is a logical implicate of decisions (major or minor) made explicit, i.e., accessible directly without deduction. An implicit consequence is a logical implicate of decisions left as such (that consequence needs deduction to be accessed). It follows that a fact may be a major decision and a consequence (implicit or explicit) at the same time. We consider four distinct entities: a decision structure, an artifact representation, an inference engine and a constraint propagator. The decision structure records two types of dependencies among decisions: cognitive dependencies stemming from the inference engine (noted \implies) and logical entailments from the constraint propagator (noted \Rightarrow). An additional supplantation

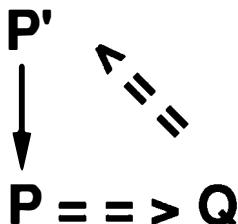
⁴ Proper consequences of a fact F are those consequences that disappear if F is removed.

relation is also considered whereby $P \rightarrow Q$ means that P supplants Q . The artifact representation encodes facts corresponding to the current state of the artifact under construction. These facts correspond to the logical closure of decisions in the decision structure (more exactly, unsupplanted decisions). The inference engine makes decisions (major decisions) given some knowledge base. The constraint propagator makes explicit logical entailments of decisions and communicates them both to the decision structure and the artifact representation.

We define below what we mean by hypothesis and immediate logical implicants of a decision.

Definition (hypothesis)

A hypothesis is defined as a fact whose implicants (cognitive or logical) are all supplanted. Example: Q is a hypothesis in the figure below.



Definition (immediate logical implicants)

We define immediate logical implicants of a decision d to be the maximal set ILI(d) of decisions such that for any e \in ILI(d) either $e \Rightarrow d$ or there exists an $e' \in$ ILI(d) such that $e' \Rightarrow e$.

Let us now make some observations regarding the supplantation process before giving the algorithm. First, if the fact to be supplanted is absent from the decision structure, then it has to be made explicit in the first place by the constraints propagator (minor decision). Second, when supplanting F, all immediate logical implicants of F have to be removed, otherwise F would still be deducible (consistency condition). It is clear that these implicants should not be discarded from the decision structure otherwise backtracking would not be possible. This may be achieved by introducing a "screening decision" whose behaviour is analogous to backtracking : a decision supplanted by a "screening decision" is simply undone and removal of the "screening decision" amounts to enforcing the decision again. This "screening decision" is attached (noted \leftrightarrow) to the supplanting decision to ensure that upon its removal, the "screening decision" is removed, hence the screened decisions enforced anew. Third, logically implicated facts need not be removed since they have the status of hypotheses upon supplantation of F by

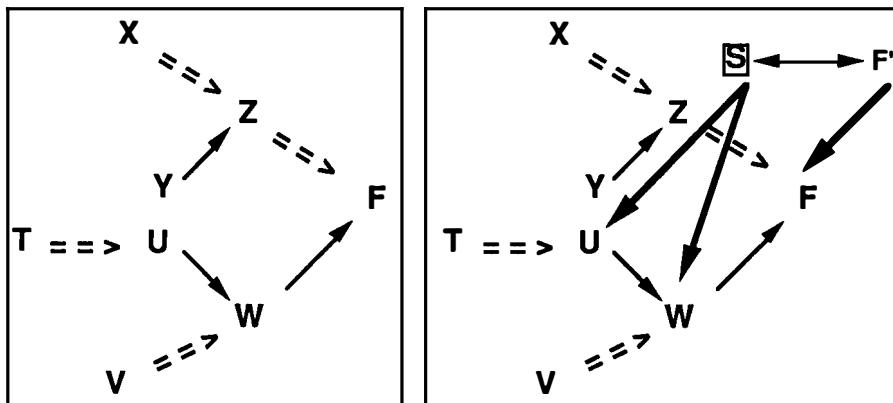
F' . In case they provoke a failure, they will eventually be removed by backtracking or supplantation. Fourth, cognitive implicants of F pose no problem.

```

program supplant ( $F F'$ )
begin
  if  $F$  implicit then make  $F$  explicit using constraint propagator
  install supplantation  $F \rightarrow F$ 
  ILI := immediate logical implicants ( $F$ )
  forall I in ILI do
    install supplantation screening-decision  $\rightarrow I$ 
  if ILI non empty then install  $F \leftrightarrow$  screening-decision
  forward-chain
end

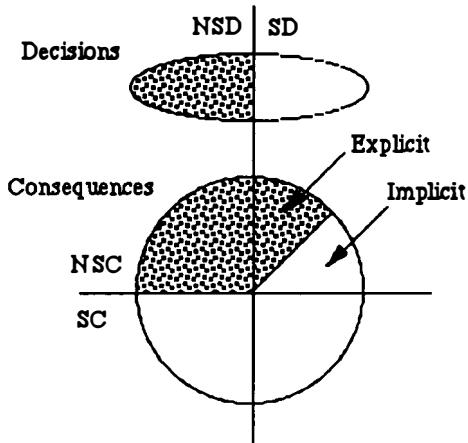
```

The example below depicts a situation where F is to be supplanted by F' . This is accomplished by determining $ILI(F)$ which evaluates to $\{U W\}$. These immediate logical implicants are supplanted by a screening decision, say S , which is attached to F' the supplanting decision of F .



We now discuss what is exactly represented when using supplantation. Let NSD and SD be respectively non-supplanted and supplanted major decisions (stemming from the inference engine). Recall that logical consequences of NSD and SD may or may not be supplanted leading to 4 distinct classes: NSD-NSC, NSD-SC, SD-NSC, SD-SC (D stands for decisions and C for consequences). Consequences may either be made explicit (by the constraint propagator) or left implicit. Consequences of NSD are accessible, but those of SD are not since SD decisions may not participate in the deductive process. Things are a bit more tricky though, since certain consequences of SD decisions are nonetheless accessible: they are

precisely those that were made explicit before supplantation of their implicit decisions and have since then not been discarded because of their compatibility with the current context. Let SD-NSC-E and SD-NSC-I designate respectively the explicit and implicit parts of SD-NSC. What is represented is exactly: $NSD \cup NSD-NSC \cup SD-NSC-E$ (see figure below).



4.3 Pros and Cons

The major advantage of supplantation is that it provides a means of handling refinement and rearrangement primitives in the same framework. In addition, it takes advantage of the fact that both the culprit and the substitute decision are known before effecting any actions hence enhanced efficacy. Another important advantage results from the presence of non supplanted minor decisions i.e. SD-NSC-E. In effect, this helps alleviate the problem of hypothesis generation. Finding the right hypotheses to get rid of quiescent states (no domain knowledge applicable) is no simple matter and supplantation provides a simple means of making relevant and plausible hypotheses. Note that though these hypotheses are not necessary conditions of the final artifact like in Finger's supersumption (Finger, 1987), it is often the case that they are used in practice. The supplantation mechanism also leads to a shift from Stefk's least-commitment principle epitomized in Molgen (Stefik, 1980) to a more attractive "weak-commitment" strategy. Note that commitments are of a particular nature: they are decisions that were in an earlier stage of the process cognitively or logically supported by some decision.

The presence of non-supplanted minor decisions, i.e. SD-NSC-E, also entails disadvantages in particular contingent formation of vestiges. In effect, some part of the artifact may serve no purpose (though it did in a previous intended version of the artifact) and yet be preserved on the basis that it is not incompatible with the current state of the artifact. This happens all the more often when ungrounded decisions are numerous (over-commitment of the whole process). In such cases, it is more reasonable to backtrack instead of employing

supplantation. One way out of this problem precisely consists of restricting usage of supplantation to fix only those conflicts that result from rearrangement primitives.

5. Discussion

5.1 Implementation

Almost all concepts discussed in this paper have been implemented in the SMACK program, an expert system for designing electronic equipment for satellites. Currently, SMACK generates satisfactory block-schema diagrams, electronic characteristics of their constituents and compliance matrices (comparison between performance of artifact and user specifications). It gives quite good results for each of the four specifications on which it has been experimented. SMACK is written in Spoke™ - an object oriented language Spoke commercialized by Alcatel-ISR - and is operational on SUN/DEC workstations. It embodies to date about one hundred rules and about the same number of building blocks (Tsang & Cardonne, 1990).

5.2 Limitations and Improvements

One of the most restrictive limitations is probably the "strongly linear" hypothesis relative to the structure of the artifact. This may be weakened to a "weakly linear" hypothesis whereby components of the artifact may either be in series or in parallel instead of being necessarily in series. This is all the more interesting since applications such as electronics require components to be effectively in parallel in certain cases. The general idea is the following. We introduce a fictive *parallel* component in addition to *start* and *end* components seen before and each time a really *parallel* decision is made regarding certain groups of components, these groups of components are merged into a single *parallel* component whose children are precisely the groups in question. The only difference with the previous framework is the increased complexity of the graph and the resulting bookkeeping (consistency checking). Note that feedback loops are still not handled: a further weakening of the hypothesis is required.

Another major limitation is the definition of the library of building blocks. Currently, when no building block is found, the user is prompted to answer whether or not it is reasonable to envisage a particular building block. A major enhancement of the system would be the possibility of defining the library of building blocks not only explicitly in terms of a list of building of building blocks, but also in terms of principles and guidelines underlying the technology of building blocks. As a result, when no explicit building block is found in the library, the system may take the initiative of designing by itself building blocks that would fit the situation at hand.

Acknowledgments. Our sincere thanks to Philippe Durand, Yannick Descotte and Reid Simmons for fruitful discussions we had with them on shortcomings of a previous version of our functional reasoning. Thanks also to Gilles Cardonne, Bernard Wrobel and Nathalie Pfeffer who contributed in making principles expounded in this paper operational in SMACK. Special thanks to Gary Walker for having improved our English.

References

- Bowen J. (1986). Automated Reasoning via Functional Reasoning Approach. Artificial Intelligence and its Applications, edited by A. G. Cohn and J. R. Thomas (John Wiley), Proceedings of AI in Simulation of Behavior, pp 79-106.
- Carbonell J.. (1983). Learning by Analogy: Formulating and Generalizing Plans from Past Experience. Machine Learning : An Artificial Intelligence Approach, pp 137-161, Eds RS Michalski, JG Carbonell, TM Mitchell, Morgan Kaufmann.
- Carbonell J.G. (1986). Derivational Analogy: A theory of Reconstructive Problem Solving and Expertise Acquisition. Machine Learning : An Artificial Approach, pp 371-386, Eds RS Michalski, JG Carbonell, TM Mitchell, Morgan Kaufmann.
- Goel A and Chandrasekaran B. (1989). Functional Representation of Designs and Redesign Problem Solving, 11th IJCAI 89, Detroit, pp 1388-1394.
- Dincbas M. (1983). Contribution to the Study of Expert Systems (in French). PhD thesis (Thèse de docteur ingénieur), ENSAE, Toulouse.
- Hall R.P. (1989). Computational Approaches to Analogical Reasoning: A Comparative Analysis, Artificial Intelligence 39 pp 39-120.
- Ingrand F. (1987). Inferencing Forms from Functions. Application to Fixture Set-up Design (in French). PhD thesis (Thèse de Doctorat de 3ème cycle), INPG, Grenoble.
- Finger J.J. (1987). Exploiting Constraints in Design Synthesis. PhD thesis, Stanford University.
- Freeman P. and Newell A. (1971). A model for Functional Reasoning in Design, 2nd IJCAI, London, pp 621-640.
- Latombe J.C. (1977). Application of Artificial Intelligence to Computer Aided Design (in French), PhD thesis (Thèse de Doctorat d'Etat), Grenoble.
- Mittal S., Dym C.L. and Morjoria M. (1986). PRIDE: An Expert System for the Design of Paper Handling Systems, IEEE Computer Magazine, July 1986, pp 102-114.
- Nilsson N.J. (1971). Problem-Solving Methods in Artificial Intelligence, McGraw-Hill.
- Nilsson N.J. (1980). Principles of Artificial Intelligence. Morgan Kaufmann Publishers Inc.
- Sacerdoti E.D. (1977). A Structure for Plans and Behavior, Elsevier North Holland.
- Stefik M.J. (1980). Planning with Constraints. PhD thesis, Stanford University.
- Tsang J.P. (1990). Reasoning Paradigms for Intelligent CAD. 4th Eurographics Workshop on Intelligent CAD Systems, Added Value of Intelligence to CAD, Chatenay-Hôtel de Mortefontaine, France.
- Tsang J.P. (1990b). Constraint Propagation Issues in Automated Design, Expert Systems in Engineering: Principles and Applications, International Workshop, Technical University of Vienna, Lecture Notes in Artificial Intelligence (eds: G Gottlob & W Nejdl), Springer-Verlag, no 462, pp 135-151.
- Tsang J.P., Wrobel B. and Pfeffer N. (1990). A Functional Reasoning for Automating the Design Process (in French), 10th International Conference on Expert Systems and their Applications, Avignon, General Conference, Vol 1, pp 151-166.

- Tsang J.P. and Cardonne G. (1990). A Knowledge-Based Design of a Telecom Equipment, Alcatel Technology Review 90, Marcoussis.
- Tsang J.P. and Wrobel B. (1991). Enhancing Constraint Propagation for multi-attribute labeling (Forthcoming).
- Williams B. (1990). Interaction-Based Invention: Designing Novel Devices from First Principles, AAAI 90, Boston, pp 349-356.

The cognitive psychology viewpoint on design: examples from empirical studies

W. Visser*

Institut National de Recherche en Informatique et en Automatique
Rocquencourt BP 105
78105 Le Chesnay Cedex France

Abstract. This paper presents, through examples from three empirical design studies, the analysis which cognitive psychology makes of the mental activities involved in design. These activities are analyzed at three levels: the global organizational, the strategic, and the process level. The claim is defended that knowledge-based design systems could benefit if they took into account the data on design revealed by such an analysis.

INTRODUCTION

This paper studies what Eastman (1970) called "intuitive design," that is "the procedures that designers have implicitly derived from their own design experience through case studies in school or from professional experience" (p. 21). Eastman judges this study important because design methodologies need to be evaluated against data from actual design activities and because the specifications of computer-aided design are to be based on these same data.

However, today in 1991, knowledge-based design systems are still not generally based on data concerning actual design activities. Results obtained in cognitive-psychology studies are not really exploited in A.I. applications. Often the relevance of these data is not even mentioned.

The purpose of this paper is therefore to present, through examples of empirical design studies, the analysis that cognitive psychology makes of the mental activities involved in design. Presenting these studies at an "A.I. in Design" conference, we assert that

- the approach taken in cognitive psychology towards design as a problem-solving activity allows relevant aspects of the activity to be revealed; and
- the development of A.I. design systems could benefit if it took into account the results of these studies.

* Thanks to A. Bisseret, B. Trouse, P. Falzon, F. Darses and R. James for help in the preparation of this paper.

Design studied from the cognitive psychology viewpoint

Cognitive psychology research on design mainly involves the study of

- the *knowledge* involved in designing and *processing* (using) this knowledge;
- the *organization* of the actual activity and the *strategies* implemented by designers.

Most empirical design studies have been conducted in the domain of software design (see Hoc, Green, Samurçay & Gilmore, 1990). Other examples are: architectural design: Eastman (1970); errand planning: Hayes-Roth & Hayes-Roth (1979); text composition: Hayes & Flower (1980); computational geometry algorithm design: Kant (1985); mechanical design: Whitefield (1986) and Ullman, Staufer & Dietterich (1987); traffic-signal setting: Bisserset, Figeac-Létang & Falzon (1988); computer-network design: Darses (1990).

These studies mainly concerned solution development by knowledge evocation; design strategies were also examined.

Organization of the paper

The cognitive aspects of design will be analyzed at three levels: the global *organizational*, the *strategic*, and the *process* level. These analyses will be discussed through the presentation of three studies conducted on three different types of design tasks:

- a study on the design of functional specifications will be used to present the global organization of a design activity (Section 2);
- a software design task will illustrate expert design strategies (Section 3);
- a composite-structure design task has been analyzed with respect to different types of problem-solving processes in design (Section 4).

The final section will show how the results of these studies are relevant for A.I.

ORGANIZATION OF THE DESIGN ACTIVITY¹

Focus on plan deviation. Early empirical design studies, especially in the domain of software design (see Visser & Hoc, 1990), generally characterize the human design activity as being hierarchically organized, in other words, following a pre-established plan. They assert that the designer decomposes his problem according to a combined top-down, breadth-first strategy. The authors conclude that decomposition using such a step-wise refinement plan is the global design control strategy.

Our observations in preliminary studies led us to become slightly suspicious of these conclusions: we observed top-down and breadth-first decomposition strategies to be implemented only *locally*, in other words, their combination did not seem to be the control strategy of the design activity at the global-organizational level. In an attempt to clarify this point, we examined the global organization of the design activity, focusing on possible deviations from a pre-established plan.

¹ This section presents in a somewhat revised way results presented in Visser (1988, 1990b).

Method

Task: Specification. A mechanical engineer had to define the functional specifications for the control part of an automatic machine tool.

Subject: A mechanical-design engineer. The observed engineer had more than ten years of professional experience in the machine-tool factory where he was working.

Data collection: Observations & Simultaneous verbalization. During a period of three weeks, full time observations were conducted on the engineer involved in his task.

The engineer's normal daily activities were observed without any intervention, other than to ask him to verbalize his thoughts during his problem-solving activity as much as possible (see Ericsson & Simon, 1984). Notes were taken on his actions; all documents which he produced during his work were collected.

Results. Preliminary remarks

The engineer's global plan. In order to compare the engineer's actual activity with his representation of his activity, the engineer was asked to describe his activity. He presented it as following a hierarchically structured plan (see Visser, 1988, 1990b). In this text, this plan is referred to as the engineer's "global plan," as opposed to local plans which the engineer was observed to formulate and/or to implement.

Notwithstanding this global plan which the engineer claims to follow, the actual activity which we observed is in fact opportunistically organized. Only as long as they are cognitively cost-effective, the control selects for execution actions proposed by the global plan ("planned" actions). As soon as other actions are more interesting from a cognitive-cost viewpoint, the engineer deviates from his global plan in favour of these actions ("deviation" actions).

Terminology: Deviation action - Planned action. Two terms will be used as shorthand:

- "deviation action" for "action proposed as an alternative to the planned action and selected by the control, which has led to a plan deviation;"
- "planned action" for "action proposed by the global plan and selected by the control."

Definition: Specification action - Design component. Specification actions consist in defining design components on several attributes, "descriptors."

The design components to be defined are machine-tool cycles or elements of these cycles, i.e., machine-tool functions or machine-tool operations.

The attributes ("descriptors") on which they are defined are: "identifier," "duration," "starting conditions," "ending conditions," and "physical device."

A blackboard model of the specification activity. Various authors have shown blackboard models to be particularly appropriate for modeling opportunistically organized activities (see Bisserset, Figeac-Létang & Falzon, 1988; Hayes-Roth & Hayes-Roth, 1979; Whitefield, 1986). We chose such a model for the specification activity.

As the organization of the specification activity is the focus of this section, the adopted control structure will be briefly presented.

Control structure. Specification actions are articulated according to the following iterative sequence:

- (a) *Action proposal*: one or several knowledge sources make action proposals because they are able to contribute to the resolution of the problem as it is defined by the state of the blackboard;
- (b) *Action selection*: one of the proposed actions is selected by the control, depending on
 - the state of the blackboard;
 - the action-proposing knowledge sources, i.e., the global plan and deviation-action proposing processes; and
 - the control knowledge (especially particular selection criteria);
- (c) *Action execution*: the selected action is executed, modifying the state of the blackboard;
- (d) back to (a).

The steps which are relevant from an organizational viewpoint, that is, steps (a) and (b), guide the following presentation of results.

Results 1. Action proposal. Processes leading to deviation-action proposals

Two types of knowledge sources propose specification actions: the global plan and processes proposing deviation actions. As mentioned above, this section focuses on these "deviation processes." Six of them have been identified through an analysis of the specification actions which did not match the engineer's global plan.

Taking advantage of available information

Acting according to a plan is a concept-driven activity. If the engineer follows his global plan, each step of the plan will impose a goal which will make him look for the information needed to achieve this goal.

Deviation, however, is generally due to data-driven processing. The most general form of deviation stems from the engineer trying to achieve the goal which available information allows him to achieve. This goal generally differs from the one imposed by the global plan.

Information may be available because the engineer is "presented" with it, or because it is being used for the current specification action.

Some examples of "information presentation" are the following:

- the client communicates to the engineer new or modified requirements;
- a colleague presents the engineer with information or comments on the solution state;
- a consulted document presents the engineer with "salient" information (information generally is "salient" because it is new or modified).

The next three processes presented below are different forms of the second case, that is, different processes for taking advantage of information used for the current specification action.

Processing information from various viewpoints leading to its use for an action other than the current one

Information used to define a component may be processed from a viewpoint that makes it useful in other specification actions or even a task other than functional specification.

This process was most frequently observed as leading to deviations from one specification action to another.

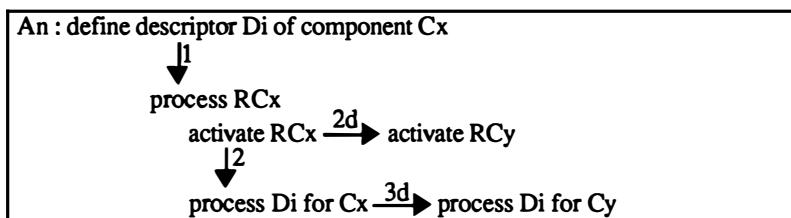
Example. Information used for defining the starting conditions of an operation was often interpreted as also being useful for defining the ending conditions of one or more previous operations.

The process also led the engineer to deviate from his functional specification to another specification task. Information used for functional specification was sometimes considered from its mechanical, i.e., physical viewpoint.

Example. During his definition of a turntable cycle operation, the engineer consults a mechanical specifications document that includes information on the electrical detectors of the turntable. In processing this information, the engineer comes to question the mechanical safety of the turntable: he is not sure that the turntable will not damage other machine-tool stations during its rotational movement. So he interrupts his turntable cycle definition to verify the mechanical specifications. Thus, he interrupts his functional-specification activity to proceed to a different task, that is, mechanical design.

Activation of a component representation other than the current one leading to a local plan for defining another component

In order to define a component Cx, its mental representation RCx has to be processed, leading to its activation in memory. This activation, in turn, may lead to the activation of the representation RCy of another component Cy, by the mechanism of "spreading activation" (see Anderson, 1983) (see Figure 1).



Key: vertical arrows: planned actions
 horizontal arrows: deviation actions

Figure 1. Two processes leading to deviation-action proposals (2d and 3d)

The main cause of RCx activating RCy -rather than, for example, RCz- is a relationship existing between RCx and RCy. Four types of these "activation-guiding" relationships have been identified: analogy, prerequisites, interaction, and opposites.

Analogy. The machine tool has two workstations: shaping and finishing. When the engineer is dealing with a component for one of these stations, he often returns to, or anticipates, the

definition of the corresponding component in the other station, because he considers their functioning to be analogous. This is reflected in the relationship between their corresponding mental representations.

Prerequisites. All work operations (doing the shaping and the finishing on the machine-tool workstations) have prerequisites, that is, operations which satisfy the conditions required for these work operations to take place (e.g., positioning a tool, or setting a stop to establish the position where a work operation must end).

The engineer was observed to "forget" certain operations, that is, he forgot to introduce them during the first definition pass and introduced them only afterwards. With one exception, these "forgotten" operations were prerequisites. The engineer generally discovered their omission when he was specifying the work operation of which it was a prerequisite. So processing a work operation led to the activation of the representation of its prerequisite.

Interaction. This relationship may be based on physical or functional relations between components.

Between machine-tool units, existing relations are generally physical. The turntable, for example, interacts with each of the other units, since these must be retracted before it can turn. The engineer partially describes the turntable cycle before the cycles for the other units. When dealing with certain of them, he goes back to the turntable cycle specification in order to introduce information in it.

Functional relations are generally between operations. For example, the starting conditions for an operation always comprise one or more of the ending conditions of the preceding operation(s). This two-directional relationship leads to two different deviation patterns. Firstly, the processing of the starting conditions for an operation has been observed to lead to the discovery and fixing of omissions or errors in the ending conditions of the preceding operation. Secondly, the processing of the ending conditions for an operation was observed to lead the engineer to jump ahead to the following operation, in order to define its starting conditions in anticipation.

Opposites. When dealing with an operation, the engineer sometimes discovers that he has omitted its opposite operation.

Definition of a component on a descriptor leading to a local plan for defining other components on the descriptor

The deviations presented above were based on exploiting semantic relationships between component representations. Deviation may also be caused by the exploitation of the particular type of definition one is engaged in. In this case the component the deviation leads to has no relationship with the currently defined component other than belonging to the same cycle. Defining a component on one of its descriptors leads the engineer to formulate a local plan for defining one or more other components on the same descriptor (see above, Figure 1).

When this process leads to deviation, it frequently leads to several consecutive deviation

actions. This may occur in several ways:

- having defined a component Cx on a descriptor Di leads the engineer to define another component Cy also on Di; next, the engineer resumes his global plan and defines Cx on Dj, which may lead him to deviate again in order to define Cy also on Dj, etc.;
- having defined a component Cx on several of its descriptors, Di to Dn, leads the engineer to define another component Cy on these same descriptors, Di to Dn;
- rather than deviate to only one other component, several other components may be defined (on one or several descriptors) before the global plan is resumed.

Difficulties with the current specification action leading to "drifting"

"Drifting" refers to involuntary attention switching to a processing other than the current one. It was observed to occur especially when the engineer was involved in a "difficult" action. During such an action, all attention is supposed to be focused on the current problem. Drifting, under these conditions, may be due to the engineer

- "looking in various directions" for possible solution elements, then
- coming upon information which more or less "obviously" applies to another component definition, and
- taking advantage of it.

The engineer may look for possible solution elements at various places, in various information sources. If he "looks" in memory, in other words, if he constructs a representation of the problem which he is trying to solve, this activates the mental representational entities of which the problem representation is built up. This activation may activate other entities via the relationships between them.

A hypothesis inspired by our observations is that drifting-caused deviation occurs especially during information retrieval for problem solving that is not guided by strict information-searching rules.

Comparative evaluation of the costs of an action: if-executed-now and if-executed-later

The cognitive costs of proposed actions are evaluated and compared in order to select the action to be executed. Plan deviation generally occurs when a deviation action is selected because it is "cheaper" than the planned action.

But the cost of a proposed action is also compared to the cost of the action itself if executed later. This may lead to different types of deviation:

- (a) *Anticipation*. A proposed deviation action A is chosen in anticipation because its relative cost is interesting, not compared to its current planned-action alternative, but to A itself when it will be the planned action.
- (b) *Postponement*. The planned action B is postponed because it costs too much, not compared to a currently proposed deviation action, but to B itself if executed later. In this case, a local plan is generally formed for re-proposing the postponed action, either as soon as the conditions leading to its postponement no longer prevail, or at a given moment later in the design process.

A reason for a planned action costing too much if executed now, is the required information being unavailable (now). This may be the case, for example, because

- the client has not yet made a decision that is necessary for the definition of the component in question (and the engineer judges that he cannot make this decision himself);
- the information source with the relevant information (usually a colleague) is absent.

Example. When, according to the global plan, the loading cycle is to be defined, the engineer postpones this definition. He examines the corresponding cycle on the example which he uses and, not understanding the way in which the example cycle has been broken down but thinking he will understand it later, he decides to postpone the loading cycle definition.

Results 2. Action-selection: a combination of control criteria

To decide which action, among the proposals, is going to be selected for execution, the control evaluates each proposed action according to selection criteria.

The global plan always proposes one action. Frequently, there are several competing actions: next to the planned action, one or various deviation proposals have been made. All proposed actions are then compared by the control.

As mentioned, only one action may be proposed, i.e., the planned action, but not be selected because it would cost too much if executed immediately.

Two selection criteria have been identified: cognitive cost and importance of the action.

First action selection criterion: Cognitive cost

For each proposed action, the cognitive cost is determined. For a planned action, this is its cost if-executed-now compared to its cost if-executed-later. For a deviation action, its cost if-executed-now includes, in addition, the cost of plan resumption.

In evaluating the cognitive cost of an action, the control considers several, not necessarily independent, factors, three of which are presented below.

The availability of a schema for executing the action. A schema provides, for each of its variables, a certain number of values, generally including one default value. Executing an action for which such a memory representation is available may cost relatively little if most or all variables relevant for execution have default values.

The engineer's global plan may be considered to be a schema of this nature. This is why its use is so profitable from the cognitive-cost viewpoint. But, as shown above in the presentation of processes leading to deviation actions, deviating from the global plan may sometimes be even more profitable.

The availability of information. On the one hand, an action may be interesting from the cognitive-cost viewpoint if the information required for executing it is available without much effort, the cognitive cost of information access affecting the cognitive cost of the action. On the other hand, an action may become interesting because available information allows it to be executed.

The difficulty of the action. Actions are more or less difficult depending on their component operations. To retrieve a value is easier than to calculate a value, an operation that is itself more or less difficult depending on the availability of the elements used in the calculation.

Example. The duration of an operation can often be found as such in a document. If not, it is to be calculated from its constituents, motor speed and advance distance/rotation. Sometimes, even these are not given in the engineer's current information sources, and still have to be looked up in technical documentation.

Second action selection criterion: Importance

Actions differ with respect to their importance. The two contributing factors which have been identified will be presented through an example.

The importance of the type of action. *Example.* Fixing the omission of an operation which has been forgotten is an important action; verifying an operation identifier is not.

The importance of the object concerned by the action. *Example.* If verifying is not an important action when it concerns identifiers, it is when it concerns durations. The engineer frequently deviates from his global plan in order to verify the duration of an operation, but never to verify its identifier.

Combination of the criteria

Among the two criteria presented above, the first one, "cognitive cost," is considered to be the most important and most general one. Our hypothesis with respect to the way in which the two criteria are combined is the following:

- (a) If the next planned action is the only proposed action and if it does not cost too much, it is selected for execution. If it costs too much, it is postponed and the next proposed actions are taken into consideration.
- (b) If one deviation action is proposed as an alternative to the next planned action, their cognitive costs are compared: the action which is lowest in cost is selected for execution.
- (c) If several deviation actions are proposed as alternatives to the next planned action, their cognitive costs are compared: if there is one action that is lowest in cost, it is selected for execution; if several actions have the same cost, the most important one is selected.

DESIGN STRATEGIES²

This section presents design strategies observed on a professional software engineer, focusing on those strategies which are specific to designing software in a work context.

In the literature on design, "problem decomposition" is a recurrent notion. For one thing, it is advocated by existing design methodologies, which provide different bases for performing it. For another, until recently, most empirical software-design studies presented it as a very important -if not the main- control strategy in expert design activity.

By definition, a design problem is "decomposed," in that it is transformed into other problems to be solved. As used in those early design studies, however, the concept conveys presuppositions such as that decomposition results from "planning" and leads to "balanced"

² This section presents in a completely revised, less detailed way, results presented in Visser (1987). For information about the programmable controller and its programming language, this paper may be consulted.

(top-down breadth-first) solution development (see Visser & Hoc, 1990, for a critical presentation).

Our study was guided by the hypothesis that, from a strategical viewpoint,

- (a) design cannot be characterized appropriately as following a pre-established global strategy (such as a plan to be followed in a top-down, breadth-first order);
- (b) the activity is governed by various locally implemented strategies.

Another hypothesis underlying the study was that these strategies differ, at least partially, from those observed in most other empirical software design studies. Those studies were generally conducted on novice, student programmers working on artificial, limited problems, whereas the designer in our study was an experienced, professional programmer who was observed in his daily work environment.

Method

Task: Software design. A programmer had to design and code the program for the control part of the machine tool described above, using the specifications defined by the mechanical engineer.

Subject: A software engineer. The programmer was a software engineer who had some four years experience in the design of software for automatic machine tools.

Data collection: Observations & Simultaneous verbalization. Over a period of four weeks, full time observations were conducted on the programmer involved in his task.

The procedure was the same as presented above for the mechanical-design engineer.

Results

After a discussion of a possible global control strategy (decomposition), the main strategies which were actually used by the programmer will be presented: re-use, considering users of the system, and simulation. These three strategies, which are at different abstraction levels, were used in combination.

Other design strategies observed on the programmer and on other software-design experts may be found in Visser & Hoc (1990).

A preliminary definition. Software-design components. The programmer's design components may be program modules, sub-modules, instructions, or instruction branches (instructions are made up of several serial and/or parallel branches).

Problem decomposition

The observed programmer used two types of decomposition, one of his task, the other of the program to be written. Both decompositions were made in the second of the three design stages in which we broke down the programmer's activity:

- (a) Studying the specifications (time taken: one day). The programmer skims rather rapidly through the specifications for the machine tool.

- (b) Program-coding planning (time taken: one hour). The programmer plans this coding along two lines, both in the form of a decomposition. The global coding task is broken down into sub-tasks according to the relative urgency with which colleagues need to receive the different parts of the program. Using this criterion, the programmer divides the program into three parts, which he plans to handle consecutively. We observed only the coding of the first part, which is referred to in this text as "the (target) program." The program is broken down into modules corresponding to machine-tool functions. This decomposition follows the order of modules on the listing of an "example" program that the programmer had previously written for a similar machine tool ("example program" will be used as shorthand for "listing of the example program").
- (c) *Coding* (time taken: four weeks). Although this third and last stage took a great deal longer than the others, no qualitatively different sub-stages can be discerned. Coding was interrupted for functional analysis and problem solving, for testing, but also for design decisions. Many of these decisions were only made at this stage; decisions taken in the previous stage were modified. These interruptions were, however, not systematic.

Although the programmer had planned to code his program modules in a certain order, this plan was not in fact used for their actual coding.

Re-use

Re-use was an important strategy in the activity of the observed programmer. Re-used components may be distinguished according to their origin: *programs written in the past*, or the *current program*. Both were among the programmer's main information sources. They were used much more frequently than the functional specifications of the machine tool, that were consulted only during the first design stage.

The programmer used two "example" programs that he had previously written for similar machine tools. One was only used for the construction of one particular target-program module. The other was used very frequently, in the design stages of program-coding planning and of coding (this program is referred to as "the" example program).

Re-used components may also be distinguished according to the level at which they are used: *structuring* the program or *coding* the instructions. In the program-coding planning stage, only functions existing in the example program were planned (others appeared later during coding): the programmer listed them by copying their titles from the example program.

A re-usable component was generally accessed by way of the semantic relationship between the mental representations of the target and source components.

Examples of exploited relationships are:

- analogy between the structures of the example and target programs;
- analogy between functions of parts of the example and target programs;
- analogy between functions of parts of the target machine tool;
- opposition between functions on the target machine tool.

Considering users of the system

This strategy had different functions. It guided solution evaluation as well as solution development.

Solution evaluation. One of the criteria used by the observed programmer in his evaluation of his design was "ease of use for future users" (system operators as well as maintenance personnel).

Solution development. Considering homogeneity an important factor of ease of use, the programmer used it as a design constraint, trying to make the program as homogeneous as possible, both for comprehension (system operators) and for maintenance reasons (maintenance personnel). This search for homogeneity was realized in different ways. Two examples are the following.

Example. Creation of uniform structures at several levels of the program

- Instruction order in the modules. Instructions corresponding to Advance operations were made to precede the related Return-movement instructions.
- Branch order in the instructions. The branch defining the automatic mode of an operation was made to precede its manual mode definition branch.
- Bit order in instruction branches. The "most important" enabling conditions of an operation were put at the beginning of its corresponding instruction branch³.
- Variable numbering. Variables are identified by numbers. Numbering of certain variables with counterparts elsewhere in the program was structured. For example, the variables corresponding to the different "Checks" on the Return movement of each machine station, defined in different program modules, were numbered B601 on one station, B701 on another, and B801 on a third station.

Next to this intra-program, inter-program variable numbering homogenizing was also observed. Certain variables are found in nearly all machine tool programs in the factory where the observed programmer is working, that is, in his own programs, but also in those written by his colleagues. The programmer gave these variables the same numbers as they have in other programs.

Example. Program changes. The search for homogeneity sometimes led the programmer to reconsider previously written instructions. In doing so, he modified either

- the current instruction in making it follow the structure of the previously written ones; or
- the previously written instructions by giving them the structure of the current instruction.

Simulation

The observed simulation was always *mental*, never concrete. Within this mental simulation, different types may be distinguished. First with respect to the problem-solving stage in which simulation took place: simulation was used for *solution development*, when the programmer explored and simulated the problem environment; or for *solution evaluation*, when the programmer ran simulations of proposed solutions.

Second with respect to the object involved in the simulation. This object could be the *machine tool operation*: this simulation was mainly used to understand the functional specifications (solution development stage). Simulation of *program execution*, generally considered to characterize novices, was one of the means used by this experienced

³ Instructions in programmable controller programs are scanned continuously. Inspection of an instruction by the supervisor is stopped as soon as a bit set to 0 is encountered in a serial connection branch.

programmer to check program modules that he had already written (solution evaluation stage).

DESIGN-SOLUTION DEVELOPMENT PROCESSES⁴

At a rather abstract level, solving a problem may be considered to proceed in three steps: construction of a representation of the problem, development of a solution for this problem, and evaluation of this solution. This section is going to present the main types of solution-development processes involved in design-problem solving.

Method

Task: Composite-structure design. The global task was to design a new type of antenna, an "unfurling" antenna.

Subject: A technician specialized in composite-structure design. The observed designer was a technician with some 30 years of professional experience in the Research and Development division of an aerospace factory. For the past four or five years, he had been involved in designing antennas.

Data collection: Observations & Simultaneous verbalization. The designer was observed over nine weeks, at the rate of 3-4 days a week. The analysis on which the presented results are based concerns the data collected during the first five weeks.

The same procedure was used as in the previous studies.

Results. Preliminary remarks

Three aspects of problem-solving which underlie the analysis will be briefly discussed.

Problem or Solution: A double status. Design may be considered as specifying a problem until an implementable solution has been reached. The problem which is the starting point for a designer are the specifications provided by a client, which specify more or less precisely the artefact to be designed. The solution to be attained is another series of specifications, which the designer provides to the workshop and which specify precisely how to manufacture the artefact.

The problem-solving path, consisting of a transition from the initial problem to a final solution, comprises a great number of intermediary states, each with a double status. Until a final implementable solution has been attained, each solution developed for the problem under focus constitutes, on the one hand, a further specification of this problem and, on the other hand, a new problem to be solved. That is why, in this text, the same entity may be

⁴ This section presents, in less detail, results presented in Visser (1990a). Examples on the antenna project would have required a detailed description of the problem of "unfurling," which would have occupied too much space. Only two simple examples can be presented.

referred to as a "solution" or as a "problem," depending on whether it is an output or an input of a problem-solving action.

Problem/Solution knowledge: Structure and access. Experts have in memory "problem/solution schemata" organizing their knowledge about classes of problems and their solutions, with "problem attributes" and "solution attributes."

However, a great part of their knowledge on problems and their solution(s) cannot -or cannot yet- be considered to be schematic: it does not concern classes of problem situations, but particular problems and their solution(s). The corresponding representations are "problem/solution associations."

In this text, except if there is evidence for problem/solution knowledge being either schematic or specific, this knowledge will be referred to as "problem/solution(s)."

The mechanism underlying the access to this knowledge is supposed to be "spreading activation" (see Anderson, 1983).

Solution evocation and solution elaboration. Solution development starts with the construction of a mental representation of the problem to be solved. The components of the resulting representation, that is, certain mnemonic entities, will be the "sources" for activating other mnemonic entities. If a problem/solution is matched, the corresponding solution is said to be "evoked." If this fails, elaboration of a solution is engaged. The next two sub-sections present these two main solution-development processes.

Results 1. Solution evocation

Different types of evocable solutions are distinguished according to

- their "historical link" to the current problem ("pre-existing" solutions vs. solutions developed for the current problem); and
- their likelihood to be activated ("standard" vs. "alternative" solutions).

Evocation of a "pre-existing" solution or of a solution developed for the current problem

Solutions acquired by experience exist in memory before the current global-problem solving (analogous to re-usable programs written in the past). But the current problem solving also leads to solutions being stored in memory (analogous to re-usable components of the current program). Depending on several factors, some are explicitly "shelved," whereas others will enter long-term memory even if the designer consciously did nothing to retain them. A solution is said to be "shelved" if it is developed, not positively evaluated, but put aside "for possible later usage" -rather than rejected.

Evocation of "the" "standard" solution or of an "alternative" solution

Design problems generally have more than one solution. The designer does not necessarily know of several solutions to a problem, but if he does, one of them is activated most strongly on evocation, and so evoked first. This is "the" "standard" solution to this problem; the other(s) is (or are) the "alternative" solution(s).

If the designer knows of several solutions to a problem, the alternative solution(s) may be evoked if the standard solution is rejected after evaluation, in other words, in later solution development cycles.

Results 2. Solution elaboration

Elaborating a solution may take several forms, according to the character of the material constituting the starting point for elaboration.

- (a) *Elaboration of an "alternative" solution.* The starting point for elaboration may be a solution which has been previously developed, but which was rejected after evaluation. This "negative" evaluation introduces supplementary constraints on the alternative solution to be elaborated. These are added to the constraints introduced by the original problem which are supposed to have already been taken into account in the previously developed solution.
- (b) *Elaboration of a "new" solution.* If a problem has not yet received a solution during the current problem-solving, the starting point for solution elaboration is the representation of the problem. Cognitively, this is the most complicated solution-development mode.

Elaboration of an "alternative" solution

For many specific problems, the designer knows of only one solution: he evokes it and evaluates it. If it is rejected and he thus has no other solution left in memory, an "alternative" solution has to be elaborated. The designer has been observed to adopt different approaches. Two general ones, leading to several possible solution-development processes, will be presented: modifying the problem by generalizing it, and modifying the solution by a heuristic.

Problem-modification by generalization. Generalizing a problem amounts to setting aside some of its attributes and focusing on others. The designer constructs this more general representation of the problem in the hope that it will activate an appropriate problem/solution.

Two possible forms of solution elaboration following this general approach will be presented. Both lead to a solution elaboration in two steps, going up and then down again in the problem/solution-abstraction hierarchy.

Generalizing the problem in order to activate a more abstract problem representation. This solution elaboration consists in first elaborating an alternative solution at a higher abstraction level (the new problem representation) and then elaborating a solution to this new problem at a lower abstraction level.

Generalizing the problem in order to activate a problem/solution schema. This second two-step solution-elaboration procedure applies to the elaboration of an alternative solution as well as to the elaboration of a "first," "new" solution to a problem.

If the representation of a specific problem Pb does not evoke an appropriate solution, the designer may see if a more abstract problem-representation activates a problem/solution schema with a default value for the attribute corresponding to Pb. If so, this default value (or a specification of it) may then be proposed as a solution.

Solution-modification heuristics. Faced with a problem for which he knows of only one solution which he has evoked and rejected, a designer is supposed to start alternative-solution elaboration by modifying his problem representation (see above). If neither directly (via a schema), nor indirectly (via a more abstract problem representation) an appropriate solution can be developed, several heuristics are available for modifying the rejected solution.

Example. "Invert the value" of the "critical" problem/solution attribute, that is, of the attribute having led to rejection of the solution.

Elaboration of a "new" solution

Alternative-solution elaboration is only applicable when one has a solution (the evaluation of which has led to rejection). This solution and the result of the evaluation constrain the alternative-solution elaboration. But without such a starting point, and without another solution stored as such in memory, a new solution has to be elaborated "from scratch."

"Design from scratch." The global design process -even of a completely "new" artefact-never proceeds from scratch: a difficult design project is never confided to a complete novice, and as soon as a person has some experience, he cannot proceed from scratch. However, solving sub-problems may proceed from scratch.

"Elaboration from scratch" remains to be defined. In this text it is considered to apply to solution development from material which is only analogous or similar, or even without any clear link to known, past analogous designs. This solution-development mode is probably the most difficult for the designer.

Various processes and strategies may be supposed to be used to this end. Those presented below were observed on the antenna project.

Brainstorming. This term, generally used for a group activity, here refers to an individually used very "weak" solution-"search" mechanism: transitions from one mnestic entity to another are not a function of rules (as in deductive reasoning), nor are they oriented by a search based on similarities (as in analogical reasoning). The activation is not "oriented" in the sense of "consciously" directed. Search is only "guided" by the links that may exist between mnestic entities, and that lead from activated "sources" to other mnestic entities, which may activate, in their turn, still other mnestic entities, the "targets." The "sources" are those entities that have been activated as a result of having been processed by the designer in his construction and further processing of the problem representation. The "targets" are the entities corresponding to the solution the designer comes up with.

The only way in which a person may "guide" brainstorming is in his elaboration of the problem representation. The solution the designer comes up with, or the "idea" he "thinks about," is (or are) the newly activated entity (or entities) which exceed(s) a threshold.

Brainstorming thus is the exploitation of the "natural" memory access mechanism, i.e., spreading activation. The observer suspects brainstorming to take place when the designer hesitates, falls silent, and/or comes up with an "idea" or "solution proposal" after a certain silence. In order to decide that brainstorming has been used -possibly accompanied by other processes- another condition must be satisfied: the observer must have a hypothesis

concerning the transition path from the sources supposed to have been activated by the problem representation to the targets corresponding to the solution proposed by the designer.

Like the other explanations proposed, such hypotheses remain of course to be verified. With these reservations, the problem-solving path for the antenna-design problem showed a frequent use of brainstorming.

Analogy-directed memory access. This process is an "oriented" form of activation. Even if it is not (yet) clear how the "orientation" from source to analogous target mental entities may proceed, the hypothesis is formulated that the targets are "searched for" through links between source and target mental entities.

along the design process. On the antenna project, their use was observed to occur especially during conceptual solution development.

Example. In a discussion with other designers, the observed designer and his colleagues developed "unfurling principles" for future antennas. They proposed conceptual solutions such as "umbrella," and other "folding" objects, like "folding photo screen," "folding butterfly net," and "folding sun hat." The designers thought of taking these objects into pieces in order to find possible ideas for the corresponding material solutions.

All proposed objects satisfied an important constraint that existed on the unfurling antenna, that is, they all had a trigger mechanism leading to such an unfurling that the resulting object constituted a rigid surface.

DISCUSSION

After a short recall of the main results of the three studies presented in this paper, their possible consequences for "A.I. and design" will be discussed.

Main results of the presented studies

The main contribution of the study on the global organization of the specification activity was the detailed analysis of how its opportunistic character was realized. Other studies have characterized design activity as opportunistic. The new idea developed in the present study is that a pre-existing plan may be considered as only one of the knowledge structures used by the control to guide the activity. That is, at the control level of his activity, an expert designer has, next to a global plan making action proposals at each step in the design process, other knowledge structures which propose alternatives to the plan-proposed actions. Six processes leading to such deviation-action proposals were presented.

The analysis of design strategies focused on the identification of local strategies specific for working on real, complex design projects. Trying to characterize an activity in terms of top-down or bottom-up and depth-first or breadth-first strategies leads to different results according to the level at which analysis takes place. At a high, abstract level, the observed programmer's activity could be described as following a top-down strategy: the program was decomposed into modules and coding was planned to follow this decomposition. As soon as the activity is analyzed at the local level, that is, at the level of the actual design actions, deviations from this plan are observed and the activity is no longer top-down: for

example, semantic relationships between the mental representations of different design components are exploited, leading to deviations similar to those described in the section on the organization of the design activity.

With respect to the local strategies, some strategies already known to be at work in "designing-in-the small" were observed, for example, simulation. Other strategies seem indeed to be characteristic of designing in a work context, especially the re-use of program components and the consideration of users of the system. Neither has generally been noticed in previous design studies. An explanation of this absence might be that their implementation depends on conditions only realized in real work situations.

Actual re-use of existing components was observed, but the programmer also created facilities for possible re-use. The observed homogenizing of inter-program variable numbering is an example of this anticipation oriented towards future re-use. Consequences of this result will be developed and discussed below.

The local strategies were implemented in combination, mostly of two. So, simulating the operation of the machine tool while considering (a particular class of) users of the system might make the programmer think of elements to be included in the design.

The analysis made on the third study was more descriptive. There were no underlying hypotheses to examine or claims to refute. Several results were especially interesting. So were the frequent use of "brainstorming" in all solution development stages, and the importance of analogy-directed memory access during conceptual solution development. The study has inspired various questions and hypotheses:

- When and how do problem/solution schemata develop? How does their appearance modify the associations between particular problems and solutions?
- The problem of the "scope" of a solution's activation. When a problem/solution is activated, what is it that is activated? Its "label" is activated, but not only that. Attributes and values are activated, but not all of them. It surely depends on the problem representation the designer constructs, but the problem remains open to question.
- What about the proposed "brainstorming"? Is there a process that may be identified as such? Is it rather a series of activations: in that case, what is their unity?
- What about "classical" problem-solving processes like deductive reasoning? Why were they not observed? Do they not play a role in other than routine-design activities?

Possible consequences for "AI and design"

In the introduction, we claimed that the results of studies like those presented above are relevant for the design of knowledge-based systems. This claim will now be defended through the examples of results on decomposition and re-use, but the other results could have been used as well. The design of support tools for strategies such as simulation is a complex task and could take advantage of empirical data on the representations constructed by designers and the way in which they are used. It would be useful to assist the switching between the various viewpoints which designers take on the design problem/solution state. Given the problems met by designers due to memory limitations, displays for helping the management of working memory could be helpful, such as facilities for:

- parallel presentation of intra- or inter-level information;
- presentation of the constraints on the problem-solving order;

- maintaining a trace of postponed problems needing backtracking.

Assistance tools for the management of memory load could also support simulation, given the frequent need for a designer involved in simulation to hold simultaneously several variables in mind. More ideas on assistance and references to other studies may be found in Ullman, Staufer & Dietterich (1987), Visser & Hoc (1990) and Whitefield (1986).

Decomposition. If the design activity is opportunistically organized, an assistance system which supposes -and therefore imposes- a hierarchically structured design process will at the very least constrain the designer and will probably even handicap him (see Visser & Hoc, 1990). Assistance tools should be compatible with the actual activity. In the present case, such tools should allow the design -that is, the solution under development- to be abandoned at a certain level when the designer prefers to process problem/solution elements at another level. The system should not only allow the design to be resumed later at the abandoned level, but it could assist this resumption by keeping a trace of such abandonments. This implies that the system knows the structure of the final design. The system should not, however, impose such a resumption. The engineer we observed used a hierarchically structured plan, to which he returned after deviation actions. Not all designers necessarily refer to such a plan, especially if the (final) design has a less constrained structure than the specifications constructed by the observed engineer.

Re-use. Re-use is an approach advocated by many computer scientists working on design environments in software and other design domains (see *IEEE Software*, Special issue on reusability, 1987; *Artificial Intelligence*, special issue on Machine Learning, 1989). Arguments in support of the approach are based on "introspection" and "common sense" considerations (economy etc.), not on results from cognitive studies on designers. Besides, if psychological studies exist on the design activity in general, empirical research on how people actually re-use design components is almost non-existent.

Systems allowing for re-use or based on re-use are being developed. However, as formulated by a computer scientist building systems supporting redesign, "The disadvantage [of this type of complex systems] is that [reusable] building blocks are useless unless the designer knows that they are available and how the right one can be found. ... Several cognitive problems prevent users from successfully exploiting their function-rich systems." (Fischer, 1987, p. 61).

In order to offer users really helpful systems, these "cognitive problems" must be identified together with the strategies actually used by designers to solve them.

REFERENCES

- Anderson, J. R. (1983). *The architecture of cognition*. Cambridge, Harvard University Press, MA.
- Artificial Intelligence*, special issue on Machine Learning (1989) **40** (2).
- Bisseret, A., Figeac-Létang, C. and Falzon, P. (1988). Modeling opportunistic reasonings: the cognitive activity of traffic signal setting technicians, *Research Report N° 898*, Rocquencourt: INRIA.

- Darses, F. (1990). Constraints in design: Towards a methodology of psychological analysis based on AI formalisms, in D. Diaper, G. Cockton, D. Gilmore & B. Shackel (eds), *Human-computer interaction - INTERACT '90*, North-Holland, Amsterdam.
- Eastman, C. M. (1970). On the analysis of intuitive design processes, in G. Moore (ed.), *Emerging methods in environmental design and planning*. MIT Press, Cambridge, MA.
- Ericsson, K. A., and Simon, H. A. (1984). *Protocol analysis. Verbal reports as data*. MIT Press, Cambridge, MA.
- Fischer, G. (1987). Cognitive view of reuse and redesign, *IEEE Software* 4: 60-72.
- Guindon, R., Krasner, H., and Curtis, B. (1987). Breakdowns and processes during the early activities of software design by professionals, in G. Olson, S. Sheppard and E. Soloway (eds), *Empirical Studies of Programmers: Second Workshop*, Ablex, Norwood, N.J.
- Hayes-Roth, B. and Hayes-Roth, F. (1979). A cognitive model of planning, *Cognitive Science* 3: 275-310.
- Hayes, J. R., and Flower, L. S. (1980). Identifying the organization of writing processes, in L. W. Gregg and E. R. Steinberg (eds), *Cognitive processes in writing*. Erlbaum, Hillsdale, N.J.
- Hoc, J. M., Green, T., Samurçay, R. and Gilmore, D. (eds) (1990). *Psychology of programming*, Academic Press, London.
- IEEE Software*, Special issue on reusability (1987) 4 (4).
- Kant, E. (1985). Understanding and automating algorithm design, *IEEE Transactions on Software Engineering* SE-11: 1361-1374.
- Ullman, D., Staufer, L. A. and Dietterich, T. G. (1987). Toward expert CAD, *Computers in Mechanical Engineering* 6: 56-70.
- Visser, W. (1987). Strategies in programming programmable controllers: a field study on a professional programmer, in G. Olson, S. Sheppard and E. Soloway (eds), *Empirical Studies of Programmers: Second Workshop*, Ablex, Norwood, N.J.
- Visser, W. (1988). Giving up a hierarchical plan in a design activity, *Research Report N° 814*, Rocquencourt: INRIA.
- Visser, W. (1990a). Evocation and elaboration of solutions: Different types of problem-solving actions, *Actes du colloque scientifique COGNITIVA 90*, AFCET, Paris.
- Visser, W. (1990b). More or less following a plan during design: opportunistic deviations in specification, *International Journal of Man-Machine Studies. Special issue: What programmers know* 33: 247-278.
- Visser, W. and Hoc, J.M. (1990). Expert software design strategies, in J. M. Hoc, T. Green, R. Samurçay and D. Gilmore (eds), *Psychology of programming*, Academic Press, London.
- Whitefield, A. (1986). An analysis and comparison of knowledge use in designing with and without CAD, in A. Smith (ed.), *Knowledge engineering and computer modelling in CAD. Proceedings of CAD86. Seventh International Conference on the Computer as a Design Tool*, Butterworths, London.

The effects of examples on the results of a design activity

A. T. Purcell and J. S. Gero

Key Centre of Design Quality

University of Sydney

NSW 2006 Australia

Abstract. The role of search processes and the knowledge used in the activity of designing has been receiving increasing attention in the areas of artificial intelligence, the study of design and cognitive psychology. More recently with the development of expert systems, considerable attention has been focussed on the representation and retrieval of expert knowledge. What has not been addressed however is whether or not an expert represents, accesses and utilises all knowledge equivalently. The design fixation effect, where pictures of design instances presented as part of a design problem result in the reproduction, by student and practicing designers, of aspects of the presented instance, would appear to indicate that certain types of knowledge are more privileged in the design process than others. The results of a preliminary experiment examining the role of pictures of different types of instances, verbal descriptions of these instances and variations in the familiarity of instances in the fixation effect are reported. Pictorial information was shown to have no effect if the instance was unfamiliar and equally familiar pictures were found to produce both design fixation and increased variety in design. Verbal descriptions of two of the pictured designs also produced effects although they were much reduced in comparison to the pictorial material that produced fixation effects. While preliminary, these results would appear to indicate a particularly important direction for research the results of which could be important for artificial intelligence, the study of design and the psychology of design processes.

INTRODUCTION

The focus of the research to be reported in this paper is on the role of knowledge in the design of three dimensional artefacts. In common with what has been referred to as the second phase of work on problem solving in an artificial intelligence context (see, for example, Feigenbaum, 1989), designing can be viewed as an activity where expert knowledge is used to fill out an initial statement of a problem in order that a solution can be developed. The initial statement of a problem that a designer works with contains two broad types of information - a statement of what is to be designed and constraints or requirements that the design has to meet. Using the general conception of expertise developed in a number of areas (Chi, Glaser and Farr, 1989; Reiman and Chi, 1989), expertise in design can be regarded as the use of knowledge relevant to the particular area to:

- (a) identify the implications of the stated constraints for the design;
- (b) access a repertoire of potential part or whole physical solutions; and
- (c) identify, possibly through the process of designing, other constraints which are relevant to the design but which were unidentified in the initial problem statement.

Given this general model of the design process, the question of knowledge in design involves issues concerning the representation of knowledge (knowledge about constraints and potential physical solutions or part solutions) and the processes involved in using that knowledge (the identification of implied constraints and the development of design solutions).

Knowledge structures

Knowledge relevant to design would appear to come from two sources. Because design is concerned with objects that do or may exist in the world, a designer would generally have knowledge based on exposure to instances. This knowledge depends on the regularities in the world and how frequently instances are encountered. In addition, the type and extent of the knowledge obtained in this way can depend on whether the knowledge accrues as a part of everyday, incidental experience or as a result of intentional learning. In the latter case knowledge can be the result of deliberately structured experiences relevant to the characteristics of particular domains and the operators that may be used to produce changes in a domain. An important issue therefore concerns the form these different types of knowledge take and the way they enter into the design process.

Everyday, incidental experience appears to be represented in knowledge structures such as schemas (Mandler, 1984;) scripts (Schank and Abelson, 1977) and E-MOPS (Kolodner, 1985). These types of knowledge structures have prototypical default values at all levels of abstraction in the knowledge structure (see, for example, Mandler, 1984). Prototypicality in this context refers to a representation of the most frequently occurring attributes and relationships and the ranges of values these attributes and relationships generally take in a set of instances from a particular domain. These knowledge structures represent generic knowledge about the world. Importantly there is considerable evidence that a particular level of abstraction, the basic, is most frequently used in information processing and easiest to access. Further, at this level, representation focuses on parts and the relationships between parts rather than more abstract attributes such as functions (Rosch and Mervis, 1975; Rosch et al., 1976; Mervis and Rosch, 1981; Tversky and Hemenway, 1984) and that this is the highest level at which an image of the objects in the category can be formed.

This type of knowledge about the world of designed artefacts would be expected to be both similar and different for designers and non-designers. The intentional learning associated with design involves both exposure to a larger number and a greater diversity of instances within a domain. For this reason expert designers' knowledge structures would be expected to be more elaborated than non-experts at the more detailed levels within the structure. However expert knowledge in general (see Reiman and Chi, 1989) and design expertise involves the development of more abstract knowledge about principles. Further expert knowledge in design appears to involve knowledge about materials and parts of objects and the relationships between these elements that can be used to produce artefacts. Expert designers' knowledge structures at the detail level could, as a result, be particularly

rich in terms of the representation of visual material resulting from exposure to larger, more diverse sets and because of the representation of materials, parts and relationships. It is these latter aspects of experts' knowledge structures that would be different to the knowledge structures of non-experts.

Design fixation

In terms of their processing, these various types of knowledge could be used in the ways described in the Simon (1981) model of search processes and subsequent developments in the area in relation to expert systems (Coyne et al., 1990; Klahr and Kotovsky, 1989). However, less attention has been paid to whether or not or under what circumstances the various types of knowledge will be accessed; an issue which is of some importance in relation to expert systems in design and in the education of designers. A possible indication that there is differential access to and use of the knowledge base in design comes from the work of Jansson and Smith (1989). In a series of experiments, groups of mechanical engineering student or professional designers were either presented with a statement of a design problem or the statement of the problem together with a picture of a possible design. The design problems used were the design of a bicycle rack for a car, a drinking cup for the blind, a spill proof coffee cup and an apparatus for taking samples and measuring speed and pressure at different points in the intestinal tract. In each case, the designers presented with a picture reproduced a number of aspects of the design including aspects that were inappropriate or incorrect. This effect is referred to as 'design fixation' and these authors relate it to the functional fixedness exhibited in a number of problem solving situations (Weisberg and Alba, 1981; Weisberg, 1988)

Jansson and Smith's results indicate that pictorial information can have a powerful effect on design and imply that access into relevant knowledge structures and processing of information may be at a corresponding level of specificity. This result is also consistent with the previously demonstrated importance of the basic level in categorisation discussed above. The impact of pictorial information may reflect the hypothesised elaboration of this material present in designers' knowledge structures. The effects of pictorial material on design may therefore reflect this preferred level of processing of information for designers. Given the role that pictorial material plays in much design education and the implications of the fixation effect for the practice of design, particularly in terms of the replication of inappropriate aspects of the presented designs, the design fixation effect and the conditions that produce it warrant careful investigation.

The Jansson and Smith (1989) experiment does not, however, unequivocally demonstrate that it was the pictorial material which influenced the designs produced. Their control group, against which the group receiving the pictorial material was compared, only received a statement of the problem. The group receiving the picture therefore received more information than the control group as well as pictorial information. In order to establish whether it is the pictorial form of the information which is important, it is necessary to equate the information presented with the form of the information differing between the design groups. The first aim of the research reported here was to attempt to replicate the design fixation effect and to examine this issue by repeating the Jansson and Smith experiment using one of their design problems but including a group that received a verbal description of

the material presented pictorially together with a control group that received only a statement of the problem. The use of a verbal description group would also address the issue of whether higher level, semantic information can produce the same effects as pictorial material.

Familiarity and design

The use of verbal or pictorial information in this way can be thought of as being similar to the second source of design information discussed above; that is, information derived from the presentation of specific design related material as opposed to information derived from everyday experience with instances of different design types. In the context of the type of experiment being proposed, the everyday knowledge could play a role in the designs that are produced, either independently of or through an interaction with the material presented in the context of the design problem.

On the basis of the work in category and concept formation, it was argued above that everyday knowledge is based on environmental regularities and repeated exposure to instances. Particular instances that are similar to the prototypical default values should be experienced as familiar while instances that depart from these default values should be experienced as unfamiliar depending on the extent of the difference to the existing knowledge structure. For design problems that involve the design of an everyday object such as the bicycle rack design of Jansson and Smith (1989), it would be expected that the participating designers would have reasonably well developed knowledge structures relating to what was to be designed. The effects of the material presented as a part of the design problem could as a result depend on the relationship between this material and the default values in the existing knowledge structure. In Jansson and Smith's work a roof mounted bicycle rack was used to produce the fixation effect. While no evidence is presented in relation to the familiarity of this type of design, an examination of available bicycle rack designs indicates that the type of design they used appears to be reasonably common. An interesting and relevant question therefore concerns whether or not the same effects will be observed if pictorial and verbal description material is used that varies in terms of its familiarity. The second aim of the experiment reported here was to investigate the role of variations in the familiarity of the instances used as fixating material with familiarity being assessed by frequency of purchase and use of various types of designs and by judgments of the familiarity of these instances obtained from the participating designers at the completion of the design session. In relation to both aims of the experiment, it is stressed that the results obtained can in no way be considered as definitive but as indicative of whether or not and in what directions empirical work might proceed in this potentially important area for design and artificial intelligence in design.

METHOD

Subjects

A total of 207 first year design students participated in the experiment in the latter half of their first year. Participants were from the Architecture departments of the University of Sydney and the University of New South Wales and from the Industrial Design departments of the University of Technology, Sydney and the University of New South Wales.

Experimental material

Jansson and Smith's (1989) bicycle rack design problem was used. Discussion with the local bicycle riders association and retailers of bicycles and accessories indicated that there were five main types of bicycle racks available. Two types were mounted on a tow ball at the rear of the car, one consisting of a single vertical tubular steel post to which the bicycles were attached and the other, two similarly dimensioned steel posts arranged to form an A-frame. Drawings of all rack designs are shown in Figure 1. There were two roof mounted designs with the bicycle attached by the seat and frame in one design and by the wheels and frame in the other design. Finally there was a car boot (trunk) mounted design with the bicycles being supported by their frames. These discussions also indicated that the single post design was the most frequently purchased design with the A-frame and the frame and wheel roof design being reasonably frequently purchased with the boot and seat support roof design being purchased much less frequently.

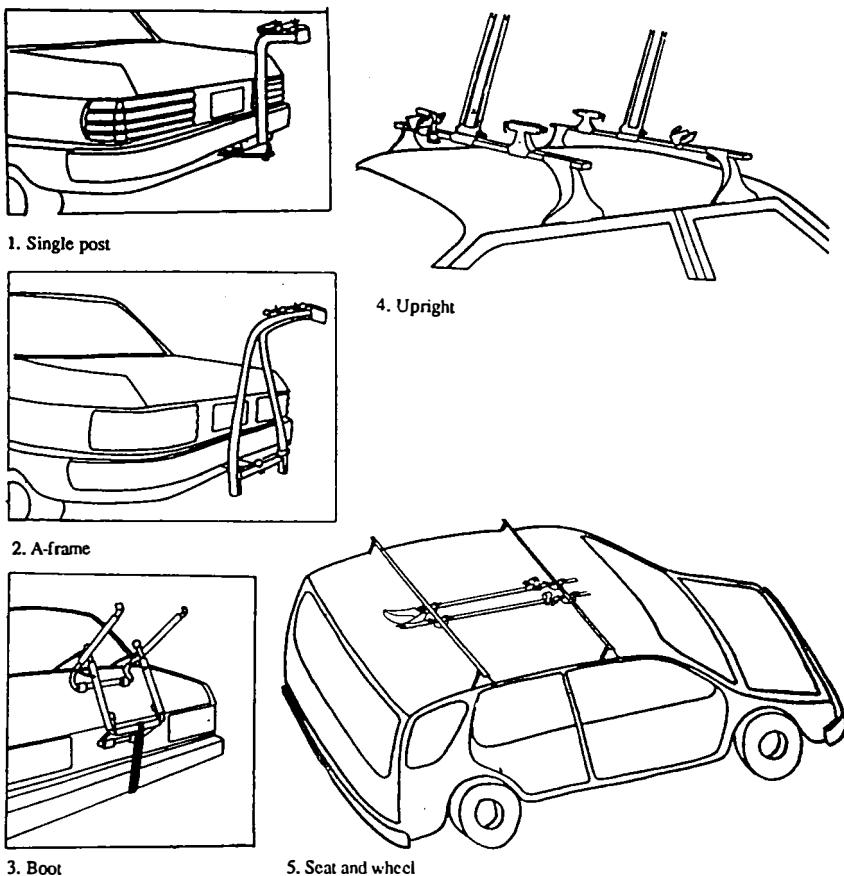


Figure 1. Five types of bicycle rack design

Three of these designs were used to create the required pictorial material: the single post, A-frame, and boot mounted designs. These were chosen to give a range of frequency of purchase and hence frequency of occurrence in the environment. In order to investigate the effect of semantic information, verbal descriptions of the single post and the A-frame designs were developed. Two rather than three of the pictorial designs were represented in verbal form in part because of limitations on the availability of participants in the experiment and in part because of the complexity of the instructions required to describe the boot design. Further, it was considered that, as the experiment was a pilot experiment designed primarily to replicate the design fixation effect and explore some factors which might be related to it, two semantic conditions would indicate whether this issue was worth pursuing. The A-frame description and the simple description of the problem for the control group are both given below. The groups that received the pictorial and verbal descriptions also received the basic statement of the problem. The pictorial material was presented as an illustration of the way the designers were to present their designs.

Verbal instructions for the A-frame design

The aim of the design exercise is to come up with a sketch design(s) for a bicycle rack for three bicycles for a car. The bicycles have to be held securely and without damage to either the bicycle or the car. The bicycles must not extend beyond the overall width dimensions of the car to avoid potential damage in passing to people or vehicles.

There are a number of key issues to be considered in designing a bicycle rack. The first is the way the rack is attached to the car. Then there has to be a structural system that will support the bicycles. Third, there has to be a way of attaching the bicycles to the support system. Fourth, both the structural system and the way of attaching the bicycles have to have a correct relationship to the car to fulfil the other more general conditions of safety, etc.

To illustrate, the bicycle rack can be attached to the car at a number of locations; for example, at the rear of the car, to a tow bar fitting, or directly to the chassis of the car. The bicycles can be supported structurally in a number of ways; for example, two steel posts in the shape of an A joined across the bottom of the A could attach to the fitting fixed to the car. This would have to be of a sufficient height to provide clearance above the ground.

Similarly the bicycles can be attached to the rack in a number of ways; for example, they can be held by the top horizontal section of the bicycle frame in short half sections of pipe of a diameter that would allow the bicycle frame section to fit into the pipe with the other half section closing over the bike frame.

The bicycles then have to have an appropriate relationship to the car; for example, to the top of the A-frame, a short steel section could be attached at right angles in line with the length of the car. The fitting with the brackets attaching the bicycles could be placed at right angles to the length of the car on this section of the post placing the bicycles parallel to and clear of the boot of the car.

Detailed and accurate drawings are not required simple, rough, outline sketches are all that is needed. As well as the sketches, you can write comments on the drawings to illustrate what you mean. You will be allowed forty-five minutes to complete the sketch design. If you wish, you may complete more than one design.

Control group instructions

The aim of the design exercise is to come up with a sketch design(s) for a bicycle rack for three bicycles for a car. The bicycles have to be held securely and without damage to either the bicycle or the car. The bicycles must not extend beyond the overall width dimensions of the car to avoid potential damage in passing to people or vehicles.

There are a number of key issues to be considered in designing a bicycle rack. The first is the way the rack is attached to the car. Then there has to be a structural system that will support the bicycles. Third, there has to be a way of attaching the bicycles to the support system. Fourth, both the structural system and the way of attaching the bicycles have to have a correct relationship to the car to fulfil the other more general conditions of safety, etc.

Detailed and accurate drawings are not required simple, rough, outline sketches are all that is needed. As well as the sketches, you can make written comments on the drawings to illustrate what you mean. You will be allowed forty-five minutes to complete the sketch design. If you wish you may complete more than one design.

Measurement variables

On the basis of existing designs and an examination of the first set of 50 designs, four characteristics of each design were identified. At the most general level, designs can be assessed in relation to the location of the bicycle rack on the car.

- (1) rear,
- (2) roof,
- (3) boot, or
- (4) other locations.

The second characteristic related to how the rack was attached to the car:

- (1) to the tow ball,
- (2) to the bumper bar,
- (3) to the roof, or
- (4) to the boot.

Of particular importance is the third characteristic of the designs which is the way in which the bicycles were supported. Seven supporting structures were identified:

- (1) a single vertical post,
- (2) an A-frame post,
- (3) a single horizontal post,
- (4) roof racks,
- (5) multiple vertical posts,
- (6) multiple horizontal posts, or
- (7) an angled post.

The fourth characteristic of the designs was the method of attaching the bicycle to the rack:

- (1) wheels,
- (2) frame, or
- (3) bicycle enclosed.

Jansson and Smith (1989) assessed the location of the bicycle rack, the method of attachment of the rack to the car and of the bicycle to the rack; they did not assess the method of supporting the bicycle by the rack. Each design was assessed in terms of these four aspects and the analyses consist of chi-square tests of significance. In addition a count was made of how many designs each participant produced. Because of the classroom setting in which the experiment was conducted and the use of design students from a number of sources, it was not possible to equate the numbers of participants in each of the experimental conditions. The numbers in each group were - Control 30, Single Post Picture 34, Single Post Description 32, A-frame Picture 42, A-frame Description 38 and Boot Picture 31. Because of these unequal numbers, the results are presented as percentages. Familiarity with each of the designs in Figure 1 was assessed using a three point scale of: not at all familiar, quite familiar, or very familiar. Tables showing the details of the relationship between each aspect of the design and each group are contained in the Appendix.

Procedure

The experiment was conducted as a class exercise. Each class was divided into groups and each group received one of the six sets of instructions. Prior to commencing designing, general instructions were given to the whole group. These introduced the general topic of the exercise to design a bicycle rack for a car. In common with Jansson and Smith (1989) participants were told that the experimenters were interested in the ideas that were generated and they were encouraged to produce as many designs as they liked. It was stressed that the designs were to be sketch and not finished designs. Both drawings and verbal annotations could be used with the aim being to provide simply enough information for the experimenters to understand what was being proposed.

Each participant worked at a separate work area with supervisors present to answer questions and to ensure independent work. The instructions were provided when the participants were seated. Forty-five minutes was allowed for the design activity.

At the completion of the design period participants were issued with a questionnaire. This collected information related to demographic characteristics, association with various aspects of bicycle riding and asked for a judgment of familiarity with the five types of bicycle designs presented as outline drawings. Finally participants were asked to write a short description of how they went about designing the bicycle racks.

Analyses

The complete data set consists of an assessment of each of the variables outlined above for each of the designs produced by each of the participants. However, there are particular combinations of the experimental treatments which are of interest rather than overall analyses of each aspect of the bicycle design. Each pictorial or verbal description treatment can be compared to the control for each aspect of the design to determine whether the different types of information produce changes in the design. From Jansson and Smith's (1989) work design fixation would be expected with each of the three types of pictorial representations. The comparison of the two sets of semantic descriptions to the control indicates whether it is the pictorial form of the information which is important in affecting the designs produced.

In total five analyses were performed involving pairs of experimental treatments. These analyses were carried out for all of the designs produced and separately for only the first design produced by each participant. Where all of the designs were included in the analysis, the number of designs produced as well as the location, method of attachment to the car, method of bicycle support and method of bicycle attachment were analysed for each of the combinations of conditions discussed above. The analysis of the number of designs produced was necessary for two reasons. First Jansson and Smith (1989) had found no difference in the number of designs using their single type of pictorial material relative to their control and the analysis was necessary for comparative purposes. Second, because there are both differing types of pictorial material and equivalent semantic material, it was of interest to determine if these variations, which were not present in the Jansson and Smith experiment, also resulted in no differences in the number of designs produced. Further, it is implicit in the Jansson and Smith result that the pictorial material simply affects the type of design produced. It is rather surprising that there were similar numbers of designs produced by their experimental and control groups. If the presentation of pictorial material has the type of effect described by those authors, it could be argued that this would act to limit the number of designs produced as it would prevent changing to a different type of design; given that both groups were requested to construct as many designs as possible.

A different aspect of the same argument leads to an analysis of the participants first designs. In order to produce more than one design, two paths are open to the designer. Variations on the first design can be produced by, for example, changing one of the four aspects of the design assessed. Alternatively the designer can switch to a different type of design, for example from a rear mounted design to a roof rack design. If the second path is common, analysing all the designs would tend to mask any fixation effect because the number of other types of designs would be inflated. This possible effect can be overcome by analysing only the first designs.

RESULTS

Differences in the number of designs

Table 1 presents the frequencies with which one, two and three or more designs were produced by each of the groups together with the probability associated with testing the total number of designs for each experimental group against the control group. Significantly more designs were produced by the group receiving a picture of an A-frame design and a description of a single post design. The last column in the table shows the average number of designs for each group. It is apparent from the data that the A-frame picture group produced the most designs with the single post description group producing fewer designs than the A-frame picture group. There is also an indication of a trend in the data with the control and the boot picture group being similar and with the lowest number of designs, with the single post picture and A-frame description groups producing an increased number of designs followed by the single post description group and then the A-frame group with the most designs.

Table 1. Frequencies with which one, two and three designs were produced by each experimental group

Group	One	Two	Three or more	p	Average no. of designs	Total
Control	28	5	2	—	1.3	35
Single post picture	31	11	9	.13	1.7	51
Single post description	26	13	19	.00	2.2	58
A-frame picture	32	21	34	.00	2.6	87
A-frame description	31	14	9	.08	1.7	54
Boot picture	27	7	3	.78	1.4	37

This result contrasts with the absence of a difference in the number of designs in Jansson and Smith's (1989) work where the average number of designs were 4.5 and 4.3 in the control and fixation groups respectively. It is also apparent that Jansson and Smith's designers produced many more designs. The most likely explanation of this difference is the difference in amount of design experience between the two experiments—Jansson and Smith's participants were senior design students while participants in this experiment were in the second part of their first year at university. This difference would appear to be the most likely cause of the effect rather than the difference in design disciplines in the two experiments—Mechanical Engineering in Jansson and Smith and Industrial Design and Architecture in the present experiment although the question of differences as a function of design background is an issue that should be investigated in future research.

The effects of pictorial information

Table 2 presents the probabilities associated with the chi-square values obtained for each of the five comparisons between the groups in the experiment for all of the designs and Table 3 the probabilities associated with the same comparisons for the first designs. From Jansson and Smith's (1989) experiment it would be predicted that each of the pictorial information conditions should result in, for each aspect of the bicycle rack design, significantly more features associated with each particular picture relative to the control group. When all the designs are considered there are no differences between the control and the three picture groups for location of the bicycle rack, the method of attachment to the car, the method of supporting the bicycle and the method of attaching the bicycle to the rack. The only exception to this result is a significant effect associated with the A-frame picture group and the method of attaching the bicycle to the rack where fewer wheel attachments and more frame and enclosure of the whole bicycle were used in the A-frame picture group.

For the first designs there was a significant difference for the method of attaching the rack to the car for the single post picture group where 49% of designs in the picture group used a tow ball attachment in contrast to 13% in the control group (Appendix Table A3). Significant differences were also found for the A-frame picture group for the method of attaching the bicycle to the rack reflecting the same effect found with all the designs and the probability associated with the bicycle support comparison approaches significance. In this

case there are fewer single post designs in the A-frame group but more multiple vertical and horizontal post and angle post designs than in the control group (Appendix Table A6).

Table 2. Probabilities associated with the chi-square obtained when all designs are considered for each of the eight comparisons for the number of designs and each aspect of the design

<i>Group</i>	<i>Location of rack</i>	<i>Car attachment</i>	<i>Bicycle support</i>	<i>Bicycle attachment</i>
Control vs single post picture	.36	.38	.53	.17
Control vs single post description	.83	.88	.72	.06
Control vs A-frame picture	.79	.64	.21	.02
Control vs A-frame description	.30	.13	.61	.12
Control vs boot picture	.89	.99	.86	.26

Table 3. Probabilities associated with the chi-square obtained for the first designs for the eight comparisons for each aspect of the design

<i>Group</i>	<i>Location of rack</i>	<i>Car attachment</i>	<i>Bicycle support</i>	<i>Bicycle attachment</i>
Control vs single post picture	.60	.02	.18	.14
Control vs single post description	.63	.29	.67	.04
Control vs A-frame picture	.16	.13	.08	.05
Control vs A-frame description	.86	.08	.25	.36
Control vs boot picture	.28	.69	.59	.48

The effects of verbal descriptions

For all designs neither of the verbal description conditions is significantly different to the control group for any aspects of the bicycle design. The one exception is associated with the method of attaching the bicycle to the rack for the single post description group where the effect approaches significance and this effect is significant when only the first designs are considered. In both cases the difference is similar to that found with A-frame picture group with the single post description group having fewer wheel attachments and more frame attachments or with the whole bicycle enclosed (Appendix Tables A7 and A8).

Familiarity of the designs

Table 4 presents the levels of familiarity with each of the bicycle rack designs presented in Figure 1. Level of familiarity is presented as a percentage of the respondents across all six experimental groups. This table illustrates clearly that there are differences in familiarity

between existing bicycle rack designs. It is apparent that a particularly high percentage of the participants were very or quite familiar with the single post design (91%). A majority were very or quite familiar with the A-frame design (66%). In contrast to this 7% of participants were very or quite familiar with the boot rack design. The roof rack design, where the bicycle is carried on the roof in a upright position, was very or quite familiar to 35% of participants. The other roof rack design, where the bicycle is carried in an upside down position attached by the seat, was very or quite familiar to 23%. The judged familiarity of the various bicycle rack designs therefore corresponds quite closely to the data regarding frequency of use.

Table 4. Levels of familiarity (%) with each bicycle rack design (Figure 1, No. 1-5) (n=207)

Rack Design	<i>Levels of familiarity</i>		
	<i>Not at all familiar</i>	<i>Quite familiar</i>	<i>Very familiar</i>
Single post	8	52	39
A-frame	34	55	11
Roof No.1	65	31	4
Roof No.2	77	16	7
Boot	93	5	2

DISCUSSION

In their experiment using a bicycle rack design problem Jansson and Smith (1989) report the following percentage differences between their control group and the single experimental group given pictorial information. For top mounted designs (our location variable) the figures were 59% for the control and 71% for the experimental group; for designs with suction cups (our car attachment variable) the figures were 6% for the control group and 54% for the experimental group and for designs with tire railings (our bicycle attachment variable) the figures were 48% and 15%. Jansson and Smith did not score their designs for the method of supporting the bicycle on the rack. These figures show a consistent fixation effect across each aspect of the design.

By contrast our results are quite different. For two of the pictured designs, the boot and the A-frame racks, aspects of these designs are found very infrequently in the designs produced. For example, a specific A-frame used as the method of supporting the bicycle occurred in only 5% of the designs of the group shown a picture of this design. Boot rack designs occurred infrequently. For example, 20% of participants shown a picture of a boot rack design located their design on the boot. In the specific comparisons between the experimental groups and the control group, statistically significant differences were only found for a limited number of the aspects of the design rather than the differences occurring for all aspects of the design as occurred in Jansson's and Smith's work. While some of these differences are consistent with a design fixation effect, for example the significantly

higher percentage of tow ball attachments in the single post picture group, other differences appear to be related to design differences that did not occur either in the pictures presented or in the control group description. For example, while the higher percentage of frame attachments of the bicycle to the rack for the A-frame picture group is consistent with a fixation effect, the higher percentage of designs attaching the bicycle by enclosing the whole bicycle represents a new aspect of the design. A similar situation exists with the verbal description of the designs with very few significant differences between these groups and the control groups.

It would appear that our attempt to replicate the design fixation effect was largely unsuccessful and as a result the issue of differences in the effects of pictorial and semantic information could not be effectively addressed. Further the largest effect in the data, the increase in the number of designs with A-frame picture condition, appears to be the reverse of the design fixation effect. Very few of these designs reflected the particular characteristics of this design; rather exposure to the picture appeared to result in both more designs and more diverse designs. The important issues are therefore why design fixation did not occur and why there was evidence for design diversity.

There appear to be two types of factors which could have contributed to the results. First there were the differences between the participants in our experiments and the Jannson and Smith's in terms of amount of design experience and type of design discipline being studied commented on previously. If the amount of design experience is the critical factor it is implied that, as designers become more experienced, the fixation effect increases; that is designers become more influenced by specific material related to what is to be designed than relative novices. In contrast to this effect, novices are more affected by existing knowledge based on experience; that is knowledge based on the most familiar examples. If this is the explanation it raises some interesting questions about how designers design and how designers are educated. If the design discipline is the critical factor with engineering designers being more likely to exhibit design fixation then this points to interesting differences between design disciplines that require further research.

The second factor which may have contributed to the absence of the design fixation effect is the variations in familiarity with different types of design. The demonstrated variations in familiarity parallel to a degree aspects of the designs that were most frequently produced. It is apparent that aspects of the unfamiliar designs occur very infrequently. The single post design is the most familiar and aspects of this design such as the rear location, method of attachment to the car and the method of support are reproduced frequently in all groups except the A-frame picture group. This is apparent from an examination of the Tables in the Appendix. These tables also indicate that there is a consistent trend in the data. The group shown a picture of a single post design consistently shows higher percentages than those found with the control group of design aspects consistent with a design fixation effect. This effect is apparent in Table 5 where the percentage of designs for each experimental group showing characteristics consistent with a single post design are shown for each aspect of the design analysed. For this group the results may therefore be made up of a familiarity component plus an effect due to design fixation. The absence of the fixation effect could therefore reflect the fact that many aspects of the designs derive from existing knowledge about bicycle rack designs based on the most frequently occurring and familiar design.

Table 5. Percentages of *all* designs showing characteristics similar to the most familiar, single post design for each aspect of the design.

<i>Group</i>	<i>Location Rear</i>	<i>Tow Ball Attachment</i>	<i>Single Post Support</i>	<i>Frame Attachment</i>
Control	54	29	31	49
Single post picture	69	43	43	67
Single post description	55	24	31	57
A-frame picture	55	29	17	61
A-frame description	49	50	28	69
Boot picture	58	30	32	80

It is also possible that this effect is not independent of the first factor discussed above. Relatively novice designers might be particularly affected by familiarity while more experienced designers are fixated by information presented within the context of the design problem. This type of interaction effect could also extend to the question of differences between design disciplines with familiarity, for example, effecting inexperienced designers in some disciplines and not others. However the role of familiarity may also be paradoxical. The A-frame design was rated as quite familiar but it appeared to produce both more designs and designs with unfamiliar aspects in contrast to the familiar single post design. This suggests that familiarity may operate in the following way. The single post design represents a highly familiar and very effective, simple design response to the problem of designing a bicycle rack which will bias all designs under these conditions. The A-frame design represents a less familiar design which is somewhat more complex and it is this difference which suggests alternatives to the designers. When the designs become very unfamiliar they do not have any effect on design because they are too different. In effect there may be an optimum amount of difference and unfamiliarity which will dis-inhibit the design process. Clearly more experiments are needed to test this hypothesis and to define more clearly what the words 'simple' and 'complex' might mean in this context and how an 'optimum' level of difference may be defined.

In conclusion, while this experiment did not replicate the design fixation effect and therefore could not address the issue of the differential effects of pictorial and semantic information in design, it has pointed to a number of other significant variables—familiarity, extent of difference, level of expertise and type of design discipline—which will need to be addressed in order to clarify the design fixation effect and the possible role of pictorial and semantic information in design. The results of this preliminary experiment also indicate that research in this area could be particularly productive both in terms of understanding the use of knowledge in design, the way in which knowledge should be presented in design education and the way in which expert knowledge should be represented in intelligent systems.

Acknowledgments. This research has been supported by a Seed Grant from the Key Centre of Design Quality, University of Sydney. The authors would like to thank Clare Callanan for her part in organising the data collection and analysis. Fay Sudweeks, in her inimitable fashion, shaped the raw text and figures into their highly readable form.

REFERENCES

- Chi, M. T. H., Glaser, R. and Farr, M. J. (1988). *The Nature of Expertise*, Lawrence Erlbaum, Hillsdale, New Jersey.
- Coyne, R. D., Rosenman, M. A., Radford, A. D., Balachandran, M. and Gero, J. S. (1990). *Knowledge-Based Design Systems*, Addison-Wesley, Reading, Massachusetts.
- Feigenbaum, E. A. (1989). What hath Simon wrought? in Klahr, D. and Kotovsky, K. (eds) *Complex Information Processing: The Impact of Herbert A. Simon*, Lawrence Erlbaum, Hillsdale, New Jersey, pp. 165–182.
- Jansson, D. G. and Smith, S. M. (1989). Design fixation, in National Science Foundation, *Proceedings of the Engineering Design Research Conference*, College of Engineering, University of Massachusetts, Amherst, pp. 53–76.
- Klahr, D. and Kotovsky, K. (eds) (1989). *Complex Information Processing: The Impact of Herbert A. Simon*, Lawrence Erlbaum, Hillsdale, New Jersey.
- Kolodner, J. (1985). Memory for experience, *Psychology of Learning and Motivation* 19: 1–57.
- Mandler, J. M. (1984). *Stories, Scripts, and Scenes: Aspects of Schema Theory*, Lawrence Erlbaum, Hillsdale, New Jersey.
- Mervis, C. B. and Rosch, E. (1981). Categorization of natural objects, *Annual Review of Psychology* 32: 89–115.
- Reiman, P. and Chi, M. T. H. (1989). Human expertise, in Gilhooly, K. J. (Ed.) *Human and Machine Problem Solving*, Plenum, New York, pp. 161–191.
- Rosch, E. H. and Mervis, C. B. (1975). Family resemblances: studies in the internal structure of categories, *Cognitive Psychology* 7: 573–605.
- Rosch, E., Mervis, C. B., Gray, W. D., Johnson, D. M. and Boyes-Braem, P. (1976). Family resemblances: studies in the internal structure of categories, *Cognitive Psychology* 8: 382–439.
- Schank, R. C. and Abelson, R. P. (1977). *Scripts, Plans, Goals and Understanding*, Lawrence Erlbaum, Hillsdale, New Jersey.
- Simon, H. A. (1981). *The Sciences of the Artificial* (2nd edn), MIT Press, Cambridge, Massachusetts.
- Tversky, B. and Hemenway, K. (1984). Objects, parts, and categories, *Journal of Experimental Psychology: General* 113(2): 169–193.
- Weisberg, R. W. (1988). Problem solving and creativity, in R. J. Sternberg (ed.), *The Nature of Creativity: Contemporary Psychological Perspectives*, Cambridge University Press, Massachusetts.
- Weisberg, R. W. and Alba, J. W. (1981). An examination of the alleged role of 'fixation' in the solution of several 'insight' problems, *Journal of Experimental Psychology* 110(2): 169–192.

APPENDIX

Tables A1. Percentages of *all* designs for each experimental group and method of location of bicycle rack

<i>Group</i>	<i>Rear</i>	<i>Boot and other</i>	<i>Roof</i>	<i>Total No.</i>
Control	54	9	37	35
Single post picture	69	4	27	51
Single post description	55	12	33	58
A-frame picture	49	13	38	87
A-frame description	70	6	24	54
Boot picture	57	11	32	37

Table A2. Percentages of first designs for each experimental group and method of location of bicycle rack

<i>Group</i>	<i>Rear</i>	<i>Boot and other</i>	<i>Roof</i>	<i>Total No.</i>
Control	56	7	37	30
Single post picture	68	3	29	34
Single post description	50	3	47	32
A-frame picture	48	24	28	42
A-frame description	63	5	32	38
Boot picture	42	19	39	31

Tables A3. Percentages of all designs for each experimental group and method of attachment of the bicycle rack to the car

<i>Group</i>	<i>Tow ball</i>	<i>Bumper bar</i>	<i>Roof</i>	<i>Total No.</i>
Control	29	34	37	35
Single post picture	43	26	31	51
Single post description	24	35	41	58
A-frame picture	30	25	45	87
A-frame description	50	26	24	54
Boot picture	30	32	38	37

Table A4. Percentages of first designs for each experimental group and method of attachment of the bicycle rack to the car

<i>Group</i>	<i>Tow ball</i>	<i>Bumper bar</i>	<i>Roof</i>	<i>Total No.</i>
Control	13	33	54	24
Single post picture	49	21	30	33
Single post description	30	23	47	30
A-frame picture	37	25	38	36
A-frame description	37	32	31	38
Boot picture	5	35	60	20

Tables A5. Percentages of all designs for each experimental group and method of support of bicycle rack

<i>Group</i>	<i>Single vertical post</i>	<i>Single horizontal post</i>	<i>Roof</i>	<i>Multiple Vertical, horizontal posts</i>	<i>Angled post</i>	<i>Total No.</i>
Control	32	11	23	17	17	35
Single post picture	42	6	24	20	8	51
Single post description	31	5	24	26	14	58
A-frame picture	17	8	21	37	17	87
A-frame description	28	9	13	28	22	54
Boot picture	32	5	30	14	19	37

Table A6. Percentages of first designs for each experimental group and method of support of bicycle rack

<i>Group</i>	<i>Single vertical post</i>	<i>Single horizontal post</i>	<i>Roof</i>	<i>Multiple Vertical, horizontal posts</i>	<i>Angled post</i>	<i>Total No.</i>
Control	34	13	27	13	13	31
Single post picture	52	9	27	12	0	34
Single post description	34	3	34	16	13	32
A-frame picture	15	8	17	37	23	40
A-frame description	19	10	23	37	11	37
Boot picture	32	3	39	16	10	30

Table A7. Percentages for all designs for each experimental group and method of attaching bicycle to the rack

<i>Group</i>	<i>Wheels</i>	<i>Frame</i>	<i>Bicycle enclosed</i>	<i>Total No.</i>
Control	40	49	11	35
Single post picture	21	67	12	51
Single post description	19	57	24	58
A-frame picture	17	61	22	87
A-frame description	20	69	11	54
Boot picture	24	68	8	37

Table A8. Percentages for the first designs for each experimental group and method of attaching the bicycle to the rack

<i>Group</i>	<i>Wheels</i>	<i>Frame</i>	<i>Bicycle enclosed</i>	<i>Total No.</i>
Control	41	52	7	29
Single post picture	19	75	6	32
Single post description	14	65	21	28
A-frame picture	14	69	15	39
A-frame description	25	63	12	32
Boot picture	27	63	10	30

Cognitive modelling of electronic design

L. Colgan and R. Spence

Department of Electrical Engineering
Imperial College
London SW7 2BT UK

Abstract Qualitative, descriptive models of design processes are relatively few in number. For those system developers attempting to provide support for the designer in the form of intuitive user interfaces, or by automation of part of the design process, this must be all too evident. This paper describes an attempt to develop and apply such a model in the context of developing a graphical, semi-automated design environment for electronic circuit designers.

1. WHY STUDY DESIGN?

Design is an interesting human activity that is prevalent in many walks of life. It is a complex example of human problem solving and decision making. 'Design is the realization of an artefact having some desired function' (Boyle, 1989). It is a creative process, often involving many iterations in which design parameters are varied. However, as a psychological phenomenon, design is little understood. This is most evident in attempts to automate the design process: 'Perhaps the key research problem in AI-based design for the 1980's is to develop better models of the design process' (Mostow, 1985).

The design of an analogue electronic circuit is a particularly complex and time-consuming instance of artefact design. The creative part of the process involves inventing the initial circuit topology - in other words, the way the constituent parts are connected together - and, for most of the time, adjusting circuit parameters (known as design variables) until the optimum set is found or a circuit topology change is forced.

Design time could be reduced and the designer's level of satisfaction increased merely by providing an integrated, supportive design environment. If, as well as supporting the conventional manual design process, this environment can also move towards the automation of some of the more repetitive aspects of the process such as parameter adjustment, then better, cheaper designs may be the result.

Of course, in providing this system, it must be ensured that the appropriate parts of the task are automated. At this stage we are not trying to automate the whole task (a flexible system that does this is probably many years away and its desirability is open to question); rather, our approach is one which involves exploiting the strengths of both the human and the computer. It is necessary then to deduce which tasks the human is best at (probably the highly creative tasks) and leave these firmly in the human domain, and automate the tedious parts or the ones the user is poor at. The assignment of roles within a semi-

automated design tool inevitably calls for an in-depth study of the design process leading to a model that is able to shed some light on these issues.

The essence of the project within which the model was developed was a strong conviction that (a) the human designer experiences considerable difficulty in adjusting the parameters of a circuit in order to minimise the discrepancy between desired and already achieved performance, (b) optimisation algorithms exist which can automate this process, and (c) the potential of these algorithms can only be realised through an effective interaction between designer and algorithm. The nature of the anticipated interaction is made more explicit in the name of the project - CoCo: the Control and Observation of Circuit Optimisation - whose objective was a working system incorporating optimisation. A fluent graphical interface supporting visualisation, together with expert knowledge-based 'Assistants', constituted the principal novel components of the CoCo system, but components whose most effective form will only be achieved with the aid of a model of the design process.

2. REQUIREMENTS FOR A MODEL OF CIRCUIT DESIGN

In order to design the proposed system, it was necessary first to understand the task which circuit designers are carrying out. To provide a usable tool which will make the process easier, the way the task is carried out presently - without optimisation - must be established. In building up some kind of model of the conventional design process we can more easily ensure that the necessary stages are permitted and supported by the new system. As the system does not attempt to automate the whole process but rather tries to relieve the user of some of the more tedious aspects of circuit design, it is essential that facilities are provided that allow the user to iterate through the usual and familiar design cycle. To be a useful tool, it must closely match the cognitive processes associated with various design activities (Baker, et al, 1989). Limitation of the load on working memory, and the provision of data in a readily digestible form requiring little additional cognitive processing, are appropriate tasks for such a tool.

In this light, an accurate model of the design process is an essential prerequisite to development. This model should capture the cycles of human activity, the knowledge sources being utilised and the reasoning mechanism used by analogue circuit designers: a subject that has been given very little attention (El-Turky and Perry, 1989). If the model can also indicate where cognitive limitations restrict progress, then the system can be further developed to alleviate their effect.

This paper describes user trials which were carried out with the aim of constructing models of the analogue circuit design process which would help with the development of a semi-automated design environment. User trials were carried out at two stages of system development, the first investigating how circuit designers are currently performing circuit design, and the second investigating how they cope with the first CoCo prototype incorporating automated design. The results were analysed and mapped into a goal-plan representation. Conclusions from this representation were then translated into implications for the system design. The end product was a model of the circuit design process which was not only useful for the CoCo system design, but may also be useful for other developers of computer-aided design (CAD) systems for the same user group.

3. USER STUDIES

As mentioned above, two studies were carried out, one in an early stage of system development as a detailed task analysis to investigate the conventional method of circuit design, and the second as a review of how the circuit design process is changed by using optimisation. The details are given below after a brief discussion of our methodology.

3.1 User Studies vs Controlled Experiments

Both sets of studies were in the form of carefully planned and executed, but unstructured interviews with circuit designers. The unstructured approach reflects the philosophy taken by the whole project: one of exploration rather than strict adhesion to any strong experimental paradigm, in keeping with the idea, advocated by Carroll and Campbell (1986), that the methodology should fit the subject matter, not the other way around. The project group has applied an incremental approach to both system development and the refinement of a model of circuit design. The intention was not to focus on a large number of strictly controlled experiments over a large number of users, but to learn as much as possible from a non-quantitative, more informal approach (Carroll and Rosson, 1985). The justification for this approach lies in the need to accommodate the diverse aims of the project and to satisfy industrial constraints.

From the project angle, the prototype system used by subjects in the second study needed to be usability tested to indicate small system changes and future system direction. A policy of incremental system improvement was adopted during the second study to ensure that low-level interaction problems did not obscure more important issues. The approach employed for these studies was qualitative rather than formal, with minor system problems corrected between trials.

The industrial constraints imposed on the studies are severe. In its natural setting, circuit design takes place over several months. This fact makes it difficult to observe the whole process with an acceptable subject population size. For any engineering establishment the time investment needed to release many designers for experimentation would outweigh the benefits they may experience from any future design system. For the experimenter, collection and analysis of data from many subjects over such a large time span would be a colossal task. Of course, taking this design process into the laboratory, and using students as subjects, may overcome many of these practical issues. Indeed, increasing the sample size in this way might ensure that the results are more 'representative', but the resulting model would *not* reflect the way circuit design is done by real circuit designers from industrial and research environments. We are left with a methodology that is a compromise between the two extremes: we try to keep conditions as consistent as possible in the subject's natural work environment, limiting the design process to a week, and studying six designers. In essence, we feel the construction of a model of how circuits are designed will be an incremental process, and that the methodology outlined below overcomes our constraints and is adequate for an early exploratory exercise.

3.2 Study 1: Conventional Method of Circuit Design

The circuit designers in these studies used a commercial design environment called MINNIE (Spence & Rankin, 1985). MINNIE is an interactive graphical user interface which allows a designer to draw a circuit design on a display screen and then examine predicted circuit performance. Performance prediction is carried out by one of several simulators with which MINNIE is capable of being interfaced.

Subjects. Two experienced circuit designers took part: one designer from an educational institution (Subject C1) and one from industry (Subject C2).

Method. Subject C1 gave a retrospective view of a circuit that he had previously designed. The subject reviewed the design process out loud in the presence of two experimenters who asked questions when he found it difficult to verbalise or when the issue being described was complex. The whole session was audiotaped for future transcription.

Subject C2, in contrast, designed a new circuit in the presence of an experimenter. Again the subject was asked to think aloud as he solved his design problem. The experimenter interrupted with questions and points of confirmation, and the session was audiotaped.

It is worth emphasising that these circuit designs were tackled *without* optimisation, thereby reflecting the conventional method of circuit design.

Data Logging. The subjects solved their circuit design problems using the MINNIE system. In the case of subject C2, this package was used during the experiment, and screen dumps were taken at strategic points to accompany the verbal protocols. A strategic point is defined as being any time that a screen selection is made by the subject or a decision point is reached. Subject C1 talked about a circuit he had designed previously using MINNIE, and so the audiotapes were the only data available.

3.3 Study 2: Semi-Automated Method of Circuit Design

The use of automated design was first facilitated by the implementation of a novel interface called MOUSE (Rankin and Siemensma, 1989) which integrated the manual design cycle of modification-and-simulation with automated circuit improvement via optimisation algorithms. MOUSE therefore provided the opportunity both to involve users in system assessment and to carry out investigations of a potentially new circuit design process with a view to obtaining a more extensive model.

Before the experiment, MOUSE was subjected to beta testing by a hostile user. A pilot study then uncovered unforeseen system errors and protocol oversights. Indeed, throughout the study, the approach of incremental refinement was applied to the experimental format as well as to the system. We required a 'clean' system, free from bugs and low-level interaction noise before we could address deeper questions; consequently the pilot study and one of the experimental design sessions (subject M1 - see below) took place at protected environments with sympathetic users. As MOUSE and the protocol became more stable, more hostile users were employed.

The experiment, which lasted 5 days for each subject, consisted of seven modules: Tutorial; Benchmark Problem; Questionnaire; Advanced Tutorial; Free Sessions; Questionnaire; and Debriefing.

Subjects. One subject from an industrial research department took part in the pilot study. Three subjects took part in the main experiment, one from a university research group (Subject M1) and two from a commercial environment (Subjects M2 and M3).

Tutorial.: The experimenter administers a standard tutorial, giving the subjects enough information to complete a simple benchmark problem. The subjects are deliberately undertrained in order to gauge the level of training needed and also to retain an element of the naive user.

Benchmark Problem. Subjects are given an electrical design problem. This problem is chosen to test low-level aspects of MOUSE, such as menu handling and the access of different system modes, when the subject is not engaged in complex problem-solving. Subjects are encouraged to attempt as much of the problem as possible without asking for help. The point at which they start to ask questions is noted. They are asked to think aloud while they attempt the problem. The session is recorded on audiotape, and accompanied with time-stamped screen dumps. Hesitations, mispicks and losses of orientation are noted.

Questionnaire A. The questionnaire consists of a mixture of rating scales, yes/no questions and open-ended questions. It attempts to test understanding of the system and identify missing facilities.

Advanced Tutorial. The experimenter administers a tutorial on the more advanced features of the system. Difficulties are noted.

Free sessions. The experimenter, supported by an adviser on mathematical and electrical aspects of the system, sits with subjects while they solve their own circuit design problems. The adviser plays the roles which the mathematical and electrical expert systems play in the final system. Screen dumps are made and the sessions are audiotaped. As in the previous study, screen dumps were taken whenever a subject made a screen selection or reached a verbalised decision point.

Questionnaire B. Again a mixture of question types, but this time focusing on the use of MOUSE for subjects' specific circuit design problems.

Debriefing. A structured session based on questionnaire answers, followed by open questions and discussion. This gives the subjects a chance to qualify their comments and discuss further issues. These sessions are audiotaped.

Data Logging. Notes were taken throughout the study, but in addition audiotaped material is available from the benchmark problem-solving exercise, the free sessions, and the debriefing. This study generated a large amount of audiotaped material (15 hours per

subject). Since transcription and editing takes approximately two days for an hour of tape, only selected segments were transcribed. Segments were transcribed when it was deemed that the visual data was insufficient in describing the design process at a particular point or when screen dumps and experimenter's notes needed confirmation. The screen dumps provided snapshots of the design process and were in a form which was easier to handle than videotape. Any notes, sketches and formulae used by the subjects were also collected.

The experiment was punctuated with discussion encouraged by the electrical and mathematical adviser. This discussion covered the aim of the circuit designer at a particular stage in the design; the design strategy; alternative strategies; and rationale for design decisions. This discussion was not tightly structured, but was deliberately intended to create an atmosphere of trust and mutual sympathy for the aims of all participants.

4. PRELIMINARY ANALYSIS: A DESIGN TRACE

Wherever possible, conventional protocol analysis (Ericsson and Simon, 1985) was avoided in view of the length of the circuit design procedure. All subjects had screen dumps available from the system they were using during the experiment*. The collected screen dumps were formed into a design trace (described below) and only selected parts of the audiotaped thought processes were transcribed. Audiotape segments were chosen for transcription when the experimenter's notes and the screen dumps did not adequately describe design at that stage. The exception was subject C1 who gave a retrospective view of a circuit previously designed: *all* his verbalisations were transcribed for analysis. Any transcription that was performed aimed to identify key concepts and recurring vocabulary, and to divide the sessions into design episodes.

As a preliminary analysis of the data, a design trace was created (as shown in Figure 1) for all subjects except C1. It was produced by pasting together annotated screen dumps to result in a series of 'design stages'. Each 'design stage' represented an episode of the design process and contained the current state of the circuit. During each design stage several issues were made explicit by annotation from the audiotape and the experimenter's notes: the plan the designer was following, the strategies being considered, and the circuit parameters being investigated were typical screen annotations. A design stage is defined as the interval between either a major screen selection or a verbalised plan, and the next major screen selection or plan. Major screen selections were those which resulted in a system mode change as opposed to those which highlighted a screen icon, for instance.

The design trace constituted a detailed case history for the circuit design being considered. It looked very much like a flow chart: a set of static screens with arrows between them. It was found to be a powerful medium to promote discussion between project members and the subjects themselves. This design trace is similar to Mostow's idealised design history (Mostow, 1985) and comprised a complete review of design

* The subjects in the study designed a range of circuits including a high gain, differential input, operational amplifier using a voltage following current mirror, a two stage Gallium Arsenide circuit to investigate whether Gallium Arsenide can be used for high speed A to D converters at high frequencies, and an active splitter circuit.

development. The trace, depicting the designer's exploration of the circuit, was then taken back to the designer for confirmation and, where necessary, correction. Since the trace was couched in circuit design terms, designers were comfortable with it. For both experimenter and designer, it was a useful way to record and condense detailed information about the circuit design problem.

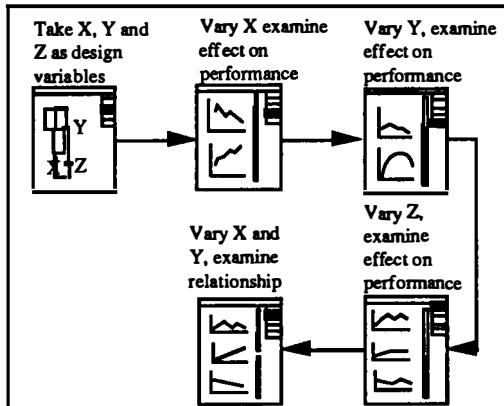


Figure 1. Extract from Design Trace for Subject M1

The generation of a design trace from experimental data offers an alternative to the complete method of protocol analysis in which the subjects' utterances are logged and classified in some way. It addresses the condensation of data problem, where data should be compressed as early as possible for ease of management but should retain key elements for productive discussion. A screen dump was considered to approximately represent an utterance since it was produced each time the subject made a screen selection or reached a decision point. The method is an alternative which is less time-consuming than verbal protocol analysis, and in many ways the trace is similar to a story-board (Mountford, 1989) of the whole design process. From the trace, it was straightforward to identify cycles of exploration the subjects were iterating through (for example, explore-change-explore cycles).

5. GOAL-PLAN MODEL

The design trace served its purpose well in facilitating a preliminary analysis of the data. The trace was useful as a means of confirming the design stages with the designer and also as a graphical description of each of the circuit design processes the subjects were carrying out. Nevertheless, the trace was still too complex to allow easy comparison with traces of other subjects. A general representation was needed which would reflect the circuit design strategy of any individual, and which would allow a comparison of the representations of all subjects in order to extract common procedures, methods and accessed knowledge sources. It would also be useful if the representation identified behaviour which was time-intensive, unsuccessful and tedious, possibly signifying tasks which could be automated.

In the context of design, data analysis can be carried out at any of several levels of detail and, because of this, a representation which made the grain of analysis explicit was needed. The representation should reflect the fact that further analysis at a higher or lower level of abstraction could be carried out as and when required, and added to the model at some future time.

Exploration showed that a goal-based analysis appeared to fit the data well. Goal-based representations of human behaviour have been important for several decades (Miller, Galanter and Pribram, 1960; Black, Kay and Soloway, 1987), and similar concepts are used in the GOMS model of Card, Moran and Newell (1983). The value of the GOMS model can be appreciated when attempting to implement a model of a user engaged in a relatively simple process such as a text editing task which has a finite number of methods by which the goal can be achieved. However, for an exceedingly complex and creative problem-solving process such as circuit design, the GOMS model is inappropriate, and a new goal-based representation was sought.

5.1 The Basic Building Block

The model was derived from consideration of the prominent features of the design traces. A goal-plan representation was suggested by the approach subjects appeared to engage in. This approach resembled design decomposition into composite goals and subgoals, although complete goal decomposition was not strictly adhered to. Subjects digressed from completion of subgoals by engaging in goal abandonment, new goal generation and partial solution of goals. We sought a goal-plan representation which, as a primary requirement, would reflect the essentially iterative and hierarchical aspects of engineering design. Thus, each design subgoal should be capable of decomposition into a separate goal-based representation of its own, and each goal abstracted into higher level goal-based representations. An additional feature was identified in the design trace. Each design stage resulted in either a change in the state of the circuit being designed, or a change in the designer's knowledge of the circuit, or both. This fact was considered important enough to merit explicit representation as 'Start' and 'End' states in the model.

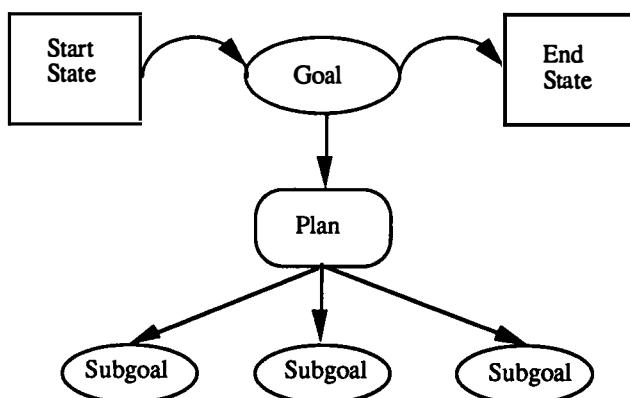


Figure 2. Goal-Plan Building Block

The proposed representation is based on the basic building block shown in Figure 2. Using this building block, part of a design process can be represented as in Figure 3. We say 'part of' intentionally: movement down the hierarchy tends towards simple actions such as mouse-clicks which may not be crucial to the sort of design process characterisation we seek. Similarly, the goal-plan structures at higher levels tend towards meta-planning and control structures. In Figure 3, the time dimension is essentially left-to-right, with the subgoals on each level being carried out in sequence. Figure 3 also illustrates the expansion of a subgoal into its own goal-plan structure, for example, the goal-plan structure for each subgoal in the top level can be located in the next level of the hierarchy by following the dashed lines. The value of the representation lies in the fact that a complete hierarchy does not have to be constructed. As much of the design process as is required can be constructed starting from a convenient point and moving upwards and downwards as far as necessary.

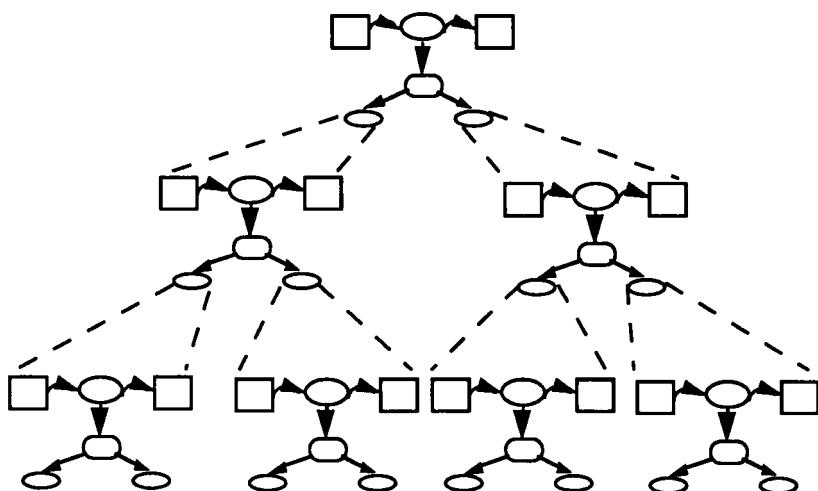


Figure 3. Hierarchical Goal-Plan Structure

The design trace relates to the goal-plan structures in two explicit ways. Firstly, those design stages from the design traces which contained major screen selections (changes of mode as opposed to selecting a delete icon, for example) were represented explicitly in the goal-plan structure, and they naturally formed what we chose to be the lowest layer of the goal-plan hierarchy. Secondly, those design stages which represented design decisions verbalised explicitly by the designer or derived from discussion, generally formed one or two layers above this. The layers higher in the hierarchy were an abstraction derived from the design trace and the experimenters' subjective interpretation of the data.

5.2 Definition of Primitive Elements

The basic building block is made up of the following elements: goals (and subgoals); plans; start states; end states; and directed arcs. Each element is described below:

Goals are states that the subject is trying to achieve. *Subgoals* are states the subject is trying to reach in order to achieve a higher level goal. The distinction between goals and subgoals is not concrete: in effect all goals are subgoals of some higher level goal.

Plans are groups of actions whose execution should result in the realization of the goal.

The *Start State* is a description of the problem space (some aspect of the circuit design) before the goal is formulated.

The *End State* is a description of the problem space after the plan has been executed. It differs from the goal state in that it is an *achieved* state rather than an *intended* state. This fact takes account of user errors in two areas: incomplete plans or choosing the incorrect plan to achieve the goal. It also accounts for those occasions when the goal is only vaguely specified such as 'investigate' a particular aspect of the circuit. The end state contains the results of plan execution. In effect it represents the state of the design at that point. This aspect of the building block permits exploratory and flawed behaviour to be represented.

The *directed arcs* are where the subject's domain and problem-specific knowledge come into play. In deciding what plan will achieve a goal, and dividing the goals into subgoals and actions, the user is relying on previous knowledge built up through experience. In a similar way, knowledge sources are being utilised to deduce which goal should be proposed from a Start State, and to evaluate the success of the plan in the End State.

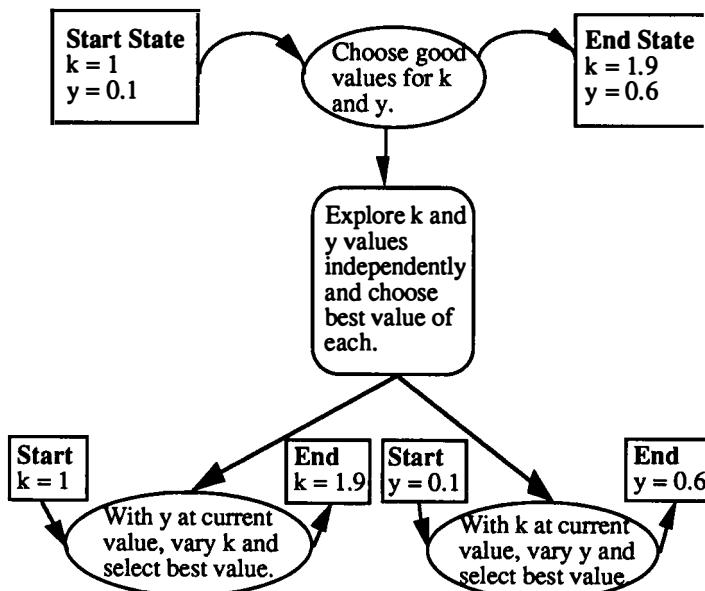


Figure 4. Goal-plan structure applied to an extract of designer M1's design trace

5.3 Populating the Representation

The model was instantiated with the data at the two lowest levels first, with higher structures added later. Each design trace was examined to identify a set of low level goal-plan structures. The goals of these low level structures were then fitted into higher level goal-plan structures. The method can be illustrated by an extract from designer M1's design trace, which led to the model of Figure 4.

The extract is modelled at one of the lowest levels of the representation of the design trace, where the designer wants to choose 'good' values for the parameters k and y. Their existing values are 1 and 0.1 respectively. Although more than one plan to achieve this goal could be formulated, that adopted by designer M1 was to *independently* explore and select a good value, first for k then for y. The subgoal structures, which are intentionally not shown in Figure 4, could be achieved by explorations of many types (for example, try minimum, try maximum, try average, define best range and repeat) and goals even further down the hierarchy would refer to keystroke level actions. The final values for k and y achieved by these lower level goal-plan structures are then inherited by the structure shown in Figure 4. Similarly, the model can extend upwards to include meta-planning and control. One perceived advantage of the this modelling approach, in fact, is the ease with which the 'grain of analysis' can be handled.

It was decided, in the first instance, to aim for a representation stretching from the subjects' verbalised goals about the overall plan for circuit design and optimisation, down to, but not including, the mechanical interactions with the system.

5.4 Comparison of Subjects' Goal-Plan Structures

A noticeable feature of this modelling method is the overall shape of the goal-plan structures. Figures 5 and 6 show the general shape of the representations resulting, respectively, from the analysis of subjects M1 and M2.

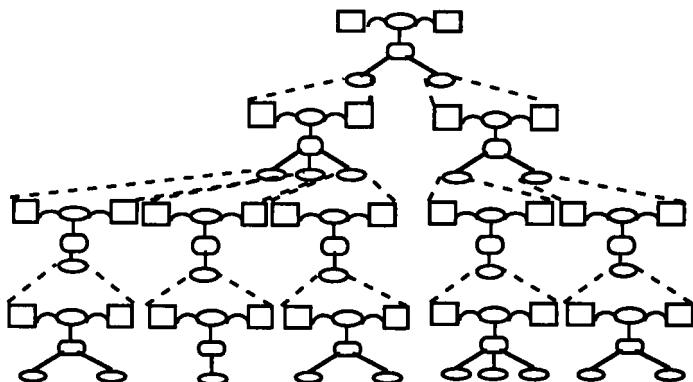


Figure 5. Goal-Plan Structure of Designer M1

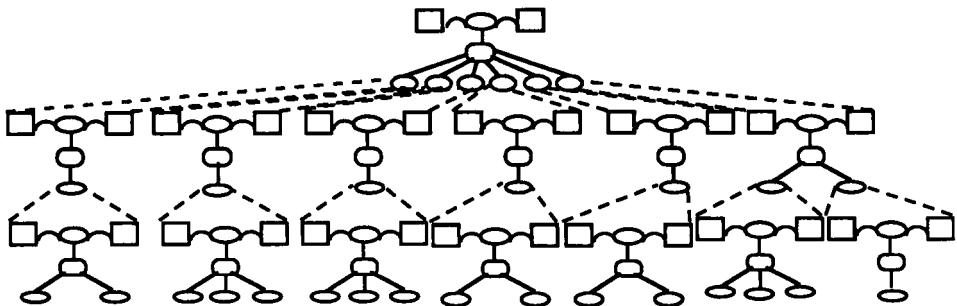


Figure 6. Goal-Plan Structure of Designer M2

Since the designers were dealing with totally different circuit design problems a direct comparison of the diagrams is not relevant. However, it is useful to point out that M1's diagram is deeper than that of M2. M1 happened to be a research designer who knew much about the circuit in question and wanted to deepen his knowledge, whereas M2 had different aims. This latter subject was working within a typical industrial context: he had inherited the circuit and the specifications, and was concerned with the rapid production of an acceptable design solution.

6. INFORMATION DERIVED FROM THE REPRESENTATION

Other features identified in the models generated for all subjects are briefly discussed below.

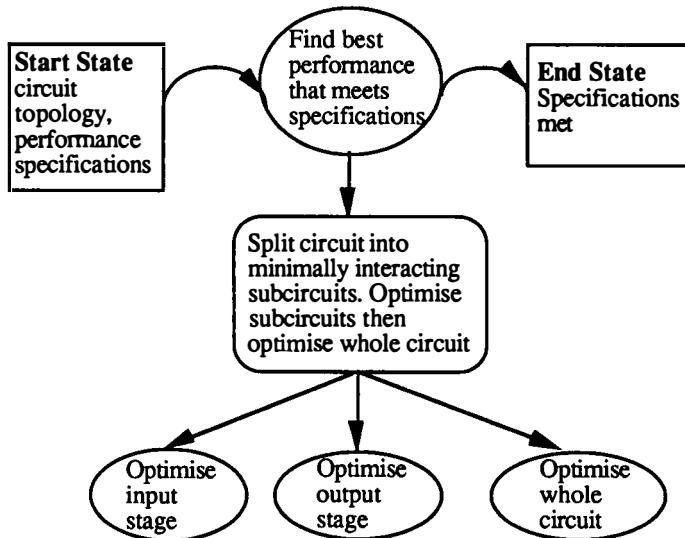


Figure 7. Subject M1 chunking the circuit

6.1 Chunking

From a high-level in MI's representation (Figure 7) the designer can be seen to be using a 'chunking' strategy to help with the overall design improvement. This observed strategy of chunking the circuit into smaller blocks about which knowledge already existed was viewed as an assignment of functional roles to groups of components. The circuit behaviour associated with these smaller blocks was, however, mentally adapted to take account of their present location in the circuit. Subjects were observed adjusting the component roles as their knowledge of the circuit grew.

Chunking is seen on other levels, the identified objects being chains of devices, current mirrors, long-tailed pairs, down to the individual component level. It was seen to trigger the recall of knowledge about the stereotyped behaviour of groups of circuit components. This idea is very similar to how knowledge is stored in frames (Minsky, 1975) and scripts (Schank and Abelson, 1977), knowledge structures which have proven successful in many A.I. programs. Thus, whenever the designer encounters a recognised configuration of circuit components, knowledge built up through experience is triggered.

6.2 Dimension reduction

It was detected that users had difficulties with thinking in more than two dimensions at once and hence in identifying interdependencies amongst circuit parameters, especially when these variables interacted in a complex fashion. Subjects attempted to overcome these difficulties by reducing the number of dimensions they had to deal with: the circuit was chunked into subcircuits, and a strategy similar to 'divide-and-conquer' was adopted to reduce the number of parameters being dealt with simultaneously. In addition, once the principal parameters and the performances they affect had been identified, subjects tended to suppress, at least temporarily, their consideration of other, perhaps stronger, variables and concentrate on optimising a small subset, as illustrated in Figure 8.

The process of identifying a minimal set of strong design variables and then of investigating how they interact and affect circuit performance are tasks which humans find difficult. As the number of design variables increases and as combinations of variables start to have an effect, the more difficult the task becomes. Of course, the use of an optimisation algorithm may eliminate the need to think in several dimensions at once since, given a set of design variables, the algorithm should arrive at optimum values; nevertheless, the user still has to choose these design variables at the outset. The length of time the optimisation algorithm takes to generate a result, together with the inability to view progress interactively, means that users still spend much time on manual investigation and do not use the full potential of the system.

Subjects often looked back at aspects of the circuit design problem that were not made explicit in the present system mode. This was not surprising, since a user cannot keep all current values of circuit components and circuit performances in his mind at once. Indeed, as a consequence, dimension reduction also took place more globally and across performance domains. Thus, a designer eventually concerned with the quiescent, frequency domain and time-domain behaviour of an amplifier would, after chunking the system into modules, concentrate on noise behaviour by suppressing all but the input stage and all designable variables except a couple of prominent ones whose relationship to the

performance had been previously determined. The phenomenon of dimension reduction can be seen as an example of subjects utilising strategies that radically shrink the search space in their attempt to satisfy multiple constraints (Chandrasekaran, 1990).

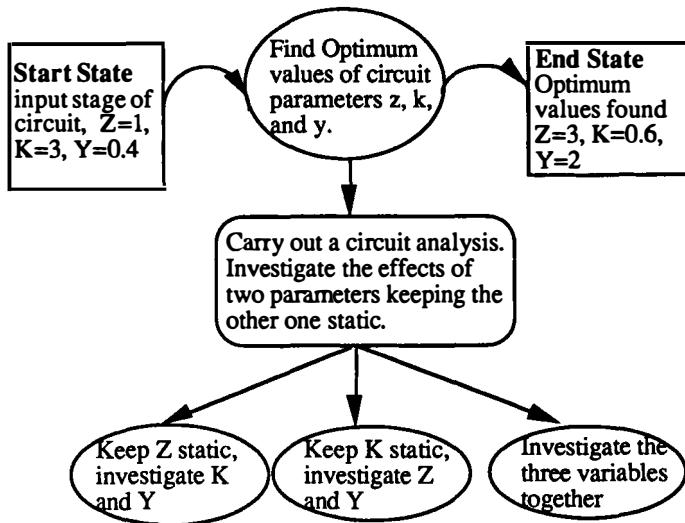


Figure 8. Dimension Reduction Displayed by Subject M1

6.3 Procedural knowledge

At regular stages throughout the process of circuit design each subject was observed to be utilising sources of procedural knowledge. This knowledge was partly based on electrical engineering first principles and partly on experience of other circuits. It was also being appended with specific information being built up through exploration. This knowledge was in the form of a set of circuit-specific rules and took the following form:

'If the loading factor is approximated to a value of X pF per micron,
then the settling time will be closer to specifications'

These rules were often hypotheses which the designers attempted to verify by simulation. However, they could often be seen to take extrapolative leaps to bypass the time consuming verification stage. Using their own set of rules, they would jump from the half finished testing of one topology to a new topology. Although not explicitly represented in the goal-plan structures, these rules can be modelled by the directed arcs, that is, in general, knowledge is employed when moving from one state to another. Our intention was not to construct a comprehensive set of design rules for automation, although many of these rules were collected as a by-product of the process. This rich source of domain-specific knowledge, that is often difficult to access, has been noted by

other researchers; for example, Goel and Pirolli (1989) have described domain-specific schemas that are acquired during years of professional training.

The formal incorporation of aspects such as dimension reduction, working memory limitations and procedural knowledge within, or allied to, the model is currently under investigation.

6.4 The effects of optimisation

Several differences were noticed between the goal-plan structures of Study 1 and those of Study 2, representing the differences between the conventional method of circuit design and the semi-automated method. The tendency to spend time investigating design variables and their behaviour was reduced, and instead subjects chose a set of likely candidates as strong design variables and left it to the optimiser to find the best values. However, one subject in particular, M1, exhibited a reluctance to rely on the system facilities and still chose to do much manual investigation himself. He manually determined the best starting values for the design variables, and then allowed the optimiser to improve on those values. Consequently, although his model still contained many goal-plan structures depicting investigation, his resulting design was better than could be achieved manually.

The knowledge sources utilised and enhanced when subjects were using optimisation differ from those used in conventional design. Subjects had to construct knowledge sources concerned with optimisation algorithms and how to use them in MOUSE. 'Setting up the optimisation profile' or specifying the performance requirements accurately was found to be difficult for subjects, and many goal-plan structures were dedicated to attempts to do this.

7. IMPLICATIONS FOR SYSTEM DEVELOPMENT

Primarily, the goal-plan structures illuminated time-consuming design activities even when optimisation was used. These time-consuming design activities represent tasks which could be automated. In particular, designers spent much time identifying strong design variables, interdependencies between design variables, and establishing relationships between components and groups of components.

One time-saving solution would be to automate the identification of powerful design variables. Moreover, identifying where the designer has difficulties, and providing adequate system support to help overcome them, would ease the design process. The models show that one of the main difficulties results from having to deal with a mass of information in multi-dimensional space. There is evidence to suggest that providing system displays which allow strong design variables to be identified through visualisation and which reduce the amount of human processing necessary to identify complex trade-off relationships, may shorten the design cycle. These user interface issues are being tackled with the development of a symbolic view of optimisation progress as reported in these proceedings (Colgan, Rankin and Spence, 1991).

The models suggest that provision should be made for the designer to interact with the system during the optimisation process. Much time was spent waiting for the system to

optimise sets of design variables without any feedback. Often a designer would wait for optimisation to finish only to find that a variable had become stuck against a bound that had been imposed on it. This would mean that the optimisation would have to be rerun with relaxed bounds on variable values. The provision of displays which alert the designer of constraint violation and allow the design process to be monitored and steered during optimisation should result in a more efficient design cycle.

Justification for a knowledge-based approach was revealed. There is evidence that a 'designer's rule book' knowledge base containing an experienced designer's 'rules-of-thumb' may be useful for the less experienced designer. This may store design rules for typical circuits or subcircuits. A rich supply of heuristics, relating to both the design variables effect on performance, and to strategies for tackling the design, was uncovered.

In general, designers were diverted from the main design task by the details of numerical optimisation. A mathematical knowledge base which allows all the mathematical aspects of optimisation to be hidden from the user unless otherwise requested would reduce the designer's cognitive load and is now implemented in prototype form (Gupta and Rankin, 1991).

Automatic identification of minimally interacting subcircuits would allow the designer to take the subcircuits off-line and treat them as independent design problems. This would help the designer in his own strategy of dimension reduction.

8. CONCLUSIONS

Even in well-modelled domains such as semiconductor devices, a range of simplified and therefore quite approximate models act as valuable tools for organised thought, and are therefore potentially of value in innovation, diagnosis and design. It may be decades before we approach a reasonably accurate quantitative model of the engineering design process but, if what has been demonstrated in other domains holds true, approximate and partial models will be of considerable value in the continuing development of computer-aided design systems. In the CoCo project, by basing our trials on industrially sensible CAD systems and by adopting an evolutionary approach to development, we have already been able to identify models that point to useful directions for future development and exploration.

Acknowledgements. CoCo is a collaborative project between Imperial College, UK, and Philips Research Laboratories, Redhill, UK. The authors gratefully acknowledge partial support from the ESRC/MRC/SERC Cognitive Science/HCI Initiative (Grant number SPG 9019856). We would like to thank Paul Rankin of Philips Research Laboratories for his many useful ideas and discussions.

REFERENCES

Baker, K.D., Ball, L.J., Culverhouse, P.F., Dennis, I., Evans, J.St.B.T., Jagodzinski, A.P., Pearce, P.D., Scothern, D.G.C. and Venner, G.M., (1989) A Psychologically Based Intelligent Design Aid, *Eurographics Workshop in Intelligent CAD*.

- Black, J. B., Kay, D. S. and Soloway, E. M. (1987). Goal and plan knowledge representations: from stories to text editors and programs, *in* J. M. Carroll (ed.) *Interfacing Thought*. Cambridge, Massachusetts, MIT Press.
- Boyle, J-M. (1989) Interactive engineering systems design: a study for artificial intelligence applications *Artificial Intelligence in Engineering*, 4(2):58-69.
- Card, S. K., Moran, T. P. and Newell, A. (1983). *The Psychology of Human-Computer Interaction*. NJ., Lawrence Erlbaum Associates
- Carroll, J.M. and Campbell, R.L., (1986) Softening up Hard Science: Reply to Newell and Card, *Human-Computer Interaction* 2:227-249.
- Carroll, J. M. and Rosson, M. B. (1985) Usability Specifications as a tool in Iterative Development, *in*: H. R. Hartson (ed) *Advances in Human-Computer Interaction*. Norwood, New Jersey: Ablex.
- Chandrasekaran, B. (1990) Design problem solving: a task analysis. *AI Magazine*, Winter, pp. 59-71.
- Colgan, L., Rankin, P. and Spence, R. (1991) Steering automated design. *Artificial Intelligence in Design*. Edinburgh, June 25-27.
- El-Turky, F and Perry, E. E. (1989) BLADES: An Artificial Intelligence Approach to Analog Circuit Design *IEEE Transactions on Computer-Aided Design.*, 8(6): 680-692.
- Ericsson, K. A and Simon, H. A. (1984) *Protocol Analysis: Verbal Reports as Data*. Cambridge Mass., MIT Press.
- Goel, V. and Pirolli, P. (1989) Motivating the notion of generic design within Information-Processing Theory: The design problem space. *AI Magazine*, Spring, pp.19-36.
- Gupta, A. and Rankin, P. (1991) Knowledge assistants for design optimisation. *Artificial Intelligence in Design*. Edinburgh, June 25-27.
- Miller, G. A., Galanter, G. and Pribram, K. H. (1960). *Plans and the Structure of Behaviour*, New York, Holt, Rinehart and Winston.
- Minsky, M. (1975) A framework for representing knowledge., *in* P. Winston (ed), *The Psychology of Computer Vision*, McGraw-Hill, New York.
- Mostow, J. (1985) Toward Better Models of the Design Process. *The AI Magazine*. Spring, pp. 44-56.
- Mountford, S. J., et al., (1989) Drama and Personality in Interface Design. *Proceedings of CHI '89 conference on Human Factors in Computing Systems* (Austin, Tx) pp.105-108.
- Rankin, P.J. and Siemensma, J.M. (1989) Analogue Circuit Optimization in a Graphical Environment, *IEEE ICCAD-89* pp. 372-375.
- Schank, R. C. and Abelson, R.P. (1977) *Scripts, Plans, Goals, and Understanding*, Erlbaum, Hillsdale, N.J.
- Spence, R. and Rankin, P. (1985) Minnie: a tool for the interactive-graphic design of electronic circuits, *IEE Colloq. on Circuit Design using Workstations*, London, 26 April.

Keys to the successful development of an AI-based tool for life-cycle design

K. Ishii[†] and L. Hornberger[§]

[†]Department of Mechanical Engineering
Ohio State University
Columbus OH 43210-1107 USA

[§]Apple Computer, Inc.
3565 Monroe Street, MS 67A
Santa Clara CA 95050 USA

Abstract. Artificial Intelligence has raised engineers expectations for a powerful tool to aid in engineering design, particularly for one that incorporates life-cycle issues such as manufacturability and serviceability. However, many experimental AI tools have not succeeded in the actual product design environment. This paper documents our experiences in implementing AI-based tools for life-cycle engineering. The paper traces the development of DAISIE (Designers' Aid for SImultaneous Engineering), a framework for computer programs that evaluate the compatibility of a candidate design with various life-cycle requirements, e.g., assembly, molding, etc. The initial programs were useful in some aspects, but received overwhelming criticisms in their long term practical utility. The paper identifies the several essential keys to a successful development of truly useful design tools. We conclude that an AI tool is a product in itself and warrants a careful marketing effort, i.e., identification of its role, clarification of customer requirements, and timing of its development.

1. BACKGROUND

In the past decade, the potential of artificial intelligence has caught the imagination of many engineers and led them to expect that it will be a powerful tool for engineering design (Barkan, 1988). Some even believe it will ultimately be the panacea for all design problems.

Many researchers have developed AI-based programs that address various stages of product design. Rinderle (1990) reported on the use of AI for the configuration stage of design, while Nielson and Dixon (1986) developed an expert system for material selection. There also seem to be many applications of AI available for detailed design. Dixon's group, for example, has developed several programs for mechanical components ranging from heat fins to V-belts. AI has also been used to evaluate functional requirements within design such as strength, heat transfer, and mechanical interference (Kamal, et al, 1987).

Agogino (1987) has used AI to implement the active constraint strategy in the optimal design of hydraulic cylinders.

As a complement to these mechanical design aids, a number of developers have used AI to assimilate the rules and guidelines for the more subtle stages of design including, for example, the areas of manufacturability and serviceability. Increasing industry interest in design for manufacturability led to the development of many computer programs that check candidate designs for compatibility with manufacturing and service requirements. While not specifically AI-based, the pioneering work of Boothroyd and Dewhurst's on their Design for Assembly (1983) program triggered national interest in this type of program. Poli's work (1988) built on this interest by addressing early cost estimating for the injection molding manufacturing process and Dixon's work (1986) did the same with the extrusion process.

Some researchers have taken the ultimate step of combining the many stages of design into a single AI program which simultaneously designs the product and chooses the process by which it is to be manufactured. Cutkosky (1988) has done this with machining operations, while Desa (1989) does this for stamping. Some of these programs are linked to CAD programs so that the AI module can even retrieve geometry information directly from CAD database, and if necessary, automatically update it based on appropriate redesign rules. Shah (1990) reports on a feature-based design CAD environment which is designed to review a machined part for manufacturability. Liou and Miller (1990) combine a parametric CAD system and an expert system shell to create a program for die casting design. Typically, these programs address detailed part designs rather than early layout designs.

Most of the programs cited above are experimental. They all hold tremendous promise as powerful tools in the design world, promise equivalent to that of CAD or robotics. Unlike these latter tools, the bulk of these programs have yet to be tested in the real world. Their effectiveness as product design tools is yet to be verified, for few of these programs have left the calm and cloistered atmosphere of the university environment and entered the dynamic and secular industrial world.

This paper reports on our experiences in introducing AI-based tools into the product design world at Apple Computer, Inc. and the discoveries we made during this experience. In particular, we focus on the key factors that we found to be helpful development of AI-based programs that aid life-cycle design, otherwise known as simultaneous engineering.

2. DESIGN COMPATIBILITY ANALYSIS AND DAISIE

For the past several years, the authors have collaborated on the development of expert systems for use in product design. This effort began in 1987 with an initial study at Stanford University on design for assembly (Ishii, Adler and Barkan, 1988) and has continued over the last three years with studies on design for serviceability (Makino, et al,

1989) and on design for plastic molding (Ishii, Hornberger, and Liou, 1989). Our primary intent in this work has been to create modules of knowledge that designers can access to check and verify their designs relative to known requirements and guidelines. Our secondary goal has been to provide designers with a tool to weigh and balance their various design concerns. Specifically, we have addressed life-cycle issues such as part function, appearance, manufacturability, assembly and serviceability.

The tool we have developed through this joint effort is a computerized model of design reviews, Design Compatibility Analysis or DCA. The central idea of DCA is to focus on the opinions of each expert who would typically be present at a design review and simultaneously evaluate a candidate design from multiple viewpoints. That is, we modeled a typical "round table" meeting in which various experts discuss a proposed design and suggest improvements (Figure 1). In a typical design review, these suggestions usually focus on design modifications needed to meet the various expert's requirements, but they may also prompt respecification of the manufacturing process (use of an alternative molding machine, etc.) or even alter the focus of the product (use of a single color design instead of a two toned one) as the various issues are weighed relative to the interests of the participants.

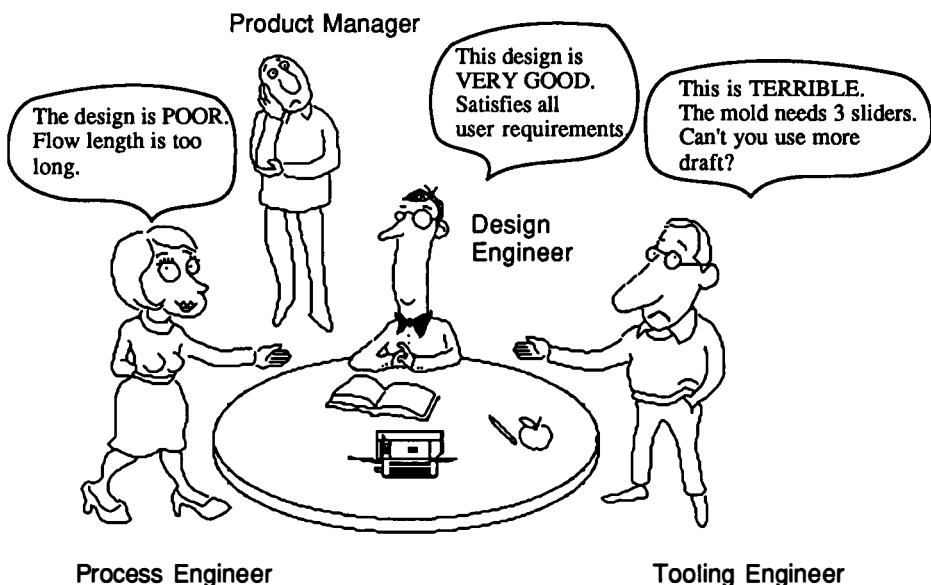


Figure 1. The "round table" model of design reviews

To simulate the review process, knowledge modules are linked to an AI program called DAISIE. In essence, it sorts through the knowledge base and seeks the design rules which apply to a particular situation. DAISIE accommodates the experts' opinions in terms of compatibility among user requirements, the candidate design, and the processing technique. Experts tend to use qualitative adjectives rather than quantitative measures to describe their

view of the compatibility between the candidate design and their field of expertise. Some compatibility issues are absolute rules, which many issues are negotiable. Consequently, adjectives such as "excellent, very good, good, poor, bad, and very bad" are common in discussion. These compatibility rules are called C-data in the DAISIE program. An example of C-data can be seen later in the paper (Figure 5). Along with this compatibility knowledge, DAISIE accommodates ordinary inference rules related to the characteristics of the candidate design or the implications of the specifications and process constraints on the design of the product.

Figure 2 illustrates DAISIE's design assessment process. Given a description of the proposed design and the design specifications, DAISIE analyzes the individual design elements or features relative to the specifications. Then, for each element that makes up the design, DAISIE selects from the compatibility knowledge-base the data that applies to it as matching compatibility data (MCD). Typically, there will be many C-data that matches a certain design. The program then computes a user-defined function that combines these comments and maps them to a numerical score between 0 and 1. Our function adopts the worst case comment if there are at least one negative comment, otherwise takes the best comment.

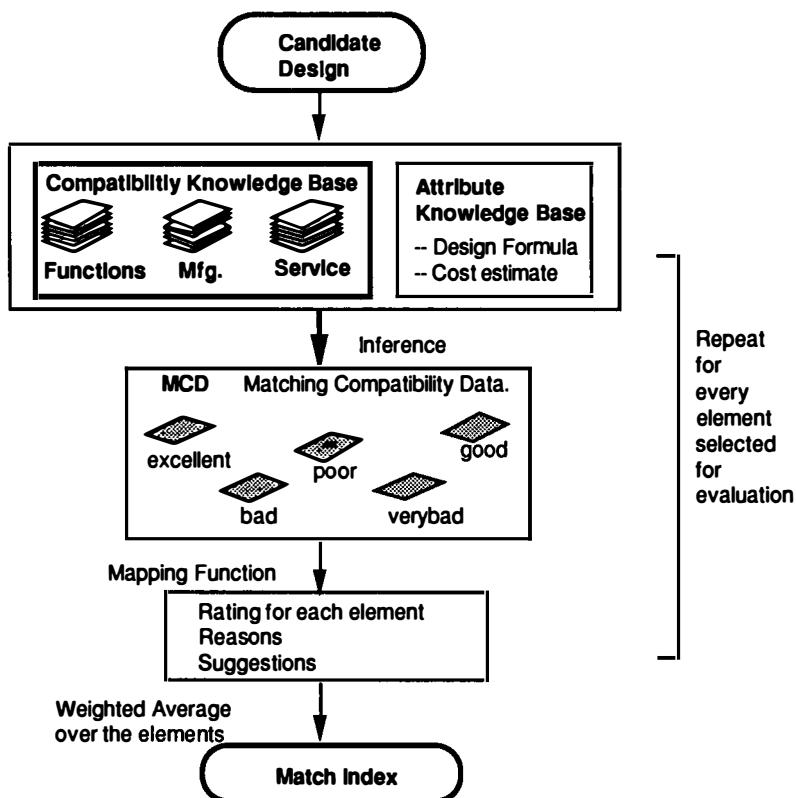


Figure 2. Computation of the Match Index

The total evaluation for the entire design, the **match index**, is a weighted average of the compatibility score for all the design elements under consideration. In addition to the compatibility rating, DAISIE provides reasons for the score and suggestions for improvement.

3. INITIAL FIELD TEST OF THE AI-BASED TOOLS

3.1 The design for assembly expert system

In 1987, DCA was used in a graduate course at Stanford to expose engineering students to the concept of design for assembly. Designers at Apple also tested the program. The initial computer program ran on a Sun III workstation in LISP. It supported only a primitive textual interface (Ishii, 1987). This system contained DFA knowledge available in textbooks and published papers only. Users described the candidate design to the program by answering a series of textual questions. DAISIE reviewed this input and gave a design rating relative to assembly efficiency.

Students who used this program commented that it was a good tool from which to learn the fundamental design guidelines for assembly. Rather than reading a thick manual that contained hundreds of design rules, students could use this software to learn the concepts from examples.

Designers at Apple were far less enthusiastic about this expert system than the students. In fact, their criticism was overwhelming. The system was too slow, too tedious, and too boring. The knowledge was old. The entry of the design was too laborious and the design ratings were poorly defined. The designers did not like having what they considered to be a computerized boss who checked their work, criticized it, and rated it. They wanted a friendly advisor which would be as easy to use as the spell checker on their word processors. They did not want to spend hours describing their design to the computer in geometric or other obscure languages. They wanted to have the design reviewed visually and comments and suggestions made simply and graphically. The designers also stated that after using the tool once or twice, they knew the fundamental rules and did not need to be continuously reminded of them. Consequently, they did not want to use the program on an ongoing basis.

Managers at Apple saw this tool as too complex. It would take a long time to train their designers to use the system and it would be too difficult to maintain. They saw it as a program which designers might use once, but never twice, since the knowledge they gained the first time would be retained for their future work. Managers considered the return on investment for this tool to be poor.

With these comments in mind, the Stanford group developed a Macintosh-based DFA program that was customizable, graphical, and fast. They designed it so that companies

could continuously upgrade the program with their own internal DFA knowledge. The program, called DAISIE / Assembly (Ishii et al, 1988; Adler and Ishii, 1989), utilized HyperCard for most user inputs and Prolog for the reasoning process. The interface was redesigned so that the user could describe the component configuration and assembly sequence through a graphical sketchpad called Linker, which is shown in Figure 3. Linker uses a palette of icons to represent common design elements. This initial graphical program led to the development of a faster, more refined program called Assembly View.

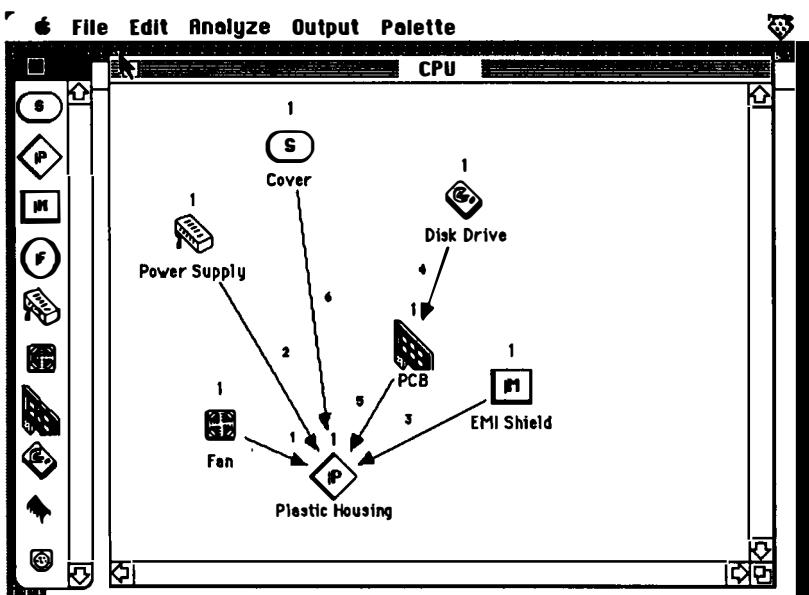


Figure 3. The "Linker" graphical interface

The designers at Apple were impressed with the graphical DFA program. They found that they could enter and evaluate complex designs in less than two hours. The system was easy to use and understand. They also found it was an ideal forum for discussion of the assembly process with the other project designers and with the manufacturing groups. The icon type graphics enabled the designers to describe their design quickly to the factory people without risking exposure of confidential design details. The factory people used the tool to offer alternative assembly suggestions.

The major limitation of the new expert system was a subtle commercial, rather than technical, problem; it was one of timing. When Apple commissioned the development of an expert system for DFA at Stanford in Fall, 1987, there was a major push within the company for the designers to decrease assembly cost. By the time the expert system was ready, the majority of the product designers at Apple had learned the fundamental principles of design for assembly and, in fact, had become experts themselves in this area and did not need or want the help of a computer program to develop their parts.

This awareness of the timing issue triggered a lengthy discussion between the authors on the fundamental nature of experts systems and their use in design. One of our main conclusions from this discussion was born out of our initial frustration with this project. We realized we had spent too much time developing a system whose knowledge base was limited in extent and usefulness. We concluded that one should pick a knowledge area for an expert system which was fairly broad, non-trivial, and needed by a large group of people. We also realized that our system should be tested early in the development stage. In other words, the expert system must have a large customer base over a long period of time.

Our second conclusion was that the knowledge base must be user accessible, so that new knowledge could be easily added to the system. Otherwise the knowledge would quickly become outdated once everyone had learned the basics contained in the initial system.

Our third conclusion was that the interest in using an expert system increased in proportion to the value of the knowledge. The more useful, complex, and obscure the information, the more designers would want access to it and the more patient they would be in working with a computerized expert. An expert system on the design of screws, for instance, would help a designer but this knowledge is readily available in textbooks and pamphlets; a designer might not feel the need for a computer to help him/her make decisions in this area. Almost any engineer can become an expert in this area in a few months. However, an expert system in the design of die cast parts would be very valuable since it takes several years to obtain experience in this area, and there are few options to acquiring this knowledge other than from an existing expert.

3.2 Expert system for the design of injection molded parts.

Having learned several valuable lessons from our experiences with the assembly system, the authors decided to develop an expert system which was needed by a larger group of people, one that could be upgraded by the user and one that contained complex information not readily accessible to a designer. We chose to concentrate our efforts on an expert system to aid in the design of plastic parts for injection molding as this was knowledge that all the product designers at Apple needed and had a difficult time acquiring on their own.

The design of plastic parts for injection molding is a very complex process. The training of a good plastic part designer is typically done through on-the-job training by trial and error, a very embarrassing way to learn a skill. There are only a handful of schools in USA which offer undergraduate training in this area and there are a limited number of textbooks which cover this field. Experts in these areas are few and they often disagree about the rules of good design. Good injection molded part design requires knowledge not only of form and function of the part but also of the processing of the material and the design of the tools to make the parts. Often, the needs of part function, processing and

tooling are in conflict. Good design requires the optimization of all three of these aspects and consequently takes years of training and experience to learn. The challenge we tackled in this project was to acquire the knowledge of the three experts involved in injection molded part design (the functional designer, the injection molder, and the tool designer) and to evaluate designs relative to their compatibility with each of these experts' knowledge.

To design this expert system, we once again chose DCA for the heart of the program as this was the ideal tool for evaluating design compatibility among several experts. We also developed a graphic interface (Figure 4) for the designer to enter the key elements of the candidate part design into the system. We acquired our plastic part design and manufacturing knowledge from published textbooks and processing literature. Figure 5 is an example of a compatibility rule for this system which is documented in a HyperCard knowledge maintenance tool. By 1989, the program was assimilated at Ohio State (Ishii, Hornberger, and Liou, 1989). It ran on a Macintosh II and used a HyperCard interface which contained knowledge cards that could be easily altered and upgraded with new information. After initial testing and debugging by the students at Ohio State, the new design tool was sent to the designers at Apple for a test drive.

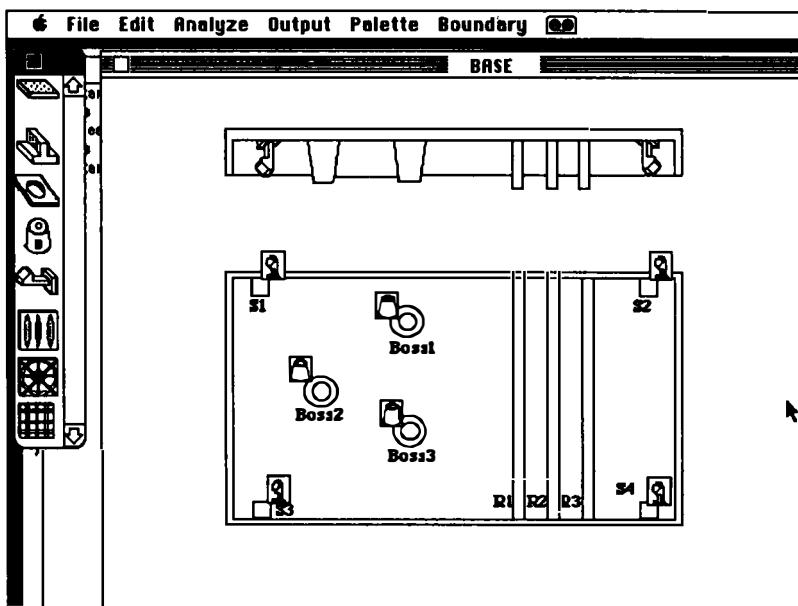


Figure 4. Sketching input for DAISIE / Plastics

The product designers and injection molding experts at Apple were enthusiastic about the potential of the new system. They liked the information potential of the new product and the subject matter it covered. However, once again, they were very critical in their evaluation of the computer program.

Top on their list of complaints was the media available for entering their designs in to the evaluation system. They did not like entering their designs through the simplistic sketch pad tool. The tool was okay for simple trial parts but it was too slow and limited for the complex parts they were designing daily. They did not want to redraw the designs they had already done in detail on the CAD system. They wanted to simply push a magic button on the CAD system and have their files sent to the plastic design expert system for evaluation and repair.

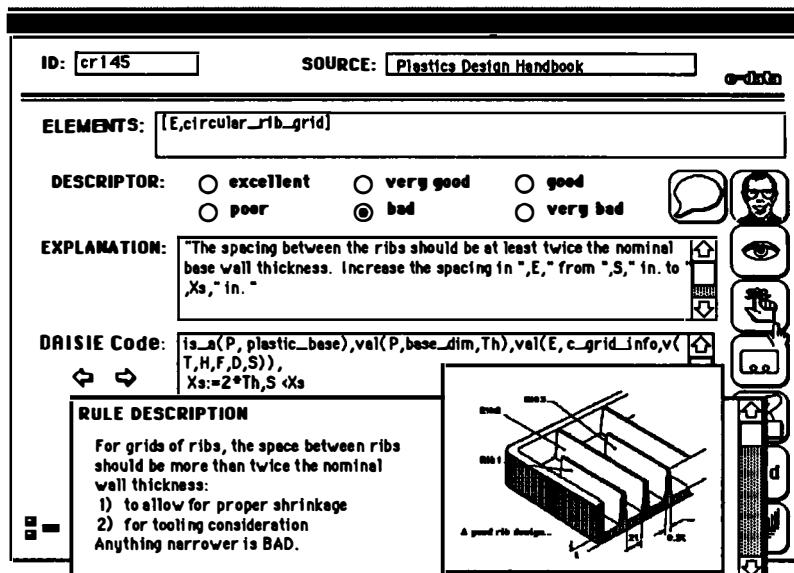


Figure 5. Knowledge maintenance module

Although the authors recognized that CAD system integration was possible, we questioned the value of such an interface feature. We thought that development of a CAD interface might turn into an extensive and fruitless task as it would require years to develop and may not lead us any closer to optimizing the use of expert systems in design. In particular, we realized that, by the time the designers have CAD representations of their parts, designers are less receptive to major change. Further, design managers at Apple had doubts about the value of such a push button checker. They felt that this approach would allow dumb and uncreative computers rather than experienced and adaptable designers to take responsibility for checking and correcting the design. They foresaw the potential for a "1984" type of design where designers manipulated a limited number of design elements to make a part but did not create a design. They speculated that in time all the designs would look the same because the computer would always make the same decisions and corrections. All parts would have the same wall thicknesses and tapers for example. Ideally designs should always be changing and improving as the designers learn and grow and find new ways to use their imagination. Design managers predicted that an unguided expert system would be a menace rather than an aid to good design.

A second, and perhaps more important, complaint about our plastic design system was that the knowledge included in the system was too general and too common. Everyone but the novice already knew the knowledge we had gathered. This comment set us thinking in new directions. Since, it is the nature of human beings to grow and learn, it seemed that no expert system could acquire knowledge as quickly as a human being. Therefore, it seemed unrealistic to us that our expert system could be a constant companion for a single designer. Rather, it appeared obvious that the designer would use this system until he or she was smarter than it was and then discard it. This realization made us refocus our design efforts.

4. DESIGNING A DESIGNER'S TOOL

With so many negative comments from our field tests of our expert system, we began to wonder if we could ever make the right tool. As we retrenched and pondered our design approach to developing an expert system, we realized that we had been trapped in a rather ironic situation. In our enthusiasm to give the designers a design expert , we had failed to use even the basic rules of good design ourselves. The multiple complaints of our customers were related to the fact that we had failed to clearly define our product, our customer, our market window and product life before we began our design of the system. The design expert system itself had been a victim of poor and incompatible design rules and goals. With these design lessons rudely brought home to us, we began our development again by correcting our own design mistakes.

4.1 Defining the product

First, we decided to define the purpose or mission of our product, an AI-based tool for life-cycle design. There were two important issues that we needed to resolve in order to define our product: its look and its function. The functions of an expert system can be many. It can be a database for designers, a design checker, a trainer for new designers, or a communication device between manufacturing and design within the company. Different functions require different interfaces and protocols. In retrospect, this clarification of purpose in the early stages of our design would have focused us much sooner and saved us many valuable development hours. We initially tried to make one system fit all the needs and ended up with a slow and unfriendly system that few people had the patience to use. To help us choose a function for our expert system, we reexamined our options.

1) The expert system as a continuously updated design data-base

At the least, this tool can serve as an updated check list on a specific area of design knowledge for designers to consult. In a dynamic environment, this tool could be an interactive information bank within a company which could be a source of stimulation, arguments, and creativity for the designers and manufacturing experts. To build this base, each expert at the company puts in their knowledge about good design as they acquire it and designers access this information as they need it. Such a system requires a dedicated maintenance staff to keep it functional. It requires an easy means of adding and accessing

knowledge. To be a useful tool in the long run, new knowledge must be continuously added to the system or the system will outgrow its value as the designers become experienced in their craft.

2) The expert system as a design checker

A fixed data base, however, would be less troublesome to maintain and could be a design checker within the company. It could contain an extensive amount of knowledge about design failures and successes generated from the company's experience with their products. Presently, this knowledge is not transferred within most companies from one design team to another or from an experienced designer to an unexperienced one. So, the same mistakes are made over and over and must be corrected over and over. As assemblers often note, "those engineers never learn." Company knowledge of this type could be put into an expert system which checks the design for common errors and corrects them. To be effective, this system must be conscientiously used by the designers, and the data in these systems must be occasionally reviewed so that the knowledge does not become outdated. The life of this type of system would be limited by the rate at which the average designer acquires the company's knowledge and experience.

3) The expert system as a company communication tool

A more direct and dynamic way of getting the manufacturers, assemblers, service people and checkers to teach the designers about common mistakes is to use the expert system as a quiet PA system or bulletin board among the groups. It could serve as a communication device between the designers and all the interested departments. In this type of system, every interested party lists their constraints and concerns for the product, and the designer tries to design within these limits. With this type of tool, the designer could document his/her decision making process relative to these historical situations. For most high paced companies, this type of tool would allow them to track and learn from their past designs in a more effective and balanced manner than they do now. Along with its documentation function, the tool could also facilitate technology transfer from technical experts to designers. New knowledge about advanced materials and processes, for instance, could be sent to the "bulletin board". Consistent with the communication role of AI-based tools, a computer-based system could be used to expose designers, manufacturers, and other experts to the changing customer preference or company specific interests.

4) The expert system as a training tool

With the accumulation of vast amounts of design information in the data base, an AI tool could serve as a powerful teacher of new skills. People who would benefit from such knowledge include engineering students, new employees, engineers with limited product design backgrounds, and experienced designers who would like to update their skills. Typically, new designers learn on the job by trial and error. It is presumed that they have the knowledge they need until they demonstrate their ignorance by making a mistake. With the aid of an AI tool, knowledge which is considered basic to the company's design knowledge could be presented to each new employee in a training session. Presenting the company's design knowledge in the form of an AI-based training session, where designer's

can learn by examples, avoids the consequences of unwise presumptions of knowledge for the employee and for the employer. In addition it exposes the new employee to the experience base of the company. An interactive learn-by-example training session is certainly more appealing than reading volumes of engineering manuals. The knowledge base for a training tool must be accurate but does not have to be updated as frequently as the other data-based design tools.

Since the design and development time for our AI tools had already been extensive, and since this development was a lengthy experience for us, we decided to choose a function for our new prototypes which would be valuable throughout this long term effort. In our initial experiences, the design rules were changing while we were developing and showing the rule base to our client, the design engineer. This time, we decided to design the expert system for training as we felt that the knowledge was less volatile in this mode than in the others. It is for this function that we are presently designing our systems.

A complement to choosing the function of the design was determining the look or appearance of the tool. Once the function was defined, the choice for appearance was obvious since training tools by their nature must be easy to use, have simple interfaces and some challenging and fun options to keep the interest of the user. As all experienced teachers know, a good training tool presents the basic information clearly and simply and then adds to this understanding through a series of increasingly more complex experiences. We decided that the "look of our system must be friendly and simplistic. This meant the main interface, the screen should be clean and uncluttered. Use directions must be obvious and the purpose of icons and symbols intuitive,

4.2 Knowing the customer

Many of our early design problems could have been avoided had we followed some simple marketing rules. The first and most obvious marketing error we made was the failure to define the customer for our product. We had not decided if our customer was a senior or junior level engineer or designer, a student, or a non-technical person. We had not asked our customer what they needed or wanted? Initially, we thought we intuitively knew the needs of the customer, but our experiences proved us wrong. We learned that although the customers had a difficult time defining the system they wanted, but they were very articulate about what they did not want and this information could guide us to the design of their tool.

In particular, the designers told us that they did not have the time to struggle with the awkward problems inherent in development software. Some of the more subtle obstacles in introducing an AI tool are based on the computer literacy of the individual customer. Since the designers' main task is to design products (integrate product functions and requirements) they do not want to spend a lot of time learning how to run a computer program. They have no patience for buggy software, obscure commands, or tedious interactions. They also do not want to read lengthy explanations or suggestions, particularly in textual form. They want their information to be in their natural media:

pictures not text. They do not want to take longer to analyze a part than they did to design it. They need the information to be timely relative to their needs. For instance, they do not need specific manufacturing details at when they are focusing on general layout. This type of information would be distracting in the early stages of design, but could be very valuable at latter stages

With this customer information in mind, we are designing our present system with a simple graphical interface. The knowledge is displayed in picture form and can be accessed on request. The system will be fully debugged at Ohio State before our first Apple tests.

4.3 Market Window (Timing)

An error we made several times in our past work is not recognizing the limited interest designers have in certain areas of knowledge. Getting knowledge to designers at the right time is crucial for the success of an AI program. For instance, there was a significant interest in a DFA program at Apple when we began our study. However, by the time we delivered a tool, the designers were already quite knowledgeable in this area, so the value and uniqueness of the tool was not as obvious as initially expected. In contrast, a later product, the forging system (Maloney et al, 1989), took only one month to develop and was delivered while the interest and need for this information was still at its peak. We believe the knowledge about plastic design in our new system will be valuable for many years as this area is only beginning to expand.

4.4 Life-cycle value of an AI tool

Based on our initial experience, we have made some observation about the fundamental requirements for a computer program to age gracefully.

1) The mission of the tool over time: An AI tool should enable engineers to think more logically and to develop designs that are more creative. As the tool makes the designer aware of the need for compatibility with production and service constraints, it should not only catch mistakes but also enhance creativity, since the design team has more time to explore various design options rather than getting bogged down in time consuming redesign remedies. An ideal training tool would also teach the designer the underlying causes of the manufacturing constraints so that the designer can essentially walk in the shoes of the manufacturing and field engineers and make productive decisions to facilitate the needs of these people and retain this knowledge for use in future designs. More specifically, a good tool should empower design teams to search through a wide range of alternatives and solutions which meet the needs of all departments within the company.

2) Simplicity of use: The long term usefulness of an AI tool depends on its simplicity of use. Graphical input / output of information attracts potential users and promotes actual use. Friendly user interfaces and simple procedures for design evaluations make it easy for the designer to use the system. Without CAD linkage, many systems tend to demand the users to input excessive sequence of information about the proposed design. Keeping user

input to a minimum by using such methods as group technology and pictorial inputs have led to the simplicity of use of our injection molding and forging tools.

3) Customizability: The life of an AI system within a company depends on how easy it is to adapt the system to the needs of the environment. Typical users prefer to customize the capability of the AI-based tools to their own needs. For instance, there are certain aspects of a proposed design that a computer may flag, but the designer may be already aware of this incompatibility and plan to remedy them later. Some users do not want to be constantly reminded of trivial problems. The user should have control over the style in which the design knowledge is presented on proposed designs; the system must be sturdy enough to allow the user to alter the screening sequence for the design.

A good computer-tool for simultaneous engineering is analogous to a well designed word processor. Its principal requirements are customizability, ease of use, instantaneous feedback, adaptability. The system must also be easy to customize in terms of knowledge. As a company increases its knowledge base this information must be addable to the system by the average user.

5. OUR CURRENT PROJECT: HyperDAISIE

Reflection on the needs of our customer and our own interest led us to the modification of our AI-based tools. We focused our new efforts on creating a training tool for designers new to the field of plastic part design for injection molding. We envisioned our customer as an engineer who is new to the company or one who needs to quickly acquire the basics of this field and its vocabulary. These customers need to be able to experience the design process under the guidance of the expert tutor and learn the rules for good designs through trial and error attempts at small designs. Corrections should be made in the privacy of the students' office. Experts in the field should be able to easily add to the knowledge base of this system through simple data entry. We expect the life of the product to be roughly five years which is the approximate design life of a personnel computer and its associated software.

To achieve these design goals with our current system, we reformatted our software with a new application: HyperDAISIE, an AI-based design assessment tool. The main platform for the input / output of information is the Macintosh application HyperCard 2.0, a graphical programming interface. Apple's version of Prolog, called the logic manager (LM), is used to perform the intelligent reasoning required for DCA.

We designed HyperDAISIE to be friendly towards the users and developers. A developer can create a new design tool by simply working on a HyperCard front-end for the graphical input and output. The programing task involved in creating a new interface is kept to a very elementary level by the use of HyperTalk scripting language. The ease of development of this platform addresses many of the requirements identified in our previous section: rapid prototyping, customizability, flexibility, etc.

5.1 Design for injection molding (DFIM)

Ohio State is currently addressing the key issues identified in section 4 by revising the design for injection molding program with the new framework, HyperDAISIE. Initial customer response from our engineering students has been noticeably improved. Figures 6 and 7 show samples of the graphical screens of the new program. In addition to reminding designers of conventional design rules, the new system accommodates various alternatives in materials and processing methods that are appropriate for a given functional need.

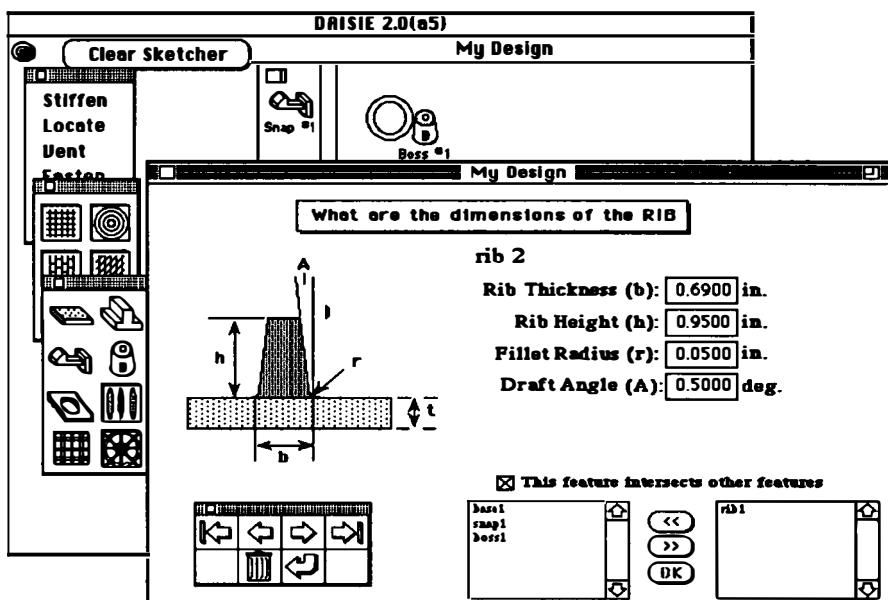


Figure 6. HyperDAISIE / Plastics

One of the new feature of the injection molding program allows the designers to scan through different alternatives based on their “design intent.” By allowing the designers to step back to their intent, we can prevent them from confining their concepts to preconceived geometry, process, or materials. We are also incorporating new and non-trivial design rules such as the relationship between geometry and dimensional accuracy of the injection molding process. In conjunction with the process selection program described in the next section, the injection molding program critically addresses the “key essence” of success as identified in the previous section:

- 1) Design knowledge that addresses dynamically advancing technology
- 2) AI tool that allows designers to search for truly creative and new solution
- 3) A program that acts as a communication bulletin board within the company.

After further testing at Ohio State, we intend to expose the new program to engineers at Apple Computer and other industrial sites, primarily for training of design methods with conventional and new plastic processing techniques.

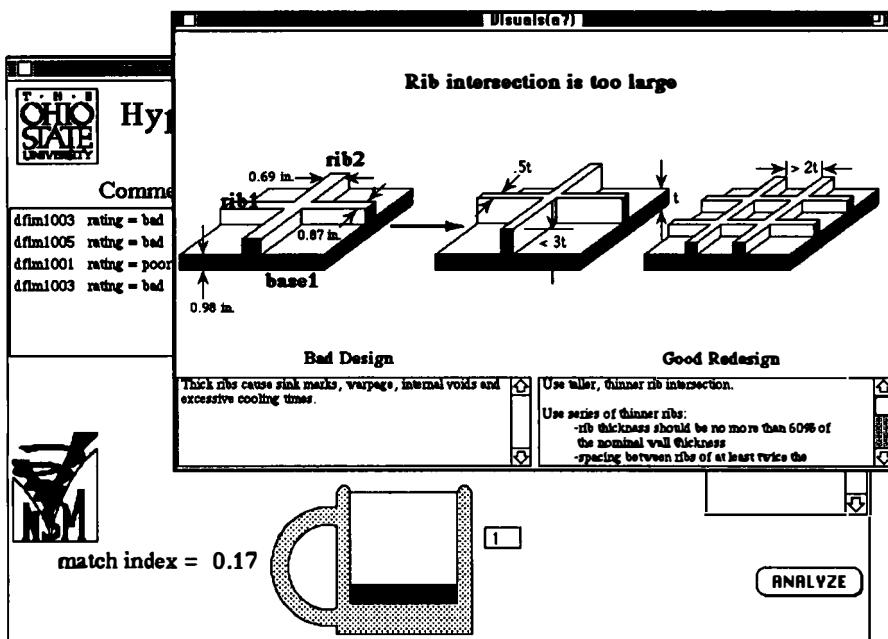


Figure 7. DCA Evaluation for HyperDAISIE / Plastics

5.2 Program for Process Selection

HyperDAISIE / COFES (Ishii, Maloney and Miller, 1990) is a system for which we solicited the voice of potential users early in its development. The program helps ensure compatibility between the forging part design, process plan, and process capabilities of the forging facility. This application was requested by the US ARMY material command because of its need to evaluate capabilities of, and best utilize, its suppliers. For this program, we had a different type of customer. The ARMY needed both a training tool as well as a supplier screening tool. The system contains a rich library of information on forging processes, making the software a perfect tool for training. The system adopts group technology codes in characterizing part shapes, which enables the user to quickly assess a candidate supplier.

This system is an example in which beta testing of the program contributed significantly to the utility of the resulting software. Forging experts were consulted early in the development process. Their feedback had a significant impact on the detailed design of the software. Figures 8 and 9 are sample screens from the software.

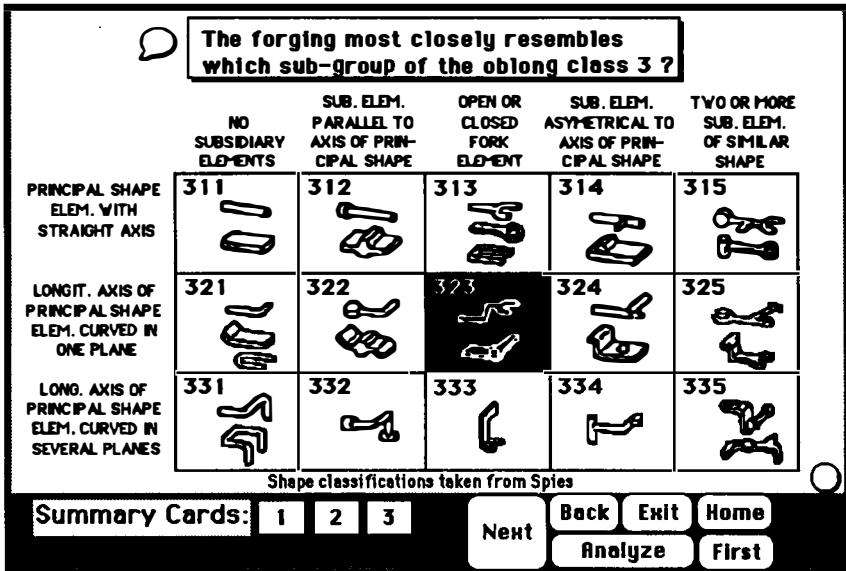


Figure 8. HyperDAISIE / COFES: User Interface

In COFES, design aid for the forging process, DAISIE asks the user to classify the candidate design using the SPIES chart as shown above.

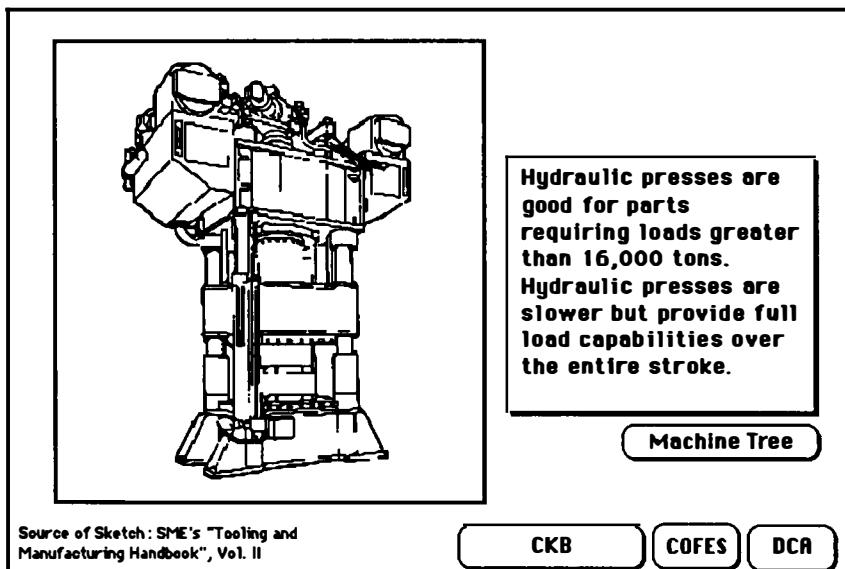


Figure 9. Visual suggestions in COFES

After giving an overall rating, COFES displays a pictorial suggestion. In this case, the system is recommending four column machines rather than the two-column currently proposed by the user. DAISIE accommodates machine characteristics, qualitative and quantitative, of machines available to the user.

We are now using this program as a centerpiece for a more general software to evaluate the compatibility of a candidate design with various manufacturing processes (Ishii, Lee, and Miller, 1990). The graphical interface will aid in communicating a wide variety of processes available to an engineer who often does not have time to explore alternative manufacturing methods.

6. CONCLUSIONS AND FUTURE WORK

This paper described our experiences with the development of AI-based tools for life-cycle design. In particular, we focused on the key requirements for a program to be successful in a practical design environment.

In short, we realized that an AI tool for life-cycle design is a product itself. As such, it warrants careful marketing effort which include identifications of its roles and values, customer requirements, and timing for product development.

We learned that it is difficult, if not impossible, to build a machine that can change and learn as rapidly as human beings. Designers are impatient to learn and gain knowledge that will help them but do not want to hear old redundant rules and myths. They want a framework for their information. They want to know the reasons and justification behind the rule or guideline so they can make future decisions on their own.

In light of the many painful learning experiences relative to product design, marketing and product life, we believe that it is crucial to find a means of rapidly prototyping the AI tool so that it can be tested in its early concept stages. The initial test program should be simple, and its knowledge base be fairly small but contain the main features intended in the design tool. The prototyping and field tests stage can even be done without computers. Storyboard testing of the concept which we call Hollywood-set testing can be effective in acquiring customer input early. A more sophisticated prototype of the AI tool can be introduced to the engineers in its core form. That is, a program with the basic system interface, evaluation technique and suggestions can be shown to the design engineer when the program has a few understandable pieces of data. It is possible to start small and build up the knowledge base through actual use. As such, the base system should be designed to accommodate incremental developments and have features that make it easy for users to add new information. A good knowledge maintenance tool is essential.

Our work in AI has taken a new direction since our market testing experience and we encourage the other AI developers to spend some time in this process. Market testing is a painful, but, as all marketeers would tell you, an extremely enlightening process.

ACKNOWLEDGEMENTS

This work is a result of support from many organizations. The primary funding for this work was provided by Apple Computer's External Research Department. The experimental work was supported by Apple's Product Design Department and the NSF / ERC for Net Shape Manufacturing at Ohio State. The authors would particularly like to thank Barbara Bowen of the External Research Department for her help in exposing their work to other researchers throughout the country and Steve Weyer of Apple's Advanced Technology Group who gave the developers the powerful inference manager tool for their program. The theoretical development of compatibility-based simultaneous engineering was developed through NSF grant DMC8810824. Sapphire Design Systems, Inc. provided us technical support needed to customize DAISIE for DFIM. The authors would like to sincerely thank their continuing support. Special thanks to Kurt Beiter who carefully edited the manuscript.

REFERENCES

- Adler, R.E. and Ishii, K. (1988) DAISIE: Designer's aid for simultaneous engineering. *ASME Computers in Engineering 1989*, Anaheim, California, July, 1989.
- Agogino, A. and Almgren, A. (1987) Symbolic computation in computer-aided optimal design. in John Gero, ed. *Expert systems in Computer-aided Design*. North-Holland. pp. 267-285.
- Barkan, P. (1988) Simultaneous Engineering: AI Helps Balance Production and Bottom Lines *Design News*, March 1988.
- Boothroyd, G. and Dewhurst, P. (1983) Design for Assembly: a designer's handbook. Boothroyd Dewhurst Inc., Wakerfield, Rhode Island.
- Cutkosky, M.R., Tanenbaum, J.M., and Muller, D. (1988) Features in process-based design. *ASME Computers in Engineering 1988*, Vol. 1., pp. 557-562.
- Desa, S. et. al. (1989) Design for producibility... *Proc. of the 1989 ASME Winter Annual Meeting: Concurrent Product and Process Design*. Dec. 1989, San Francisco, CA.
- Dixon, J.R., et al. (1986) Designing with features : creating and using a features database for evaluation of manufacturability of castings. *ASME Computers in Engineering 1986*, Vol. 1, pp. 285-292.
- Dixon, J.R., et al. (1986) Dominic I: Progress towards domain independence in design by iterative redesign. *ASME Computers in Engineering 1986*, Chicago, IL. pp. 199-207.
- Duffey, M.R. and Dixon, J.R. (1988) Automating the design of extrusions: a case study in geometric and topological reasoning for mechanical design. *ASME Computers in Engineering 1988*, Vol. 1, pp. 505-511.
- Ishii, K. and Barkan, P. (1987) Design Compatibility Analysis: a framework for expert systems in mechanical system design. *ASME Computers in Engineering 1987*. Vol. 1, pp. 95-102.

- Ishii, K., Adler, R. and Barkan, P. (1988). Application of Design Compatibility Analysis to Simultaneous Engineering. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing (AI EDAM)*, Academic Press. 2(1), pp. 53-65.
- Ishii, K., Hornberger, L., and Liou, M. (1989) Compatibility-based design for injection molding. *Proc. of the 1989 ASME Winter Annual Meeting: Concurrent Product and Process Design*. Dec. 1989, San Francisco, CA.
- Ishii, K., Maloney, L. and Miller, R.A. (1990) Compatibility-based concurrent engineering for forging. *Artificial Intelligence in Engineering: Manufacturing* (G. Rzevski, ed.) Computational Mechanics Institute. Proc. of the 5th Int. Conference on AI Applications in Engineering, July, 1990, Boston. pp. 343-360.
- Ishii, K., Lee, C. H., and Miller, R. A. (1990) Methods for process selection in design. *Proc. of the ASME Design Theory and Methodology Conference*, September, 1990, Chicago, IL. Vol. DE-27. pp. 105-112.
- Kamal, S.Z., Karandikar, H.M., Mistree, F., and Muster, D. Knowledge representation for discipline-independent decision making. in John Gero, ed. *Expert systems in Computer-aided Design*. North-Holland. pp. 289-319.
- Liou, S. Y. and Miller, R.A. (1990) Design for die casting. *Int. J. of Computer Integrated Manufacturing*. To appear.
- Makino, A., Barkan, P., Reynolds, L., and Pfaff, E. (1989) Design for serviceability expert system. *Proc. of the 1989 ASME Winter Annual Meeting: Concurrent Product and Process Design*. Dec. 1989, San Francisco, CA.
- Maloney, L., Ishii, K., and Miller, R.A. (1989) Compatibility-based selection of forging machines and processes. *Proc. of the 1989 ASME Winter Annual Meeting: Concurrent Product and Process Design*. Dec. 1989, San Francisco, CA.
- Nielsen, E.H., Dixon, J. R., and Simmons, M.K. (1986) GERES: A knowledge-based material selection program for injection molded resins. *ASME Computers in Engineering 1986*, Chicago, IL, July, 1986. pp. 255-263.
- Poli, C, Graves, J. and Sunderland, J.E. (1988a) Computer-aided product design for economical manufacture. Proc. of the ASME Computers in Engineering Conference, Vol.1, pp 23-27, San Francisco, July, 1988.
- Poli, C and Fernandez, R. (1988b) How part design affects injection molding tool costs. *Machine Design*, November, 24. pp. 101-104.
- Rinderle, J. and Balasubramaniam. (1990) Automated modeling to support design. *Proc. of the ASME Design Theory and Methodology Conference*, September, 1990, Chicago, IL. Vol. DE-27. pp.281-291.
- Shah, J.J. and Rogers, M.T. (1988) Feature based modeling shell : design and implementation. *ASME Computers in Engineering 1988*, Vol. 1, pp. 255-261.
- Simmons, M.K. and Dixon, J.R. (1985) Expert systems in a CAD environment : injection molding part design as an example. *ASME Computers in Engineering 1985*.

The Castlemaine Project: development of an AI-based design support system

P. Buck,^{*} B. Clarke,^{*} G. Lloyd,^{*} K. Poulter,^{*} T. Smithers,[†]
M. X. Tang,[†] N. Tomes,[†] C. Floyd[§] and E. Hodgkin[§]

^{*}Logica Cambridge Ltd
104 Hills Rd, Cambridge CB2 1LQ UK

[†]Department of Artificial Intelligence
University of Edinburgh, Edinburgh EH1 2QL UK

[§]British Bio-technology Ltd
Brook House, Watlington Road, Oxford OX4 5LY UK

Abstract. The development of effective computer support for designers needs to be driven by better modelling of the design process, and an effective way of developing such an understanding is by building Artificial Intelligence systems through which this process can be modelled. This methodology has been adopted by the Castlemaine project, a two and a half year industrial and academic collaborative research project investigating the application of AI techniques to the support of the design process. In the first phase of the project, a prototype design support system integrating various knowledge based system techniques to aid molecular drug designers has been constructed which is to further test the "exploration-based model of design" developed at Edinburgh University. This position paper describes the background to the Castlemaine project, the research programme and the status of the project at March 1991.

INTRODUCTION

Research work at Edinburgh University has demonstrated that the development of effective computer-based support for designers needs to be driven by better modelling of the design process and that an effective way of developing such an understanding is by building artificially intelligent systems which can be used to evaluate the knowledge processes involved in design (Smithers et al 1989). This methodology has been adopted in formulating *Castlemaine*, an Information Engineering Directorate (IED) project¹. This

¹ Castlemaine is one of the industrial and academic collaborative projects set up under the Department of Trade and Industry's Information Engineering Directorate initiative, the UK government directed information technology research programme which followed on from the Alvey programme.

project proposes to evaluate and develop the domain independent model of the design process and an associated knowledge-based design support system with an architecture which attempts to embody some of this model. The applicability of the "exploration-based model of design" developed by the Artificial Intelligence department at Edinburgh University will be tested in two ways. Firstly, by constructing a design support system integrating various knowledge-based systems to support the design activities of molecular drug designers. Secondly, by using a design case study from another design domain against which the design model can be evaluated.

In order to achieve the objectives set, the project participants are required collectively to have an appreciation of the common design processes, to be expert in the fields of molecular design, to be capable of applying knowledge-based system techniques and, preferably, to be a user of computer-based molecular design tools. Given these requirements, the Castlemaine project is a collaboration of four partners:

- Logica Cambridge Ltd, with experience of developing conventional software and knowledge-based systems;
- the Department of Artificial Intelligence at the University of Edinburgh, where a domain-independent understanding of the design process is currently being developed;
- British Bio-technology Ltd, bringing a knowledge of molecular design to the project and who are users of computer-based molecular modelling systems;
- CamAxs Ltd, a company with extensive database experience.

The Castlemaine project spans two and a half years and is organised around the construction of two prototype systems. The first of these, the preliminary prototype to be built in the first half of the project, is to assess the applicability of the exploration-based model of design developed by Edinburgh University, to the domain of molecular design. Evaluation of the preliminary prototype will support and inform the definition of the architecture of the main prototype which will be constructed in the latter half of the project. Towards the end of the project, a case study in software design will be selected against which the design model and main prototype can be assessed. Presently, the project is completing the integration of the major components of the preliminary prototype.

The following sections present early project results. The domain of indirect drug design is described, the knowledge acquisition and representation techniques are presented, and an outline of the architecture of the preliminary prototype drug design support system is given.

THE DOMAIN: INDIRECT DRUG DESIGN

Towards rational drug discovery

Knowledge elicitation sessions with British Bio-technology Ltd have emphasised the division of rational drug design into the two conceptually different approaches of *direct drug design* and *indirect drug design*. These are located in the hierarchy of drug discovery approaches in Figure 1.

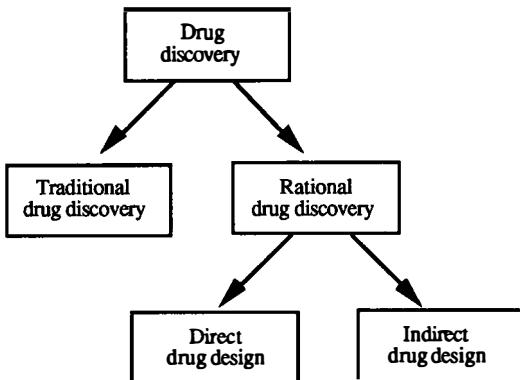


Figure 1. Hierarchy of drug discovery approaches

Traditional drug discovery has relied upon the highly serendipitous method of screening a large set of existing candidate substances for ones with the desired physiological effect. As pharmacological knowledge has increased, so the typical candidate set has grown to contain many thousands of potential molecules. The screening method, though successful in the past, is now increasingly seen as wasteful of research resources and also unlikely to produce the ideal drug.

Increasingly, the approach is to design the drug at the molecular level, based upon an understanding of the relationship between drug structure and drug activity. This approach can be termed rational drug discovery and is based upon the precept that the pharmacological activity of a drug is a direct consequence of its binding to the "target" receptor molecule. That is, when a drug binds to a receptor, some biological change takes place such as the opening of an ion-channel. Any small molecule, of whatever origin, which binds to a receptor is more generally termed a "ligand": we will restrict ourselves to talking about "drugs" for the sake of simplicity.

Drugs with a better "fit" to a receptor will bind more strongly and will thus have an improved activity. The popular metaphor of the *lock-and-key* is a useful one here, in which the drug "key" is specifically designed to fit and operate the receptor "lock" (Figure 2). Fit, however, is not purely a matter of geometry but is largely determined by the complementary distribution of chemical and electronic properties across the molecule and its receptor.

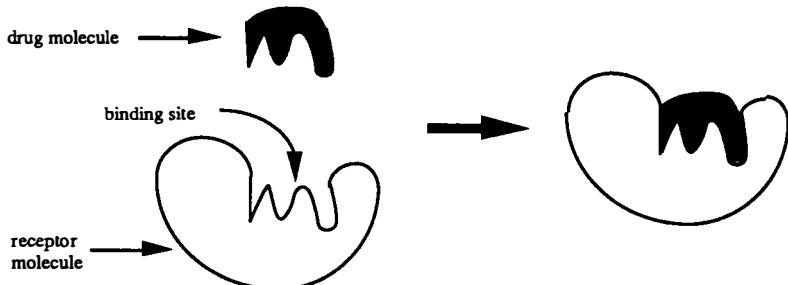


Figure 2. A drug fits into a target receptor's binding site

The requirements of the receptor's binding site, the part to which the drug specifically attaches, need to be modelled with a view to defining a drug molecule which can meet these requirements. When the nature of the binding site is known, the task of drug design involves the production of a complementary molecule.

If the molecular structure and geometry of the receptor is known, for example through X-ray crystallography, then the drug designer knows the receptor's requirements exactly and can use the direct approach in designing drugs for that site. Unfortunately, the geometry of most pharmacologically interesting receptors has not been characterised at an atomic level and the process of obtaining the molecular structure for a single receptor type may take many years.

Indirect drug design

More often, the designer has to take the indirect approach to drug design. The nature and size of a receptor binding-site can often only be inferred from the drugs that the receptor most readily accepts. This approach is analogous to attempting to infer the inside configuration of a lock from an examination of the keys which best fit it. Most molecules are flexible and it is not known which three-dimensional configuration any given molecule will take. This means that indirect approaches cannot directly model the geometry of the receptor. Obviously, the binding-site model produced from these techniques can only be an incomplete approximation.

The indirect drug-design approach models the "pharmacophore", an arrangement of chemical properties which are thought to be necessary for binding specifically to a particular receptor type. This model is derived from analysis of a set of drugs, all of which produce the desired biological activity to some degree and which are assumed to bind to the same receptor. This model is used to guide the design of new chemical molecules which target the same receptor type. The drug designed must be specific in that it binds to the target receptor but not to any other type of receptor: over-generalisation produces unwanted side-effects in the recipient of the drug. In indirect drug design, the structure of the receptor and its binding-site can be entirely unknown or perhaps only partially postulated in terms of important features; the effort here is concentrated on elucidating the features and functional groups of the drugs which contribute to high activity. Analyses of structure activity relationships (SAR) identify the salient structural features of the compounds contributing to the pharmacophore model, ranking these structures in terms of their importance for activity.

Conservative modifications of existing compounds are made with the goal of producing a novel drug with improved receptor fit and which shows high activity. The basic assumption of this approach is that properties which are related to binding are effective over a short range. This means that any changes made to modify a compound are local in extent and do not affect the behaviour of the molecule as a whole. There are additional critical factors which affect the way in which the drug is transported through and metabolised by the body, and these form part of the context for the novel compound's design. These pharmacokinetic factors are given increasing importance later on in the design project once an active compound has been successfully described and the focus of development moves to optimising the compound for activity in the mammalian body.

The Castlemaine consortium has chosen indirect drug design as the subdomain of molecular design to be focussed upon for the first prototype system. The reasons for this choice are:

- a. indirect drug design relies on substantial amounts of expert knowledge and experimental techniques. This is the kind of task where knowledge-based systems

have a valuable part to play. The Product Formulation Expert System (Skingle 1990) provides an exemplar of experientially-based synthesis (design) support;

- b. more than 95% of the drug design work carried out by pharmaceutical companies is indirect drug design. The popularity of indirect design is not the main reason for choosing it for support. However, it is naturally more comfortable to support a mainstream approach;
- c. direct drug design is essentially based around geometric manipulation of well-defined objects using well-defined data. This kind of task is one for which traditional computing techniques are more suitable than knowledge-based systems techniques, though "expert systems" for this approach have been postulated (Lewis and Dean 1989).

KNOWLEDGE ACQUISITION FOR DRUG DESIGN

Approach adopted

For the preliminary prototype analysis, a *formal* knowledge acquisition methodology has not been used on the Castlemaine project. The pattern has been to initially use semi-structured interviews to gather overviews of the drug design domain and to impose increasing structure on knowledge elicitation sessions at successive meetings between elicit and expert (an approach described by Shadbolt 1988).

A top-down approach was followed in the domain analysis for the preliminary prototype. As the knowledge engineers became more familiar with the domain of drug design and the objectives for the preliminary prototype were defined, more focussed knowledge acquisition techniques were progressively applied. The techniques used for the preliminary prototype fall into four main categories and were pursued in the following order:

- a. case studies
- b. task analysis
- c. concept structuring
- d. protocol analysis

Case studies

Analyses of previous drug design projects were carried out early in the course of the knowledge elicitation process. These sessions were mainly conducted in an unstructured fashion; the expert related the historical course of development of an actual past drug design project and the elicitors asked questions to clarify points or to elicit more detail on points of interest. This process provided a familiarisation for the knowledge engineers of the concepts, concept relationships and problem-solving strategies involved in the domain and demonstrated the difference between the two main categories of direct and indirect drug design. The overview of the drug design domain thus gained assisted the later decision by the Castlemaine consortium to focus effort on supporting indirect drug design. From the case studies, a set of domain concepts was compiled and a task analysis was then carried out.

Task analysis

The first formal task analysis that was conducted with the drug design experts at British Bio-technology Ltd was aimed at providing an analysis of the top-level tasks that take place in a drug design project, through from the formal instigation of a design project to the synthesis and optimisation of the novel drug. Subsequent task analyses concentrated on the lower-level tasks which had been chosen for support in the Castlemaine preliminary prototype. These knowledge elicitation sessions were structured using the dataflow diagram drawing tool MacCaddTM (Logica 1989). The MacCadd facilities allow the rapid development and modification of dataflow diagrams and also the ability to maintain a hierarchy of diagrams and subdiagrams though the "zooming in" facility (a diagram box can be selected to expand and display an underlying dataflow diagram). The format of the sessions was for an elicitor and an expert to jointly construct diagrams directly onto the computer screen. This interactive technique allows the expert to see his or her knowledge displayed in a structured form and to review and modify this knowledge during the session.

The set of task analysis MacCadd diagrams were reviewed and four main subtasks within indirect drug design were identified which appeared appropriate for support within the preliminary prototype system. These subtasks spanned the whole of the initial drug design process and were:

- a. *Analyse a compound.* Compounds are partitioned into pharmacologically meaningful building-blocks which form the basis of an analysis of functional properties.
- b. *Build a pharmacophore model.* Patterns of functional properties are identified across a selected group of compounds. The overall pattern is called the pharmacophore model and this is described in terms of a number of property features and the relationships between these features.
- c. *Explain the structural relationships within the pharmacophore model.* Structure-Activity Relationship (SAR) data is elucidated: the relationship between the activity of a compound and its structure is described in relation to the features within a particular pharmacophore.
- d. *Specify a novel compound.* The pharmacophore patterns and SAR data are used to specify a new drug or to suggest modifications that can be made to an existing drug to produce a novel drug with improved activity.

Figure 3 shows the breakdown of the indirect drug design task into these four main subtasks. For the preliminary prototype system, each subtask has a separate support system (rectangular boxes) which, in turn, is comprised of a number of independent knowledge sources (rounded boxes).

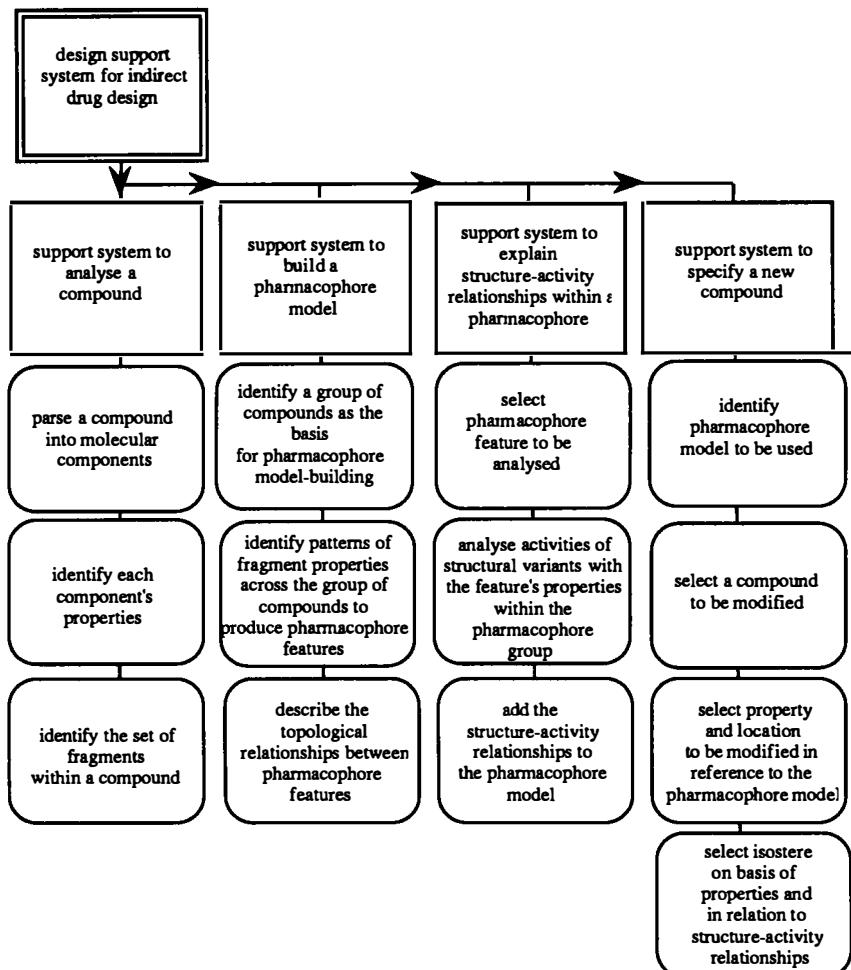


Figure 3. The support system and knowledge source hierarchy

Concept structuring

An integral part of recording the knowledge within the domain is the construction of a taxonomy of domain objects. The drug design experts were introduced to the frames system of *GoldWorks II™* by the elicitor. Using the set of drug design concepts produced from the earlier task analyses, a domain concept hierarchy was begun on the GoldWorks system. The format was for the elicitor and the expert to jointly construct the hierarchy directly in the GoldWorks environment. The elicitor "drove" the system whilst the expert provided the information needed. This interactive approach, similar to that used in the earlier task-analysis, enabled the expert to see the hierarchy as it was built up and to

TM GoldWorks is a trademark of Goldhill Computers Inc.

request modifications as necessary. For some of the more subtle concept distinctions, a version of card sort, the repertory grid (Williams and Holt 1988), was used with the aim of creating a "multi-dimensional" map of the elements of the domain.

The hierarchy thus constructed was intended to be a complete taxonomy of the domain. This complete hierarchy of concepts contains many frames that are not needed for the dynamic inferencing of the preliminary prototype system. There is, therefore, a second version of this concept hierarchy which is reduced to those objects essential for the knowledge-base of the preliminary prototype, with this implementation-specific version being derived from the full taxonomy.

Logica Cambridge Ltd. has developed a tool for the computer-aided acquisition of attributes in a domain as part of its *Rule Induction* programme. This tool guides the user to compare domain elements and to enter attributes for the domain, prompting him to fill in missing data. One possibility is that the results from this tool could be fed into a rule induction module which would generate classification rules for the domain elements. The tool will be evaluated for its role in knowledge acquisition in the main prototype.

Protocol analysis

Protocol analysis (Burton et al 1990) involves the expert solving a problem in front of the elicitor, and being asked to "think aloud" during this process. This technique was used to elicit more implicit knowledge involved in reasoning about chemical properties. Protocol analysis seemed the most natural way to proceed where the task was not one that is normally explicitly done by a drug designer and therefore requires a good deal of consideration and revision.

MODELLING REASONING AND DEFINING OBJECTS

The task analysis led on to modelling the decision processes involved in the drug design activities. This model is reflected in the design of the prototype system which supports the decisions which a drug designer would normally make. The early stages of drug design involve an analysis of the design problem's characteristics: that is, parameterising and constraining a model of the design problem. In pharmaceutical design, this analysis is largely evidence-driven and is based on a theoretical understanding of how chemical properties are distributed and how they contribute differentially to the binding of drug molecule to receptor. The medicinal chemist takes a variety of different views of a molecule focussing, for example, on the molecule's electronic properties, its size, on how different pieces of it respond to the presence of water. It was necessary to reflect these different views in the objects which the prototype design support system could reason with.

Firstly, a compound can be viewed as a number of constituent physical structures, each of which has homogeneous chemical characteristics which can be reasoned about unambiguously. For this project, these have been termed *molecular components*.

Secondly, a compound can be seen as delivering a number of chemical properties and these properties are distributed locally across the compound. Any physical structure may be involved in delivering more than one type of property, so there is not a one-to-one mapping between structures and chemical properties. The units delivering chemical properties within a compound have been termed *molecular fragments* and a fragment is made up of one or more molecular components. The fragments may, by this definition, overlap each other in structural terms. Both components and fragments are defined for individual molecules.

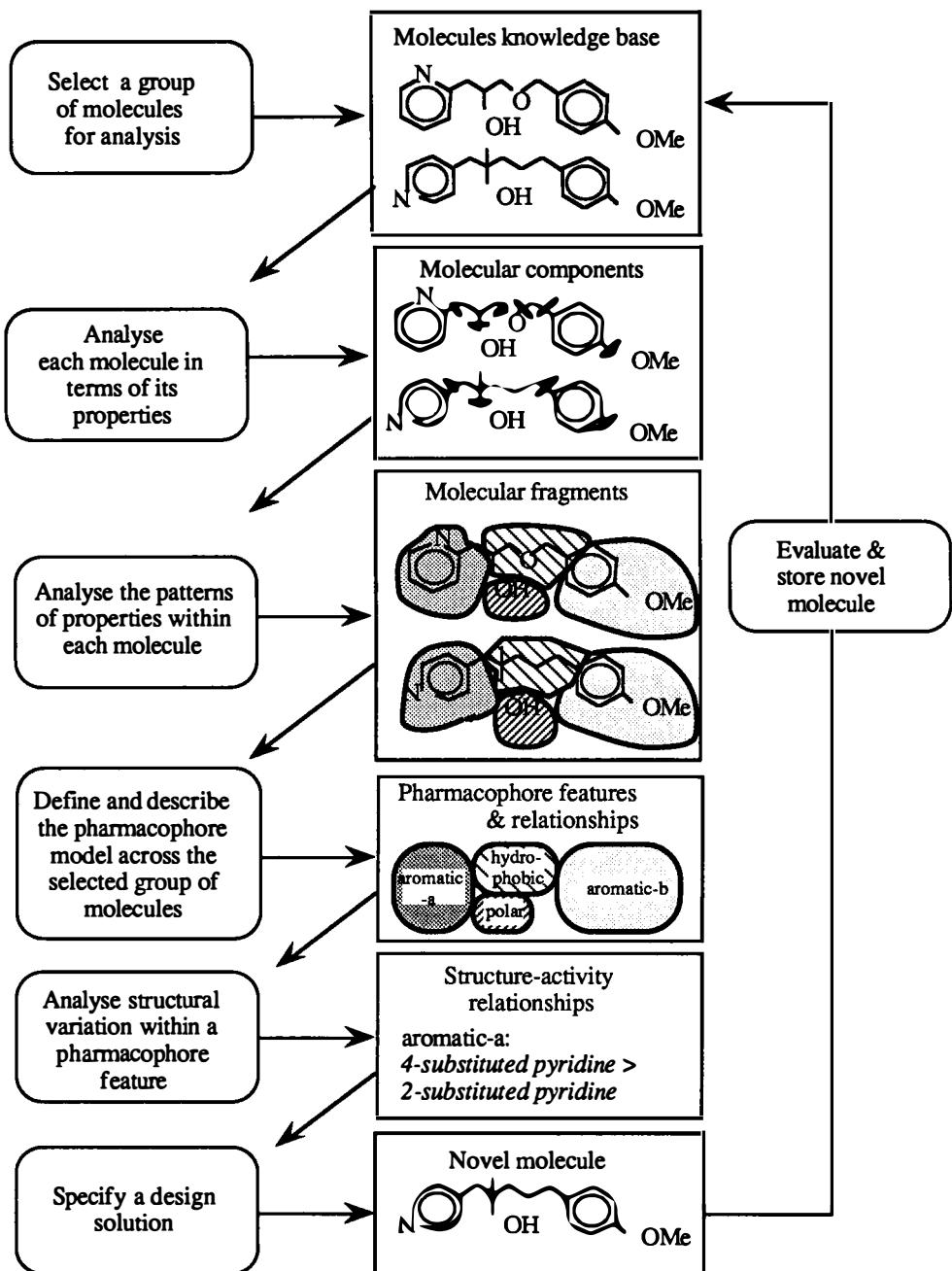


Figure 4. The Castlemaine model of indirect drug design

Two other objects are defined which relate to the modelling of the drug design problem. These are the *pharmacophore model* and the pharmacophore model's *features* and these relate to a group of compounds. The feature is defined by a pattern of chemical properties found across a set of compounds being analysed. As such, it is contextual and is abstracted from the physical structures of the molecule.

Figure 4 represents the main subtasks being supported (rounded boxes) and the representations which they operate on (rectangular boxes). In this diagram, physical structures are represented as molecular diagrams and property groupings are represented as shaded areas. The top-level feedback loop is given for simplicity but there are, of course, intermediate feedback loops in which aspects of problem analysis and synthesis are explored.

One of the major objectives of indirect drug design is the determination of the pharmacophore. The pharmacophore can be seen as a hypothesis about the arrangement of chemical properties required to achieve the desired biological activity. As such, it integrates across the patterns of properties found for individual compounds, identifying the common properties and the relationships between these properties. Each of the common subpatterns within this model is a *feature*. In terms of medicinal chemistry, a feature is an identifiable set of properties which work cooperatively on a local basis to affect the way in which the compound binds to its receptor.

The pharmacophore model is analysed in terms of the structural variation of the molecules which contribute to the definition of each of its features. The usefulness of any piece of structural information is related to how active the compound is in binding to the target receptor. This analysis of *structure-activity relationships* forms an important part of the problem definition which, in combination with the pharmacophore model, allows possible design solutions to be generated.

A pharmacophore model is described in terms of the *properties* (that is, the functions) delivered by each of its features. There are a number of ways in which variant structures can deliver the same set of properties. Design options are generated by specifying variants in molecular structure within each of the pharmacophore features, choosing these in such a way as to maximise the activity of the compound as a whole. A molecular structure which can be used to replace a different molecular structure, whilst retaining a given profile of properties, is called an *isostere*. A pharmacophore model in combination with its related structure-activity analysis can be used to determine plausible isosteric replacements for designing novel compounds.

DESIGN

Pharmaceutical drug design projects fall largely into the category of creative design, rather than being innovative or routine design (Gero 1987). The search is for compounds of significant and demonstrable novelty which will merit patents. This is not to say that each novel drug does something different from its predecessors: it may achieve the same physiological effect but it must use new science to achieve this effect in a creative way. If a design solution is not innovative, then it is not patentable. Design can be divided broadly into phases of formulation, synthesis and evaluation (Maher 1990) and much of the effort of indirect drug design is concentrated on the formulation phase. Most current research on design bypasses this early stage of formulation and considers design projects from synthesis onwards. Design formulation is consequently poorly understood and poorly modelled.

An artefact can be defined in terms of its function, behaviour and structure and Gero (1987) suggests that the interrelationships of function behaviour and structure are stored as schemata or prototypes which the designer calls upon. Gero and Roseman (1989) suggest that the functional description is used to derive the artefact's expected behaviours. These expected behaviours dictate which actual behaviours are derived from the structural description.

The functionality of a novel drug is stated in physiological terms: for example, *to lower blood pressure*. This physiological level of functional description cannot readily be used to provide a description of the expected behaviours for a drug molecule at the atomic level as there is no known set of correspondences between physiological function and the anatomy and behaviour of the receptor molecule. The functionality for the design project can be restated at a lower-level of description to make the design problem more tractable: *to produce biological activity at a particular level in a specified receptor*. But, this does not alter the basic fact that the expected behavioural attributes of the drug molecule cannot be derived, as the behaviour of the drug's receptor is not known.

The structural vocabulary of pharmaceutical design is straightforward: a molecule must be composed of atoms and these will largely be drawn from the organic subset of atoms. The laws of physics dictate how atoms combine into molecules and determine the behavioural properties of the resulting structures and substructures. In most other domains of design, there is a known mapping between behaviours and structures: in molecular design, the behaviours of individual atoms are contextual because of the complex interrelationships of physical forces. In medicinal chemistry, behaviour can be derived from structure only on a local basis. That is, given a small number of connected atoms, their chemical behaviour can be hypothesised with reasonable certainty as many physical properties are short-range and thus largely non-contextual. There are, however, longer-range properties which modulate local behaviours, reducing the certainty of the hypothesis about their relationship to structure. As the number of atoms under consideration increases so the certainty about behaviour decreases because of the complexity of the interrelated physical properties. In this respect, pharmaceutical design is quite unlike design in other domains such as architectural or engineering design where the relationships between behaviours and structures can be enumerated.

How well does indirect drug design fit well with a description of schema-based design? There is no pre-existing schema which can be used to inform the novel design because of the creative nature of the design and the lack of knowledge of the receptor's behaviours and so there is no means of deriving expected behaviours from the functional description. Instead, the expected behaviours must be generalised from the actual behaviours and structures of a number of molecules which have similar functionality. Only when the expected behaviours have been inferred can the design of a novel molecule proceed into the synthesis phase and these expected behaviours may have to be revised in the process of exploring design solutions. It might be possible to look at the process of modelling the pharmacophore as analogous the construction of a schema which is used to inform the specification of design instances: but as the model relates to a specific body of evidence, the notion of the pharmacophore as a schema lacks generality.

The starting point for a drug design project is usually some molecule which exhibits the desired functionality and which provides a lead. The molecule may not exhibit this functionality very strongly and it may, indeed, exhibit the inverse of the desired function: all that is required is some correlation with the desired functionality. Often several candidate molecules can be identified which show similar functionality or the single starting compound is used to generate structural variants which provide evidence for analysis. These molecules are solutions (of varying quality) to aspects of the design problem in hand and are the starting point for the new design. This suggests some similarity with case-based design (Zhao and Maher 1988, Maher 1990), where earlier design episodes are used to provide analogical solution processes or methods of exploration. A case-based model of

the design process has been applied by Maher to design synthesis but not to design formulation. Case-based reasoning relies upon having episodic information of previous similar designs which can provide a link from function to behaviours and structure. Episodic information of sufficient specificity is seldom available in relation to a particular receptor target. A starting molecule's value is principally as an analogous *solution* and is limited to what behaviours can be inferred from the interrelationships of its structural elements. Episodic information is brought to bear on a new problem in the more general form of the designer's repertoire of domain methods.

Dasgupta (1989) characterises design as an evolutionary process modelled on the scientific hypothesis-testing method. In Dasgupta's hypothesis-testing model, the designer sets out to test and evaluate the current hypothesis. Indirect drug design formulation has much in common with scientific hypothesis-testing and is consonant with the exploration-based model of design (Smithers et al 1989) which is outlined in the next section. The designer specifically seeks evidence from which hypotheses about the receptor's expected behaviours can be derived and against which these hypotheses can be tested. These hypotheses are evolved through exploration of aspects of the design problem which are dependent on the strength and utility of the available evidence. Where evidence is lacking or hypotheses are not testable, further data may be generated. The exploration is based on both substantive and methodological considerations. This exploration results in a model which embodies a set of hypotheses about the nature of the drug receptor and this model is used to generate testable design specifications. The evaluation of a design specification may lead to refinement or reconstruction of the model through further exploration of the characteristics of the design solutions which can be derived from the current model. There are two types of exploration, then: the first being part of the design formulation which is directed at producing a set of expected behaviours and the second being part of design specification and evaluation which is directed at producing the set of actual behaviours which specify the final design. These types of exploration are interdependent.

The exploration-based model of design

The exploration-based model of design characterises design as an exploratory process, and describes how knowledge underlying the design is organised, applied and generated, as shown in Figure 5. The design exploration process evolves a user-defined initial design requirement description into a final design requirement description. The initial design requirement description is usually a weak initial statement of the anticipated functionality of the artefact to be designed which may be incomplete, ambiguous and inconsistent. As the design proceeds, more of the space of possible designs is explored. Through this exploration, the initial design requirement description becomes more thoroughly defined resulting finally in a complete and consistent description of the artefact's functionality. This is the final design requirement description. A final design specification, consistent with the final design requirement description, is also developed. This is a technical description of the solution that will deliver the functionality described in the final requirement description.

The design exploration process is a collection of interconnected activities such as search, analytical assessment, problem decomposition, parameterisation, synthesis and optimisation, performed in a sequential or concurrent manner. The record of activities, decisions made, and rationale behind each decision constitute a history of the design process. The history, the final requirements description and the design specification together form the Design Description Document. The design space is explored by applying knowledge of the domain together with knowledge about how to design in the domain. Domain knowledge partially defines the space of possible designs to be explored. Domain knowledge together with knowledge about the methods which can be used to explore this particular domain define the Design Knowledge Base.

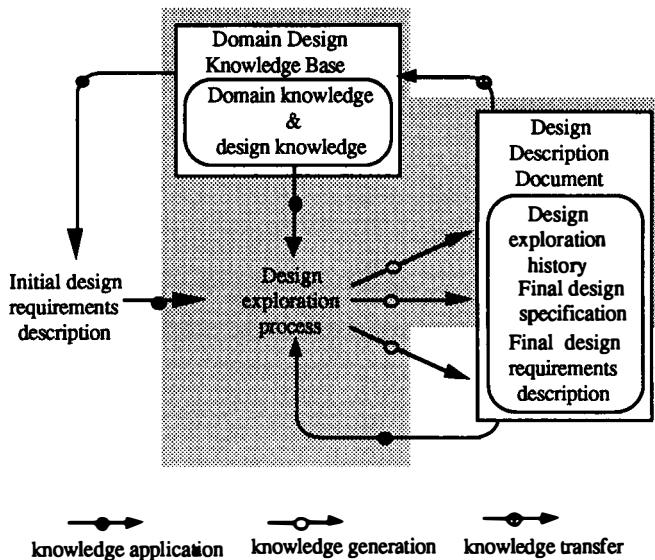


Figure 5. The knowledge process underlying exploration-based design

THE ARCHITECTURE OF THE PRELIMINARY PROTOTYPE

The exploration-based model of design grew out of and informed the design of the Edinburgh Designer System (EDS), part of the Alvey *Design to Product* project (Smithers 1987). EDS was a system which integrated various representations, reasoning and control subsystems to support mechanical design. Both the exploration-based model of design and EDS have influenced the architecture of the Castlemaine preliminary prototype system. The architecture of the Castlemaine preliminary prototype encompasses the portion of the model with a grey background as shown in Figure 5 for the domain of indirect drug design.

In developing the architecture, the primary goal has been to provide a design support system where the user is actively involved in problem solving rather than an autonomous problem solving system, the aim of many conventional expert systems. This is because the designer's exploration of a problem may take many different routes and involve different problem-decompositions, posing problems of control for an autonomous system. A design support system must take on the role of assistant and appear to be competent in the design domain. For this to be so, a number of requirements have been identified which must be addressed by the architecture:

- Complexity management:* The system should enable complex design explorations to be undertaken by transferring some of the cognitive load from the users to the system. This cognitive load is associated with various design activities, such as recording and structuring the information generated by considering large numbers of possible design choices.
- User support:* The system should act as an assistant to the user by cooperatively working with the user, by alleviating users of some of their workload by performing the mundane design subtasks and providing support for the more

complex ones. The system should be able to account to the user for the reasoning processes which it follows.

- c. **Context management:** For many design tasks, the user will wish to explore simultaneously multiple avenues to a solution. This requires some form of context management to allow mutually-inconsistent solutions to be examined.
- d. **Reason maintenance:** Some design subtasks may employ default reasoning. A reason maintenance system enables the dependencies between inferences made during problem solving to be recorded and consistency between them to be maintained.
- e. **Knowledge representation:** The system must be able to represent in a structured fashion the different types of knowledge commonly used during design, such as those found in textbooks and design handbooks. In addition, the system must be capable of applying this in a timely fashion to perform various design subtasks as the user requires.

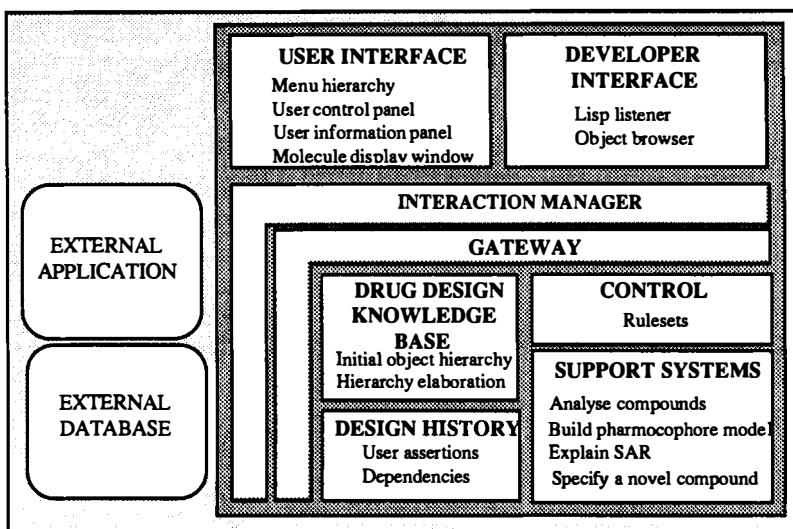


Figure 6. The architecture of the preliminary prototype

These requirements have been satisfied to various extents by the architecture of the preliminary prototype, shown in Figure 6, which is described in the remainder of this section.

The architecture is based upon a blackboard model of control which is centred around the concept of multiple agents communicating via a global workspace, referred to as the blackboard, to cooperatively solve a problem (Nii 1986). In the Castlemaine prototype system, there are two types of agents: the support systems with their related knowledge sources; and the user who is treated as a high-priority knowledge source.

There are four support systems, each of which performs some design support task. Each support system corresponds to one of the four major tasks within indirect drug design determined from the task analysis described previously. A support system's overall task

can be decomposed into self-contained subtasks, each of which is implemented as a knowledge source. A knowledge source, in this implementation, is comprised of forward-chaining rules. The support system and knowledge source hierarchy was presented above in Figure 3.

There is a need to control the invocation of knowledge sources so that design subtasks are performed in a computationally efficient manner and at the user's command. This is achieved through a mechanism based on GoldWorks rulesets. A ruleset is an object which controls the activation and deactivation of a set of rules assigned to it. When a ruleset is deactivated, none of its rules are used to generate any items for the agenda in the pattern-matching phase of the match-fire cycle. If all rulesets are deactivated, then nothing gets onto the agenda. Rules within a ruleset can control the activation or deactivation of any ruleset, which means that a ruleset can have a low-priority rule whose role is to deactivate the ruleset itself and then activate an alternate ruleset. In this way, different rulesets can be chained together to perform some complex task, efficiently using the limited system resources. Rulesets can be used for forward-chaining, backward-chaining and goal-directed forward-chaining.

This provides a higher level control over knowledge sources and allows some sort of system resource control and focusing of control to be achieved. Multiple rulesets can be defined and organised as a chain by putting generic rules into each ruleset such that their activation and deactivation can be controlled. Usually, when a ruleset is activated, all other rulesets are deactivated. This effectively focusses the system's resources on the relevant knowledge sources. It also prevents two knowledge-sources which share the same objects in the dynamic knowledge base from generating conflicting agenda items. Rulesets can be constructed in line with the design task decomposition so that each knowledge source is assigned to a unique ruleset. The invocation of knowledge sources can thus be controlled by activating and deactivating the relevant ruleset in the system. This may be done either by the ruleset itself or by the user of the system.

In the Castlemaine drug design support system architecture, the user has been given the highest priority in controlling the knowledge source invocation. A main menu hierarchy is provided in the user interface to allow the user to activate a ruleset so that explicit control of the inference process can be achieved. This ruleset-based control scheme allows several important issues such the design task decomposition, control and focussing of the system's resources to be consistently embodied in the architecture.

The core of the knowledge representation for the preliminary prototype is the drug design hierarchy that has been constructed within the GoldWorks frames system. As well as the comprehensive hierarchy of drug design concepts generated during knowledge acquisition, there is a version which has been optimised for the implementation of the preliminary prototype. The drug design hierarchy at the outset of a session initially holds a set of compounds and concepts for modelling compounds at different levels of abstraction, as outlined above. This corresponds to the domain knowledge in the design model. During a session, as the design process proceeds, the hierarchy is elaborated with new instances representing structural and functional aspects of compounds, and pharmacophore models with their features. The design specification for a novel compound would be derived from the information stored in the frame hierarchy, given in Figure 7. In this, boxes represent chemical concepts and arrows represent relations between them. For example, a *molecular fragment*, implemented as an object, is comprised of *molecular components*, also implemented as objects. A *molecular fragment* has the attributes *primary property* and *secondary property* determined from the attributes of the *molecular components* comprising it.

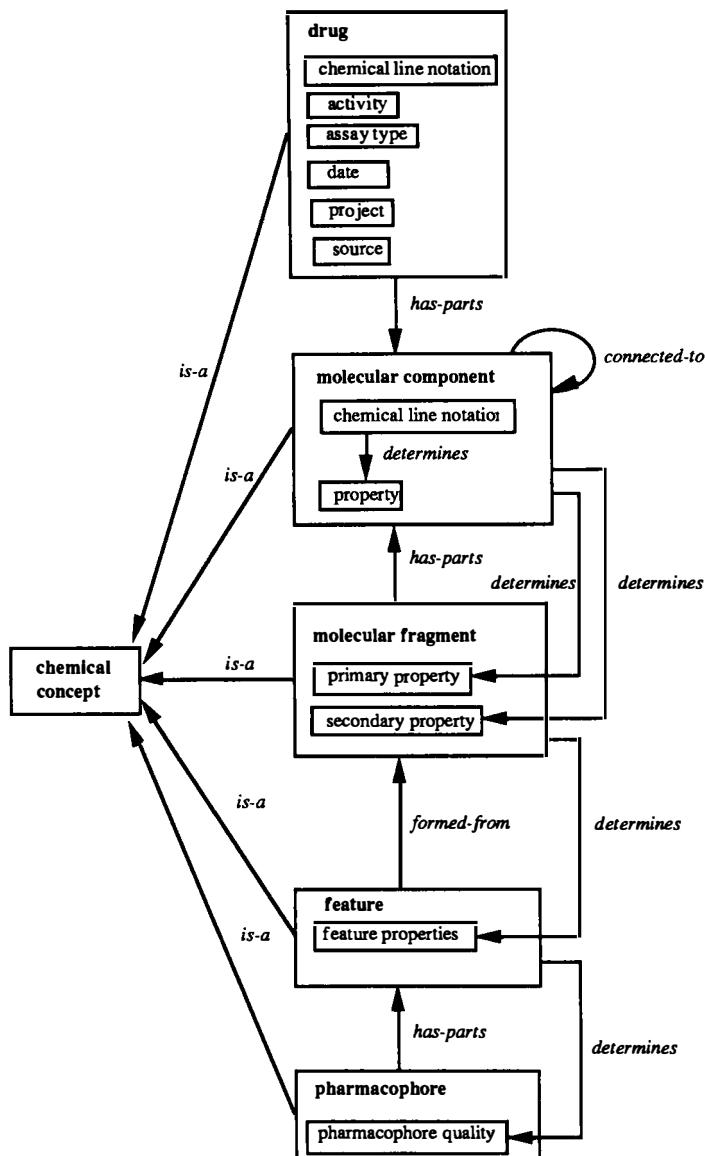


Figure 7. The relationship between chemical concepts

The *design history* records the changes taking place in the dynamic database and holds a justification-based chronological record of the design process. The design history is built up during inference by the support systems and from input given by the user. It records:

- a. assertions made by the user during a design session;

- b. the reasoning history, that is, the inferences in the form of changes to the drug design hierarchy made by rule firings, and the supporting justifications for a rule's triggering;
- c. the temporal ordering of rule firings.

This information is used by the system to provide an explanation of its inferencing. For the first prototype system, the history record also includes snapshots of significant past states which enables the user to return to an earlier stage in the design process and take an alternative route. This is a simple form of context management.

The success of a design support system largely depends upon the ease of interaction with the user. The user interface of the preliminary prototype has been designed so that the user has full control over the system enabling them to pursue their intentions as desired during the design process with the system acting as an assistant. Also, the interface allows the communication of large volumes of data associated with indirect drug design between the user and system. This functionality is provided by the four major subcomponents comprising the user interface:

- a. the menu hierarchy allows the user to invoke support systems through the control module;
- b. the user control panel enables the user to load/save compounds from/to text files, or add, delete or modify compounds in the drug design hierarchy. Also, a facility is provided so that users can search and browse through the the drug design hierarchy. The user control panel makes use of the gateway;
- c. the user information panel displays system status information from which informs the user whether some expected work has been performed by the system;
- d. the molecule display window is a multiple window display area to display simultaneously up to nine molecules. Active graphic images are used so that the user can interact with the system through the graphical displays. This is particularly important in drug design where medicinal chemists are used to thinking about and manipulating two-dimensional graphical representations.

There is a requirement to convert textual chemical formulae of compounds into the internal representation (instances in the drug design hierarchy) which can be reasoned about, and vice versa, to retrieve formulae from the internal representation. These chemical formulae may be provided or needed by the user, a chemical database or an external application package, such as a molecular modelling system. In the preliminary prototype, this function is performed by the *gateway*.

The *interaction manager* is concerned with the working of subsystems within the whole design support system and is consequently closely linked to the functioning of the control system and the user interface. It manages the interactions between the subsystems, between foreign languages, to different interfaces and to external applications or databases.

SUMMARY

The architecture of a molecular drug design support system has been developed based upon the exploration-based model of design and integrating various knowledge based system techniques. The research work in the first half of the Castlemaine project has shown how different knowledge acquisition techniques can be used in a top-down approach to elicit general to specific knowledge for the domain of drug design.

Future work in the second half of the project will evaluate the preliminary prototype system. This evaluation will be used to extend the exploration-based model of design and develop a main prototype design support system. The aim is to make the architecture of the main prototype more generic so that systems based on this architecture can be applied to other domains, such as software design, as well as to drug design.

ACKNOWLEDGEMENTS

The work referred to in this paper is supported by grant number GR/F3567.8 from the UK Science and Engineering Research Council and through the UK Department of Trade and Industry's Information Engineering Directorate. We would like to thank the conference reviewers and Brian Logan of the University of Edinburgh for their comments on an earlier draft of this paper.

REFERENCES

- Breuker, J. (1987) Model-Driven Knowledge Acquisition: Interpretation Models. *Deliverable task A1, Esprit Project 1098. Memo 87, VF Project Knowledge Acquisition in Formal Domains.*
- Burton,A.M., Shadbolt, N.R., Rugg, G. and Hedgecock, A.P (1990) The Efficacy of Knowledge Elicitation Techniques: A Comparison Across Domains and Levels of Expertise. *Knowledge Acquisition 2:* 167-178
- Dammkoehler, R.A., Karasek, S.F., Berkley Shands, E.F. and Marshall, G.R. (1989) Constrained Search of Conformational Hyperspace. *J. Comp. Mol. Design,* **3:** 3-21
- Dasgupta, S. (1989) The structure of the design process, *Advances in Computers,* **28:** 1-67
- Gero, J. S. (1987) Prototypes: a new schema for knowledge-based design, *Working paper, Architectural Computing Unit, University of Sydney, Australia*
- Gero, J.S. and Roseman, M.A. (1989) A conceptual framework for knowledge based design research at Sydney University's Design Computing Unit, in Gero, J. (ed) *Artificial Intelligence in Design*, Computational Mechanics Publications, Springer-Verlag, Southampton
- Hickman, F.R. (1989) *Analysis for Knowledge-Based System: A Practical Guide for the KADS Methodology*, Ellis Horwood
- Johnson, L. and Johnson, N. (1987) Knowledge Elicitation Involving Teachback Interviewing, in A.Kidd (ed) *Knowledge Elicitation for Expert Systems: A Practical Handbook*, Plenum Press
- Lewis, R.A. and Dean, P.M. (1989) *Automated Site-directed Drug Design: The Formation of Molecular Templates in Primary Structure Generation*. *Proc. R. Soc. Lond. B,* **236:** 141-162
- Lewis, R.A. (1989) Determination of Clefts in Receptor Structures, *J. Comp.Mol. Design,* **3:**133-147

- Logica UK Ltd. (1989) *MacCadd Version 5.0 - A New Generation of CASE for the Apple Macintosh*. January 1989.
- Maher, M.L. (1990) Process models for design synthesis, *AI Magazine*, Winter 1990:49-58
- Nii, H.P. (1986) Blackboard Systems: Part 1, *AI Magazine*, 7:38-53.
- Rajan, T. (1988) Goldhill Finds the Midas Touch, *Expert Systems User*: 14-16, May 1988.
- Rawlings, C.J. (1987) Analysis and Prediction of Protein Structure Using Artificial Intelligence, *4th Seminar Computer Aided Molecular Design*
- Rich, E. (1988) *Artificial Intelligence*, McGraw-Hill
- Saldanha, J. (1988) Application of the Logic Programming Language PROLOG to Prediction of the Location and Orientation of Inhibitor Residues in Enzymatic Active Sites, *Imperial Cancer Research Fund Publication*
- Shadbolt, N. (1988) Knowledge Elicitation: The Key to Successful ES, *Expert Systems User*: 22-25, October
- Sheridan, R.P. and Venkataraghavan, R. (1987) Designing Novel Nicotinic Agonists by Searching a Database of Molecular Shapes, *J. Comp. Mol. Design*, 1:243-256
- Skingle, B. (1990) An Introduction to the PFES Project. *Proc. Avignon 90: Tenth International Workshop on Expert Systems and Their Applications*: 907-922.
- Smithers, T. (1987) The Alvey Large Scale Demonstrator Project Design to Product, in Bernhold, T. (ed) *Artificial Intelligence in Manufacturing, Key to Integration*, North Holland, pp251-261
- Smithers, T., Conkie, A., Doheny, J., Logan, B., and Millington, K. (1989) Design as Intelligent Behaviour: An AI in Design Research Programme, in Gero, J. (ed) *Artificial Intelligence in Design*, Computational Mechanics Publications, Springer-Verlag, Southampton
- Williams, N. and Holt, P. (1988) *Expert Systems for Users*, McGraw-Hill
- Zhao, F. and Maher, M.L. (1988) Using analogical reasoning to design buildings, *Engineering with computers* 4:107-119

Automating the design of telecommunication distribution networks

C. Rowles, C. Leckie, H. Liu and W. Wen

Telecom Research Laboratories
PO Box 249
Clayton VIC Australia

Abstract: This paper describes an automated system for the design of telecommunication distribution networks. The paper discusses models for the design process and shows how appropriate modelling and representation of the problem domain can simplify design. In particular, the paper shows how the decomposition of the problem in terms of the design components and the constraints on the design solutions can simplify the problem of generating new design structures. The embedding of the automated design process into a logical framework based on this decomposition is described, including the difficulties associated with representing all of the information required by the system that is taken for granted in manual design. The process of resource allocation to dimension the structure and provide a design solution is then described as well as how this process is optimised. Finally, the paper describes a technique for the optimisation of the resource allocation process.

INTRODUCTION

As systems become more complex, there is an increasing drive to automate their design. This is partly due to a desire to reduce design time, especially by automating the more mechanical aspects, but also to reduce errors and reduce system costs by optimising their design.

Increasing system complexity and an increasing range of implementation media mean that the solution space to many design problems is increasing, with a resultant increase in the resources required to determine the optimum solution. A result of this increased complexity is the increasing trend to the application of artificial intelligence (AI) technology to reduce the search space.

AI can contribute to design in a variety of ways: for the evaluation and critiqueing of manual designs to ensure conformance with design rules (e.g. Balachandran & Gero 1990); to assist in the design process by elaborating on manually generated design outlines (e.g. Kowalski & Thomas 1985); or by performing the automated design of systems given functional and performance specifications (e.g. Rowles & Leckie 1988).

This paper describes a system for the automated design of telecommunication distribution networks from a geographical description of the service area, a specification of the services to be provided and a description of the implementation technology. The paper outlines our problem domain, shows how appropriate modelling of the domain can simplify the design task, describes our approach to design, including design optimisation (of both design and design process) and finally, describes how we embed the design system in the domain.

DESIGNING

Over the last few years, our understanding of design processes has grown to such an extent that design automation systems have begun to appear in many applications (e.g. Rowles & Leckie 1988, Mittal & Araya 1986, Gero & Rosenman 1989). The design of Very Large Scale Integrated (VLSI) circuits, for example, is an area that has received much attention (e.g. Rowles & Leckie 1988, Parker & Hayati 1987). Here the goals have been to greatly reduce the design time, and at the same time minimise the occurrence of errors. VLSI designs can typically take several months, while undiscovered errors can produce costly and time-consuming design and prototyping iterations.

For our purposes at least, design can be described as a sequence of phases: *synthesis*, *resource allocation* and *evaluation*. Synthesis is the generation of new structures to meet some functional specification, while resource allocation is the dimensioning of that structure to satisfy a design specification (c.f. generative and interpretive design, (Coyne & Gero, 1986)). Evaluation is the process of comparing the predicted performance or behaviour of the design with the expected performance or behaviour of the functional specification. It may be followed iteratively by further phases of synthesis and resource allocation, hopefully leading to the design's behaviour converging on that required. In some applications, the domain theory may be insufficiently understood to allow evaluation based on predicted performance, and an actual working system or model must be constructed.

Many design tasks are well understood at the resource allocation phase. In VLSI design, for example, there are equations that describe for a given structure how system performance changes as parameter values are varied. These equations are the domain theory, and such a design domain may be automated via the application of Artificial Intelligence (AI) techniques. Here, equations and rules that describe their application become rules in an expert system.

The synthesis phase, however, is often regarded as *ill-structured* (McDermott, J. 1981), suffering from the inability to specify in advance a specific sequence of design steps that will lead to a solution. In many cases, human design experts appear to reason more intuitively and heuristically than novices. Sometimes a deep understanding of the relationship between structure and behaviour in a domain may suggest new structures that lead to novel designs. Often, experts reason analogically, drawing upon past designs to form the basis for new problem solutions.

While many applications may appear to require specific design approaches, there are nevertheless, several generic models of synthesis.

Simon (1969) regards design as a tree searching problem. Different design decisions lead down different branches, eventually terminating at leaf nodes representing final designs. As typical design trees can be very broad and deep, the problem becomes one of how to prune the search tree to allow good (or optimum) designs to be reached given reasonable computing resources. Realistically, the difficulty in predicting the behaviour of the final designs from the higher levels in the tree make pruning difficult. In addition, the tree representation is not especially well-suited to iterative synthesis or non-monotonic incremental improvement in general, although there are approaches to handling these problems (Rowles & Leckie 1988).

Maher (1990) describes three synthesis models: *decomposition*, *case-based reasoning* and *transformation*. Decomposition is a common analysis tool in engineering, and seeks to break a problem down into parts sufficiently small that a structure for the design solution be-

comes apparent.

Case-based reasoning is an analogical approach that attempts to find structure for a given design problem by recognising existing structures and solutions which have similar functional or behavioural specifications to the current task. It is similar the concept of *prototypes* (Gero 1987) synthesis for design solutions. In both cases, the “prototype” solution is refined to satisfy the new specification. Transformation is also similar in that it uses a set of generalised transformational rules to incrementally refine a design until it achieves the desired result.

In practice, most design tasks rely on a mixture of different design approaches according to the novelty and the complexity of the domain. In almost all cases, the search for a structure for the design solution (i.e. synthesis) remains the most difficult part of the design process.

TELECOMMUNICATION NETWORK DESIGN

The design of telecommunication networks is a complex design task encompassing synthesis, resource allocation and evaluation. The range of implementation technologies, spatial design aspects, mixed demands (i.e. functionality), long service life (i.e. requirement of low maintenance costs) and minimised installation cost demand complex trade-offs to be made. Such is the cost of installation and maintenance that design automation offers practical and economic potential by reducing design times and lowering costs by producing optimised designs. In particular, the distribution network that services residences and businesses from exchanges is a high cost component of any network due to its extent and diversity.

The design process is further complicated by a proliferation of alternative implementation technologies. For example, cables can be above or below ground, sharing trenches with other utilities or not. Cables may be copper or optical fibre, and may also carry multiplexers to increase the number of virtual channels above the number of physical channels. There may also be dedicated point to point links, and some links may be via radio. In most cases, the appropriate choice is dependent on the implementation cost of the final design.

Telecommunication distribution networks are hierarchical. Trunk cables connect exchanges. Exchanges service well defined exchange distribution areas via a main cable that connects the exchange to a distribution point (called a pillar). The pillar services a subregion of the exchange area called a distribution area (or DA) via a tree-structured network between the pillar and customers. The distribution network consists of cables of different sizes in pipes placed underground with joints placed in pits wherever cables of different sizes must connect to each other. Some cables may be suspended above ground from poles, and cables may be copper or optical fibre. Cable sizes taper toward the leaf nodes of the distribution tree. Copper cables contain individual pairs of wires each carrying one channel. Optical fibres and multiplexed cables contain larger numbers of virtual channels.

The key design problems in this domain are to determine the optimum distribution point in the DA, to determine the optimum tree structure and routing of cables for distribution, and to determine the optimum allocation of capacity and plant (such as cables) to satisfy the customer demand. *Optimum* is defined as the lowest cost solution that is consistent with a comprehensive set of design guidelines (i.e. design rules).

A DA consists of land parcels (containing buildings and having varying telecommunication network connection demands), reserves and streets. A typical DA consists of 300 telephone services, and DA topologies can vary widely. A simple DA is shown in Figure 1. All

plant items such as cables, cable joints, pillars and pits must be on public land, except for the *lead-in* cable to each building. Generally, cables routes follow road sides. The service requests are located on land parcels. Numbers in parentheses are the numbers of service requests on each street segment.

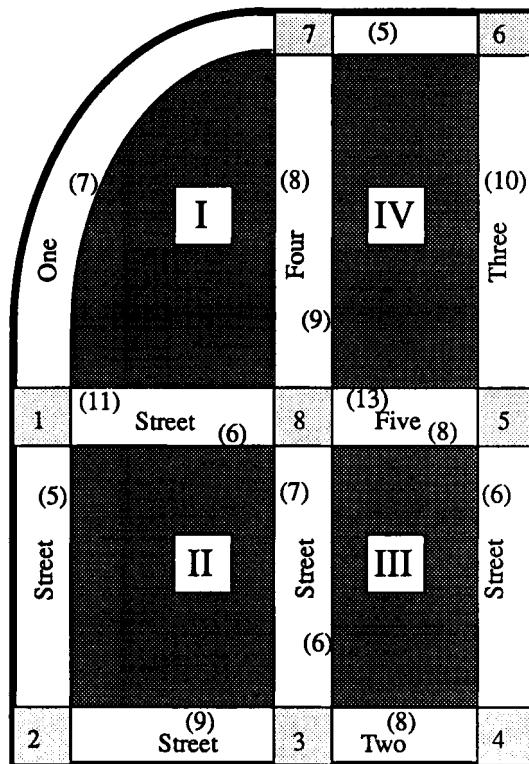


Figure 1. A Distribution Area.

The Logical Relationship Between Objects

All entities in our design are deemed as objects. A DA consists of streets and land parcels. Streets are composed of street segments and intersections. The unit of these objects is arcs.

The objects are represented as frames. Each object has its own properties which describe its pre- and/or post processing data. For example, a street contains side information (side A or side B), how the sides are made of arcs (along which cables are routed), and how many joints are on the street. In Figure 2, the frames of these objects are shown. Service requests

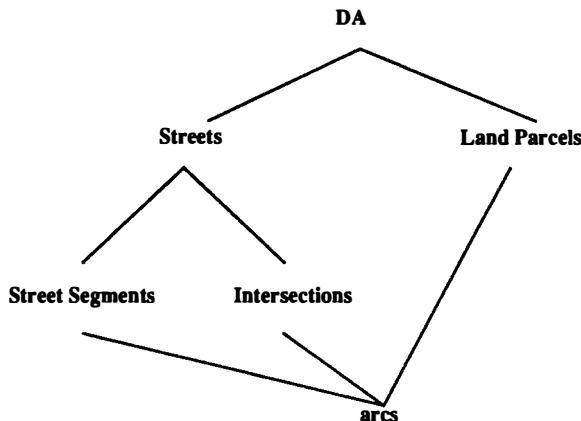


Figure 2. The relationship between objects in a DA

indicate a customer's telephone line requirements, and are represented in objects in which service location and number of requests are specified. The relationships between these objects are shown as "#" or "*". The reverse relation is indicated in the parentheses ().

A complete DA network design includes all cable routes, cables, pipes, pits, joints and road-crossings. Information on cable lengths, number of wire pairs per cable and even the labour required to install various components must be calculated to determine the total cost of the network. Each network component is represented as a frame with their properties. They are shown in Figure 3.

<i>arc</i>	<i>land_parcel</i>
instances #:	instances #:
list:	consist_of * (reverse_consist_of)
service_requests:	
<i>intersection</i>	<i>street</i>
instances #:	instances:
consist_of * (reverse_consist_of):	contains * (contained):
neighbof * (neighb):	joint_list:
<i>street_segment</i>	<i>sideA</i> :
instances #:	<i>sideB</i> :
consist_of * (reverse_consist_of):	<i>prefer_side</i> :
neighbourof * (neighbours):	<i>restriction</i> :
residual_request:	

Figure 3. Frame prototypes of objects in a DA.

Streets, land parcels (both represented topologically), service requests and catalogues of plant items are static data forming the starting point of the design task. All other information must be determined. Streets and land parcels are represented as polygons in a geographical database called the *cadastral*.

<i>pit</i>	<i>cable</i>
instances #:	instances #:
linked_by_cables * (cable linking):	arcs:
serve_requests * (served_by):	joint_begin:
location:	joint_end:
pair_number:	length:
	size:
<i>joint</i>	
instances #:	
serve_pit * (served_by_joint):	<i>terminal_cable</i>
list:	instances #:
location:	from_joint_or_pit:
out_cables:	to_pit:
pre_joints:	
	<i>road_crossing</i>
	instances #:
	from_joint_or_pit:
	to_pit:

Figure 4. The objects in the design solution.

A DESIGN AUTOMATION SYSTEM FOR TELECOMMUNICATION NETWORKS

The distribution network design problem comprises synthesis and resource allocation. The domain theory is reasonably well-defined; how various network components interwork to achieve a solution is known, and the design solution is characterised as the lowest cost option that is consistent with current design guidelines and satisfies the projected demand. Other factors such as unknown geographical constraints (objects such as trees and other obstacles are not digitised into the cadastre), modifications to the DA boundary and future flexibility are handled as manual modifications before or after the automated design process.

The synthesis problem is constrained, resulting in a simpler design task. The DA boundary is determined with regard to trunk cable routes, exchange areas and geographical features such as rivers, major roads and reserves. Thus, for our purposes, it can be regarded as fixed. As the exchange position is pre-determined, so is the main cable route to the DA. Thus, the synthesis problem is limited to determining the optimal pillar location and optimal cable routing, given the cadastre. Land parcels and streets are represented in a geographical data base as polygons. As network plant is generally positioned along streets but not on land parcels, the arcs forming the boundaries between streets and land parcels become potential cable routes. Similarly, a pillar can only be located on a boundary arc. By regarding the distribution network as a tree, the problem is to determine where the root of the tree should be (i.e. the pillar location), to determine which arcs result in an optimal cable routing and to break loops formed by intersecting streets (to avoid loops in the tree).

Thus, there exists a rudimentary structural prototype for a distribution network produced by the topology of the DA. Because of this inherent structure for a design solution, we do not have to attempt to represent potential solutions as a canonical set of distribution topologies or attempt the incremental generation of a solution via transformations. Instead, the prototypical design structure results from the decomposition of the logical representation of the problem domain (see later for a more detailed discussion of this decomposition).

Similarly, this inherent "prototype" and a logical representation of the network tree neatly avoids complex spatial reasoning. Since cables are only available in discrete sizes, and routes will determine the number of pairs required in a cable, however, synthesis and re-

source allocation do require an iterative solution.

The design is optimised to satisfy the following objectives:

(a) *Flexibility for future extension*: future potential development should be considered when a new DA design is carried out. This reduces the likelihood of having to carry out major network modifications when the number of service requests increases.

(b) *Reliability*: minimum access to underground plant and minimum number of joints are two guidelines for reliable design since joints are a major source of faults.

(c) *Minimum cost*: once the first two considerations are satisfied, the lowest cost solution is preferred.

(d) *Consistency*: as DA network design is currently performed by many individuals nationally, standardisation on the latest design rules is important.

(e) *Multiple objective balance*: balance these objectives in order to achieve a design with the best overall performance.

In satisfying these objectives, the following features are desired in the final design: (1) optimum pillar location; (2) optimum cable routes; (3) minimum joint numbers; and (4) minimum equipment costs.

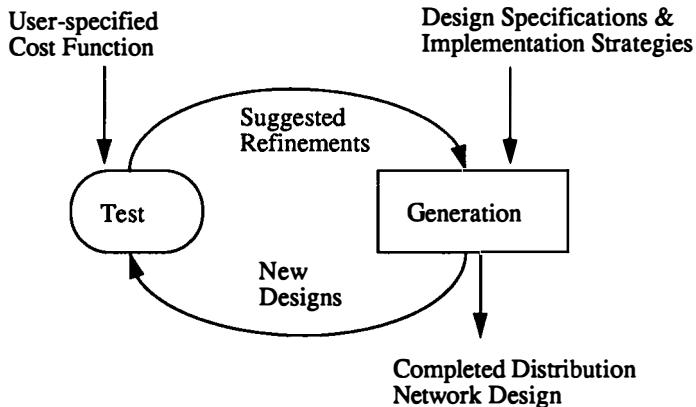


Figure 5. The Design Process

The system operates iteratively as shown in Figure 5. Optimisation is divided into two stages. The *generate-and-test* loop implements the stage of global optimisation in which cost functions are used to guide the optimisation of the synthesised structure and the design process itself (see later for more details). Iteration continues until the threshold set by the cost functions has been reached (Liu, Wen & Rowles 1990). The other stage is local optimisation within the generate phase, in which heuristics can be used to select between alternative design options. Design generation is performed as a series of layers. The top layer is pillar placement. This is a process of trading-off main cable costs versus the cost of distribution cables. The second layer performs joint assignment and cable layout. The appropriate design strategy at this stage is chosen according to the type of DA. The final layer assigns terminal cables, road-crossings and pits to the design, and refines the location of cable joints. Very

limited optimisation is possible at this stage, since the earlier two stages have largely constrained the design.

The complexity of the domain precludes the use of exhaustive search to achieve the design objectives. Instead, DENIS uses heuristics to prune the search space and suggest a near-optimal solution. These heuristics are encoded either as approximate evaluation functions or production rules, depending on the particular design task.

The design system is implemented in LASER (an AI programming environment) and C. Briefly, LASER allows structured production rule design, by providing the rule types as global rule groups, global rules, local rule groups, and local rules. In addition, global rules or groups can be assigned priorities. Since a network design problem is treated as tree pruning, the design process is simplified by using a production rule system together with a hierarchical frame-based representation. The use of polygons and arcs simplifies the reasoning process. For example, implied in a polygon representation is the knowledge that cables may not cross land parcels or street intersections by only allowing cables run along the boundaries of polygons of streets, land parcels and intersections. As mentioned earlier, the design flow proceeds sequentially: first, pillar placement; second, joint assignment and cable routing; and third, resource allocation. In different phases, the focus of design is on different problems. Each of these design phases and the heuristics that they use are described in the following subsections.

Pillar Placement

The first task of pillar placement involves trying to find a central distribution point within a DA that minimises the total cost of main and distribution cable. The range of positions is restricted by practical considerations, since the pillar must be on a street boundary in a safe and accessible position. However, the true cost cannot be calculated unless the cable routes have been determined. The computational complexity of calculating cable routes prohibits us from doing this while searching for a pillar position. Instead, we decouple the problems by using an approximation of the cable costs to determine an initial pillar position that is highly likely to be near the optimum, and then use this result in solving the cable routing sub-problem. If the final result differs significantly from the estimate then we can iterate through this generate and test loop. The advantage of this approach is that the number of iterations can be reduced by using the solution to an approximation of the original problem as a heuristic.

We approximate the costs of distribution cable by assuming that each service is connected to the pillar in a direct line of sight, rather than following the street network. From this linear approximation we can derive a closed-form solution for our initial estimate of the pillar location. This turns out to be the “centre of mass” of the DA, using a weighted sum of the coordinates of each service. Of course, this position may appear anywhere in the DA, and so it must be mapped onto the street network to find the nearest practical position on a street boundary. This position may require further modification if it is too close to an intersection or in an awkward position. A rule-based approach is used to test for these situations and, if necessary, suggest where the pillar should be moved to.

An outline of the algorithm is as follows:

Given the coordinates of each service request,
 the number of connections to each service request,

the main cable entry point,
the street boundary cadastre.

- (a) Estimate the main cable size from the total number of service requests,
- (b) Calculate the coordinates of the “centre of mass” for the service requests,
- (c) Project the centre of mass onto the nearest street boundary,
- (d) Apply safety rules to test the street boundary position,
- (e) If the safety rules recommend that the pillar be moved
- (f) Translate the pillar along the street boundary accordingly.

At present, a best-first approach is used in the pillar placement algorithm. This can be extended to a Branch And Bound search by projecting the pillar onto all the surrounding street boundaries, rather than merely the nearest. The approximate cost of each of these positions can then be evaluated, and the best used for cable route design. If it then turns out that the true cost of the chosen position exceeds the estimated cost of one or more of the alternatives, these may be used as the basis for alternative designs. Since the “line of sight” cable cost function consistently underestimates or equals the true cable cost, it is an admissible heuristic for use in Branch And Bound search (Pearl 1984). Such a search would provide a method for validating the initial choice made by the heuristic cost function.

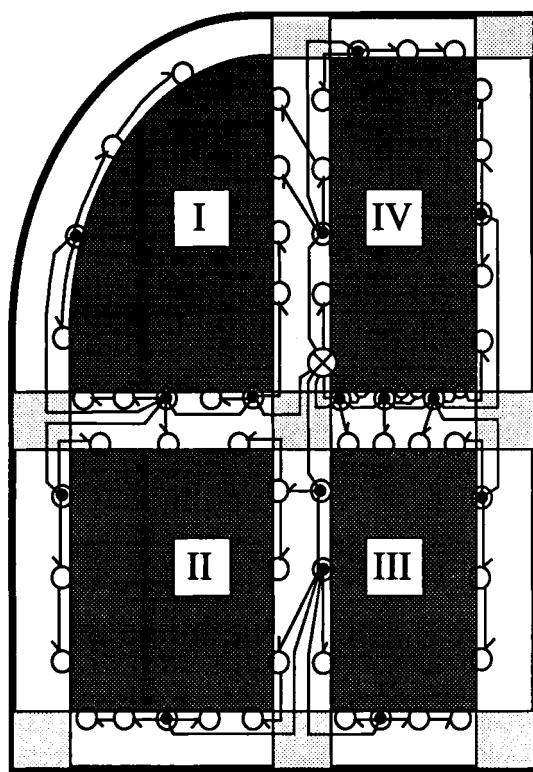
Cable Routing and Joint Assignment

Cable routing determines how cables run through the streets in order to serve customers. In general, if there is a service request on a street, a cable should be laid along that street. In some situations, the preferred reticulation practice is to run cables down only one side of the street and use road crossings to connect customers on the other side. Accordingly, the cable routing algorithm needs to determine which side of a street should be chosen. As a heuristic, the side that contains more service requests is chosen.

The result of joint assignment and cable routing is a tree in which a pillar is its root, joints are nodes, and cables are links. The leaves of the tree are joints that have no outgoing cables. At every joint, a ten-pair cable is split out along the cable route. These ten pairs can be used to service eight customers with two pairs spare for future demand.

Cable routing is modelled as a search for a tree spanning all the intersections, with the pillar as its root. In order to guide the search, each intersection is labelled with the number of neighbouring street segments that have not yet been assigned joints, referred to as its out-going degree. Each time a street segment is included in the cable route tree, joints are allocated to that street, and the out-going degree of each intersection is updated. This process repeats until the out-going degree of each intersection is zero. We use four heuristics to guide the search, implemented as a set of prioritised rules:

- (a) Start cable routing from the DA boundaries.
- (b) Use intersections along the DA boundaries that have least out-going degrees to start a cable route.
- (c) If no such intersection exists, use intersections whose out-going degrees are the least.
- (d) Among the intersections which satisfy (c) & (d), choose the one which is farthest



- | | | | |
|-----|--------------------------------|---|----------------|
| ■ | Intersection | ■ | Street segment |
| ■■■ | Aggregated Land parcels | — | DA boundary |
| ⊗ | Pillar | ○ | Pit |
| ● | Joint | — | Cable segment |
| ← | Terminal cable / Road crossing | | |

Figure 6. Diagrammatic representation of completed layout.

away from the pillar.

The purpose of these heuristics is to minimise the number of joints needed to serve the DA. In doing so, the susceptibility of the network to faults should be minimised, since many causes of faults centre around cable joints.

An example of the results of this cable routing and joint allocation process appear in Figure 6. In order to validate the efficacy of these heuristics, it is necessary to compare these results against those from a more exhaustive search. For this purpose, we have used a much slower method, namely simulated annealing, to provide an independent solution. The heuristics can then be refined if they have missed any of the optimisations that were found by simulated annealing. This refinement process is described in more detail later.

Resource Allocation

Cable dimensioning is carried out by traversing the tree in a depth-first fashion. For every joint encountered, ten pairs of cables are added to the previous cable dimension. When the root is reached, the actual need for cable pairs on each cable is defined. Similarly, pits, terminal cables, and road-crossings are determined while the tree is being traversed. The exact locations of joints are then adjusted according to where services are required and where pits are deposited.

At the end of cable routing and joint assignment, cable routes are terminated at the pillar. Usually, the joints assigned along each route exceed the service requests. Often there are redundant joints. Thus, some joints should be deleted under certain conditions, and the relevant cables should be re-routed. Heuristics for this process are given below:

- (a) Keep the most isolated joints (such as joint J3 in Figure 7) that have no close neighbouring joints.
- (b) Delete the joints that serve the least number of requests among the non-isolated joints, such as joints J1 and J2.
- (c) If (b) still cannot determine which joint to prune, delete the joints whose farthest neighbouring joints (these should be isolated joints) serve less requests.

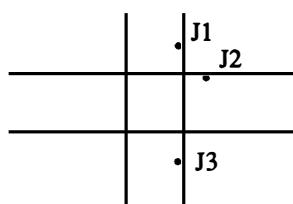


Figure 7. Joint allocation

EMBEDDING DESIGN

The form and availability of design data is a major constraint on the function and structure of any automated design system. Rather than standing alone, our design system must integrate with existing sources of data describing the telecommunications network. In many cases, this

data has been collated for other purposes, and hence may not explicitly encode the entities and relationships that are most relevant to the design process. As a result, providing easy access to the information that is required, in a form that reflects the uses to which it will be put, is a key factor in simplifying the design process.

During the design, we need to access three main types of information: the goals for the system, the constraints on the design, and the resources that are available to implement the network. The goals of the system in this case are the customer service requests for connection to the network. Each service request is represented by a physical reference point and the type of service required. The constraints on the design come in many forms. They include the cadastre, the type of technology to be used, any existing plant in the DA, and physical obstacles which must be avoided in the design. The resources available to implement the design are described by the conditions under which each item of plant can be used and its cost.

As mentioned earlier, by making use of the inherent structure present in the domain we can constrain the choices in our design to a point that makes the search for satisfactory solution tractable. Our internal representation of the domain should contain those features which encode this inherent structure. However, the existing encoding of the domain may contain extraneous information used for other purposes while lacking certain features that are central to the design. For example, cable routing is simplified if the DA can be modeled as a graph of street and intersection boundaries along which the cables may run. In deciding between alternative routes, we may require additional information about the proximity between streets and their lengths. This model must be reconstructed from the existing representation for the cadastre, which has been primarily designed for graphical display. Thus, detailed information about the shape of each street can be discarded, while a model needs to be built of alternative routes between intersections. The advantage of this mapping process is that it maintains a faithful representation of the problem domain while highlighting the key features which structure the design. Without it, the design problem would lack sufficient constraints to make its solution tractable.

This data needs to be assembled from a variety of sources. Some of it will be entered directly into the system by the human designer, e.g., service requests, choices of technology and physical obstacles. The rest must be gathered from external databases and translated into the object-oriented representation used within our system. It is important that each feature that is referred to directly during the design is explicitly represented in our system. In many cases, this external data will not be encoded in a form that is suited to our application, and will need to be manipulated into a form that extracts the features used in each stage of the design. The encoding of the cadastre for the DA is a case in point. It describes the layout of the streets and land parcels within the DA boundary in terms of points, arcs and polygons. These can be encoded either topologically or spatially. The topological encoding describes the connectivity between arcs and polygons, while the spatial encoding records the shape and coordinates of each of these features. Each of these encodings is required at different stages of the design. For example, in calculating the pillar position, the main focus is on the spatial distribution of the service requests in the DA. By comparison, the cable routing subtask is more concerned with the topology of the street boundaries when generating the structure of the cable layout. Both of these encodings are kept in a geographical database that is external to the design system. It maintains the topological relationships between features, and can support spatial queries that arise during the design.

However, not all the information that is relevant to the design is directly available to the system without manual intervention. For example, the task of designing the DA boundary is beyond the scope of our system as it is strongly influenced by knowledge of the physical lo-

cation of features such as major roads, rivers, railway lines and so on, which are not encoded in the cadastre. Similarly, there is no information directly available to the system about obstacles to plant allocation such as trees. Thus, the design must be open to modification by the human designer who can take such information into account as a result of a site inspection. This has a significant impact on the structure of our system. It must try to act autonomously given the information available, yet be able to accept and evaluate changes made by a human designer. Even though the initial design produced by the system may need to be changed, it provides a starting point for the human designer in deciding which additional information is relevant. It also provides a reference point for comparing alternative strategies that the human designer may suggest.

OPTIMISING DESIGNS

In this section, we discuss the system design methodology and optimization of the design. The former is about how we can refine a rule-based system which can approach to the performance of Simulated Annealing (SA) but with much less computational complexity. The latter is about the method of Branch And Bound (BAB) method.

Refinement of the Rule-Base by Simulated Annealing

Although an optimum or nearly optimum solution can be always guaranteed by using SA, SA is too slow. An alternative to SA is to pursue designs that are good from an engineering point of view but which are not necessarily the absolutely optimum. However, to approach nearly optimum solutions, a good knowledge base is needed. This section will discuss the refinement of the knowledge base using SA so that nearly optimum solutions will be produced.

Knowledge eliciting is a well known bottle neck of the design of knowledge based systems. It is painful to rely on the domain experts to provide the whole set of examples. This is because the time of domain experts is often very expensive and it is very difficult to built up an efficient communication between the knowledge engineers and the domain experts. Our method is:

1. Ask the domain experts to give a very basic theory about the relevant domain. This theory is neither necessarily error-free nor complete. This theory is called the initial knowledge base.
2. A set of typical example design specifications is collected by the designer of the system and is handled by an SA system to generate a set of corresponding design solutions. This set of solutions can be reckoned as at least nearly optimal.
3. The design specifications and the corresponding SA design solutions are then used to refine the initial rule base. The designers and the experts will assess the rule base comparing initial rule-base decisions with those made during simulated annealing. If the initial decisions do not result in satisfactory solutions then the rule-base is modified to replicate the SA system decisions. This procedure is shown in Figure 8.

In this refinement procedure, SA plays the role of an example generator. It converts our problem from an unsupervised one into supervised one. The user is the tester. Because we think the results generated by SA are at least nearly optimal, the domain theory or rule base

has to be changed according to the examples if the user cannot explain them as permissible exceptions.

Evaluation of Generated Designs Using Branch and Bound

In operation, the design system follows a generate and test paradigm during design synthesis. Since the detailed costing of a design is only available after the resource allocation phase, a heuristic evaluation technique is required to test decisions during various stages of synthesis. For this we use a branch and bound (BAB) approach.

In a BAB system, we need an algorithm to evaluate the cost of a design decision. For our application, we adopt the following cost function:

$$COST = \alpha \cdot C + \beta \cdot R + \gamma \cdot F + \delta \cdot D \quad (1)$$

where

C -- cost per customer service.

R -- the ratio of utility of non-standard materials and practice.

F -- factor for the non-economic application of alternative technologies.

D -- deviation from current service requirements.

and $\alpha, \beta, \gamma, \delta$ are constants selected by the designer to reflect the importance of the above four cost parameters. This cost value can be considered as a measure of the merits of a design generated by the system.

To see how to calculate the cost of a scheme, note that any valid design generated by the system can be thought of as a tree, the root of which is the pillar. The node set of a tree is the set of distribution joints (there may also be some empty pits without joints in them) for underground reticulation. The pillar also belongs to the node set and is the root of the plant tree.

The set of links between nodes is the cables (and pipes) connecting the corresponding nodes. Each node and each link between nodes have costs associated with them. For a node the cost assigned to it is:

- (a) the cost of joints,
- (b) lead-in cost if any, and
- (c) pits needed if any, and
- (d) labour for installation and maintenance.

For a link, the cost associated to it is:

- (a) The cable runs between the two terminal nodes of the link.
- (b) Pipes needed if any,
- (c) Labour for installation and maintenance.

For each node and link, parameters of R and D are also assigned. For a network tree, a count of pair gain used is also maintained and so is a count of non-economic application of alternative technologies.

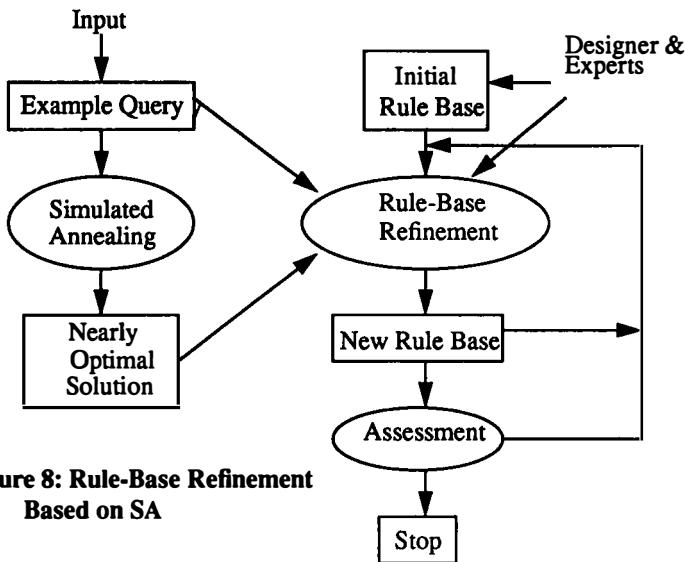


Figure 8: Rule-Base Refinement Based on SA

Data Structures

Frames are used for data representation. The only things need to be mentioned are:

- (a) The attached procedures contained in *If-Needed* slots and *If-Added* slots can be implemented with pointers to the corresponding functions.
- (b) A slots itself may have subslots or may also be an array of subslots of the same type. The latter may be the main difference of our system from the conventional frame-based systems.

For example, suppose we have defined the following enumerate types:

```

enum Dominant_type
{single,flats, multi_story,shop_centre, complex high_
rise};
enum Class
{ I, II, III, IV, V, VI };

```

and

```

enum Cons_form
{ shar_UG, sepa_UG, aerial, dp };

```

frame DA_PLANT can be implemented by the following C struct:

```

struct DA_PLANT {
    struct *design;
    enum Dominant_type dominant_type;
    int default_type=single;
    enum Class class;
    int default_class=I;
}

```

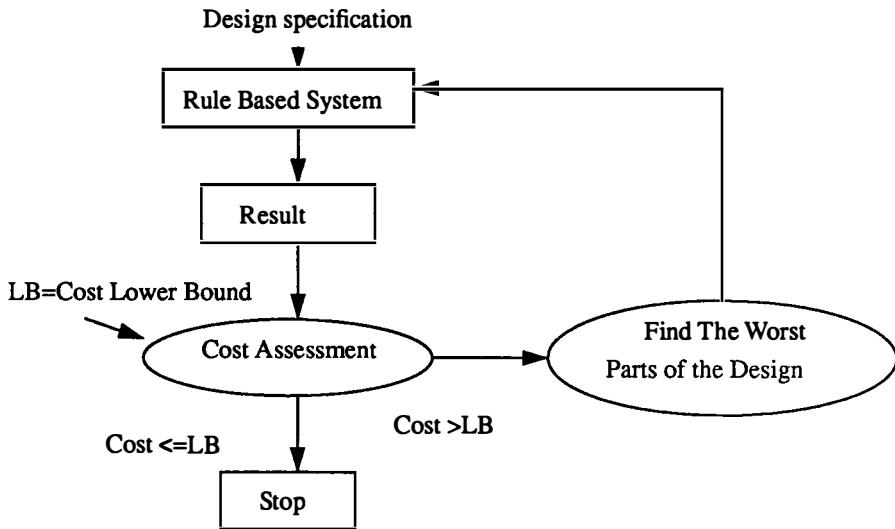


Figure 9: BAB Algorithm

```

int (*class_det)();
NODE *Pillar;
int No_services;
int (*no_services)();
enumerate Cons_form cons_form;
int default_form = share_UG;
float merits;
float (*merit_calc)();
}

```

where function `class_det()` is a C function as follows:

```

class_det(type)
int type;
{
    switch(type) {
    case single: return(I);
    case flats: return(II);
    case multi_story: return(III);
    case shop_centre: return(IV);
    case complex: return(V);
    case high_rise: return(VI);
    }
}

```

It is easy to write C functions for `no_services()` and `merit_calc()`.

The Algorithm

To evaluate a design, a depth-first search algorithm is adopted. Starting from the pillar, the root of the tree, the algorithm traverses the tree. Whenever a node whose all children have been assessed is encountered the algorithm accumulates the costs of the children and the corresponding links to the cost of the current node and then marks the node as assessed. The procedure continues until the node corresponding to the pillar is assessed. The framework of the algorithm is as follows:

Algorithm 1 (*Design Evaluation*):

Input: A weighted plant tree rooted by the node corresponding to the pillar.

Output: The cost of the tree of the form (1).

Method:

1. The main function:

```
struct NODE *pillar;
assess(pillar);
C =  $\alpha$  * pillar->cost;
R =  $\beta$  * (pillar->NonStd / pillar->TotalMat);
F =  $\gamma$  * pillar->NonEconomic;
D =  $\delta$  * pillar->DevFromStd;
printf("The merit factor of the plant tree is: %f.\n",
      1 / (C + R + F + D));
```

2. `assess(node)` is a recursive function traversing the plant tree:

```
void assess(node)
struct NODE *node;
{
    struct NODE *np;
    for (np=node->links;np!=NULL;np=np->next)
        assess(np->end_node);
    node->assessed = TRUE;
    accumulate(node);
}
```

3. `accumulate(node)` accumulates the costs of all child nodes of node and the costs of the links between node and its children, and then adds the result to the cost of the current node. The result of accumulation is the cost of node:

```
void accumulate(node)
struct NODE *node;
{
    struct NODE *np;
    for (np=node->links;np!=NULL;np=np->next) {
        node->cost += CostEstimate(np);
        node->TotalMat += TotalMaterial(np);
        node->NonStd += NonStandard(np);
```

```

    node->NonEconomic += NonEcoUtil(np);
    node->DevFactor += DevFromStd(np);
}
}

```

where,

(a) `node->cost`, `node->TotalMat`, `node->NonStd`, `node->NonEconomic`, and `node->DevFactor` have been initialized to the local values within the nodes in the input weighted plant tree.

(b) `CostEstimate()`, `TotalMaterial()`, `NonStandard()`, `NonEcoUtil()`, and `DevFromStd()` return the corresponding cost factors of the child nodes as well as the corresponding links.

Branch and Bound Design Algorithm

The basic points of BAB algorithm (see Figure 9) are as follows:

- (a) Based on previous design experiences, give a reasonable lower bound of the average cost LB.
- (b) Run the design algorithms once to get a solution S with an average cost C.
- (c) If $C < LB$ the algorithm terminates and the final solution is S,
- (d) Otherwise, find the worst part (or parts) in the current design, modify that part (or those parts), goto step 2.

By “the worst part (or parts)” we mean that part (or parts) with the highest local average cost (or costs). Normally, this is a subtree or a subforest of the whole DA network tree. It is easy to find out the worst part (or parts) with a depth first search algorithm.

CONCLUSION

An automated system for the design of telecommunications distribution networks has been described. The design task is divided into three separate problems: selecting the central cable distribution point, designing the cable network layout, and allocating plant to the network structure. While many aspects of the problem can be abstracted in terms of standard optimisation methods, practical considerations significantly complicate the construction of a working system. Consequently, our approach is to combine heuristic and rule-based techniques with traditional optimisation.

This problem illustrates the importance using an internal representation for the design that reflects the salient features of the domain which constrain the structure of the final solution. We have demonstrated that without these constraints on the problem, the task of automating the design becomes intractable. How this information can be provided to the system from existing descriptions of the domain has strongly influenced the implementation of our system.

ACKNOWLEDGEMENTS

The permission of the Executive General Manager, research, Telecom Australia to publish the above paper is hereby acknowledged.

REFERENCES

- Balachandran, M. & Gero, J.S., (1990), *Role of Prototypes in Integrating Expert Systems and CAD Systems*, Report on *Design Process Recognition*, Dept. of Architectural and Design Science, University of Sydney, September.
- Coyne, R.D. & Gero, J., (1986), *Expert Systems that Design*, Proc. of the 1st Australian AI Congress.
- Gero, J., (1987), *Prototypes: a new schema for knowledge-based design*, Working Paper, design Computing Unit, dept. of Architectural and Design Science, University of Sydney.
- Gero, J.S. & Rosenman, M.A., (1989), *A Conceptual Framework for Knowledge-Based Design Research at Sydney University's Design Computing Unit*, in J.S. Gero (ed), *Artificial Intelligence in Design*, CMP/Springer Verlag, Southampton and Berlin.
- Kowalski, T.J. & Thomas, D.E., (1985), *The VLSI Design Automation Assistant: What's in a Knowledge Base*, Proc. of the 22nd Design Automation Conference.
- Liu, H., Wen, W. & Rowles, C.D., (1990), *Optimising Design of a Knowledge-based Design System*, IEEE Conference on AI Applications, February, 1991.
- Maher, M.L., (1990), *Process Models for Design Synthesis*, Report on *Design Process Recognition*, Dept. of Architectural and Design Science, University of Sydney, September.
- McDermott, J., (1981), *Domain Knowledge and the Design Process*, Proc. of the 18th Design Automation Conference.
- Mittal, S. & Araya, A., (1986), *A Knowledge-Based Framework for Design*, 5th National Conf. on AI.
- Parker, A.C. & Hayati, S. (1987), *Automating the VLSI Design Process Using Expert Systems and Silicon Compilation*, Proc. of the IEEE, 75(6).
- Pearl, J., (1984), *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, Reading Massachusetts.
- Rowles, C. & Leckie, C., (1988), *A Design Automation System Using Explicit Models of Design*, *Engineering Applications of AI*, Vol.1, December.
- Simon, H.A., (1969), *The Sciences of the Artificial*, Cambridge, Mass., MIT Press.

Knowledgeable assistants in design optimization

A. Gupta and P. J. Rankin

Philips Research Laboratories

Crossoak Lane

Redhill Surrey RH1 5HA UK

Abstract. This paper describes two cooperating knowledge based systems, prototyped as part of a CAD tool for electronic circuit designers. One knowledge based system operates in the electrical domain and the other in the mathematical domain. By basing the implementation of these systems on different knowledge representation schemes - rule based and object based - we have gained an insight in their relative merits and pitfalls. By coupling these systems with other CAD tools and developing an appropriate user-interface, we have tackled problems of how such systems can be successfully integrated and embedded into mainstream environments. In this paper we describe the lessons learnt and conclude with some recommendations for those considering the integration of knowledge based systems with other tools in large industrial projects.

1. INTRODUCTION

The design of most complex artifacts is an exploratory activity. Given a certain objective, parameters of the object are adjusted step-by-step until the design is either perfect or no further improvement is possible. Humans attempt simple design tasks without any assistance - for example, the layout of aisles and shelves in a supermarket or the layout of seating in a room. However, some problems involve the resolution of not one but various desirable objectives, some of which may be mutually conflicting. The relationship between parameters and objectives may also be more complex - a given parameter affecting one of the objectives adversely but another favourably. Faced with problems, the end product may be acceptable but far from optimum.

An inability to make all factors explicit, manipulate them and perform trade-offs, inevitably leads to inferior designs. Humans are particularly poor at efficiently searching through high dimensional design spaces of alternative parameter values. This is exactly the area where appropriate use of computers can give major assistance. Automation of this search for an optimum is possible provided the design problem is couched in mathematical terms. The computer can then be set loose on the problem and deliver a perfect result. If only this were the whole story. Inevitably, new

solutions bring new problems: designers are now expected to be mathematicians. Moreover, perfect mathematical solutions are not always acceptable in the real world unless the designers have explicitly made *all* their possible design concerns known to the automaton. Solutions must be sensitive to the design domain. There is a need therefore, both to make a design automation tool more accessible to designers (i.e. to map its input and output back into the domain they understand) and to steer the process to a solution which makes sense in the designer's eye.

From the above discussion, it is clear that the penetration of numerical optimisation methods in any engineering domain will be inherently limited without knowledge-based assistance. In agreement with MacCallum (1990), we believe that the role of knowledge-based systems in CAD should be to provide support for the creative process - an *amplifier* rather than a *substitute* for the actions of a human designer. Our requirements for knowledge encapsulation condense around two areas:

- i. A master designer never has the time or patience to voice all the constraints and factors which he/she takes into account during the iterative process of manual design. Low-level knowledge of the application domain is taken for granted from any assistant - whether in the form of a human apprentice or a software tool.
- ii. Experts in optimisation are too few to help every designer in person. Any numerical art in the way in which optimisation problems are formulated and run must be encapsulated in the software tool.

2. CoCo - CONTROL AND OBSERVATION OF CIRCUIT OPTIMISATION

Industrial circuit designers have to create circuits containing perhaps thousands of transistors, consuming the lowest possible power whilst also having the best electrical performance and being cheap to manufacture in an available technology.

Circuit optimisation is the process of *iteratively* changing some designable quantities (i.e. design variables), such that some (simulated) measure of performance is improved, subject to constraints (Brayton, 1981). The iterative changes are performed using numerical procedures (i.e. optimisation algorithms) which choose improved sets of design variable values. Typical design variables are resistors, capacitors, and transistor dimensions. For some time now, circuit optimisation techniques have demonstrated great potential both in reducing the design time and in significantly improving a manually-produced design, but have hardly been used in industry.

Study of the requirements for integration of numerical optimisation tools, together with an electrical CAD system, led to the CoCo concept shown in Figure 1. The CAD system contains a graphical interface which enables a designer to create a circuit diagram (a "schematic"), set up computations of the electrical behaviour with a circuit simulator, and view the results before changing the circuit topology or component values to improve the design. To this system was added:

- a. A general-purpose optimisation kernel, 'Optimize', to couple the circuit simulator to a number of numerical optimisation algorithms (Rankin and Siemensma, 1989)
- b. An on-line monitoring and diagnosis system capable of performing tests on data generated during the optimisation process
- c. Two knowledge-based sub-systems ('KBS')
- d. Considerable extensions to the user-interface (Colgan, Rankin and Spence, 1991)

The complete system is an exemplar of the potential for interactive optimisation. It comprises parts written in C and Pascal - 250k lines for the user-interface, 20k lines for the optimiser and 200k lines for the simulator. The knowledge-based components (shaded in Figure 1), written in Lisp and Smalltalk, form the focus of this paper.

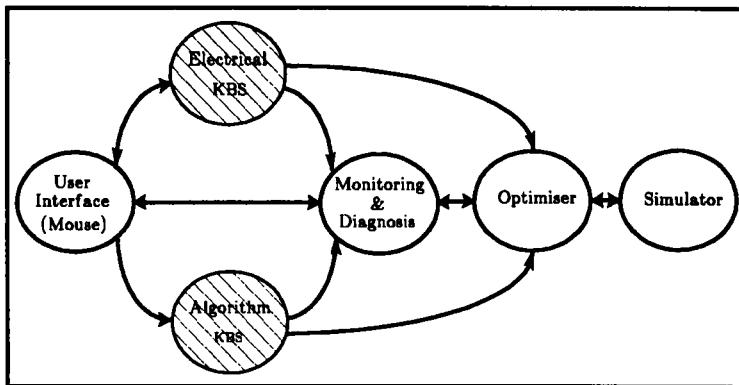


Figure 1 - The Structure of CoCo

A designer interacts with CoCo via the graphical user interface, 'Mouse' (Rankin and Siemensma, 1989) to draw a circuit, measure its simulated electrical performance and add specifications on the circuit responses. The electrical KBS acts as a watchdog, setting up monitors to ensure electrical 'sensibility' or physical realisability of the design. The "algorithm" or mathematical KBS advises the user on which optimisation algorithm is best suited to the problem at hand and on the adjustment of its parameters. Mouse then invokes the optimiser, Optimise, which handles the simulation requests. During optimisation, a novel user-interface, the 'Cockpit', provides an overview of the progress and the monitors are regularly tested. Any violations are signalled to the user via a special alarm display. When certain criteria terminate the algorithm, the designer can view any improvements made in the design as a function of algorithm iterations, or request a diagnosis from the mathematical KBS if the algorithm failed.

Initially, the overall architecture of CoCo was elusive. We set out not to develop a CAD product, but to write a *specification* for one through experience with an exemplar. Rather than develop deep knowledge bases, our aim was to arrive at a framework which would enable their coupling with the other subsystems. The prototyping of both of the knowledge-based assistants, and the development of the mainstream system of user-interface and optimiser proceeded in parallel. This posed problems, in that feedback from actual use of the assistants by designers on industrial circuits was delayed until the main system had been established and the two systems connected to it. Until then, their specifications could only be discussed with users already cognisant with optimisation.

3. THE ELECTRICAL KBS

When refining circuits manually, designers bring a considerable amount of domain knowledge and experience to bear on the problem. For example, if the designer adjusts say, a transistor length, this will impact on the transistor's performance. The designer will know how connected components will be influenced electrically and how to guard against adverse changes. Many 'rules-of-thumb' are employed during manual design iterations. If instead of the designer, the transistor dimension is adjusted by an optimiser, the designer must still guard against adverse repercussions in the same way. With a small circuit this is not a problem, but when industrial-size circuits containing large numbers of transistors are optimised, the designer is faced with a very laborious and error-prone supervisory task. Designers vary in their ability to appreciate subtle interactions and so the thoroughness of such checks, varies from designer to designer.

The algorithms underlying optimisation are domain independent and may be used in fields such as economic modelling or mechanical design. This lack of electrical "sense" on the part of the algorithm, introduces new problems discussed below.

Problems appearing before optimisation is started :-

- a) *The choice of design variables.* Component models used in simulation have many parameters. Some of these are under designer control - such as transistor widths and lengths. Others are determined by the manufacturing process and device physics - such as gate-oxide capacitances. Only those *designable* parameters which impact on the problem need to be adjusted. Their identification however, is not easy, making it difficult for designers to choose a minimum, orthogonal set of strong variables. A 'sensitivity analysis', (Brayton, 1981), could be invoked to assist in the choice.
- b) *Setting bounds on design variables.* Given a design variable, the optimiser may adjust it between any number - negative or positive - representable by the computer. Obviously, say, negative capacitances are not valid and even positive values must be constrained in ways sensitive to the technology and process limitations. Unrestrained exploration of values outside these ranges not only wastes computational time, but may even lead to failure of the simulator.

Problems appearing during optimisation :-

- c) *Maintaining component & cluster functionality.* When manually adjusting a circuit, designers ensure components remain in their original operating regions. As design variables are adjusted by an optimiser, the operating region of components may change - saturated transistors in a CMOS op-amp may be turned off, for example. Furthermore, where components form functional clusters - such as long tail pairs or current mirrors - the latter may deviate from their intended behaviour. Optimisation by a 'blind' numerical engine would soon deliver nonsense circuits: if the designer wanted to minimise power consumption, and specified no further constraints, the optimiser would turn all transistors off ! This is unlikely to be what the designer really intended. Many such electrical rules comprise what, for the designer, is 'common-sense'. The optimiser must be constrained to electrically sensible designs, or warn the designer as soon as violations occur, without requiring all the labour of setting up such monitors or 'demons' by hand.

Problems appearing after optimisation is complete :-

- d) *Interpretation of results.* As optimisation is a mathematical process, it is often not at all obvious to engineers, why a run may have failed or what (often very minor)

changes in specifications could give vast improvements. Inappropriate problem formulation or misinterpretation of results can lead expert designers to fail to benefit from optimisation even on the simplest circuit with undemanding specifications. Often, the designer's multiple objectives will conflict. For example in integrated circuit design, circuit speed, power and area may be mutually traded-off. Alternatively, the design space may allow for more than one circuit topology or combination of design variable values that satisfies requirements, but the best design will not be apparent. An optimisation algorithm normally gathers data (such as second-order derivative matrices) which could be used by an electrical KBS to reveal subtleties in the design. Facilities to extract this information and present it to the designer in a natural form are not yet implemented in the prototype.

3.1 Implementation of the Monitoring Tasks

Essentially, monitoring involves the checking of certain electrical parameters between optimisation iterations. Prior to invoking the optimiser, the electrical assistant scans the circuit and builds up a candidate set of components, clusters and nodes which require monitoring. The user may engage in a graphical dialogue to discard some entities from the set, so as to disable checks deemed inappropriate.

The electrical KBS then generates a list of the monitoring tests and simulation requests (for electrical data required in the tests) for the entities. Generated tests involve arithmetic expressions of results at the current iteration, the previous iteration or their initial values. These tests are invoked by procedures external to the KBS, thus avoiding the overhead of running the latter during optimisation. Upon detection of a violation of any test, the user-interface alerts the user, who can interrupt the optimisation, modify the problem, eg. to add explicit constraints to prevent the violation, and re-start optimisation. The user's view of this interaction is described elsewhere (Colgan, Rankin and Spence, 1991)

The electrical parameters which are checked depend on the device or cluster under consideration and will typically be simulated voltages or currents. The sub-system generates device-specific formulae appropriate to the check. This is best illustrated with an example, showing the monitoring of transistor states. (The same mechanism is used in other monitoring rules.) The following set of tests determine if a MOS transistor in an integrated circuit changes its operating region during the optimisation process from its operating region at the start. (Similar tests are used for monitoring integrated transistors in other technologies such as bipolar):

For every iteration :-

If $V_{gs} < V_t$

and $\text{Initial_Value_of_}V_{gs} > V_t$

Then conclude that the transistor is now in the 'off' state

If $(V_{gs} - V_t) > V_{ds}$

and $(\text{Initial_Value_of_}V_{gs} - V_t) < \text{Initial_Value_of_}V_{ds}$

Then conclude that the transistor is now in the linear region

If $(V_{gs} - V_t) = < V_{ds}$

and $(\text{Initial_Value_of_}V_{gs} - V_t) > \text{Initial_Value_of_}V_{ds}$

Then conclude the transistor is now in the saturation region

where V_{gs} and V_{ds} are the gate-source & drain-source voltages and V_t , the MOS threshold voltage is a technology dependent constant. The sequence of events (illustrated in Figure 2 below) is therefore as follows :-

- i. Prior to optimisation, the designer invokes the rule 'check-transistor-state'
- ii. The electrical sub-system requests the designer to confirm the selection of transistors it identifies for monitoring (see Figure 3)
- iii. The sub-system determines the correct threshold voltage
- iv. For each instance of these types of transistor, the sub-system generates the above monitoring formulae, referencing specific voltages for each transistor
- v. Again for each instance, simulation requests are generated, so that the simulator can deliver computed values for V_{gs} and V_{ds} after every iteration.

This done, the sub-system's tasks are completed. It plays no further role in monitoring transistors. After every iteration of the optimiser, the Monitoring & Diagnosis sub-system uses the latest voltage values computed by the simulator and by substituting these in the testing formulae, determines whether any transistor has changed state, and signals the designer via the user-interface.

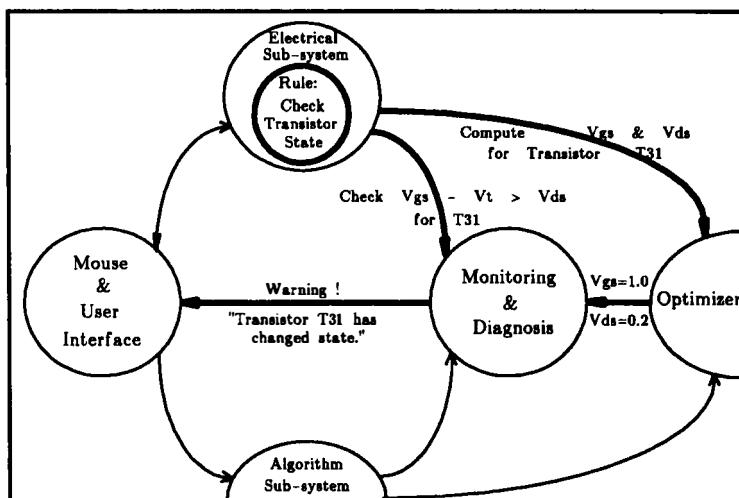


Figure 2 - Component Monitoring

Other checks, generated by rules in the electrical assistant, test that:

- o changes in nodal voltages between iterations remain less than a certain threshold
- o currents in transistor pairs or mirrors remain balanced within certain margins
- o total power consumption of the circuit remains below an upper limit
- o power dissipations within individual circuit components remain within the components' ratings for reliability
- o branch currents remain in the same direction as at the start.

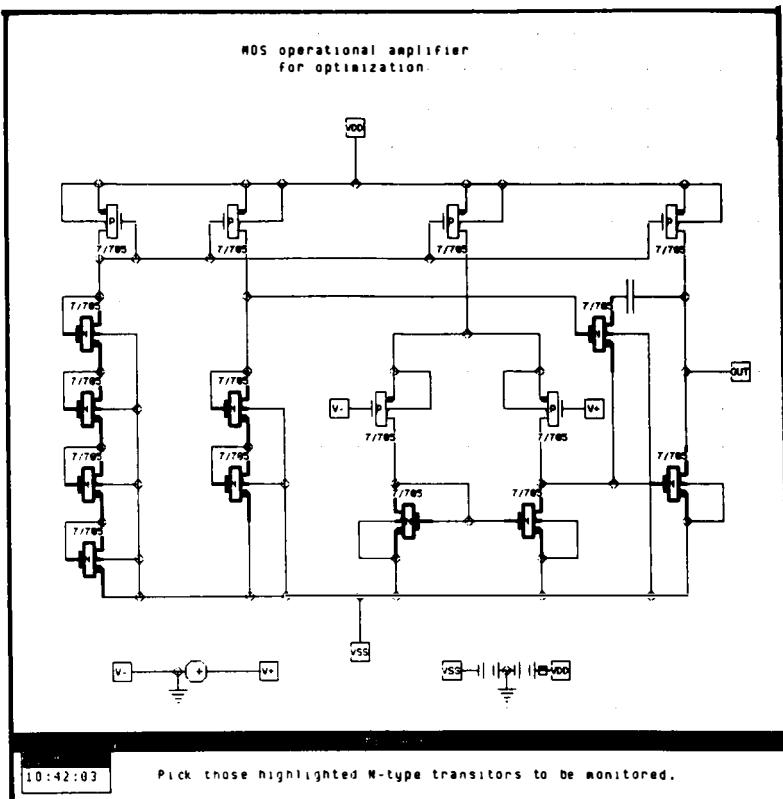


Figure 3 - User Confirmation

Note the analogy to an insurance policy - there is a compromise to be struck between the computational premium introduced by the additional simulation measurements needed for the tests and the security against unwanted electrical side-effects produced by the optimiser. Most of the rules require only additional dc quantities to be output by the simulator, and are thus cheap to compute compared with, say, a transient simulation. Notice also the use of these extra tests only for *monitoring* an optimisation problem which the designer has formulated explicitly. In principle the additional tests could be passed on as nonlinear constraints which must be observed, so augmenting the optimisation problem presented to the algorithm.

In summary, the electrical assistant acts as a junior technician, taking over some low-level tasks but never replacing the designer. Its most important role is to act as a watchdog, constantly monitoring the changing circuit looking for unacceptable electrical behaviour and informing the designer when occurrences are detected. To do this, it needs a considerable amount of domain-specific knowledge of the kind designers themselves possess. There is of course a spectrum of such electrical knowledge, from simple considerations such as the sensible bounds on the values of design variables, to primary design issues such as the prioritising of major design issues or rules about how to synthesise new circuit structures (see Figure 4). Our approach was to start by implementing rules which were simple both to acquire and express and which were expected to be of immediate benefit for increasing the success rate of optimisations submitted by a designer.

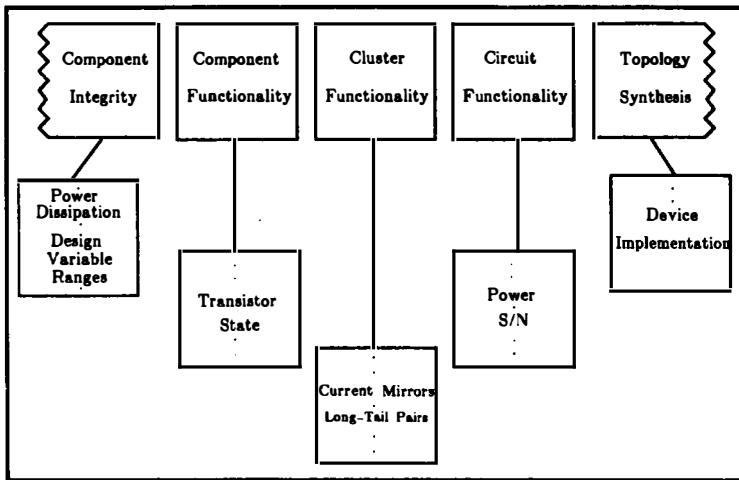


Figure 4 - The Spectrum of Electrical Knowledge

3.2 The Knowledge Acquisition Process

It is widely recognised that the development of knowledge based systems is severely limited by the knowledge acquisition process. With the modest aim of developing an assistant for circuit designers, *not* a synthesis system that replaces them, we restricted the scope of the system and the type of knowledge required. The system needs low level knowledge about transistors, resistors etc. It needs topological information about each generic component - such as the number of component terminals, their type (i.e. whether input *or* output), and their class (i.e. whether the terminals can be interchanged). To detect higher level structures, it needs to know how generic clusters are formed, in terms of the interconnections of the constituent components. Finally, to monitor them, it must know the electrical formulae applicable to the tests.

3.2.1 Knowledge Elicitation - Interviews with Designers

The first series of interviews, took place over a three month period and averaged a few hours each. The fourteen designers interviewed came from our organisation and from our collaborators, Imperial College, UK. They were drawn both from academic and industrial research *and* from advanced industrial development. Designers were chosen according to their expertise and application area. These initial interviews were exploratory, centering around case-histories of projects on which the designers had been engaged; they were unstructured and not audio-taped. Notes were made during the interview, and the findings documented and verified, to form the basis for defining the tasks required of the electrical assistant.

Through this process, we began to understand the circuit-design domain and become conversant with the vocabulary. The knowledge was formulated in terms of if-then rules and a taxonomy developed which classified each rule according to various orthogonal criteria, such as their relevance to optimisation, whether they were low-level or demanded much associated information, and their ease of

implementation. An important distinction was made between knowledge and meta-knowledge which enabled us to separate the electrical knowledge *within* rules from the control knowledge about *when* to apply them. A rule from each class was selected for implementation to be sure that the system architecture was sound and so that the full span of the assistant's capabilities could be demonstrated.

Subsequent interviews were structured and audio-taped. The most fruitful results came from discussion around specific circuits and exploration of the stages of manual optimisation. Designers were encouraged to think in terms of the fundamental circuit design problems, without dwelling on the use of a computer. Otherwise, the scope of the discussions would have been limited to the capabilities designers perceived in their present CAD tools. Minimal notes were made, relevant ideas being noted conveniently on circuit drawings. As Hoffman (1989) has noted, ten hours of interviews with an expert planner of aircraft schedules may take over a hundred hours to transcribe and analyse. Therefore, the audio-tapes were *not* transcribed but simply played-back, in the designer's absence. Important points were noted and as before, documented findings were verified with the designer.

3.2.2 User Trials & Protocol Analysis

When Mouse became sufficiently robust, structured user trials were organised. Designers were asked to solve their own optimisation problems using Mouse and were encouraged to think aloud; the session being audio-taped and supplemented with screen-dumps and questionnaires. When designers encountered problems concerning the formulation of an electrical design as an optimisation problem or the running of the optimiser, we helped personally and recorded the assistance given. Thus we acted as the very "assistants" we were prototyping. By recording all such cases of personal assistance and analysing their use of Mouse via *protocol analysis*, (Colgan 1991), we identified areas where a knowledge based assistant could help. Justification was again found for some of the rules acquired in the earlier interviews. This formed useful input for the design and specification of the electrical assistant.

3.2.3 Literature

The electrical assistant requires much low-level knowledge which can be collected from graduate level text-books and manufacturers' data-sheets. The topological information required for the automatic recognition of clusters came from circuit drawings, while transistor threshold values were gathered from data-sheets.

3.3 Knowledge Representation

Both domain and meta knowledge are coded as if-then rules. The tool used in developing the rule base, was VERA, (Kosteljik, 1989), on an Apollo workstation. VERA enables rules about circuits to be written hierarchically i.e. rules are able to call other rules and can also call code in Common Lisp, the language VERA is written in. It is specially-designed to parse very large integrated circuits to rapidly check their electrical, topological and timing properties, acting as a design critic to reduce the need for simulation. (General-purpose expert system shells are totally inadequate for this application.) Figure 5 shows VERA's internal architecture and the enhancements we made. As explained by Kosteljik (1989), 'primitives' are the building-blocks of rules and are analogous to the operators of any programming

language. Match primitives are based on pattern-matching, searching for structures in a circuit or for the validity of facts. Action primitives allow changes to be made to the circuit topology, such as the removal of electrical components, the expansion of clusters into constituent components, etc. Rules are written in if-then form, match-primitives being used in the "if" part and action-primitives in the "then" part. Some of the match-primitives are :-

- o *recognise* - finds all occurrences of a specified cluster
- o *get-attribute* - returns the values of specified attributes of elements or nodes
- o *fork* - finds clusters of elements connected in parallel

Some of the actions-primitives are :-

- o *abstract* - replaces a collection of elements by one cluster
- o *expand* - the reverse of abstract, it replaces a cluster by its constituent elements
- o *create-element* - adds a new element to the circuit

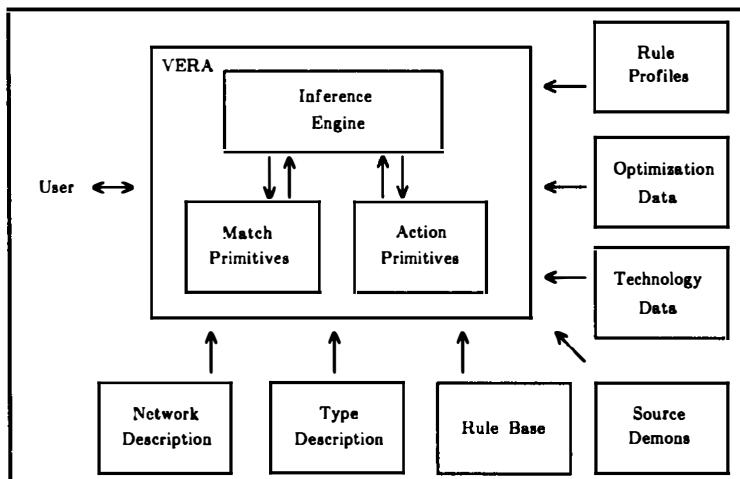


Figure 5 - The Electrical KBS

An application using VERA must supply at least three distinct forms of data - a network i.e. circuit description, a type description and a rule-base. The type description file defines the external terminals of components & clusters and the properties of these terminals. It would for example, define a resistor to have two terminals and state that it is invariant under topological transformations. It would also define new clusters by stating how previously defined components are inter-connected. In this regard, the type description must be regarded as forming part of the knowledge base (as distinct from the rule-base). To the original system we added :

- a. *source demons*, templates of monitoring formulae for components and clusters
- b. *optimisation data*, bounds on the values design variables may take
- c. *technology data* such as transistor threshold values and power rating values

- d. *rule-profiles* which group rules according to function, such as "low-noise circuitry" rules and "high-gain circuitry" rules
- e. *restriction of the scope of rules to circuit areas* so that the designer can bring the knowledge-base to bear on part of the circuit, excluding other optimised parts
- f. *user-confirmation* so that the designer is able to accept or reject the advice of the electrical sub-system.

The following rule fragments give an indication of the rules in the knowledge base.

If technology is CMOS
 and a design variable is layout area of a double-polysilicon capacitor
 then suggest limits of $10 \mu\text{m}^2$ and $100000 \mu\text{m}^2$

Rule :- Constrain-Design-Variables

If current-mirror identified
 then check currents in mirror are proportional to emitter areas
 if not within 5% warn user

Rule :- Monitor Current-Mirrors

The encoding of the second rule in the VERA language is shown in Listing 1.

```
(define-rule monitor-current-mirrors nil (range)
  ((el-a el-b inpa inpb modela modelb demon) nil)

; The match part ...
; Find all current mirrors in the circuit

((recognize ((el-a 'idl)(el-b 'id2)(inpa 'in1)(inpb 'in2))
            ((network-of 'cur-mirror-1)
             (restrictions-of 'cur-mirror-1 1) 'no-network-list))

(get-attribute (el-a modela) ('element '(model)))
(get-attribute (el-b modelb) ('element '(model)))

; Discard potential clusters with different transistor model types
(test (modela) ('equal (instance-of modela) (instance-of modelb)))

; Get and initialise appropriate demon from source demon file ...
(instantiate (demon) ('(,(get-cm-demon (instance-of el-a) (instance-of el-b]

; The action part ...
; Set up a message to tell the designer when current-mirrors fail ...
(message ()
  ('( !Current in the transistors | ,(instance-of el-a) | and | ,(instance-of el-b) |
    mis-match by more than 5% | )))))
```

Listing 1 - The Monitor Current-Mirrors rule

4. THE MATHEMATICAL KBS

The use of optimisation involves several tasks requiring mathematical knowledge which is not of direct concern to the designer. Examples are the choice of a suitable numerical method (the algorithm) and appropriate instantiation of its parameters. These tasks impose the burden of acquiring considerable mathematical knowledge additional to that required for circuit design. The need to acquire such knowledge at best distracts the designer from the primary task of design but more often dissuades the designer from venturing to use circuit optimisation tools at all. It is these difficulties that the mathematical sub-system aims to overcome. It can be argued that what is needed is a "mathematical assistant" to whom the circuit optimisation problem is handed and who then returns a set of component values which meet the desired objectives. Six activities in the optimisation process which the mathematical sub-system should perform are (Arora & Baenziger, 1986) :

- a. *Algorithm selection* - from a library according to the problem's properties
- b. *Parameter values assignment* - using knowledge of the algorithm and of the design problem's mathematical properties
- c. *Problem transformation* - application of transformations to variables and other functions to lower numerical error and make the algorithm more effective
- d. *Results monitoring* - analysis of algorithm output after every iteration to check on possible failures, slow convergence and bad conditioning
- e. *Diagnosis* of algorithm termination conditions
- f. *Restarting or reselecting algorithms* - if an algorithm has failed to find an optimum or is converging very slowly it may be possible to adjust some algorithm parameters and re-run the algorithm, starting from the best point found; alternatively it may be necessary to try another algorithm.

These are explained below in more detail:-

4.1 Algorithm Selection

There are many commercially available algorithm libraries such as those from Harwell, NAG and CERN, all of which contain implementations of most of the commonly used optimisation algorithms. We decided to enable the designers' use off-the-shelf algorithms, developed by the Numerical Algorithms Group (NAG, 1988). Each algorithm is designed to solve a special type of problem. Considerable documentation, written in mathematical terms explains how an algorithm can be chosen, via a decision tree, according to properties of the problem, such as the number of design variables, whether the variables are constrained (i.e. only allowed to take values from a predetermined set), whether computer memory is at a premium, whether function calls are expensive etc. A subset from the range of NAG algorithms to match the diverse optimisation problems posed in circuit design was selected. These allowed access to constrained and unconstrained sequential quadratic programming, conjugate gradient and quasi-Newton/least-square methods.

4.2 Algorithm Parameter Values

The parameter value assignment task is one which shows the importance of this type of 'assistant' quite clearly. The numerical strategy employed by the algorithm is 'tuned' via its parameters. Three types of algorithm parameters need to be assigned values before an algorithm can be run. These are algorithm-related, problem-related and machine-related parameters. Algorithm-related parameters allow the user to set details such as the accuracy of linear searches, maximum linear search step size and maximum number of iterations. These are assigned initial values according to known theoretical rules and accumulated experience. Problem-related parameters such as the number of variables, constraints and constraint types are derived from the user input. Machine-related parameters are quantities such as machine accuracy.

4.3 Problem Transformation

The wide range of values taken by design variables frequently causes numerical problems in optimisation: they may well be measured in different physical units. Simple linear transformations, eg. so that all the initial values are unity, usually help the numerical conditioning of the problem. Logarithmic transformations of circuit component values often produce a closer approximation to a quadratic problem, aiding algorithms which are based on this assumption. When attempting multiple-objective optimisation, the individual circuit response requirements measured in different electrical units need to be balanced. Here a normalisation based on dividing by a range of acceptability, indicated by the user for each circuit requirement, may be used.

4.4 Examination and Diagnosis of Results

On completion, most algorithms return a termination code, indicating the reason for termination and the quality of the solution obtained. The algorithm writers supply explanations of termination conditions which are couched in mathematical terms. These may indicate that a constrained problem does not have a feasible solution, that poor numerical conditioning (due to redundancy in the variables or poor scaling) prevents further progress, that the problem has been solved within the convergence criteria set, etc. In the case of NAG library routines, an error flag points to printed documentation, but a certain level of familiarity with the algorithms in particular and optimisation in general is assumed. The 'mathematical assistant' captures this diagnostic and remedial information and interprets it for the designer. It may for example, recommend a 'warm restart' of the optimisation, if this is appropriate.

4.5 Restarting or Reselecting Algorithms

An interrupted algorithm may be restarted after rescaling the problem, changing the set of design variables, adjusting some parameter or choosing a new starting point. It is possible to have more than one of these conditions active simultaneously. Which action should be initiated depends on the analysis of the results obtained thus far. For example an ill-conditioned hessian may prompt rescaling of the problem; a set of consecutive iterations limited by the maximum step length may lead to increasing the parameter controlling the maximum step length. If none of the available rules for restarting the algorithm are applicable then, if the algorithm selection procedure has

recommended more than one suitable algorithm, an alternative one may be tried.

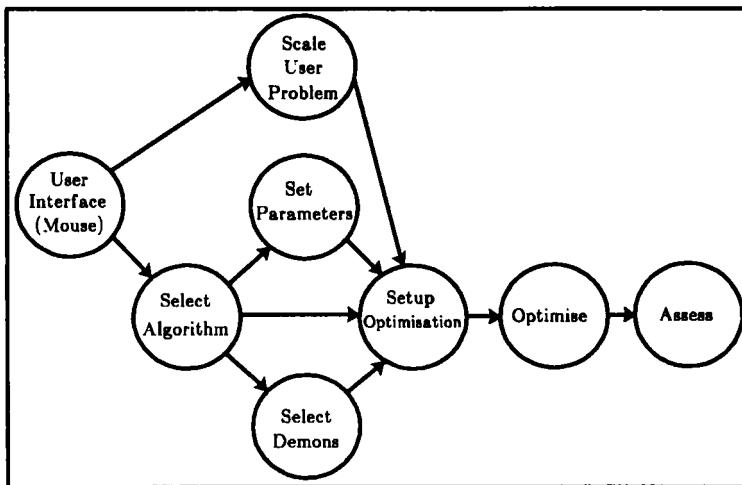


Figure 6 - Functional Decomposition of the Mathematical KBS

We describe below, the prototyping of a mathematical KBS that acts as an assistant to the designer and takes over many tasks in a transparent manner i.e. *without* requiring any interaction, other than its activation as a kind of 'help' function. The tasks covered by the mathematical assistant are shown in Figure 6. The following sections describe the knowledge acquisition process and its representation in Smalltalk (Goldberg, 1989).

4.6 Knowledge Acquisition for the Mathematical KBS

Two sources for the knowledge were employed: literature and interviews. The main source for the literature study was the supplied documentation (NAG, 1988). On trying to encode some of the rules, various ambiguities and imprecisions emerged, which were resolved by consulting optimisation experts.

Thus far, we have concentrated on acquiring and coding algorithm dependent mathematical knowledge which applies in any application. Future effort will concentrate on acquiring *domain dependent* mathematical knowledge, empirically. There is a pressing need to do extensive numerical experimentation with real optimisation problems. For example, the convergence rate should be measured as a function of factors such as algorithm parameter settings. Such experiments would lead to the establishment of improved methods of scaling and transformation, better default algorithm parameters tuned to the characteristics of circuit problems and more strategies for dealing with failures. Tseng and Arora (1988) have shown the value of gathering statistical data. Their recommendations are applicable for obtaining better domain dependent data: an area for us to pursue in the future.

4.7 Knowledge Representation

The structure of algorithm knowledge is best organised so that related topics can be grouped together according to their reliance on some common numerical strategy. For example, the knowledge about the algorithms' parameters or the knowledge about the algorithms' failure diagnosis mechanism can be clustered. We represent algorithm knowledge as objects. Object-oriented knowledge representation enables one to represent knowledge about objects *in* those objects, in terms of data and operations on the data. Relations between objects are represented in the form of a directed-acyclic graph; objects *inheriting* data and operations from more general objects. The object-oriented paradigm has been exhaustively described in the literature.

We used the ParcPlace Systems Smalltalk-80 implementation of the Smalltalk object-oriented programming environment (Goldberg, 1989). The algorithm sub-system is organised into the following classes : Algorithms, Nodes and Results. Class Nodes represents nodes in the algorithm selection decision tree and is explained later, in this section. Class Results contains the results for all optimisation iterations. Another class, namely UserProblem, maintains knowledge about the current optimisation problem i.e. the number of variables, the number and types of constraints etc. We also envisage a Simulator class to encapsulate properties of different simulators, such as their accuracy. The Server, Comms and MouseRequests classes, implement the communication protocol with Mouse and enable the functionality of the mathematical sub-system to be accessed. Figures 7 to 12 illustrate the hierarchy & organisation of the objects. The Smalltalk method names shown (only a small selection of those actually implemented), give a flavour of the functionality provided.

```
Object
Algorithms
Newton
ConjugateGradient
ModifiedNewton
QuasiNewton
SQP
```

Figure 7 - Hierarchy for class SQP

```
Object subclass: #Algorithms
instanceVariableNames: 'parameters name'
classVariableNames: 'ChosenAlgorithm CondHzLargeThreshold CondHzSmallThreshold
CondTLargeThreshold'
Method category 'recovering' :
method names :- converged doNotRestart determineNewStartingPoint handleFailureIfAny
illConditionedHessian
Method category 'algorithm selection' :
method names :- selectAlgorithmsFor:
```

Figure 8 - Organization of class Algorithms

```

Algorithms subclass: #SQP
instanceVariableNames: "
classVariableNames: "
Method category 'recovering' :
method names :- alterUTCAHLF applyConvergenceTests applyTheFourConditions
bothStepAndItQPEqualOne condHzLarge condHzSmall condTLarge
determineNewFunctionPrecision determineNewLinearTolerance
finalNormGzLessThanAtStartPoint findRedundantLinearConstraints

```

Figure 9 - Organization of class SQP

Figure 8 shows how the place-holders for data, i.e. instance- and class- variables, common to all algorithms, are provided by the Algorithms class which also provides code, i.e. methods, for operations on those data. The variables and methods are inherited by sub-classes of the class Algorithm, which in turn provide their own particular methods (shown here for class SQP, Figure 9) for algorithm parameter setting, self diagnosis and reselection. Classes involved in the algorithm selection process are illustrated below, omitting interaction with class UserProblem for brevity.

```

Object subclass: #SelectionTree
instanceVariableNames: 'root'
classVariableNames: "
Method category 'tree making' :
method names :- buildTree makeAlgorithms
Method category 'algorithm selection' :
method names :- selectAlgorithmsFor:

```

Figure 10 - Organisation of class SelectionTree

```

Object
Nodes
ComputationalCost
Constraints
Discontinuities
FirstDerivatives
SimpleBounds
StoreSize
SumOfSquares

```

Figure 11 - Hierarchy for class Nodes

```

Object subclass: #Nodes
instanceVariableNames: 'offspring'
classVariableNames: "
Method category 'accessing' :
method names :- setVarOffspringYes:andNo:
Method category 'algorithm selection' :
method names :- selectAlgorithmsFor: selectAllAlgorithmsFor:withFirstOffspringAt:

```

Figure 12 - Organisation of class Nodes

The process of algorithm selection involves the interrogation of the mathematical problem to determine properties such as the number and type of constraints etc. We represent each property as a sub-class of class Node (Figure 11), and create a *decision tree* connecting these classes. Each class has two offspring classes, (as can be seen from the description of the Node class in Figure 12), the leaf classes having offspring that are algorithm instances. (Note, this tree must not be confused with the directed-acyclic *graph* representing the hierarchical organisation of the classes. Unlike the graph, the offspring classes in the tree do *not* inherit from their parent class.) Each class provides methods that operate on its own data and the firing of these methods result in one of the offspring test-methods firing and so on. By traversing the tree in this depth-first fashion we reach a leaf class and therefore instances of algorithms appropriate for the mathematical problem.

If, on incorporating knowledge of a new algorithm in the assistant, we find that its selection requires more information on the mathematical problem; or if we wish to make the algorithm selection process more sophisticated, we only need to create new classes that represent more features of the problem (these would be sub-classes of class Nodes); provide methods for the classes and connect them to the tree.

5. COUPLING THE SYSTEMS

A fundamental problem with the design of knowledge based systems is one of apportioning control. It was apparent after numerous interviews with circuit designers that they needed an automated junior electrical "technician" but an automated mathematical expert. They wished to interact with the former to oversee its activities, but wanted the latter to take over some tasks completely, without even giving explanations of its inferences. These requirements provided a basis for the design of the overall CoCo system.

However, we faced other problems arising from the languages in which the knowledge-based systems were implemented, namely Lisp and Smalltalk, and that in which Mouse was implemented, namely Pascal. Though it is easy for Lisp and Smalltalk to call programs written in other languages via the foreign-function interface and user-primitives respectively, the reverse is not so straight forward. These languages are in fact sophisticated programming environments and do not readily relinquish control. We therefore implemented a client-server protocol for the coupling between Mouse and these two sub-systems, based on the exchange of files.

On CoCo startup, Mouse spawns processes in which both the knowledge based systems are started. Each execute self-initialisation procedures which consist of an infinite loop looking for a special file, reading, parsing and validating the command it contains, deleting the file, executing the command and writing the result to another special file. Four such files, each uniquely named, enable conflict free, asynchronous communication between Mouse and the sub-systems. Mouse writes a command to a file which the electrical sub-system is looking for and which it reads and deletes. The electrical sub-system writes its reply to another file, which Mouse is searching for, and which it reads and deletes. The same mechanism is implemented in the mathematical sub-system. When the sub-systems need to communicate with the user, messages to Mouse activate menus and graphical dialogues in a style consistent with the rest of the user-interface. We have found that though such a simple system is somewhat slow, this has not been a problem; our aim in research being to explore a

novel framework in which intelligent circuit design assistance can be provided.

It is interesting to contrast the user-interface requirements of the two knowledge-based systems. On the one hand, the electrical assistant works in the user's own electrical design domain, it does not know all the facts in that domain, so needs to interact with the user. On the other hand, the mathematical assistant works in a foreign domain - dialogue with the user would be inappropriate and anyway not comprehensible by the user.

6. SOME LESSONS

Use the right tool for the job

Our motivation in designing and implementing CoCo has always been to explore how intelligent optimisation assistance can be provided. Issues relating to efficiency were necessarily secondary. Rather than off-the-shelf expert system shells, we chose software development tools on the basis of their prototyping power and knowledge representation scheme. Both Lisp and Smalltalk are mature programming environments in their own right, as well as being implementations of different programming paradigms. We found the richness of the tools they provide give more power to the programmer's elbow. Though there is a learning curve in using them varies from person to person, it should *not* be over-estimated.

Match the nature of the knowledge with its representation

Domain knowledge *per se*, is of many kinds. It is important to understand the kind of knowledge one is trying to capture and to represent it appropriately. Having decided that the electrical sub-system would be a "technician" we wanted to encode a subset of circuit design knowledge, namely procedural knowledge, capturing condition-action forms. This is best represented in the form of rules. Having decided that the scope of the mathematical sub-system would be to take over all mathematical tasks, it was apparent that its knowledge representation scheme would have to be more sophisticated. It was also clear that the nature of algorithmic knowledge was such that there was a lot of commonality between different algorithms. The object-oriented representation was therefore chosen. This enables one to store deep knowledge, compared to rule-based systems which are shallower in their representation.

Develop and evaluate prototypes

It is very important to get feedback from the end-users. However, one must strike the right balance between showing them a sophisticated system which is too novel for them to use and one which only offers very little extra. Most likely, both cases would be evaluated unfavourably. In our project we engineered a new CAD system for circuit optimisation in parallel with prototyping its knowledge-based assistants. The long delay before the KBS prototypes could be assessed by users on real industrial circuits carried the risk that they might not prove to be suitable.

Engineer the prototypes

Large projects such as CoCo have to be managed carefully. From the outset, we decided to engineer the prototypes. It is vital to make all factors explicit in the design phase, otherwise fundamental problems manifest themselves very late, costing much investment in development time. We used the Yourdon methodology, (Page-Jones, 1980), to create data-flow & state-transition diagrams and structure charts. This

enabled us to express and document design issues in a uniform style and formed a basis for discussion amongst project members.

Don't replace the expert

Attempting to automate as complex an activity as analogue circuit design would be presumptuous and bound to fail. It is salutary to reflect that it may take twenty years to become a master designer. In fact, it is far better to adopt an evolutionary approach and provide assistance with those tasks that are fully understood. In this way, one can first get the confidence of the end-user and then incrementally refine the system.

7. CONCLUSIONS

Some ambitious artificial-intelligence projects aim to automate high-level tasks that humans perform. In the field of design, the poorly-understood process by which creative design is done, and the range of knowledge of diverse manufacturing, testing and marketing issues which designers deploy make complete automation unlikely for a long time to come. Much more success will result from a judicious augmentation of human skills - an amplification of human endeavours rather than a substitute. In this paper we have described just such an approach. Numerical methods offer great potential for relieving designers of difficult and tedious searches through design parameter space, while the major creative changes in the concept are best left to the designer. Significant improvements in the reliability and usability of large, mainstream CAD systems can result from augmentation with two kinds of knowledge-based systems. These carry much lower risks than an attempt at complete design automation - there is no threshold of intelligence that the system must possess, below which it is useless.

We have explained how such knowledge-based assistance can be provided in the domain of analogue electronic circuit design and demonstrated its potential. Two assistants have been prototyped - one for containing simple electrical knowledge, the other for hiding mathematical knowledge. The roles of these sub-systems differ considerably - one absorbs low-level tasks in the user's domain, the other substitutes for (expert) knowledge in a foreign domain. By letting their implementation be based on the appropriate knowledge representation paradigms we were able to build the systems relatively easily. Both the object-oriented and rule-based approaches chosen were successful. The architecture chosen reflects a decision to try to keep as much knowledge processing off-line from the computationally intensive optimisation process, the former being confined to pre- or post-processing functions. The major bottleneck left is now the knowledge acquisition process. This will remain the case until we fully understand how to *engineer* this process.

8. ACKNOWLEDGMENTS

Thanks go to all contributers, in particular colleagues at our labs., namely: M. Burchell, C. Meewella and K. Hollis; academic collaborators, L. Colgan and Prof. R. Spence (Imperial College); the team that developed Vera, A. Kosteljik et al (Philips Research Labs., Eindhoven); M. Brouwer-Janse, L. de Jong and P. v. Loon (Philips CFT, Eindhoven); and D. Mehandjiska Stavreva (Inst. of Elec & Mech. Engineering, Sofia, Bulgaria). Lastly, we are indebted to the designers who took part in the

knowledge elicitation exercise.

REFERENCES

- Arora, J. and Baenziger G. (1986). Uses of artificial intelligence in design optimisation, *Computer Methods in Applied Mechanics and Engineering* **54**: 303-323.
- Brayton, R. et al. (1981). A survey of optimisation techniques for integrated circuit design, *Proceedings IEEE*, **69**(10): 1334-1362.
- Colgan, L. and Spence, R. (1991). Cognitive modelling of electronic design, *Proceedings of AI in Design*, June 1991, Edinburgh (to appear).
- Colgan, L., Rankin, P. and Spence, R. (1991). Steering automated design, *Proceedings of AI in Design*, June 1991, Edinburgh (to appear).
- Goldberg, A. and Robson, D. (1989). *Smalltalk-80: The Language*, Addison-Wesley, Reading, Mass.
- Hoffman, R. (1989). A survey of methods for eliciting the knowledge of experts, *SIGART Newsletter, Special Issue on Knowledge Acquisition* **108**: 19-27.
- Kosteljik, A. P. (1989). VERA, a rule based verification assistant for VLSI circuit design, *Proc. VLSI'89 Conference*, pp.89-99.
- MacCallum, K. J. (1990). Does intelligent CAD exist? *Artificial Intelligence in Engineering* **5**(2): 55-64.
- Numerical Algorithms Group FORTRAN library Manual—Mark 13 (1988). Numerical Algorithms Group Ltd, Oxford, UK.
- Page-Jones, M. (1980). The Practical Guide to Structured System Design, Yourdon Press.
- Rankin, P. J. and Siemensma, J. M. (1989). Analogue circuit optimisation in a graphical environment, *Proc. IEEE ICCAD-89*, Santa Clara, CA, pp.372-375.
- Tseng, C. H. and Arora, J. S. (1988). On implementation of computational algorithms for optimal design 1 and 2, *International Journal for Numerical Methods in Engineering* **26**: 1365-1382 and 1383-1402.

Qualitative models in conceptual design: a case study

B. Faltings

Laboratoire d'Intelligence Artificielle
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne (EPFL)
MA (Ecublens), 1015 Lausanne Switzerland

Abstract Design is an important but little understood intelligent activity. *Conceptual* design is the transformation of functional specifications to an initial concept of an artifact that achieves them. Human designers rely heavily on the use of sketches, which can be thought of as qualitative models of a device. An appealing model of conceptual design is that of a mapping from *qualitative* functional specifications to a corresponding *qualitative* object model.

As a case study, I have investigated this model for the conceptual design of part shapes in elementary mechanisms, such as ratchets or gears. I present qualitative modelling formalisms for mechanical function, the *place vocabulary*, and for shape, the *metric diagram*. However, I show that qualitative functional attributes can not be mapped into corresponding qualitative attributes of a device that achieves them, and consequently that qualitative function can not be computed based solely on a qualitative object model. Only a significantly weaker functional model, *kinematic topology*, can be derived based on qualitative object models alone.

This result means that at least in mechanical design, sketches do not represent a *single* qualitative model, but must be interpreted as a *set* of possible precise models. Each step in the design process then refers to a particular precise model in this set. This novel interpretation of the use of sketches suggests alternatives to the popular model of conceptual design as a symbolic mapping of functional into object attributes.

ELEMENTARY MECHANISM DESIGN

Designing an artifact is a complex intellectual process of much interest to AI researchers. Most research in intelligent CAD systems has focussed on *detail* design, the adaption of an initial concept to precise specifications. Little is known about the process of *conceptual* design, the transition between functional specification and concept of an artifact that achieves it. As the artifact is only vaguely defined, conceptual design heavily involves qualitative reasoning and representations.

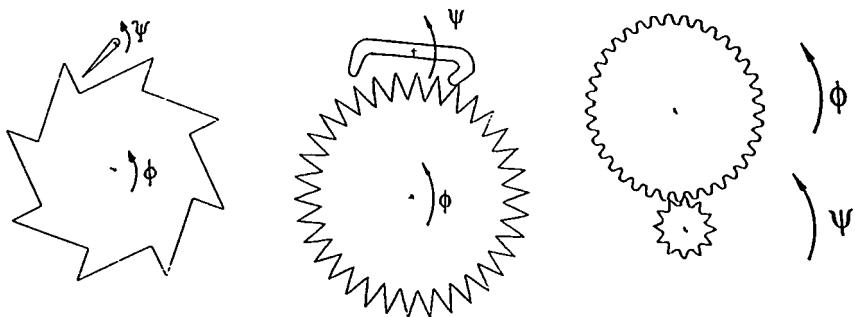


Figure 1: *Examples of higher kinematic pairs: a ratchet, a clock escapement, and gearwheels.*

I am investigating the problem of conceptual design for a specific and particularly intriguing sub-problem: the analysis and design of *higher kinematic pairs*, often called *elementary mechanisms*. A higher kinematic pair consists of two parts which achieve a kinematic function by interaction of their shapes. This class contains most of the “interesting” kinematic interactions, such as ratchets, escapements or gearwheels, as shown in the examples of Figure 1. Important properties of kinematic pairs are that each object has one degree of freedom (rotation or translation) only, and that their interaction can be modelled in two dimensions. The domain of kinematic pairs is ideal for studying design methodologies and representations because it is very rich in possible designs, but is also contained enough so that it can be studied in isolation without the need for extensive assumptions.

Design is a mapping of functional specifications to an actual physical device. A function of a mechanism is its behavior in a particular environment: a ratchet blocks a particular rotation, a Peaucellier device transforms circular motion of one point into straight line motion of another. A functional *specification* is a condition on a function of the device. A numerically precise specification of all functions of a device is often overly restrictive, and it would be pure coincidence if there actually existed a mechanism which satisfies them. In practice, the functional specifications are intentionally *vague*: they admit a whole range of numerical values for functional properties. In order to exploit the possibilities admitted by this vagueness, it must be represented in the functional specification, using a *qualitative* functional model. In this paper, I describe how *place vocabularies* ([Faltungs, 1987a, Faltungs, 1990]), a qualitative functional modelling language for elementary mechanisms, can form the basis for formulating qualitative functional specifications.

A common belief among designers is that the design process has a hierarchical structure: a very rough conceptual design is done first, and successively more and more details are filled in. For example, architects do their first sketches with a very coarse pencil so that they are not tempted to fill in too much detail. This suggests

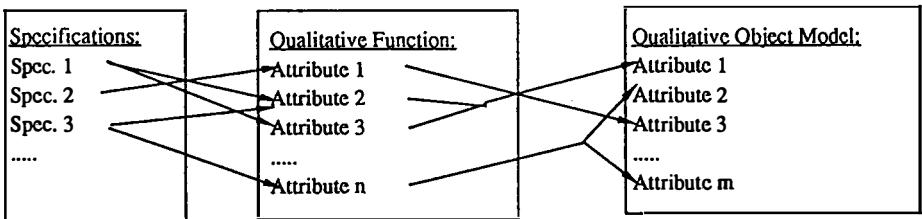


Figure 2: An appealing model of conceptual design: design specifications determine characteristics of a qualitative functional model, which are then mapped to attributes defining the qualitative object model. Note that the mapping can involve combinations of attributes.

that at the stage of conceptual design, the design object should be modelled qualitatively. Conceptual design would then be a mapping from the qualitative functional model to the qualitative object model, as shown in Figure 2. This is the model of design implicitly or explicitly assumed by many researchers in intelligent CAD ([Yoshikawa, 1985, Tomiyama et al., 1989, Williams, 1990]). For exploring this model of conceptual design, I define a qualitative modelling formalism for shape, the *metric diagram*. The metric diagram is designed to represent the information contained in sketches used by human designers.

The most important problem in conceptual design is then the mapping between the functional and object models, i.e. between place vocabularies and metric diagram. It turns out that this mapping requires quite precise information (or assumptions) about the metric dimensions of the parts of the device. Only very weak functional representations, such as kinematic topology ([Faltings et al, 1989]), can be based on qualitative object models. I argue that this problem is not a result of the particular representations used in this paper, but is a general property of the domain of elementary mechanisms.

If precise metric dimensions are required in the design process, qualitative models of the artifact are not enough. As a result, I will argue that for most conceptual design problems, the model of Figure 2 should be replaced by the one in Figure 3. In this model, first-principles reasoning relates the *precise* dimensions of the object model to the qualitative functional characteristics of the device. The new model offers another, less obvious explanation of the fact that human designers like to use sketches: in conceptual design, the precise dimensions have to be changed very often, and a sketch can be reinterpreted as different (precise) models rather than having to be redrawn each time a dimension is changed! The results of my research provide strong evidence for this alternative interpretation.

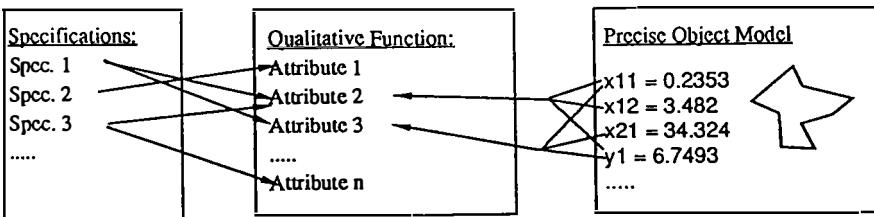


Figure 3: *A more adequate model of the conceptual design process: attributes of a precise object model map to a qualitative functional model. Conceptual design involves frequent modifications of precise dimensions.*

What is a qualitative representation?

As there is no generally agreed on definition of what makes a model “qualitative,” it is necessary to define the term for the purposes of this paper. The word “qualitative” is derived from “quality”, which is a synonym of “property”. Consequently, a qualitative model is composed of properties of the modelled domain, represented as predicates in first-order logic. More precisely, a qualitative model is

a model of a system in first-order predicate logic where symbols correspond to independent entities of the modelled system, and predicates correspond to connections between or properties of the symbols.

This definition is consistent with all the major approaches to qualitative modelling, in particular it entails compositional and local models (as defined in [Bobrow, 1984]). Note, however, that models are usually represented using higher-level constructs, but could be translated into predicate calculus.

A first characteristic of models following this definition is that they are *compositional*: no symbol can refer to combinations of independent entities, so that the set of symbols in a composed system is simply the union of the symbols associated with each of its parts. This allows a model to be instantiated from a finite library of physical knowledge, as for example in Qualitative Process Theory ([Forbus, 1984]).

Of particular interest in this paper are the representations of quantities allowed in a qualitative model. The restriction that symbols and predicates must correspond to entities of the real world restricts the use of precise numbers to *landmark values*, fixed distinguished values for which such a correspondence can be established. It also rules out representations of quantity values by symbolic algebra, as this would require introducing predicates (such as multiplication) and individuals (subexpressions) which have no correspondence in the real world. The above definition of a qualitative model leaves only two ways of modelling the values of quantities:

- by a fixed set of qualitative values expressed as predicates on the quantity, such as $\text{POSITIVE}(x)$, $\text{ZERO}(x)$ or $\text{NEGATIVE}(x)$.
- by relations between quantities or landmark values, such as $\text{GREATER}(x,y)$.

The same applies to any individual which represents a continuously variable entity of the world, such as shapes. This is the definition of a “qualitative” model which I shall use throughout this paper.

QUALITATIVE REPRESENTATIONS OF FUNCTION

The model of conceptual design as a mapping from function to design object requires first of all a language for qualitatively specifying function. For many domains, the issue of what such a language should express is an open problem. For the limited domain of elementary mechanisms, however, a clear set of requirements can be stated. In the following, note that I use the word *function* to mean something distinct from the *purpose* of the device. As an example, consider a list of specifications that might lead to a ratchet device, shown in Figure 1:

- the device should permit continuous rotation of an axis in the counterclockwise direction, but mechanically block it in the clockwise direction.
- the permissible backlash in the clockwise direction is at most 0.25 rotations.
- the maximum torque required for turning in the counterclockwise direction is at most 3Nm.

The first element specifies a desired function as a list of required behaviors in two different environments¹:

- given a torque on the shaft in one direction, the device should permit the rotation.
- given a torque in the opposite direction, the device should eventually produce a reaction force to it, and thus block the rotation.

The second specification is a kinematic restriction on the behavior of the device in response to a change in its environment, namely when the direction of the input torque is reversed from counterclockwise to clockwise. Likewise, the third specification imposes a condition on the numerical parameters of the behavior in an environment where there is both a torque and motion of the shaft in the counterclockwise direction. There do of course exist other restrictions, such as those on the size of the device, but those are not functional specifications.

The point of this example is to show that functional specifications in general take the form:

$$\begin{aligned} \text{Environment} &\Rightarrow \text{Behavior, or} \\ \text{Environment} &\Rightarrow \text{Restriction on Behavior} \end{aligned}$$

¹An automatic translation of the original form of specifications into this more complete form would require extensive knowledge about mechanisms and their use, and is not addressed in this paper.

where the \Rightarrow often implies a causal connection, i.e. a property of the environment must produce the particular behavior.

We thus define a *function* as a pair of (Environment, Behavior), giving the behavior of the device in a particular environment. Note that many functions may never be specified: a pair of gearwheels may function as a conductor (or a space heater), but if it is never put in that environment this function is irrelevant. A functional *specification* is a condition on a function.

In almost all cases, the desired device has to function not only in particular, precisely specified environments, but in a whole range of them. Consequently, the environments in the functional specification are qualitative, most often given by ranges of parameter values. This means that the resulting behaviors, and the functional model as a whole, are also qualitative, although restrictions on behaviors could refer to precise numerical parameters.

The environment in which a mechanism is used can be specified as a set of qualitative values (or history of values) for the external parameters which influence the device. The types of qualitative values used in actual mechanism specifications are either:

- signs of parameters, e.g. "turns clockwise"
- intervals of parameters, e.g. "requires a force less than 3 N"

Specification of an admissible interval can be seen as specification of a sign with an added numerical restriction. For the sake of simplicity, I have limited this research to specifications which involve the sign only. Adding numerical restrictions requires additional modeling and can only increase the complexity of the models, so that this does not affect the main point of this paper.

For expressing qualitative behavior or restrictions on qualitative behavior in a qualitative functional model, a general modeling language for qualitative mechanical behavior is necessary. A good qualitative model of behavior is the *envisionment* ([De Kleer, 1977]) of the device. However, the envisionment itself is not appropriate for modeling function, as it cannot express the functional connection between the environmental inputs and the resulting behavior.

These functional connections are given as a set of inference rules or equations that relate individual parameters of the device. As an example, consider a contact between two objects whose positions along some axis (not parallel to the contact surface) are given by parameters a and b . The (unidirectional) properties of a the contact are captured by the following inference rules:

$$\begin{aligned} \frac{da}{dt} = + &\Rightarrow \frac{db}{dt} = + \\ \frac{db}{dt} = - &\Rightarrow \frac{da}{dt} = - \end{aligned}$$

An increase in the parameter a results in an increase in parameter b (pushing), but decreasing a has no influence on b - one can not pull with the contact.

In general, a complex device has an internal state which defines the applicable functional connections between inputs and outputs. A formalism for representing mechanical function must represent the different states and possible transitions between them. A functional representation that fulfills these criteria is the *place vocabulary*

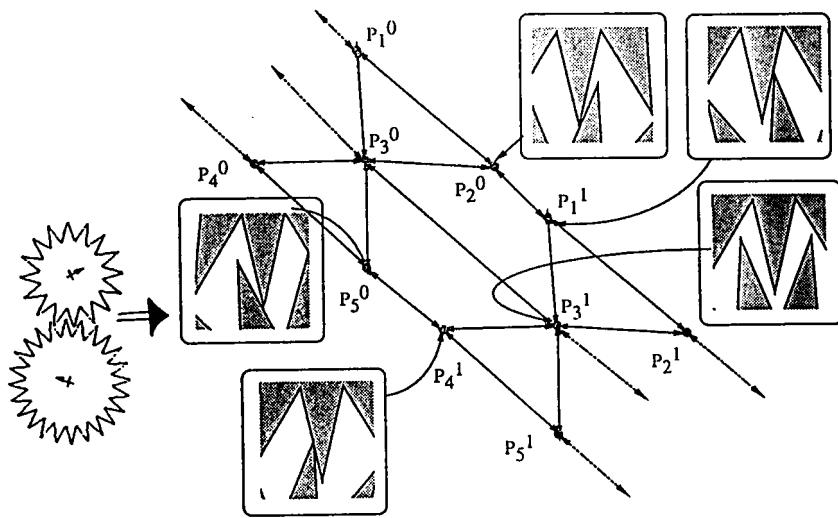


Figure 4: *The place vocabulary for the interaction of a pair of gear teeth. Note how it distinguishes places P_1 and P_2 where counterclockwise motion of gear 1 can push gear 2, places P_4 and P_5 where gear 2 can push gear 1 counterclockwise, and the slack state, P_3 , where the two gears are not in contact. The superscripts indicate the periodic repetition of the interactions.*

([Faltungs, 1987a, Faltungs, 1990]). Each place² is characterized by a particular kind of object contact and the set of applicable qualitative inference rules between the dynamic parameters of the device. The places are arranged in a graph, whose edges define the possible transitions and are labelled with the conditions on qualitative motion under which the transitions can occur.

As an example, consider a typical place vocabulary for gearwheel interactions, shown in Figure 4. The set of places represents the qualitatively different kinds of object contacts and corresponding functional relations in the form of inference rules between motion parameters. Thus, the inference rules for P_1 and P_2 are different from those for P_4 and P_5 , and P_3 has no attached inference rules at all. The place vocabulary provides a detailed model of the functions of a pair of gears, including the slack when the direction of rotation is changed.

Transitions between places depend on motions of the device. The adjacencies between places are marked with the qualitative derivatives of each of the motion parameters which might result in the given transition between places. This is an essential

²While "state" would be a better term, we use "place" for historical reasons, and to avoid confusion with kinematic states.

element of the detailed functional model, and necessary to compute an actual environment of behavior based on the place vocabulary. A detailed description of place vocabularies can be found in [Faltings, 1990].

The place vocabulary is a minimal formalism for functional modelling. Every change in the place vocabulary entails different states or inference rules which represent a potential difference in qualitative function of the device, so that all elements of a place vocabulary are *necessary* for the functional model³ On the other hand, every change in qualitative function of the device must manifest itself as a variation in the place vocabulary, so that the place vocabulary is also a *sufficient* model of the qualitative function.

Realistic specifications almost never specify the complete functional model, but only part of it. The example of the ratchet specifications above refers only to forces on the input shaft, and says nothing about the movement of the pawl. Such specifications have to be completed with more detail before they can be used to define an actual device. This can be done either by instantiating a device from memory, or by using first principles to search for a satisfactory complete model. In both cases, a qualitative language for expressing complete functional models is required: either for representing and indexing the library of known functions, or for constructing a search space of functions. This task of *completing* the specifications is a very difficult part of design. In this paper I only address the representational issues involved.

A complete place vocabulary can then be matched to the functional specifications, either by

- envisioning the behavior in response to the environments of interest ([Nielsen, 1988]), computing a causal analysis and comparing it with the desired one, or
- using the behavioral rules to directly infer relations between input parameters and behavior, for example inferring the direction of motion of a gear in response to an input motion.

Place vocabularies thus provide a representation for modeling mechanical function and, consequently, functional specifications.

However, in some cases the place vocabulary shows more functional detail than is required by the specifications. For example, in many cases the slack in a pair of gear-wheels is so small as to be negligible in the specifications and functional models. Several researchers have investigated the use of abstractions, either on the level of place vocabularies ([Nielsen, 1988]), or on the level of configuration spaces ([Joscowicz, 1989]). Such techniques can be used to construct abstracted place vocabularies which can be matched more efficiently to specifications.

MODELLING THE DESIGN OBJECT

When discussing concepts, people like to refer to *sketches*, which appear to be qualitative representations of some form. In fact, designers often insist on using an extra wide pen in order to make purposely rough sketches of their initial ideas. In this section, I

³Some place distinctions are due only to changes in object contacts. They could be abstracted away if this should not count as a functional distinction.

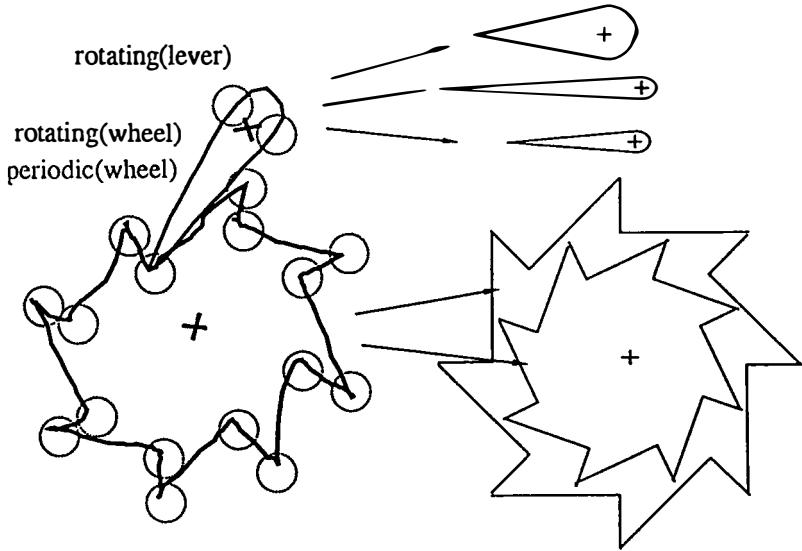


Figure 5: *Example of a sketch of a ratchet. Note that the sketch correctly models the discontinuities of the shapes (marked by circles) and the types of boundaries between them. However, the metric dimensions are much less precisely defined, leaving open the different interpretations shown on the right. This characteristic is shared with the metric diagram.*

develop a qualitative modelling formalism for shape which is designed to capture the information represented in a sketch.

In the domain of elementary mechanisms, a sketch is a rough drawing of the shapes of a mechanism which achieves a desired function. An example of such a sketch is shown in Figure 5. The sketch shows the example of a ratchet, a device which blocks clockwise rotation of the wheel (state shown in the sketch), but allows counterclockwise rotation. The most striking fact about the sketch is that the wheel as it is shown in the sketch can not even be turned a full rotation. Consequently, the device as shown in the drawing does not even achieve the function of a ratchet that the sketch is intended to demonstrate! Clearly, understanding of a sketch must involve an *interpretation* as a desired function, rather than just a simulation of the device as shown. A theory of how such an interpretation might be constructed is described later in this paper.

If a sketch can not be understood as a precise shape representation, what is the information it conveys? Note that a sketch correctly shows the discontinuities of the shape, as indicated by the circles in Figure 5, as well as the type of edges between discontinuities. The inaccuracies in metric dimensions leave open a range of possible interpretations, some of which are shown on the right in Figure 5. For elementary mechanisms, the information conveyed by a sketch consists of:

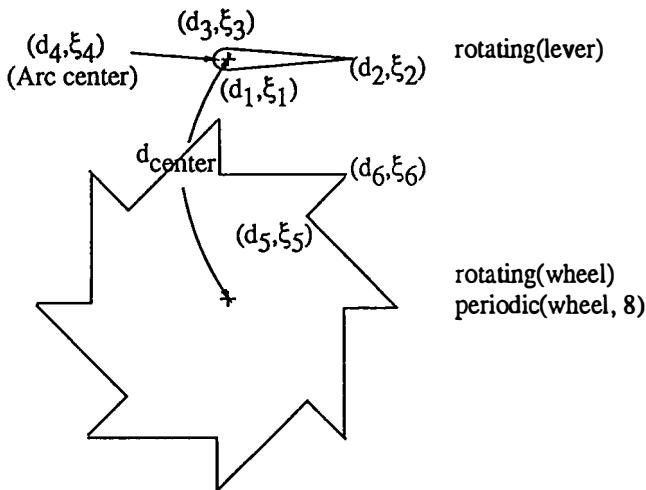


Figure 6: The metric diagram for a ratchet mechanism. The object dimensions are defined as symbolic coordinates, which can then be represented by qualitative values. The coordinates of discontinuities are local to the particular object and defined in polar coordinates.

- a graph of discontinuities and edges between them, and
- very approximate metric dimensions of the shape, and
- annotations, such as freedom of motion or periodicity of a shape.

This is the information modelled in the *metric diagram* of a shape, shown in the example in Figure 6. The structure of the shape is modelled as a graph of vertices and edges between them, indicated by the drawing of the shape in Figure 6. In my implementation, edges can be either straight lines or circular arcs.

Metric dimension parameters are associated with each of the elements of this structure. They are the positions of the discontinuities in polar coordinates, the positions of the centers of circular arc edges, and the distances of the objects in the frame of the mechanism. They can be represented in quantity spaces, systems of fixed intervals, precise numbers, or any other representation suitable to express the knowledge available about them. Annotations to the metric diagram express the freedom of motion of the parts, and the periodicity of the ratchet wheel.

The metric diagram is a qualitative representation: the graph modelling the connections can be expressed in predicate logic, and parameter values can be represented by a fixed set of qualitative values. I now show that there exists no simpler shape representation which is both sufficient to predict the different possible object contacts - an important precondition for any kinematic analysis - and is also qualitative according to the definition in the introduction. This claim follows from a proof that all of the elements of the metric diagram are required in any qualitative shape representation useful for qualitative kinematic analysis.

According to the definition of qualitative, each individual in a qualitative representation must correspond to an entity in the real world. Each part of a mechanism must therefore be modelled by distinct and independent individuals. The elements of the model of a single part shape can be justified as follows:

- **Discontinuities:** depending on the shape the part interacts with, each discontinuity can cause an isolated object contact to appear, so it must be an independent element of the model.
- **Edges:** different types of edges can result in different propagation of motion by the mechanism. Connections between edges and discontinuities define possible transitions between contacts. Edges and the connections they define must be another independent element of the model.
- **Positions of discontinuities:** changing the position of a single discontinuity can make an object contact possible or impossible, so each must be represented individually.
- **Distance between objects:** is required to predict object contact, and must be a separate quantity because of the independence of object models.
- **Annotations:** are either shorthand (periodicity), or express important information for kinematic analysis (freedom of motion).

I conclude that the metric diagram is a required part of any shape representation which is to be related to a model of mechanism kinematics.

Other types of parameterizations or decompositions can be imagined to decouple the structure from the dimensions of a shape, but they do not change the nature of the representation. The formalization by polar coordinates illustrated in Figure 6 is optimal in the sense that functional attributes depend as directly as possible on attributes of the representation. For example, the functional attribute that particular parts of two rotating shapes can touch depends only on the distance of these parts from the respective centers of rotation: a single parameter in each of the shape representations. For other attributes, the conditions are often more complicated, but the fact that only polar and cartesian coordinates are widely used by engineers makes it unlikely that there exists a different formalism to represent coordinates which results in more direct mappings.

An interesting fact about the metric diagram is that by changing the representation of quantities to real numbers, it can be turned into a precise object representation. This makes it a good framework for comparing qualitative predictions to precise calculations. In particular, the complexity of kinematic predictions can be compared on the basis of what accuracy of the values of metric diagram quantities is required to make the prediction.

MAPPING BETWEEN FUNCTION AND ARTIFACT

The purpose of modelling in design is to allow reasoning about the relationship between a designed object and its function. Ideally, this would be accomplished by a direct mapping between attributes of the functional model and attributes of the geometric

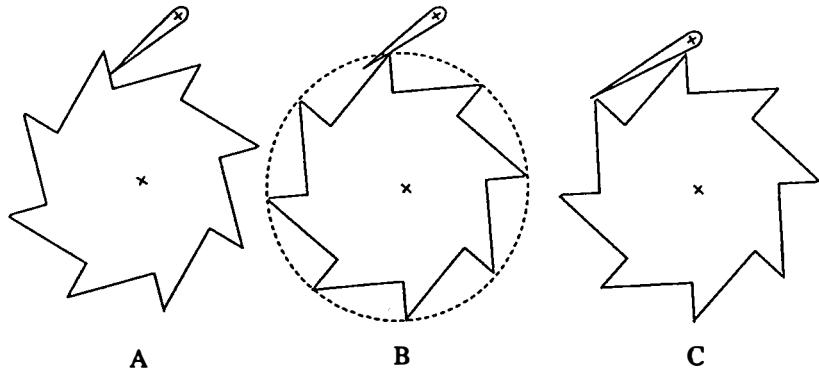


Figure 7: *Three different states of a ratchet which illustrate the three example attributes. Deciding whether the contact shown in A is possible amounts to linear distance comparisons. Determining the inference rule which holds in B requires evaluating a complex algebraic expression on dimensions. Showing that the transition between contacts shown in C is actually possible requires iterative numerical analysis.*

object model. A first difficulty here is that this is a one-to-many mapping: physical objects that achieve a particular function can be constructed in many different ways. The task of identifying (or adding) the object features intended to achieve a particular functional attribute is the responsibility of a particular design strategy. This paper addresses the issue of reasoning about what the features must look like to achieve the intended function.

Using the place vocabulary as a functional model, the following are examples of functional attributes which are of interest in design:

1. the feasibility of a particular object contact (place).
2. the inference rules which hold in a particular place.
3. the conditions for possible transitions between a pair of places.

Each of these attributes can be mapped into a condition on the model of the designed object, as shown in [Faltings, 1988c]. Figure 7 shows example positions of a ratchet which serve to illustrate the three types of attributes. The three examples illustrate a progression of complexity: for the first, the mapping can be based on a purely qualitative model, the second requires rather precise approximations of the metric dimensions, and the third example can in practice only be validated on a numerically precise model.

Mapping to a qualitative object model

For an intelligent CAD system based on a qualitative object model, it is important to be able to map the qualitative functional attributes into equivalent attributes of

the object model. In this section, we show the attributes in a metric diagram which correspond to the functional attributes shown in the three examples of Figure 7.

Feasibility of an object contact: The functional attribute of the existence of a state where two particular object parts touch maps into an attribute on the relative dimensions of the objects. In particular, for two rotating objects such as in the ratchet example, two object parts with distance d_1 and d_2 from the respective centers can touch whenever

$$\begin{aligned} d_1 + d_2 &\geq d_{\text{center}} \text{ and} \\ |d_1 - d_2| &\leq d_{\text{center}} \end{aligned}$$

where d_{center} is the distance between the centers of rotation of the two objects.

Situation A in Figure 7 is representative of the touch between the tip of the lever and the side of the wheel's tooth. The place corresponding to this situation exists whenever the tip of the lever can touch *some* point on the edge, so that (using the notation of Figure 6 given earlier) the attribute of the metric diagram becomes:

$$\begin{aligned} \min(d_5, d_6) + d_2 &\geq d_{\text{center}} \text{ and} \\ |\max(d_5, d_6) - d_2| &\leq d_{\text{center}} \end{aligned}$$

The mapping of this functional predicate into the object model is thus a linear distance comparison. If the distance between the centers of rotation, d_{center} , is kept constant in the design problem, it can be expressed as an attribute of the *relative* values of the object dimensions. As these could be expressed qualitatively in a quantity space ([Forbus, 1984]), a purely qualitative metric diagram is sufficient as an object model which allows one to express this attribute. However, if the distance between centers of rotation becomes variable, the quantity space representation becomes insufficient. Note also that the chosen metric diagram is optimal for this case: it only involves a single parameter for each object.

Inference rules for a given place: In the situation B in Figure 7, it might be of interest to know in which direction the pawl will turn when the wheel is turned counterclockwise. This depends on the direction of the edge of the lever with respect to the dashed circle which describes the incremental freedom of motion of the wheel. If the edge "points" to the inside, as shown in the figure, the lever will turn clockwise, otherwise, it will turn counterclockwise. The attribute can be generalized from a single contact point to all configurations with the given contact, but for one contact, there may be up to three regions with different inference rules. The functional attributes are not single inference rules, but the different combinations of regions with different inference rules which can exist for the type of contact.

The condition is equivalent to the sign of a linear expression of the sines and cosines of the rotation angles in the configuration where the points of interest touch. However, these angles in turn depend on a nonlinear combination of object dimensions ([Faltings, 1987b]). For the example B shown in Figure 7, the inference rule to be applied in the place containing the configuration where the tip of the lever touches a tip of the wheel depends on the sign of (using the notation of Figure 6:

$$\sqrt{d_2^2 + d_3^2 - 2d_2d_3\cos(\xi_2 - \xi_3)} \left(d_2^2 + d_6^2 - d_{\text{center}}^2 \right) - 2d_2d_3d_6$$

This attribute is dependent on nonlinear combinations of distinct parameters of the metric diagram, which furthermore belong to independent objects. Since qualitative attributes can not depend on combinations of independent objects, it is not possible to store its value as part of a qualitative object model, but it must be composed from individual qualitative representations. Only symbolic algebra is sufficient to do this, and consequently symbolic algebra is also required to express the attributes corresponding to rules for force and motion propagation in the metric diagram. As any qualitative shape representation must contain the metric diagram, and the power to express such attributes makes the metric diagram non-qualitative, *there is no qualitative shape representation that would allow one to express this attribute.*

Conditions for place transition: Situation C in Figure 7 shows a configuration where the lever and wheel touch in two different points. This instantaneous situation represents a transition between two places, called a *subsumption*. The attribute which states the possibility of this direct transition, i.e. the existence of this subsumption configuration, is an important element of the functional model of the device.

If we attempt to map this attribute to the object model, represented by a metric diagram, it amounts to the existence of a configuration which simultaneously satisfies two nonlinear constraint equations. Using symbolic algebra, it is possible to derive an equivalent condition as the existence of a root of a six-degree polynomial. By applying algebraic decision methods ([Ben-Or et al., 1986]), it is possible to reduce this to a complex combination of algebraic predicates which express the condition, but these are highly nonlinear in the parameters of the metric diagram - so complex that it is impossible to print them readably on a single page. Besides the fact that the nonlinear combination of parameters of independent objects violates the condition of compositionality, the expression of the subsumption condition is much too complicated to be effectively used for reasoning.

The example of subsumptions points to even deeper problems with the mapping between qualitative models of function and objects. It is due to the fact that qualitative representations are *local*: all relations between individual symbols are defined and reasoned about individually. Consequently, in a qualitative analysis of kinematics, each object contact is reasoned about individually. This fails to take into account *interference* between object contacts: a particular state may in reality be impossible because it would create an overlap of other, not directly related parts of the mechanism. Such interference can be reliably inferred only from the presence of subsumption configurations - but these attributes in turn can not be formalized in a qualitative object model. I conclude from these arguments that qualitative object models are almost useless for making even qualitative predictions about function. However, they may have a limited usefulness in design for controlling search processes, as indicated later in the paper.

Mapping to a precise object representation

Even though most functional attributes can not be mapped directly into attributes on a *qualitative* object representation, they do define attributes of a *precise* object representation which can be reasoned about. Given a numerically precise model of the designed objects, its place vocabulary as a representation of the qualitative function can be computed using the methods described in [Faltungs, 1990, Faltungs, 1987b]). Each

of the attributes of the place vocabulary can be labelled with the conditions on the object representation which are necessary to maintain its existence ([Faltungs, 1988c, Faltungs, 1988a]). For the existence of places or inference rules associated with them, these are the algebraic conditions on the object dimensions, as shown in the examples given earlier.

For reasoning about the existence of a subsumption, it is now sufficient to express the condition for maintaining the *particular way* in which the subsumption is achieved by the object shapes, not a condition for the existence of the subsumption in general. For this reason, it turns out to be possible to formulate the conditions for maintaining or achieving a particular subsumption in closed form ([Faltungs, 1988a]). Even subsumptions can be reasoned about if a precise object model is used.

The many advantages offered by precise object representations leave open the question of why designers prefer rough sketches to precise drawings at the stage of conceptual design. I discuss possible explanations in the next section.

INTERPRETATION OF SKETCHES: KINEMATIC TOPOLOGY

The results of the preceding sections leave open the question why human designers often insist on using sketches. There are two possible interpretations of this phenomenon: either the sketch represents a single *qualitative* model, or the sketch is a representation of a family of *precise* models.

The sketch as a single qualitative model

As has been shown by the preceding discussion, interpreting a sketch as a single qualitative model can not be powerful enough to infer qualitative kinematic behavior. However, it turns out that a qualitative metric diagram - equivalent to a sketch - is sufficient to predict the *kinematic topology* ([Faltungs et al, 1989]) of the device. Kinematic topology expresses the topology of the device's *configuration space*, the space spanned by the position parameters of the mechanism's parts. For many devices, the topology of its configuration space already says a lot about its function. For example, in a pair of gearwheels, the fact that the two gears can only move in coordination can already be deduced from the fact that the configuration space consists of several doubly-connected regions which "wrap around" both dimensions of configuration space ([Faltungs et al, 1989]). On the other hand, topology is too weak for a qualitative simulation of the meshing of the gear's teeth, or the blocking behavior in a ratchet.

The computation of kinematic topology is most easily explained by reformulating the metric diagram as a decomposition into adjacent shape primitives, of which there are two types: *pieces* for convex sections and *cavities* for concave sections, as shown in the example of Figure 8. Note that this primitive decomposition is very similar to the discontinuity-based representation in the metric diagram, with the addition of the distinction between convex and concave discontinuities.

The interaction of a pair of shape primitives generates topological primitives. As shown (intuitively) in Figure 9, an interaction between two pieces generates a potential area of illegal configurations, called *obstacle*, and an interaction between a piece and a cavity generates a potential area of legal configurations, called *bubble*. Initial

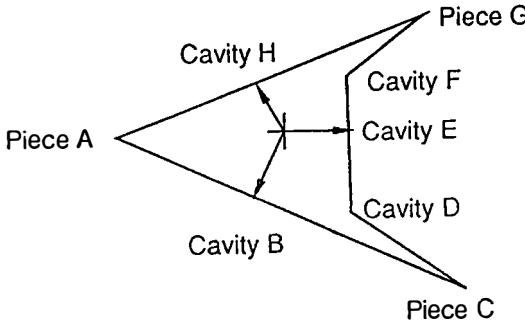


Figure 8: *Example of the representation of a shape by primitives.*

connections between these topological primitives are given by the adjacencies of the shape primitives on the objects themselves. However, for determining whether primitives actually exist, and whether adjacent primitives intersect or not, it is necessary to know whether the extremal points of these primitives can touch. This is the same condition for possible object contacts which has already been discussed earlier, and can be expressed in a qualitative model.

However, even kinematic topology depends crucially on the existence or absence of global subsumptions, which establish additional connections between topological primitives and can have a profound effect on configuration space topology. In spite of this problem, kinematic topology and the associated primitive-based representation of object shape are useful for conceptual design. Because of the high degree of abstraction, the amount of ambiguity which results when subsumption conditions can not be evaluated is manageably low. For example, for a pair of gearwheels described in the primitive decomposition, there are only five different topologies to be considered ([Faltings et al, 1989]). With only approximate metric information, such as that provided by a sketch, the analysis of kinematic topology already allows us to predict that the gears either mesh or jam - a prediction which rules out many other forms of behavior and provides a focus for subsequent detail design. An analysis at this level also explains how people can pick out the desired function out of the many functions permitted by the inaccuracies of a sketch. Furthermore, as shown in ([Faltings et al, 1989]), kinematic topology can be computed for any shape which can be decomposed into segments of convex and concave curvature. To my knowledge, it is the only form of kinematic analysis which does not require a precise representation of object shapes.

The sketch as a family of precise models

An important characteristic of a sketch is that its precise dimensions often do not represent a correctly functioning device. The sketch requires an *interpretation* as a device with different precise dimensions in order to support the desired explanations. More precisely, the sketch defines a metric diagram in which the parameter values are

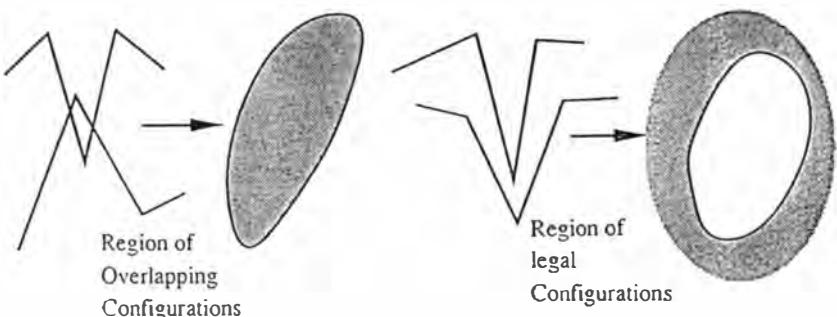


Figure 9: *Interaction between two object pieces creates illegal configurations, interaction between piece and cavity allows legal configurations.*

underdetermined. This vagueness means that a single sketch can be interpreted as any of a family of possible precise models. In conceptual design, the sketch thus allows the designer to make frequent mental changes to his design without having to change the drawing - an important economy when such changes are frequent.

The most likely explanation of the designer's use of sketches is a combination of the two possibilities. On the one hand, the sketch itself defines a restricted domain in which a precise solution is searched. This is based on a single interpretation, for example based on kinematic topology. On the other hand, it allows the designer to reuse the same drawing throughout the frequent changes inherent in conceptual design. The arguments in this paper have shown that the interpretation of sketches as precise models is inevitable for design, and consequently that the popular model of design being based on mapping functional attributes to a single qualitative model represented in a sketch is wrong.

CONCLUSIONS

I started this paper with the hypothesis that conceptual design is a mapping from a qualitative model of function to a corresponding qualitative model of the designed artifact, a common model among researchers in intelligent CAD. This model of design, motivated by observation of human designers, is corroborated by early work on automatic circuit design ([De Kleer and Sussman, 1989]), but its application to other domains such as mechanical design has in fact never been investigated. This was the starting point for the case study presented in this paper.

I have shown how qualitative models of kinematic function and of object shapes can be constructed for the limited domain of this case study. However, it has proven impossible to generate useful direct correspondences between the qualitative functional models and the qualitative shape models. Even though this result is limited to a narrow domain, it shows that the hypothesized model of conceptual design can at least not be

generally applicable.

This result suggests an alternative model of conceptual design: the iterative refinement of a *precise* model of the design object ([Faltings, 1988b]). In this approach, shape modifications are obtained by reasoning about the limits of validity of the functional attributes of a current design. This avoids the difficulties with qualitative object models, while maintaining the use of a qualitative functional model which can easily be related to specifications. A precise object model is also used in the work of Joskowicz and Addanki ([Joscowicz and Addanki, 1988]), who present an incomplete algorithm for mapping exact functional specifications into corresponding exact object shapes.

Why, then, do human designers like to use sketches? One reason is that a representation like kinematic topology is useful for controlling search: a design whose kinematic topology does not correspond to the desired one should not be pursued any further. But another, more important reason may be that while the sketch defines the correct metric diagram, its precise dimensions can be freely reinterpreted according to the interests of the analysis. The sketch thus allows the designer to make frequent mental changes to his design without having to change the drawing - an important economy when changes are frequent. Because of the difficulties with kinematic predictions discussed earlier, this latter reason seems much more plausible than the use of a sketch as a qualitative shape representation. It is also likely that this model of the designer's use of sketches also holds for other domains, such as architecture.

On a larger perspective, the results in this paper cast some doubt on hopes for useful qualitative object representations. This doubt is confirmed by a survey of the research results that have been achieved in qualitative physics. Of the 55 papers in a recent representative collection of papers dealing with qualitative physics ([Weld and De Kleer, 1989]), only three put an emphasis on the qualitative representation of *objects*, and these only in the context of functional predictions. All successful research in qualitative physics is primarily motivated by qualitative models of behavior and function, which in some cases can be mapped successfully to qualitative object models. The results of this case study may thus indicate a deeper truth about qualitative physics: qualitative function does not always correspond to qualitative object attributes!

References

- Ben-Or, D. Kozen, J. R. (1986). The Complexity of Elementary Algebra and Geometry, *Journal of Computer and System Sciences* 32: 251–264
- Bobrow, D. (1984). Qualitative Reasoning about Physical Systems: An Introduction, *Artificial Intelligence* 24.
- de Kleer, J. (1977). Multiple Representations of Knowledge in a Mechanics Problem Solver, *Proceedings the 5th IJCAI*, Cambridge.
- de Kleer, J. and G. Sussman, G. (1980). Propagation of Constraints Applied to Circuit Synthesis, *Circuit Theory and Applications* 8,
- Faltings, B. (1987a). Qualitative Kinematics in Mechanisms, *Proceedings of the 10th IJCAI*, Milan.

- Faltings, B. (1987b). The Place Vocabulary Theory of Qualitative Kinematics in Mechanisms, *University of Illinois Technical Report UIUCDCS-R-87-1360*, July.
- Faltings, B. (1988a). A Symbolic Approach to Qualitative Kinematics, submitted to *Artificial Intelligence*, March.
- Faltings, B. (1988b). Qualitative Kinematics and Intelligent CAD, *Proceedings of the 2nd IFIP WG 5.2 workshop on ICAD*, North-Holland, Amsterdam.
- Faltings, B. (1988c). A Symbolic Approach to Qualitative Kinematics, *Proceedings of the Third International Conference on Fifth Generation Computer Systems*, Tokyo, November.
- Faltings, B. (1990). Qualitative Kinematics in Mechanisms, *Artificial Intelligence* 44(1) June.
- Faltings, B., Baechler, E. and Primus, J. (1989). Reasoning about Kinematic Topology, *Proceedings of the 11th IJCAI*, Detroit, August.
- Forbus, K. (1984). Qualitative Process Theory, *Artificial Intelligence* 24.
- Joscowicz, L. and Addanki, S. (1988). From Kinematics to Shape: An Approach to Innovative Design, *Proceedings of the AAAI*, St.Paul, August.
- Joscowicz, L. (1989). Simplification and Abstraction of Kinematic Behaviors, *Proceedings of the 11th IJCAI*, Detroit, August.
- Nielsen, P. (1988). *A Qualitative Approach to Rigid Body Mechanics*, PhD Thesis, University of Illinois.
- Weld, D. S. and de Kleer, J. (eds.) (1989). *Readings in Qualitative Reasoning about Physical Systems*, Morgan-Kauffman.
- Williams, B. (1990). Interaction-based Invention: Designing Novel Devices from First Principles, *Proceedings of the 8th Conference of the AAAI*, Boston, August.
- Tomiyama, T., Kiriyama, T., Takeda, H., Xue, D. and H. Yoshikawa, H. (1989). Metamodel: A Key to Intelligent CAD Systems, *Research in Engineering Design* 1.
- Yoshikawa, H. (1987). General Design Theory and Artificial Intelligence, *Artificial Intelligence in Manufacturing*, North-Holland, Amsterdam.

A graph-based representation to support structural design innovation

J. Cagan

Department of Mechanical Engineering
Carnegie Mellon University
Pittsburgh PA 15213 USA

Abstract. We introduce a graph-based representation to make structural design innovation possible. In this representation, an abstract delineation of geometry is modeled via graph nodes and links, while a generic expression of design knowledge is associated with similar classes of graph nodes. Design space expansion occurs by expanding the nodes in the graph model, automatically expanding the symbolic model of the design. One expansion technique based on optimization criteria and implemented with the graph representation is called *Dimensional Variable Expansion (DVE)*. This expansion is extremely efficient; because the design knowledge is independent of the graph structure, the expansion of the graph automatically expands the complete set of design constraints. The connectivity of the graph acts as a filter for operations of region removal which are not valid based on discontinuous load paths. Although the graph representation is employed for structural innovation, it can be utilized to model general design problems of various physical domains.

INTRODUCTION

Representation of a design problem is one of the major research issues which must be investigated if computer systems are to properly support the design process. With a well defined representation, a design problem can be sufficiently modeled for computational algorithms to reason about and manipulate. A proper representation can also support theoretical development of the design process; if a design can be formally modeled computationally, then processes which utilize that representation as a foundation have a basis from which to observe and measure change.

In this paper we introduce a graph-based representation responsive to the design of structural bodies. This representation is concise and efficient. The graph itself is an abstraction from the detailed geometric level of design; however, the graph links to more complete and detailed knowledge about the geometry, constitutive relations, and other pertinent design information. By linking to this more detailed knowledge, the graph is

capable of organizing symbolic descriptions of the behavior and geometry of the body. Although the graph representation is motivated by design innovation, it is also amenable to the routine levels of design.

The graph-based representation models geometric topological information based on graph nodes, links, and the coordinate system. The graph nodes have a one-to-one correspondence with the regions in the body being designed while the node links model the connectivity of the regions. Constitutive and other life-cycle knowledge and constraints are stored in separate data structures (called *knowledge modules*), independent of the specific node regions. When the design space is expanded, new nodes are spliced into the graph and they link to the appropriate knowledge module. By expanding the geometric topology, the design knowledge is automatically expanded. Because the graph models connectivity between regions, it will sometimes be referred to as a *connectivity graph*.

Cagan and Agogino (1987, 1990) and Cagan (1990) introduce the 1stPRINCE design methodology for innovative design of mechanical structures. By reasoning from a deep level utilizing qualitative optimization techniques and engineering first principles, 1stPRINCE expands the design space to innovate new structures. The 1stPRINCE methodology utilizes an implementation of the graph representation to model and manipulate mechanical designs. We will show how expansion of the connectivity graph equivalently expands the design space and makes design innovation within 1stPRINCE possible.

The next section will briefly review the concepts of innovative designs and the expansion technique called Dimensional Variable Expansion (DVE) utilized by 1stPRINCE. Next the graph-based representation will be introduced and then utilized to implement DVE. After a body is expanded, it may be beneficial to remove regions of the body to improve a design; region removal will be explored within the connectivity graph.

BACKGROUND

We emphasize that design is optimally directed. Cagan (1990) defines *optimally directed design* as: an approach to design which attempts to determine the optimum by directing the search toward the optimum, reducing the problem to a bounded space surrounding the optimum while providing information about how to reduce the complexity of the search space. The global minimum cannot always be reached; rather at times the space containing the optimum can only be reduced, directing the search toward the optimum. Although the graph representation may support processes that do not utilize optimization information, we will discuss only optimization-oriented approaches to design in this paper.

We define a *primitive-prototype* as the model of a design problem specified by an objective function and a set of inequality and equality constraints within a design space. A *prototype* is defined as a solution from the analysis on a primitive-prototype which can be instantiated to at least one feasible artifact. We can then define an *innovative design* as one which demonstrates new design variables or features in a prototype based on existing variables or features from a previously known prototype, whereas a *routine design* describes a prototype with the same set of variables or features as a previously known prototype.

With innovative designs new variables are introduced into the design space by manipulating the mathematical formulation of the primitive-prototype. First it is determined which variables are potentially *critical*, defined as those variables which have an influence on the objective function and which, when expanded, will create new variables which also influence the objective. Section 4 briefly discusses how those variables can be determined. Once critical variables are selected, mathematical manipulations of those variables expand the design space. One expansion technique, called *Dimensional Variable Expansion* (DVE), introduces new variables in the primitive-prototype by expanding a single region into multiple regions, where a *region* is a section of a body which may be independently modeled and may have independent properties and features. During DVE, a body is subdivided into a group of regions which together define a new primitive-prototype and body. If there is a coordinate system which is not of physical dimensions, DVE remains valid although not necessarily physically intuitive.

Cagan (1990) presents the formal theory for Dimensional Variable Expansion. Intuitively, DVE can be understood by observing a continuous integral of a function of variables χ and w divided into a series of continuous integrals over smaller ranges as:

$$\int_{z_0}^{z_n} f(\chi, w) dw = \int_{z_0}^{z_1} f(\chi_1, w) dw + \int_{z_1}^{z_2} f(\chi_2, w) dw + \dots + \int_{z_{n-1}}^{z_n} f(\chi_n, w) dw, \quad (1)$$

where n is the number of divisions, often of number two, and subscripts designate distinct variables. If the body remains homogeneous after division, the equality in equation (1) remains consistent. In DVE, however, the properties within each subregion are made discontinuous. The equality of equation (1) no longer applies; rather a completely

different prototype than the one described by the left hand side of equation (1) may result. It is the discontinuities in properties which produce innovative prototypes. By division of an integral over a critical variable which models a dimension (called a *dimensional variable*) and by permitting discontinuities across the geometric axes, new variables are introduced and the design space is expanded. For example a beam of one region under flexural load may be expanded into a four region beam of independent properties (a sketch of which can be seen in Figure 10), thus introducing new variables to model each region.

DVE has been incorporated into the 1stPRINCE design methodology to perform innovative design of mechanical structures. 1stPRINCE utilizes optimization information to search the design space after application of DVE. Monotonicity analysis (Papalambros and Wilde, 1988) is utilized to provide a symbolic, qualitative technique to reason about design optimization from fundamental principles. The manipulations performed by 1stPRINCE are domain-independent; however, their application may require domain specific knowledge. This knowledge specifies which design variables can be expanded as dimensional variables. In addition, in the mechanical structures domain, constitutive relations are formulated as constraints to give 1stPRINCE the power to reason about material properties. 1stPRINCE has been applied to various structures and dynamics problems. By minimizing weight, 1stPRINCE has innovated hollow tubes and composite rods from a solid cylindrical rod under torsion load. Also by minimizing weight for the same beam under flexural load a stepped beam was derived; applying inductive techniques to the beam a tapered beam was finally innovated. From a solid rectangular cross-section rod under flexural load, a hollow tube and an I-beam were innovated. Also, by minimizing resistance to spinning, a wheel was invented from a solid rectangular block. In each of these examples, a qualitative optimization analysis after DVE produced optimally directed prototypes in closed-form when possible.

Although the graph-based representation and DVE have been utilized in the 1stPRINCE methodology, both have applications beyond 1stPRINCE. DVE has potential as a technique to expand a design space outside of the structures domain and independent of optimization information if so desired. The graph-based representation itself makes implementation of DVE straightforward, but it is developed as a representation of design knowledge *independent* of structural design innovation. The graph representation, then, is a useful tool to represent design knowledge for general design methodologies. In this paper we will use 1stPRINCE to demonstrate application of the connectivity graph.

GRAPH-BASED REPRESENTATION

This section formulates the graph-based geometric connectivity representation and data structure. Graph representations are not new; however, this application to represent structures, the proposed semantics associated with the representation, and the manipulation of the graph to derive new designs are unique. The representation must support a model of a body in which geometric sub-regions of the body properly connect to other sub-regions to form a whole body. The environment must also support region expansion where new regions are created and connected to other regions to again form a complete model. Further, the representation must support the constraint knowledge of a primitive-prototype in a form that supports the expansion of regions with computational efficiency without requiring detailed manipulation and bookkeeping of those constraints.

We now introduce the graph-based representation where the nodes, links, and coordinate systems define the legal topology. A region is represented by a *node* of a graph; a node is a data structure with information about the associated region and its neighboring regions. Links between nodes contain information on how the body regions connect to each other. Model representations are limited by the number of links from each region where each region can connect to one unique neighboring region in each direction. Each graph utilizes a coordinate system in which to orient different nodes and links and direct their manipulation. Links pertain to the coordinate system so that nodes connect to other nodes only in each coordinate direction. Thus for a 1-2-3 coordinate system, links can connect in directions +1, -1, +2, -2, +3, and -3.

Figure 1a shows a planar body of eight regions in rectangular coordinates which is represented as a connected graph in Figure 1b. Link “L1” shows a 1-direction link, link “L2” a 2-direction link. In the graph representation for this planar body, a node can have connections in the +1, -1, +2, -2 directions; for example, node 3 has +1, -1, and +2 connections to neighboring nodes while node 2 has connections only in the +1 and -2 directions. All other links which do not connect to other nodes are nil, represented as the ground sign (the “free” boundary condition).

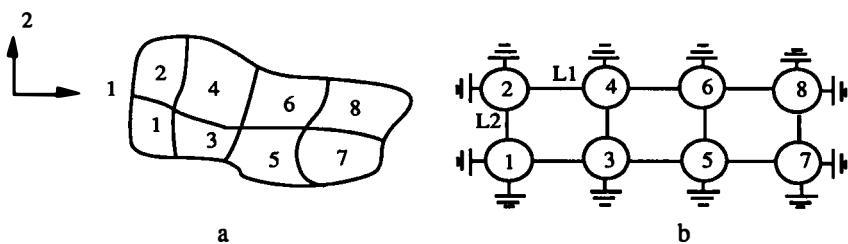


Figure 1. Body of eight regions (1a) represented as a graph structure in 1b; the ground sign represents the free boundary condition (nil)

Connectivity links represent *face connections* which imply that a region face of one region connects to a region face of different region, where a *region face* is a boundary of a region of dimension $n-1$ and n is the dimension of the body. In linear space, a face is of zero dimensions (a point); in planar space, a face is of one dimension (a contour); and in three dimensional space, a face is of two dimensions (a surface). The graph connection, *face links*, are associated with a single coordinate axis in that one axis must be traversed to obtain the face connection. By limiting connections to face links, only such connections are valid. Further, DVE requires a one-to-one interface between region faces. Thus Figure 2a, although of highly non-linear shape, is a valid representation while Figure 2b is not. The region faces of Figure 2a are said to be *equivalent surfaces* which are required in this representation and defined as:

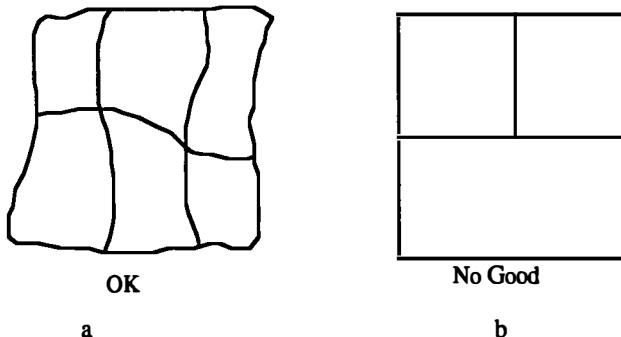


Figure 2. Bodies of equivalent (2a) and non-equivalent (2b) surfaces

A surface formed by a set of regions A is an *equivalent surface* to a surface formed by a set of regions B if for each face of the surface formed by A which is common to a face of the surface formed by B, there is a one-to-one mapping and the faces geometrically match.

Figure 3 shows a planar body in polar coordinates. The $r-\theta$ polar coordinate system is shown in the graph representation associated with rectangular coordinates. The hollow composite tube of Figure 3a is modeled by the connectivity graph in Figure 3c. The solid composite rod for Figure 3b is represented in Figure 3d. Note that for the hollow tube, the interior edge regions connect in the $-r$ direction to nil; however, for the solid rod the same

regions connect to opposite regions (1 to 3 and 2 to 4). Note also that for the solid rod these interior faces have zero length.

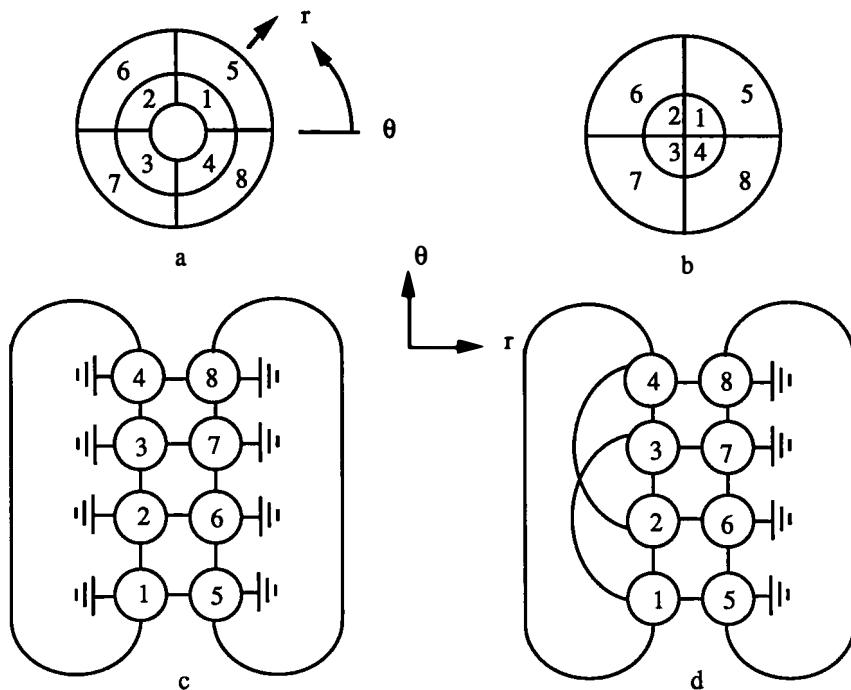


Figure 3. Hollow composite tube (3a&c) and solid composite rod (3b&d) in polar coordinates with associated graph representations

In general, a node is a data structure with slots for all information pertaining to the corresponding region. A body is represented in a three-dimensional space (be it rectangular, spherical, or cylindrical); a given node must connect to neighboring nodes in the three dimensions and thus that node has two, four, or six connecting links¹. These connecting links can connect to other region nodes, boundary condition nodes (BCs), or nil (which represents the free boundary condition)². Thus a node has six links and up to six neighboring regions.

In this paper we use the graph-based representation to implement DVE. For DVE, the graph node must store the type of coordinate system, the objective function, and constraint

¹ A one- or two-dimensional body can also be represented in the three-dimensional space by utilizing only two or four connecting links of the node.

² Note that a region at a free surface connects to nil; thus although nil represents the free boundary condition, it actually represents the boundary of the body but is not represented by a node.

'category' as the system has no way of telling the type of such attributes, i.e. if function, behaviour or structure.

The other type of objects are objects which are not design objects but are objects to the expert system. These are also formulated as frames but, obviously, there will be no reference to any design prototype. Examples of such frames are shown below:

clause	subclause	
	applicability	
applicability		
options: determined		options: determined
affects: [BCA_R60, BCA_R70]		affects: [BCA_R80]
subclause		dependent_on: [BCA_R70]
affects: [BCA_R60]		exemption
compliance		options: applicable
options: satisfactory		affects: [BCA_R80]
dependent_on: [BCA_R60]		dependent_on: [BCA_R90]
		compliance
		options: satisfactory
		dependent_on: [BCA_R80]

The frame part of the system makes use of the usual properties of frame systems.

Design Description—CAD Database

Here we will assume a level of information in the CAD database corresponding to objects in the design. That is, we are not concerned with the interpretation of arrangement of pure geometrical features, such as lines, to derive features or objects as is the work of Nnaji and Kang (1990). We are therefore assuming that the CAD system allows the users to create 'objects'. We have also previously stated that all objects to be described using the CAD system must exist as design prototypes. Nevertheless, the representation of such objects in the CAD database will be in a format incompatible with that of the domain knowledge and must be interpreted to produce descriptions which are meaningful to the rest of the system.

CAD Database Interpreter

In the process of integrating CAD systems with expert systems, the information presented graphically must be converted to a form that can be understood by the expert system, that is to a form commensurate with that of the design prototype format. For this purpose a CAD database interpreter (DBI) must be used. Where the CAD system database uses a standard graphic database format such as IGES or DXF or higher level format the interpreter must convert these formats to that of the design prototype format while if the CAD system database uses some non-standard format the interpreter must be written specifically for the particular CAD system used. CAD systems consist of a graphical interface and a database. The database of a CAD system contains representations of drawing elements and numeric or alphanumeric information associated with those elements. The syntactical information, namely, in the form of dimensions, locations, shapes, etc. can be mapped onto the structure properties of a design prototype. The graphic database interpreter consults the appropriate design prototypes

connectivity: The region is removed from the graph; if the resulting graph is connected then removal of the region is permitted (Cagan, 1990).

The graph representation described is a highly abstract representation of the geometry of the structure. The semantic interpretation of the geometric connectivity is unique to DVE and the 1stPRINCE methodology but may also be utilized by other design algorithms. With these semantic considerations of the geometric relationships between different regions, much information is made easily available for problem formulation, analysis, and manipulation.

DIMENSIONAL VARIABLE EXPANSION ALGORITHM

In this section, an algorithm is presented to automate DVE with the graph representation. The concept of the Dimensional Variable Expansion Algorithm is shown in Figure 5. The body which is to be expanded in the i dimension is shown in Figure 5a. For each node location in the i dimension the algorithm selects the slice in the j - k plane at the i location. In Figure 5b that slice (i) is duplicated (i') and linked into the graph next to slice i . By repeating this process for all slices of i , the graph is expanded along the i direction.

The algorithm is given in Figure 6. Note that more than one critical variable (CV) can be selected implying that the body can be expanded across more than one dimension. If more than one dimension is selected then each dimensional variable expansion is completed before the next dimension is expanded. The algorithm first determines critical variables. In 1stPRINCE, a variable is candidate to be critical if it appears in an active constraint in some case from a monotonicity analysis or numerical optimization analysis. Currently if a dimensional variable is a candidate to be critical, it is assumed to be critical and then later verified.

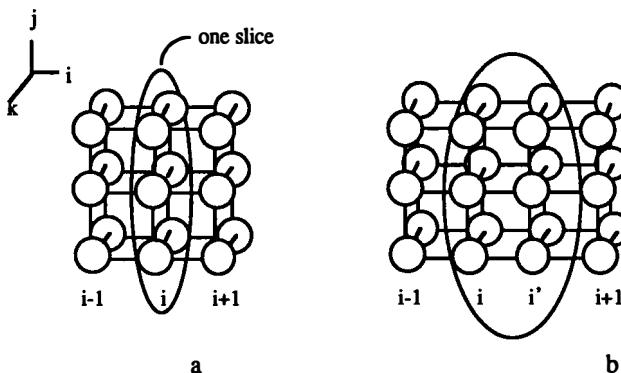


Figure 5. Demonstration of the Dimensional Variable Expansion Algorithm in the graph representation

```

Begin
    Determine critical variables (CVs);
    For each dimensional CV Do
        Begin
            For each location of nodes in dimension of CV Do
                Begin
                    Let slice of nodes be called S1;
                    If S1 is not a slice of only boundary nodes Then
                        Begin
                            Duplicate S1 and call it S2;
                            Link all CV+ nodes of S2 to nodes linked by
                                analogous CV+ nodes of S1;
                            Link all CV+ nodes of S1 to analogous nodes of S2;
                        End
                    End
                End
            End
        End

```

Figure 6. Algorithm for DVE

Once the CVs are determined, then for each CV a slice is selected, duplicated, and re-linked into the graph. Selecting a slice is not trivial. In Figure 7, referring to the i-direction, there are three levels of slices. At levels 2 and 3 the slices are non-connected. In order to determine all of the nodes in a given slice, a depth-first search must be performed keeping records of levels of the slices by adding and subtracting level numbers as the algorithm progresses up and down in the appropriate direction. Pointers to all of the nodes in the desired level are stored on a list and then each node is duplicated. To guarantee that all nodes are duplicated in the body, the node of smallest level in that direction must be found. For efficiency, when searching for that smallest level node, all nodes are checked and their relative level recorded. Nodes are sorted by level and then each node can be taken off the acquired list of nodes of ascending level.

In Figure 7, nodes 1, 2, and 3 are at level 1 of the i-direction; nodes 4 and 6 are at level 2; and nodes 7 and 9 are at level 3. To expand the body in the i-direction, first the slice at level 1 (nodes 1, 2, and 3) is divided and duplicated, then the nodes at level 2 (nodes 4 and 6), and finally the nodes at level 3 (nodes 7 and 9). The resulting body appears in Figure 8 where node numbers take the form: *parent-number.child-number*, i.e., the number of the region before DVE dotted with the current number of the node as a descendant. The slice at level 0 consists of all boundary nodes (BC) and thus is not expanded.

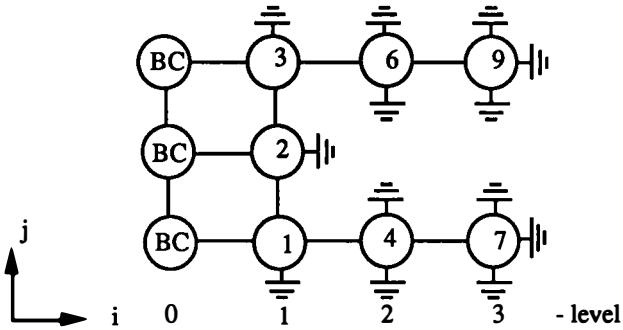


Figure 7. Graph showing levels of nodes with non-connected slices

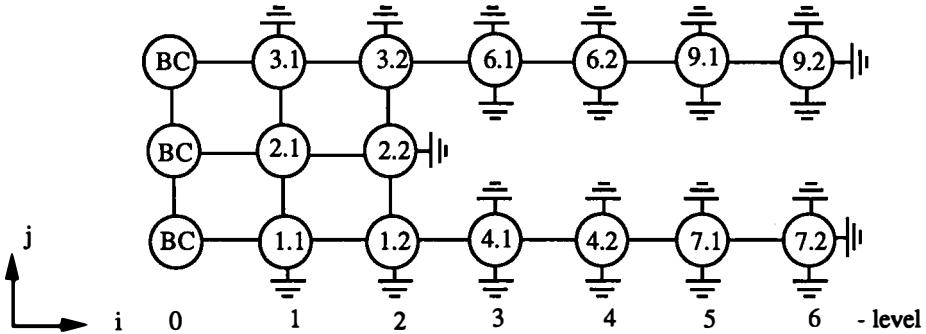


Figure 8. Graph of Figure 7 after DVE

The algorithm performs a depth-first search visiting each node via each link to determine node levels and then visiting each node and link again to duplicate each slice. This portion of the algorithm runs in time $O(|N| + |L|)^4$, where N designates all nodes and L designates all links. Boundary nodes are treated differently than region nodes and may or may not be duplicated.

Sorting can be performed with any sorting algorithm. Mergesort runs in $O(|N| \log |N|)$ in both average and worst case (Kruse, 1984). Thus the order of the sort algorithm controls the order of the entire algorithm unless a sort routine can run in less than $O(|N| + |L|)$. The DVE algorithm, using mergesort, runs in $O(|N| \log |N|) + O(|N| + |L|)$. In the implementation, $|L| \leq 6|N|$; thus, the algorithm runs in $O(|N| \log |N|)$.

* Given a function $f(x)$ and constant $K \geq 0$, $f(x)$ is $O(x)$ means $|f(x)|/x \leq K$ as $x \rightarrow 0$ (Luenberger, 1984).

REPRESENTATION EXAMPLE

As an illustration of the use of the graph representation for storing constraint information, Figure 9a shows a clamped beam with four regions and a flexural load at the end, and Figure 9b shows the graph-based two-dimensional representation. The ground sign represents nil, implying free boundary conditions. Boundary information is incorporated into the region constraints; however, at the two ends there are "LIMBC1" and "LIMBC2" nodes which represent the geometric limits of the beam. The following is a set of constraints for all the regions and boundary nodes. Constraints limit geometry and bending stress; however, for this discussion, only the number of the regions is important; symbols such as "x1" mean the "x" value of region number 1 and "xLIMBC1" and "xLIMBC2" mean the "x" value of the LIMBC1 and LIMBC2 boundary nodes. The constraints are:

For Region 1:

$$\begin{aligned}x_1 &> x_{\text{LIMBC1}} \\x_2 &> x_1 \\ \sigma_1 &\leq \sigma_y \\ \sigma_1 &= \frac{4 p x_1}{\pi r_1^3}.\end{aligned}$$

For Region 2:

$$\begin{aligned}x_2 &> x_1 \\x_3 &> x_2 \\ \sigma_2 &\leq \sigma_y \\ \sigma_2 &= \frac{4 p x_2}{\pi r_2^3}.\end{aligned}$$

For Region 3:

$$\begin{aligned}x_3 &> x_2 \\x_4 &> x_3 \\ \sigma_3 &\leq \sigma_y \\ \sigma_3 &= \frac{4 p x_3}{\pi r_3^3}.\end{aligned}$$

For Region 4:

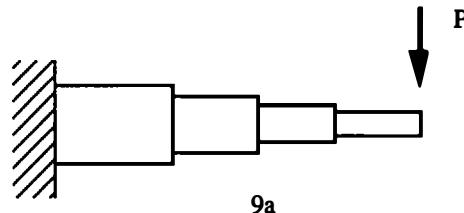
$$\begin{aligned}x_4 &> x_3 \\x_{\text{LIMBC2}} &> x_4 \\ \sigma_4 &\leq \sigma_y \\ \sigma_4 &= \frac{4 p x_4}{\pi r_4^3}.\end{aligned}$$

For Region LIMBC1:

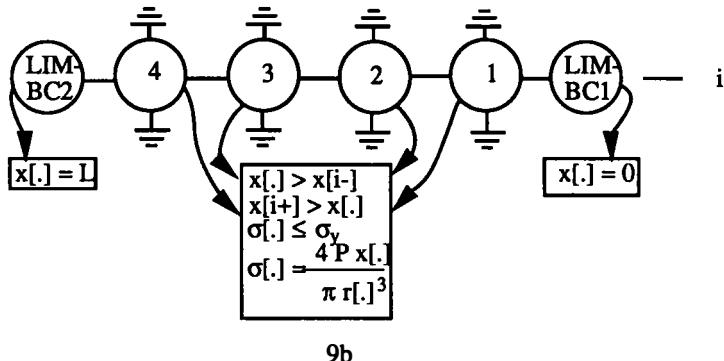
$$x_{\text{LIMBC1}} = 0.$$

For Region LIMBC2:

$$x_{\text{LIMBC2}} = L.$$



9a



9b

Figure 9. Graph representation for a clamped beam under flexural load

Note that each region has a separate set of equations for similar constraints and that the boundary condition regions are only used to give limits on variable dimensions. Each time a dimensional variable is expanded, a new set of constraints must be formulated for each new region taking care to organize appropriate region numbers for bookkeeping purposes. These same constraints can be described utilizing a generic representation, called a *knowledge module*, which references the node links instead of the neighboring node by node number. For the region nodes the generic representation is:

$$\begin{aligned} x[.] &> x[i-] \\ x[i+] &> x[.] \\ \sigma[.] &\leq \sigma_y \\ \sigma[.] &= \frac{4P x[.]}{\pi r[.]^3}, \end{aligned}$$

and for the boundary conditions they are:

$$\begin{aligned} x[.] &= 0 \text{ for region LIMBC1, and} \\ x[.] &= L \text{ for region LIMBC2,} \end{aligned}$$

where,

- [·] designates referenced node,
- [i+] designates node in the i+ direction,
- [i-] designates node in the i- direction.

Constraint $x[\cdot] > x[i-]$ dictates that the x-position of the region which is being referred to must be greater than the x-position of the region neighboring the referenced region in the i- direction. Similar interpretations can be made for the other constraints.

From this example we can recognize an advantage of the graph-based representation that *a general set of constraints can be formulated and utilized for each node by referring to neighboring nodes via these links rather than the actual node names*. When a dimensional variable is expanded, instead of formulating a new set of constraints for each new region, a link from each new node simply points to the same, generic set of constraints as discussed here. When a neighboring node is replaced by a different node, the set of constraints need not be modified because they refer to the node *linked* to the present node and *not* to an instance of the node. This property is fundamental in making methodologies like 1stPRINCE general techniques able to reason about and expand structural features. Without this generality, every change made to the body would require detailed record-keeping and name modification which would make a program inefficient.

IMPLEMENTATION

The graph-based representation with face links has been implemented to automate DVE and show feasibility in CommonLisp and Flavors on a MAC II. This research is part of a long-term effort which will develop a computational representation in which a design can be conceptualized, modified, and improved while reasoning from a fundamental level of knowledge.

RELATION TO OTHER WORK

In this section we examine geometric modeling and finite element methods (FEM), demonstrating the differences between these methods and the proposed work. We also discuss future extensions to the current work to incorporate geometric modeling and FEM, describing a potentially powerful design tool.

Geometric Modeling

The representation described in this paper provides a framework with which to perform mechanical/structural design. Geometric modeling techniques (Mortenson, 1985; Requicha and Voelcker, 1982; Requicha, 1980; Eastman, 1970; Gursoz, Choi, and Prinz, 1990) provide tools which aid designers in visualization, analysis, and manufacturing of a pre-defined design. Geometric modeling techniques are utilized to define the geometric features and shape of an artifact for computational purposes. A geometric model can be sliced to see the interior of the object, two objects can move together to check interference between their geometries, the inertial properties of the object can be calculated, paths for numerically controlled (NC) machining can be calculated, and it can be used as a preprocessor for finite element mesh generation (creating the series of nodes for a finite element model).

Current geometric modeling representations offer no means of modeling behavioral design constraints and are thus not sufficient for design innovation. For example, the Boolean trees utilized in solid modeling are currently not a useful representation for 1stPRINCE because they do not provide the tools necessary to divide structures via DVE and remove regions while maintaining connectivity; there is no implemented mechanism to have one region linked directly to another for use with knowledge modules as is done with the graph representation presented in this paper. Boundary representation (b-rep) methods utilize a graph-based representation; however, their topology is limited to vertices, edges, and faces, but not regions which are expanded. There is currently no way to map the regions of a b-rep to sets of constraint equations. Thus current geometric modeling techniques do not contain the necessary mechanisms to determine which nodes to select for expansion or how to store the new set of nodes to maintain their geometric connectivity relationships as required by 1stPRINCE and DVE.

Geometric modeling is a visualization modeling tool rich with geometric information. In future research, a mapping between the geometric modeling environment and the connectivity graph described in this paper will be developed to make possible design innovation within a geometric modeling environment. Within this powerful environment, analysis of the new designs will be possible with the finite element method, as well as analysis of manufacturing information from NC machining.

The Finite Element Method

DVE may appear similar to the finite element method (FEM) since a set of regions is divided into a larger set of smaller regions to describe a structures problem, however the

similarities stop there. FEM (Cook, 1981) is a numerical approximation technique to model the behavior of a structure. DVE is a technique to *expand* the design space to make innovative *design* possible, utilizing a set of symbolic design constraints. There are techniques to automatically generate and refine a nodal mesh from a solid model (e.g., Kela, Perucchio, and Voelcker - 1986), but they do not expand the set of design variables to reason about and are only dividing a fixed geometry into a group of smaller elements. Cagan (1990) discusses the need to utilize the FEM to analyze complicated structures within 1stPRINCE, however the current approaches are quite distinct from the FEM theory.

FUTURE EXTENSIONS

The connectivity graph introduced in this paper has been implemented in the 1stPRINCE program. A possible extension is to consider the cross-coordinate axis links (Figure 10 shows a planar sketch of these links). Such an extension to the representation would give the 1stPRINCE design methodology more flexibility for domains outside the structures domain. In addition, even within the structures domain, this would create a richer environment in which to perform design tasks; however, the extensions would have the same complication as with the current representation in that physically they are poor at transmitting load in structures. These extensions could be utilized in the structures domain as long as some additional load path is included in a design to transmit the load. We are also currently investigating other applications of the graph representation beyond the structures domain.

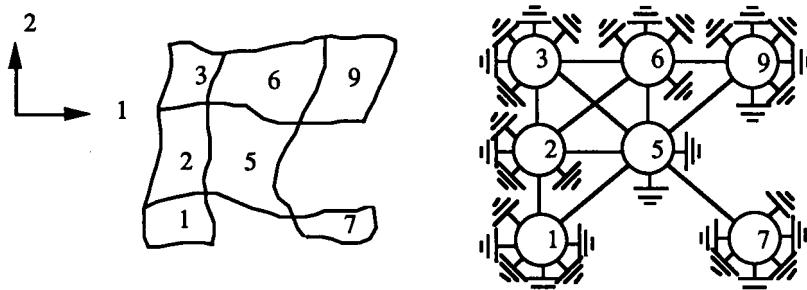


Figure 10. Demonstration of coordinate axis and proposed off-coordinate axis links

CONCLUSIONS

This paper discusses a theoretical foundation for representing physical bodies that is used in implementing the 1stPRINCE methodology for innovative design. A graph-theoretic approach is utilized where body regions are represented by graph nodes and graph links represent connections between different regions. With this model, symbolic information about constitutive, geometric, and other behavioral information can be assembled and manipulated. Utilizing this geometric connectivity representation, a computational methodology is able to perform DVE, node removal, and general manipulations of the body which aid in non-routine design.

The connectivity graph is an abstract representation of the geometric object; the graph maintains the major features of the object without the detailed information required in a geometric model. It is because of these abstractions that the representation is computationally useful. If computational environments are to reason at a conceptual level, they will need to make use of different levels of abstraction. Conceptual design is combinatorially explosive; reasoning at abstract levels could significantly control the combinatorics. How general and versatile this representation is still needs to be determined; however, in all of the problems solved by 1stPRINCE with the connectivity graph, the representation was sufficient to model the designs.

Acknowledgements. The author would like to thank Alice Agogino for her important discussions about this work and Steve Bradley for his comments on this manuscript.

REFERENCES

- Cagan, J. (April 1990). *Innovative Design of Mechanical Structures from First Principles*, Ph.D. Dissertation, University of California at Berkeley.
- Cagan, J. and Agogino, A. M. (1987). Innovative Design of Mechanical Structures from First Principles, *AI EDAM: Artificial Intelligence in Engineering, Design, Analysis and Manufacturing*, 1(3): 169-189.
- Cagan, J. and Agogino, A. M. (1990). Inducing Optimally Directed Non-Routine Designs, *Preprints Modeling Creativity and Knowledge-Based Creative Design*, Design Computing Unit, University of Sydney, Sydney.
- Cook, R. D. (1981). *Concepts and Applications of Finite Element Analysis*, John Wiley, New York.

- Eastman, C. M. (1970). Representations for Space Planning, *Communications of the ACM*, 13(4): 242-250.
- Gursoz, E. L., Choi, Y. and Prinz, F. B. (1990). Vertex-based Representation of Non-manifold Boundaries, in M. J. Wozny, J. U. Turner, and K. Preiss (eds), *Geometric Modeling for Product Engineering*, North-Holland, New York, pp. 107-130.
- Kela, A., R. Perucchio and Voelcker, H. (1986). Toward Automatic Finite Element Analysis, *Computers in Engineering*, July, pp. 57-71.
- Kruse, R. L. (1984). *Data Structures & Program Design*, Prentice Hall, Englewood Cliffs, NJ.
- Luenberger, D. G. (1984). *Linear and Nonlinear Programming*, Addison-Wesley, Reading, MA.
- Mortenson, M. E. (1985). *Geometric Modeling*, John Wiley, New York.
- Papalambros, P., and Wilde, D. J. (1988). *Principles of Optimal Design: Modeling and Computation*, Cambridge University Press, Cambridge.
- Requicha, A. A. G. (1980). Representations for Rigid Solids: Theory, Methods, and Systems, *Computing Surveys*, 12(4): 437-64.
- Requicha, A. A. G. and Voelcker, H. B. (1982). Solid Modeling: A Historical Summary and Contemporary Assessment, *IEEE Computer Graphics and Applications* 2(2): 9-24.

Evaluating the patentability of engineered devices

S. M. Kannapan and K. M. Marshek

Department of Mechanical Engineering

University of Texas at Austin

Austin TX 78712 USA

Abstract. Utility, novelty and non-obviousness are key requirements that are used in evaluating patentability. Notwithstanding the extensive patent literature and judicial case history, clear criteria for evaluating novelty and non-obviousness are yet to be developed. This paper presents an approach to the evaluation of patentability based on a representation scheme for a design library of prior art that integrates the structure, behavior and function aspects of design descriptions. The determination of the novelty and obviousness/non-obviousness of synthesized designs are then shown to be expressible in terms of the designs present in the design library. A means for quantification of non-obviousness (*degree of non-obviousness*) is proposed. Examples of mechanical devices are used, including those from an actual patent evaluation process, to illustrate the approach.

INTRODUCTION

The evaluation of the patentability of a device based on technical criteria rests on three conditions with respect to the prior art:

- (a) the utility of the device;
- (b) the novelty of the device;
- (c) the non-obviousness of conceiving the device
by a person of ordinary skill in the art.

Notwithstanding the extensive patent literature and judicial case history, clear criteria for evaluating novelty and non-obviousness are yet to be developed. The determination of utility, novelty and non-obviousness of a given device in relation to a specified prior art remains a matter of natural language argument. This paper presents an approach to the evaluation of utility, novelty and non-obviousness based on a representation scheme for a

design library that integrates the structure, behavior and function (purpose) aspects of design descriptions. The prior art is viewed as a "closed world" design library. The determination whether a device is novel and an obvious/non-obvious synthesis of prior art is then shown to be expressible in terms of the structure, behavior and function of designs present in the design library. The approach described is intended as an aid to the intellectual process of patent evaluation and not as a conclusive consideration for patentability that is automatically computed. Simple mechanical devices are used to illustrate the ideas of utility, novelty and non-obviousness. An actual patent examination situation dealing with the kinematics of mechanical actuators is also analyzed.

An Example

Figure 1 shows schematics of three examples of inventions: a tricycle, a unicycle, and a gyro. The questions that characterize the issues addressed here with respect to a design library comprising the tricycle and gyro are:

- (a) Can the unicycle be shown to have utility?
- (b) Is the unicycle novel with respect to the tricycle?
- (c) Is the unicycle obtainable by synthesis using the tricycle and the gyro?
If so, is it a non-obvious synthesis?

PATENT EVALUATION IN THE UNITED STATES

The history of recent patent evaluation in the United States is an interesting reflection of the evolution of legal, political and technical expertise in protecting the interests of inventors as well as society at large (Vaughan, 1972). Here is a brief non-legal view of the events.

Before the middle of the nineteenth century, patents were meant to protect an inventor's interest by registering novel and useful devices at the Patent and Trademark Office. In 1851, in *Hotchkiss v. Greenwood*, an additional requirement for patentability was introduced in a lawsuit involving a clay doorknob. The additional requirement was that a patentable device should require more skill to conceive than ordinarily available (Vaughan, 1972) (Harmon, 1988). In the following years the granting of patents required such high standards that only one in five patents were held valid in courts (Harmon, 1988). In response to protests from Congress and the Patent Bar, U. S. C. 103 of the Patent Act was passed in 1952 that formalized the requirement that the device to be patented be non-obvious to persons of ordinary skill to conceive with respect to the prior art at the time the device was invented. However, this act was interpreted differently by different courts giving rise to non-uniformity on the central question of non-obviousness (Harmon, 1988).

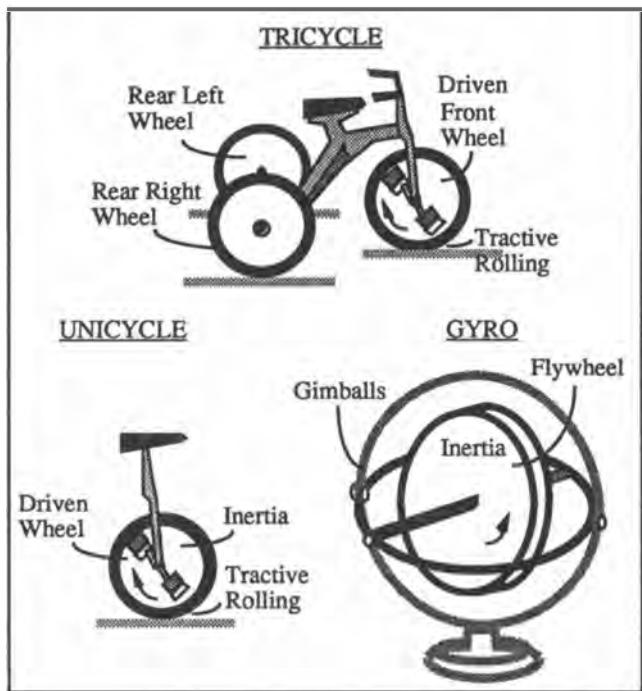


Figure 1: Schematics of three inventions.

In 1966, in *Graham v. John Deere*, the supreme court declared that the determination of patentability was essentially a legal question and set forth a procedure for determination of non-obviousness. But the technical process of determination of obviousness was still undefined. Several notions of novelty and non-obviousness such as synergy, combination patent, etc. were used to aid technical judgement without satisfactory results (Harmon, 1988) (Pressman, 1986).

Failing clarity in technical evaluation, so called secondary considerations, such as, commercial success, long felt need and past failure were used to judge patentability claims. Finally, in 1982, a Court of Appeals for the Federal Circuit was set up to handle all such cases. The Federal Circuit has since attempted to deal uniformly with all patentability claims essentially using the *Graham v. Deere* procedure for determining non-obviousness but generalizing cases to the minimum degree while also using secondary considerations (Harmon, 1988) (Pressman, 1986). As a result of all these developments the question of technical evaluation of devices for utility, novelty and non-obviousness has essentially been sidestepped -- see for example, (Harmon, 1988), and (Pressman, 1986) pp. 74-77.

REPRESENTATION SCHEME FOR PRIOR ART

A useful interpretation of prior art is that of a design library representing the designs of known devices. The design library represents designs in terms of systems and relationships, their hierachic structure, behavior/communication, and function; details of the representation scheme and definitions of terms can be found in (Kannapan and Marshek, 1990, 1991a). Laws of nature, methods of science, and ideas are not patentable (Harmon, 1988) (Pressman, 1986) and are not represented in the design library as prior art.

The representation scheme for designs is summarized as follows. The design library represents the design of a device as a system (Sys) that is decomposed to a configuration or Structure of specific instances of systems (Sys_1, Sys_2, etc.) and specific instances of relationships (Rel_1, Rel_2, etc.) -- see Fig. 2. Systems have Attributes and associated Behavior. Similarly relationships have Attributes and associated Communication. Behaviors and Communications are predicates that hold on "items" such as force, motion, etc. that "flow" through the Ports of Sys and Rel (Kannapan and Marshek, 1989, 1990, 1991a). In Fig. 2, B1 is the Behavior of Sys_1, C1 is the Communication of Rel_1 etc., B-Reqd. is the required Behavior for the device, and a1, a2, etc. are Attributes. The composite behavior obtained from the Behaviors/Communications of the instances of Sys/Rel in the Structure produces the required Behavior of the device (Kannapan and Marshek, 1990, 1991a). A Primary Function of a Sys/Rel instance used in a device is a part of the definition of its Behavior/Communication that is utilized to achieve a required device behavior. The Secondary Function is the part of the definition of required device behavior that is achieved by a Primary Function (Kannapan and Marshek, 1989, 1990, 1991a). For example, in Fig. 2, a Primary Function of Sys_1 is B11, and its Secondary Function is BR1.

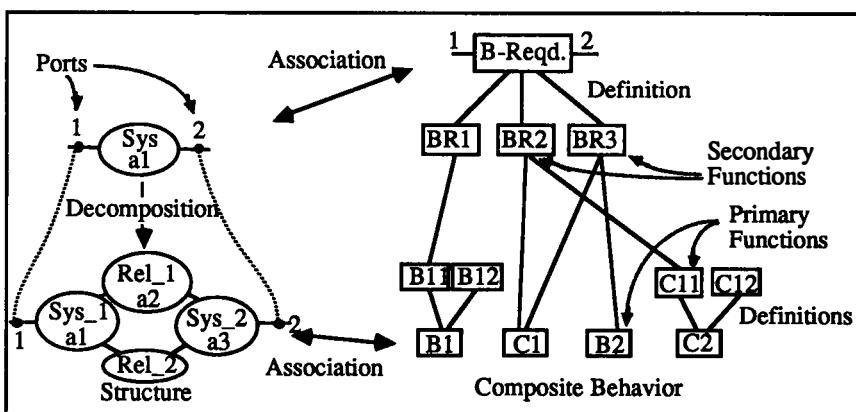


Figure 2: A scheme for representing a prior art device in a design library.

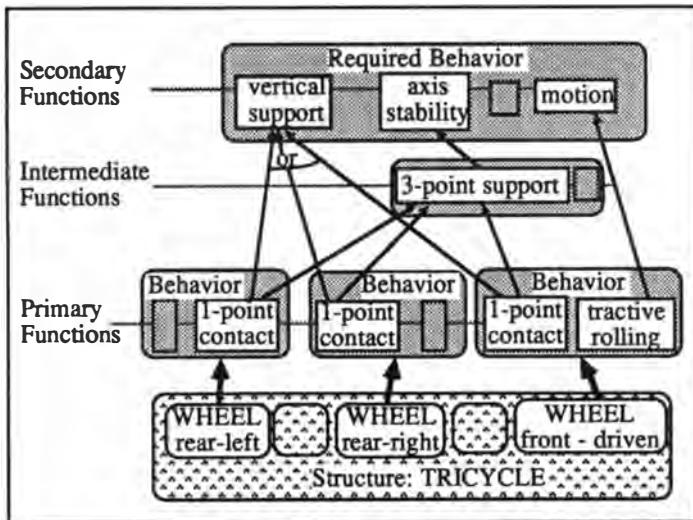


Figure 3: Functions and required behavior for the tricycle.

EXAMPLE:

UTILITY, NOVELTY AND NON-OBVIOUSNESS OF DEVICES

The approach developed in this paper for evaluation of novelty and non-obviousness is intuitively motivated in this section.

Consider the example of the tricycle, the unicycle, and the gyro introduced earlier. Figure 3, Fig. 4 and Fig. 5 show relevant components of structure, behavior, and function for the three devices. Figure 3 shows the required behavior of the tricycle as: vertical support, axis stability and motion. The vertical support requirement and the axis stability requirement together allow the vehicle to stay erect. Relevant components, i.e., the three wheels of the structure of the tricycle, and their associated behaviors are also shown in Fig. 3.

There are many facets of component behavior with only the primary functions being utilized to produce the required behavior. For example the primary functions of the rear wheels of the tricycle are to support the tricycle while the primary functions of the front wheel is both to support and move the tricycle. The primary functions directly or indirectly satisfy one or more behavior requirements (the secondary functions). Here, the "1-point contact" primary functions of the wheels together produce the intermediate function of "3-point support" that produces the secondary function of "axis stability". One or more of the "1-point support" primary functions is sufficient to satisfy the secondary function of "vertical support". The "tractive rolling" primary function of the front wheel produces the

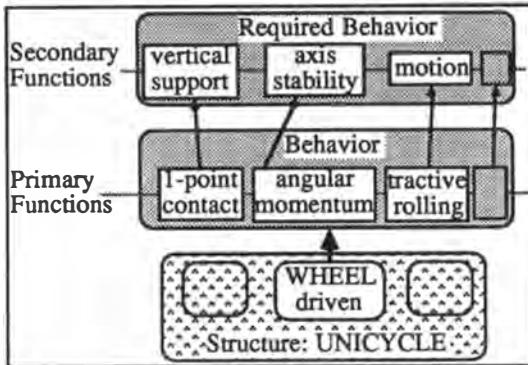


Figure 4: Functions and required behavior for the unicycle.

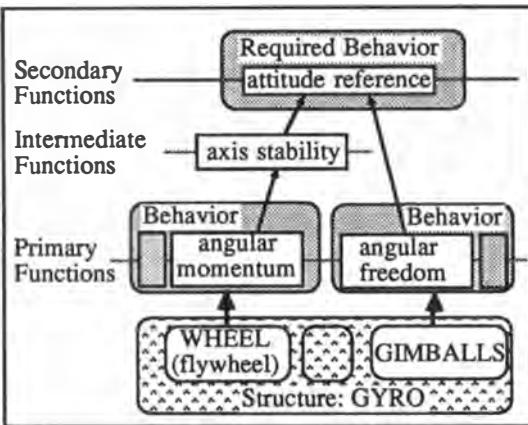


Figure 5: Functions and required behavior for the gyro.

secondary function of "motion". The mappings from the primary functions to the secondary functions for the components are shown by arrows in Fig. 3.

The functions and required behavior shown in Fig. 4 and Fig. 5 for the unicycle and the gyro are organized as in Fig. 3. The required behavior for the unicycle in Fig. 4 is the same as that for the tricycle but the relevant component of structure is simply a single **wheel**. In Fig. 5, an important required behavior of a directional gyro is that of providing an attitude reference (Siff and Emmerich, 1960). The relevant gyro structure components are a rotating wheel with inertia (flywheel) that produces angular momentum (primary function) that in turn produces axis stability (intermediate function). The angular freedom (primary function) in two axes (Siff and Emmerich, 1960) provided by the gimbals of the gyro, together with the axis stability provided by the flywheel, produce the secondary function of attitude reference.

We now consider the basis for evaluation of utility, novelty and non-obviousness.

Utility

What is the utility of a device? A simple but intuitively powerful notion is that utility arises from the use of the device to achieve an aspect of the behavior of a larger device or system. Because, if the device cannot be used in any way to achieve some larger purpose, it has no utility. Thus the utility of a device can be evaluated on the basis of its purpose, i.e., the secondary function of the device. In the context of a hierarchic design library of prior art, the utilities of a *component* are its secondary functions *in a device* that uses the component, the utilities of the *device* are its secondary functions *in a larger device* and so on. For example, the utility of the flywheel in the gyro is to provide attitude reference. The utility of the gyro in, say, an aircraft is to provide a homing capability, and so on. Ultimately, in a finite hierarchy of prior art in a design library, the top level devices will not be used in a larger system and its purpose will have to be defined external to the design library.

Novelty

Let us evaluate whether the unicycle is novel with respect to the tricycle. At first glance it might seem that the unicycle is not novel (i.e., the tricycle is prior art, and the tricycle anticipates the unicycle -- see (Harmon, 1988)), since the unicycle is simply a tricycle with its rear wheels removed, and also has the same required behavior. That is, the structure of the unicycle is essentially obtainable from the structure of the tricycle, and is meant to achieve the same result. However, a closer look at the primary and secondary functions of the wheel in the unicycle (Fig. 4), and the front wheel of the tricycle (Fig. 3), indicates that the wheel in the unicycle distinguishes itself in a novel way from the front wheel of the tricycle by also serving the purpose of axis stability by utilizing angular momentum. This example illustrates that the determination of the novelty of a device with respect to another device requires not only comparison of structure and required behavior but also primary and secondary function. Thus, a design is novel with respect to a prior design if the components, structure, functions or required behavior aspects of the design, expressed as in Figs. 3-5, are not identical in the prior design.

Non-obviousness

The question now is whether, the unicycle can be synthesized from the tricycle and gyro. Inspection of the gyro shows that to achieve the requirement of axis stability the angular momentum behavior of the flywheel can be utilized. The tricycle has an axis stability behavior requirement that can directly use the teaching of the gyro design. By replacing the

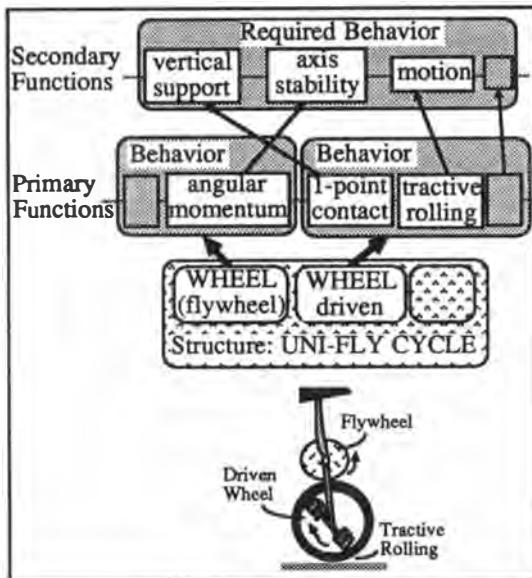


Figure 6: Functions and required behavior for the uni-fly cycle.

manner of achievement of the axis stability requirement in the tricycle by that of the gyro, the resulting design and its function and required behavior aspects are shown in Fig. 6. We call it a uni-fly cycle and picture it schematically at the bottom of Fig. 6. *However, this does not produce the same structure as the unicycle.* The design of the unicycle uses the knowledge that a driven wheel used for tractive rolling can also act as a flywheel (see Fig. 4); a piece of knowledge not taught by the gyro or the tricycle. Thus, although the structure of both the uni-fly cycle and the unicycle can be obtained by synthesis from the structures of the gyro and tricycle, we propose that the uni-fly cycle is an "obvious" synthesis that exhibits re-use of known purpose -- i.e., Function Re-Use (Kannapan and Marshek, 1990), while the unicycle is a "non-obvious" synthesis that exhibits a new use of a component -- i.e., Function New-Use (Kannapan and Marshek, 1989, 1990, 1991a). Also, the unicycle achieves axis stability by a wheel that is already used for other purposes in the unicycle; this demonstrates Function Overload (Kannapan and Marshek, 1989, 1990, 1991a).

The notion of obvious/non-obvious synthesis presented here is based on the theory that a "closed world" design library representing prior art is principally indexed by its function (primary, intermediate and secondary). By Function Re-Use, if a need arises to achieve a behavior requirement, we match the behavior requirement to an index of the design library and attempt to re-use a component or subsystem for the same purpose it was used previously. We propose that it can be considered "obvious" to use knowledge of past utilization again. By Function New-Use and Function Overload, an object is used in a way that it has not been used in the past; this requires the use of objects in the design library

without the principal index of function for the design library. We propose that a synthesis of components can be considered "non-obvious" not necessarily because the components are novel but because they are utilized in a new way to achieve some purpose (also see (Pressman, 1986) pp. 74-77, for similar views). By these notions of obviousness and non-obviousness, the uni-fly cycle is an obvious synthesis while the unicycle is a non-obvious synthesis from prior art of the tricycle and the gyro.

Further, a degree of non-obviousness can be devised to quantify non-obviousness based on the mapping of primary and secondary functions once the structure is synthesized from two or more past designs to satisfy the behavior requirement. The degree of obviousness is simply the number of mappings between primary functions, intermediate functions and secondary functions that are new with respect to a design library of prior art. It is to be noted that even if the number of components and the number of function mappings increase, only the mappings that are new with respect to the design library will contribute to the degree of non-obviousness. The uni-fly cycle in Fig. 6 has no new primary to secondary function mappings with respect to Fig. 3 and Fig. 5, and hence its degree of non-obviousness is 0 with respect to a design library comprising the tricycle and gyro. Comparison of Fig. 4 with respect to Fig. 3 and Fig. 5 shows that there is one function mapping from angular momentum to axis stability for the wheel that is new in Fig. 4. All other function mappings for the unicycle in Fig. 4 are present in the tricycle and gyro designs. Hence the degree of non-obviousness for the unicycle is 1. Such an evaluation of the degree of non-obviousness, in general, will depend on the completeness and level of detail of the design library and designs that are modeled.

UTILITY, NOVELTY AND NON-OBVIOUSNESS EVALUATION

With respect to a "closed world" design library containing the designs D_x and D_y , the utility, novelty and non-obviousness of a design D_z can be evaluated as follows:

Utility: if there exists a secondary function for D_z in some other design D_x , or there exists a secondary function for D_z identical to the required behavior of a design D_y , then D_z has utility. (The utility of D_z may be absent within the design library but present outside the design library, i.e., in the real world. This cannot be avoided in any finite design library.)

Novelty: if there exists no design D_x or any of its sub-designs D_y that is equivalent to D_z in structure and required behavior, and with identical mappings between primary, intermediate and secondary functions for all components in the structure, then D_z is novel.

Non-Obviousness: if there exist one or more mappings between primary, intermediate, and secondary functions in D_z that are absent from every design D_x synthesized from the design library and equivalent in structure to D_z , then D_z is non-obvious. (The presence of such mappings demonstrate Function New-Use, and sometimes Function Overload also, in the device. The degree of non-obviousness is the number of such new mappings. A design with a zero degree of non-obviousness is an obvious design that demonstrates Function Re-Use only.)

The devices are limited here to single state, time invariant structures, behaviors, communications and functions. The meanings of the terms, equivalence of structure, sub-design, and synthesis (Kannapan and Marshek, 1989, 1990, 1991a) are explained briefly as follows.

The equivalence of structure between two designs D_x and D_y requires recognition of identity between systems and relationships in the structure of the two designs by matching attributes and behavior/communication of systems and relationships. For example, D_x may use a component called "gear" and D_y may use a component called "cog". Their identity must first be recognized using identical attributes of cylindrical body shape, an axis of symmetry, peripheral teeth etc., and identical rigid body kinematic behavior. Once the identity of systems and relationships in D_x and D_y can be mapped one-to-one, D_x and D_y are equivalent in structure if the same port connections and attribute constraints exist between the equivalent Sys and Rel in the structures of D_x and D_y (Kannapan and Marshek, 1991b).

A sub-design of a design D_x is another design D_y with a structure equivalent to a substructure of D_x , and with all systems and relationships in the substructure having primary functions and secondary functions present in D_x (Kannapan and Marshek, 1991b).

A synthesis of designs D_x and design D_y replaces one or more sub-structures of D_x by one or more sub-structures of D_y to produce a design D_z . The synthesized design D_z satisfies a required behavior with potentially new primary, intermediate and secondary function mappings (Kannapan and Marshek, 1991b).

Uniqueness and Complexity

The uniqueness of a design representation and the complexity of a design are two issues that affect patentability evaluation. The first issue is that in an unrestricted representation language for designs, the representation of a design may not be unique. That is, a design that is to be evaluated (i.e., a subject design) may be represented in alternative ways making it difficult to recognize its equivalence to a prior art design. The problem of recognition can be simplified by introducing the restriction that the subject design must be modeled only using components, relationships and behavior/communication predicates already present in

the design library. If the modeling of the subject design requires the addition of a component, relationship or predicate to the design library, then the addition is not prior art and is treated as a discovery (Kannapan and Marshek, 1991b).

The second issue arises if the design of a system is complex in terms of the number of components it involves or in the number of predicates required to describe its behavior. It is usually convenient to model a complex design as hierarchical decompositions of its structure and behavior (Kannapan and Marshek, 1989, 1990, 1991a). In that case, the recognition of equivalence is possible at any level in a decomposition hierarchy. If a sub-system of a subject design and a sub-system of a prior art design are recognized to be equivalent based on their attributes and behavior, the patentability evaluation problem can be divided into two parts. The first part is the evaluation of patentability of the matched subsystem of the subject design. The second part is the evaluation of patentability of the subject design with the matched subsystem considered as a component with no internal structure. This division is permitted since the internal structure of the matched sub-system of the subject design cannot exhibit new function mappings in achieving the required behavior of the subject design (Kannapan and Marshek, 1991b).

In the patent evaluation examples that follow, designs are modeled by a common set of structure and behavior primitives thus simplifying the recognition of equivalence. Also, a single level of decomposition of structure and behavior is used.

APPLICATION TO PATENT EVALUATION -- AN EXAMPLE

Fig. 7, Fig. 8 and Fig. 9 show the components, relationships, structure and required behavior of three patented devices, viz., the Allen device (Allen, 1965), the MacDonald (or "McD") device (MacDonald, 1973), and the Kannapan and Marshek (or "K&M") device (Kannapan and Marshek, 1988) respectively. Schematics of the devices are also shown in Figs. 7-9.

The Allen device converts translation motion of piston_14 to rotary motion of shaft_20 through a helical spline about axis "ax1". In Fig. 7, "1_m", "2_m", and "3_m" represent motions at Ports 1, 2 and 3 respectively. The conversion of translation to rotation is represented by the predicate "trans-rot". A radial relationship (or cylindrical joint) between piston_14 and cylinder_12 about an axis "ax2" that is parallel but distinct from axis "ax1" equalizes the rotation of the piston_14 and the rotation of the cylinder_12. The predicate "roteq" represents equality of rotation motions. The radial and axial support relationship "radax_26" (like a bearing) equalizes the translation of shaft_20 and cylinder_12. The predicate "transeq" represents equality of translation motions.

Both the McD and K&M devices convert rotary motion of an input shaft to rotary motion of an output member. In Fig. 8 and Fig. 9, the conversion of a rotary motion to

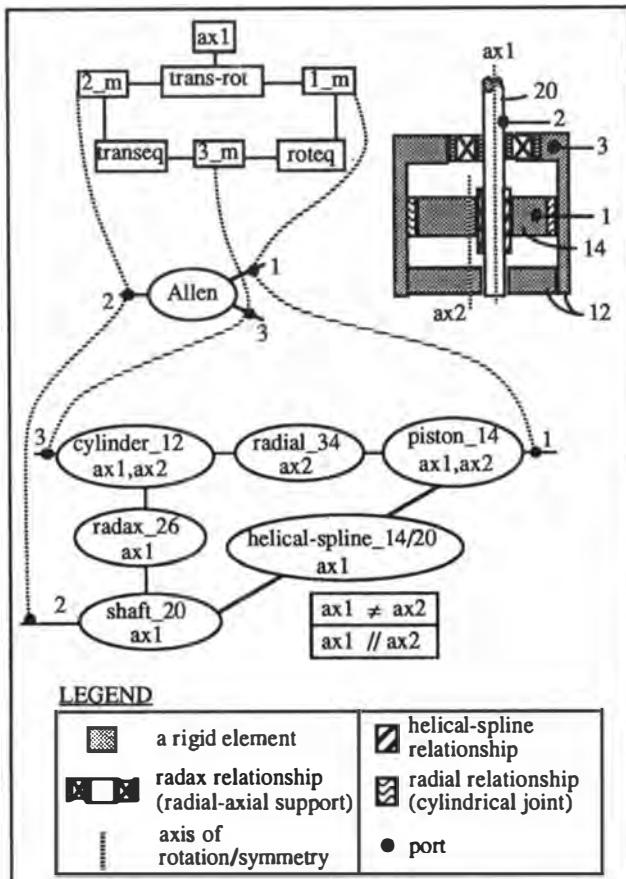


Figure 7: Structure, behavior and schematic of the Allen device.

another rotary motion is represented by the predicate "rot-act". The means of operation of the McD and K&M devices are described in (Kannapan and Marshek, 1989, 1991a).

The questions we address are the following with respect to a design library comprising the Allen and McD devices:

- Does the K&M device satisfy the requirement of utility?
- Does the K&M device satisfy the requirement of novelty?
- Does the K&M device satisfy the requirement of non-obviousness?

Firstly, we must analyze the designs from the points of view of functions and required behavior -- see Figs. 10-12. The bottom half of Fig. 10 shows the behaviors and communications of components and relationships of the Allen device, the top half shows the

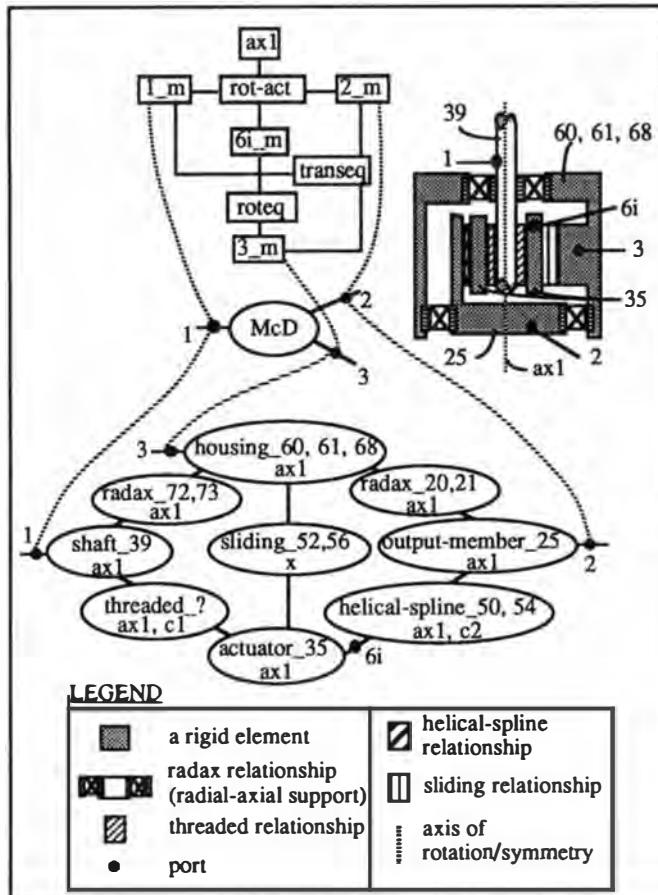


Figure 8: Structure, behavior and schematic of the McD device.

behavior requirements of the Allen device. The mapping lines between primary and secondary functions show how components and relationships are utilized to satisfy the behavior requirements. Fig. 11 and Fig. 12 similarly show the functions and required behavior of the McD and K&M devices respectively (details of the derivation of required behaviors are available in (Kannapan and Marshek, 1989, 1991a)).

Utility

A comparison of required behaviors in Fig. 11 and Fig. 12 shows that the McD and the K&M devices have identical behavior requirements. Since the McD device is prior art in this case, by the definition of utility given earlier, the requirement of utility for the K&M device is satisfied.

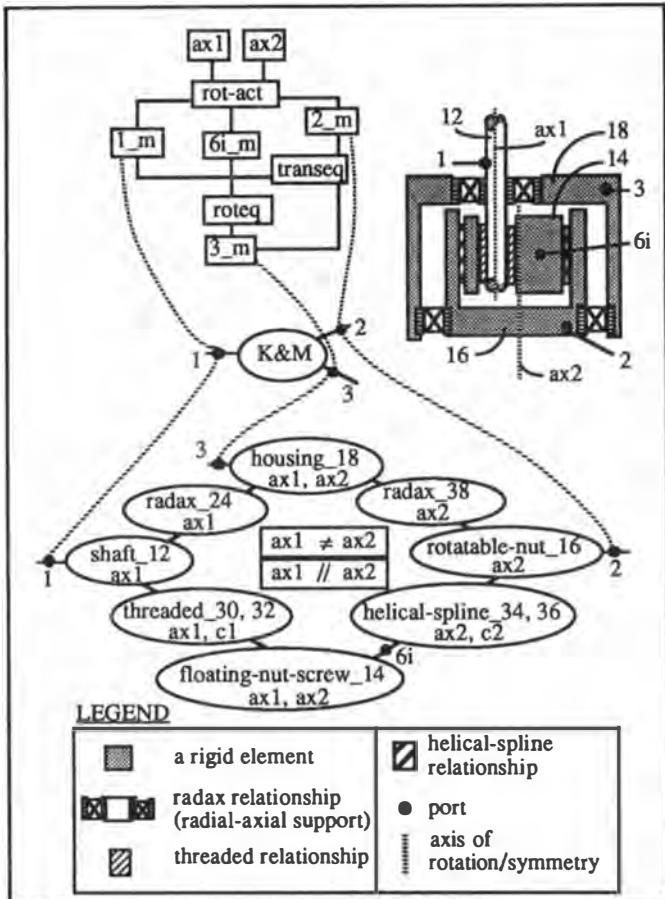


Figure 9: Structure, behavior and schematic of the K&M device.

Novelty

A comparison of systems and relationships having identical attributes in the K&M device with those in the other two devices shows that

- (a) The K&M device is novel over the Allen device since the threaded_30,32 relationship in Fig. 12 does not have an equivalent in the Allen device (Fig. 10).
- (b) A substructure of the McD device (obtained by removing the sliding_52,56 in Fig. 8), has identity equivalent systems and relationships with the K&M structure (see Table 1). However the attribute constraints between axes in the K&M device, viz., inequality and parallelism of ax1 and ax2, are not present in the McD device or its substructure. Hence, the K&M device is novel with respect to the McD device.

Non-Obviousness

As described above in the evaluation of novelty, a substructure of the McD device is equivalent to the structure of the K&M device except for the attribute constraints. The attribute constraints of inequality and parallelism of axes can be introduced into the substructure from the Allen device to obtain the K&M device structure. However, comparison of the primary and secondary functions in Fig. 10, Fig. 11 and Fig. 12 shows that some of the mappings from the primary functions to the "roteq" secondary function for the K&M device (Fig. 12) are not present in the Allen or McD devices and hence demonstrate Function New-Use. Other primary to secondary function mappings in the K&M device are present in either the McD or the Allen device and hence demonstrate Function Re-Use. Since the K&M device demonstrates Function New-Use, the K&M device is non-obvious over the prior art of Allen and McD devices. Further, the degree of non-obviousness of the K&M device is 3; the number of new mappings between primary and secondary functions in the K&M device, shown by bold lines in Fig. 12. Also, if a new design uses the same mappings as past designs but eliminates one or more primary functions for the same secondary function, then the new design is non-obvious (Kannapan and Marshek, 1991b). This source of non-obviousness can be included by adding to the degree of non-obviousness the minimum of the number of mappings present in a past design, but absent in the new design for the same secondary function (Kannapan and Marshek, 1991b).

A design of degree of obviousness equal to zero (i.e., an obvious design) can also be synthesized from the McD and Allen devices. The functions and required behavior of the "obvious" synthesized design called the McD-Allen device is shown in Fig. 13. The

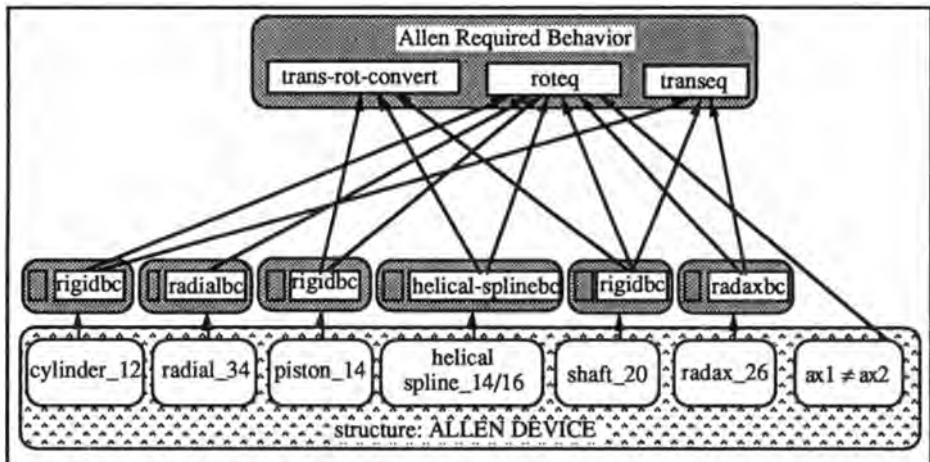


Figure 10: Functions and required behavior of the Allen device.

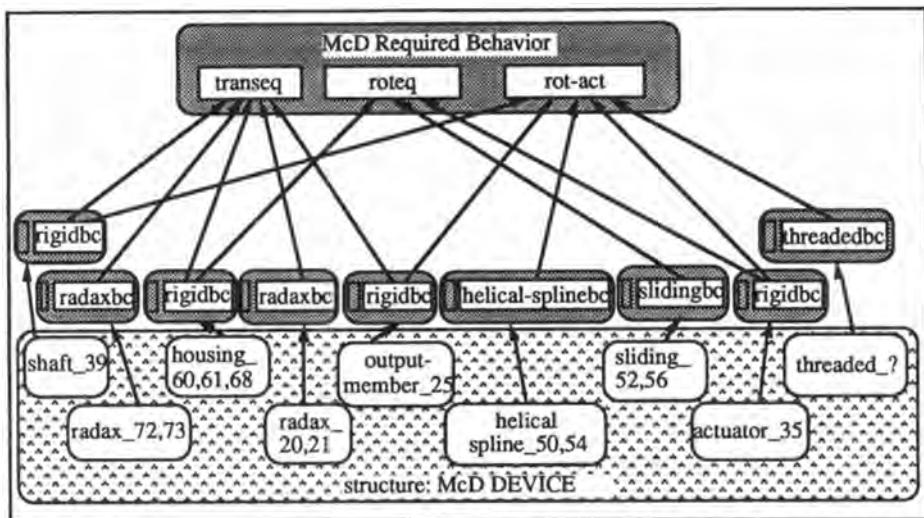


Figure 11: Functions and required behavior of the McD device.

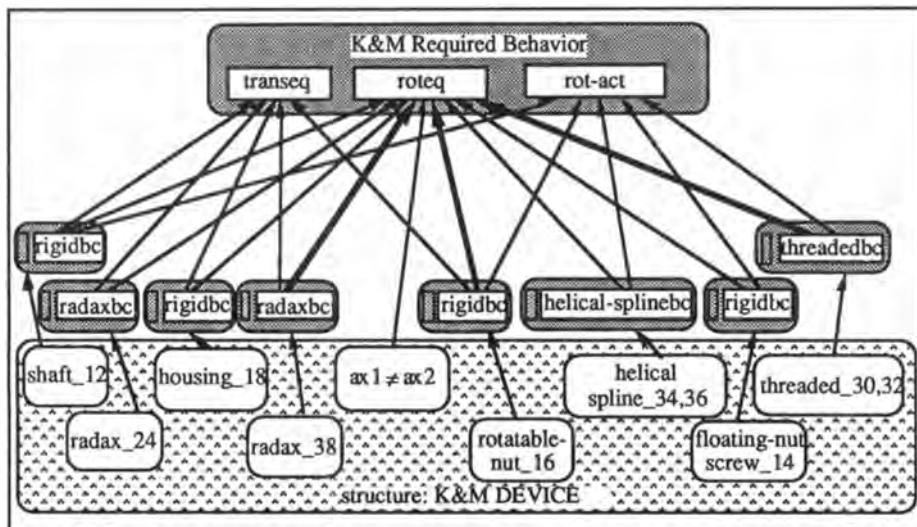


Figure 12: Functions and required behavior of the K&M device.

process of synthesis of the McD-Allen device (recall the synthesis of the uni-fly cycle in an earlier section) replaces the manner of achievement of the "roteq" behavior requirement of the McD device by the manner of achievement of the "roteq" behavior requirement of the Allen device. The corresponding structure produced essentially replaces the sliding_52,56 relationship in the McD device by the radial_34 relationship of the Allen device and also introduces the inequality and parallelism axis constraints between ax1 and ax2. The structure, behavior, and schematic of the McD-Allen device is shown in Fig. 14. The

Table 1: Equivalence of Systems and Relationships in the Structure of the McD and K&M Devices.

Identity Equivalence	
McD Sys/Rel	K&M Sys/Rel
shaft_39	shaft_12
radax_72,73	radax_24
housing_60,61,68	housing_18
radax_20,21	radax_38
output-member_25	rotatable-nut_16
helical-spline_50,54	helical-spline_34,36
actuator_35	floating-nut-screw_14
threaded_?	threaded_30,32
sliding_52,56	-none-

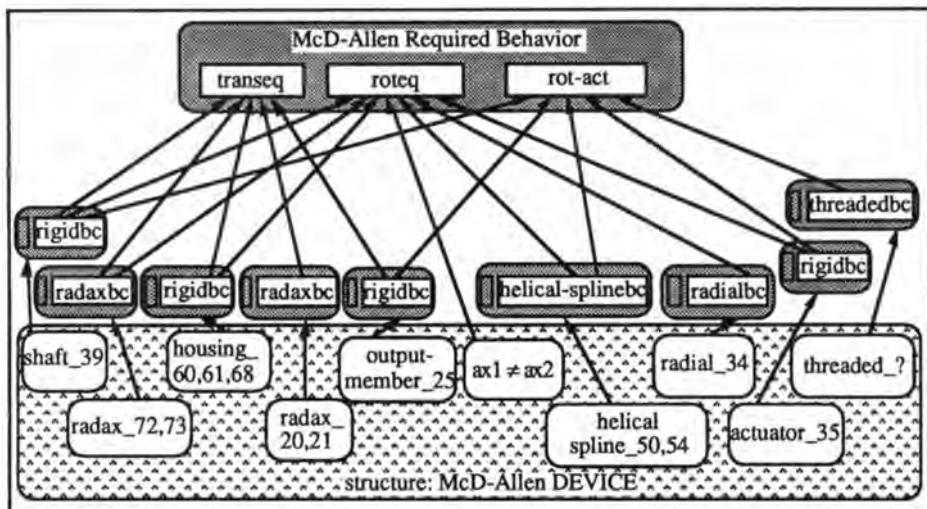


Figure 13: Functions and required behavior for the McD-Allen device.

degree of obviousness of the McD-Allen device is zero since all mappings between primary and secondary functions exhibit Function Re-Use from those of the McD and Allen devices.

CONCLUSION

Utility, novelty and obviousness are key requirements that are used in patent examination. This paper presents an approach to the evaluation of utility, novelty and non-obviousness based on a representation scheme for a design library that integrates the structure, behavior

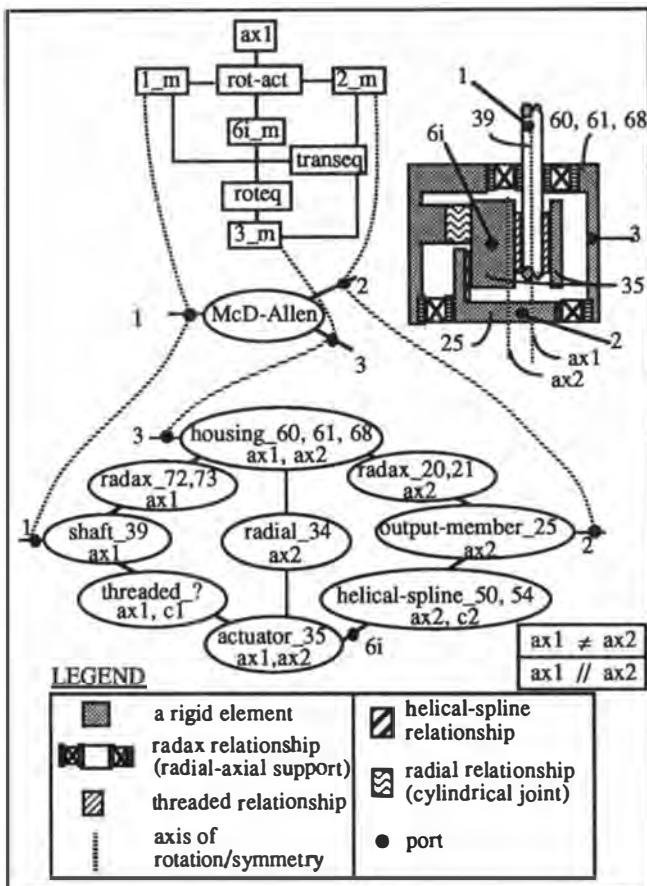


Figure 14: Structure, behavior and schematic of the McD-Allen device.

and function (purpose) aspects of design descriptions. The prior art is viewed as a "closed world" design library. The utility of a subject device is established by the presence of a purpose for the device (i.e., a secondary function) in the design library, or a required behavior for the subject device identical to another prior art device in the design library. The novelty of a device is determined by the absence of any other device in the library that has an equivalent substructure with identical function mappings and required behavior. The non-obviousness of a device is determined by evaluating whether at least one of the function mappings in the device is new with respect to prior art designs in the design library. The degree of non-obviousness is quantified by the number of such new mappings in the device. The notions of utility, novelty and non-obviousness are illustrated using examples of mechanical devices, including those from an actual patent evaluation process. The synthesis of "obvious" devices (i.e., with a zero degree of non-obviousness) is also demonstrated using the examples.

Acknowledgements. This work is supported in part by the National Science Foundation, grant no. DMC8810503, the Texas Advanced Research Program (1989), and by a grant from the Challenge for Excellence endowment of the University of Texas at Austin. The support is gratefully acknowledged. We thank M. K. Narasimhachar for suggesting the tricycle and unicycle device examples used in this paper, and the reviewers for their useful comments and suggestions.

REFERENCES

- Allen, A. K. (1965). Free Piston Oscillator, *United States Patent 3,183,792*, May 18.
- Harmon, R. L. (1988). *Patents and the Federal Circuit*, Bureau of National Affairs Inc.
- Kannapan, S. and Marshek, K. M. (1988). Eccentric Differential Screw Actuating, Torque Multiplying and Speed Changing Device, *United States Patent 4,723,453*, Feb. 9.
- Kannapan, S. and Marshek, K. M. (1989). Design Synthetic Reasoning, Mechanical Systems and Design Technical Report 216, University of Texas at Austin, Texas, June.
- Kannapan, S. and Marshek, K. M. (1990). An Algebraic and Predicate Logic Approach to Representation and Reasoning in Machine Design, *Mechanism and Machine Theory*, 25(3): 335-353.
- Kannapan, S. and Marshek, K. M. (1991a). Design Synthetic Reasoning: A Methodology for Mechanical Design, *Research in Engineering Design*, to be published.
- Kannapan, S. and Marshek, K. M. (1991b). An Approach to Developing a Computational Aid for Patent Evaluation, *Mechanical Systems and Design Technical Report 218*, University of Texas at Austin, Texas, in preparation.
- MacDonald, J. G. F. (1973). Power Operable Pivot Joint, *United States Patent 3,731,546*, May 8.
- Pressman, D. (1986). *Patent it Yourself*, ed. S. Elias, Nolo Press, Berkeley, California.
- Siff, E. J. and Emmerich C. L. (1960). *An Engineering Approach to Gyroscopic Instruments*, Robert Speller and Sons Publishers Inc., New York.
- Vaughan, F. L. (1972). *The United States Patent System*, Greenwood Press, Connecticut.

A segment-based approach to systematic design synthesis

I. Horváth

Institute for Machine Design
Technical University of Budapest
Müegyetem rkp 3, Budapest XI H-1521 Hungary

Abstract. Design is a typical field for application of artificial intelligence and creativity. The paper outlines a possible approach to expert system oriented mechanical design. First, evolution of design methodologies is surveyed. Then general aspects of systematization, theoretical fundamentals and a possible way of principled formalizing based on postulates are discussed. As a real entry point of a DES-oriented design process the assignment of the set of required functions followed by the specification of the possible topologies of functions is defined. A methodology is devised through the introduction of formalized design operations, classification of design problems and working out methods for systematic combination of predefined segments and segment-elements.

INTRODUCTION

As an outgrowth of the application of digital computers and softwares based on original paradigms, continuous evolution can be observed in mechanical engineering design, Figure 1. While during the sixties computer application in design was centered around 2D computer-assisted drawing and numerical calculations, the seventies brought the wireframe approach and the proliferation of interactive computer-aided design (CAD) systems. As it is well-known, by the end of this decade the practice of computer internal solid modelling of objects have been worked out. Solid modelling opened up new ways of product analysis, manufacturing support and quality control. Several integrated (interactive) systems became available, furthermore, methods and tools of both internal and external system integration were raised to a considerably high level.

Practically drawing, geometric modelling, analysis, simulation, database management, documentation, data communication and technological preprocessing belong to the scope of integrated computer-aided design in the field of mechanical engineering. It means, that integrated CAD does not comprise the conceptual design at all. In fact, conventional integrated CAD is able to give indirect support to conceptual design by relieving designers from the less creative activities. These systems are data-based, therefore, they are unable (because not intended) for heuristic problem solving and spontaneous adaptation. These factors, however, are indispensable ingredients of any autonomous operation and intelligent behaviour, respectively.

Having recognized the limitations of data-based CAD systems, researchers started to investigate the possibility of raising the intelligence of design systems. This endeavour is propagating to-day, since experts hope a significant leap in the productivity and effectiveness of computer-based design. The conception of intelligent, integrated and interactive design arose at the beginning of the

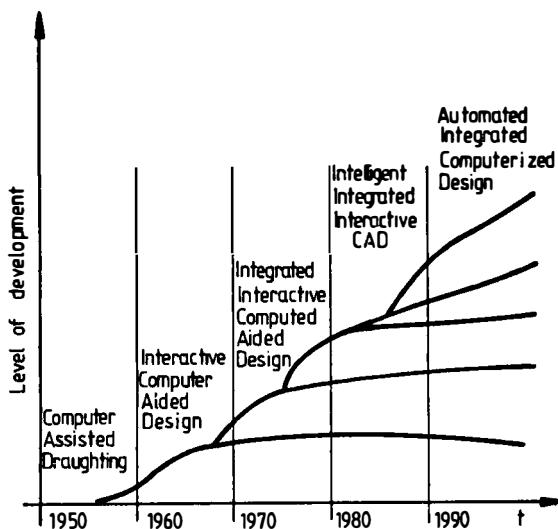


Figure 1. Evolution of design methodologies

eighties - mainly as an idea of Japanese and American researchers. Implementation of design problem solving intelligence finally ends at automating all of the machine solvable parts of mechanical engineering design, starting out of the functional specification and finishing with the evaluation of design prototypes. However, applying complex knowledge processing, in itself, is not enough for intelligent automation. Seeing that the computerized equivalent of the human problem solving capability and creativity, and not only their emulations, are to be implemented, effective computer-oriented design methodology is needed. In the foreseeable future it will definitely be required to establish the highest level of computer support to design that can be called automated integrated computerized design (AICD).

Research activities related to design automation are distributed on several areas. The most important ones are given below:

- (a) Design science and specific design knowledge on technical systems, Hubka and Eder (1988), Roth (1982), ten Hagen and Tomiyama (1987), Rosenthal (1990).
- (b) Operational and environmental processes of intelligent design systems, Ohsuga (1989), Yoshikawa (1981), Yoshikawa (1983).
- (c) Functional components and architecture of intelligent CAD systems, Akman et al. (1990), Dym (1985), Allen (1986), Papalambros and Chirehdast (1990).
- (d) Strategy and control of design problem solving, Coyne et al. (1987), Horváth (1990), Márkus (1989), Tomiyama et al. (1989).
- (e) Representation of designer's knowledge, know how, skill and experience, Krause (1989).

- (f) Principles of non-geometry oriented design, Gero (1989), Grundspenks (1983), Yoshikawa (1972).
- (g) Methods of semantic product modelling, Ohsuga(1983), Tomiyama (1988),
- (h) Embedding design expert systems into human and production environment, Jain and Maher (1990), Koegel (1988), Chen (1988).
- (i) Self-organising and learning systems in design, Hoeltzel and Chieng (1989), Yang (1989).

As references given above indicate a great deal of 'milestone' results are available. Some of them were directly used as starting blocks at the set-out of our project. At that time it seemed to be rather sensible to integrate the different aspects to see how they work together. The research work behind this paper was primarily aimed at reaching this goal. Programming techniques, e.g. procedure-oriented, data-oriented, rule-oriented and object-oriented, were also considered. Development of the design expert system PANGEA was initiated to investigate the possibilities of building up mechanical object from segments and/or segment-elements by systematic combination.

GENERAL ASPECTS OF AUTOMATING DESIGN

It is advisable to accept as a rule, that there is no chance to develop any automated design system with ignoring the fundamental ascertainties of system theory. The general system theory describes systems abstractly. A system, or a component of it, can practically be anything if exists at all. To apply the ascertainties of system theory for mechan'cal engineering design at least three restrictions are to be considered.

First of all, a design system (ie. the system being designed) is always considered as one built up from components. Components are defined by dissectioning the system. It can be both logical and physical. Physical dissectioning results in a set of parts being manufactured as individual entities. Logical dissectioning is needed to help evaluating systems. Secondly, the number of the components of a design system must be limited. There is no sense to take into consideration an infinite number of parts for practical reasons. Thirdly, from the point of view of possibilities of merging the components into a design system, restrictions are to be made on the nature and attributes of components.

Only these three restrictions discussed above make it possible for us to take the design system as a finite set. We must add, however, that any real system is (and behaves as) a total oneness of its mutually related components. From the point of view of forming a system, totality means synergic connections between the constituent parts of the system. It means, that the attributes of any component being combined to form a design system should reflect the relations of that particular component to others.

Note that, as it has been agreed earlier, components can be anything. By our interpretation, components that are subsystems, group of parts or individual parts are considered as segments. In general they are known standard parts that exist physically. For the purpose of automated design, however, more elementary components are to be identified. These may be functional surface groups or pairs of any known or designed segment (or segments). These geometrically incomplete objects are called segment-elements.

It has been concluded by Sadowskij (1976), that there are some initial contradictions in the system oriented way of thinking. These may appear either

in the definition of the systems or in their analysis. Each of these contradictions is a paradox of system theory. From the six paradox identified by Sadowskij, three directly apply to system oriented automating of design. These are the paradox of hierarchy, totality and methodology. Their importance, however, not necessarily follows the order given.

Paradox of hierarchy says that components of a design system can be defined only if the structure of the system is known. The structure, however, depends on the selected set of components. This paradox can be solved by supposing that the system components are definitely known. Design systems are allowed to be built up as a certain combination of a set of known components.

Paradox of totality says that components do not have the total attributes of the design system, but they are segregated with their own (total) attributes. It is not possible to specify those attributes of a component that are inherited from all of the embedding design systems. Consequently, if the requested total attributes of the design systems are known, components are to be selected to cover the attribute set at the possible largest extent. Otherwise, if the total attributes of the requested components are known, the sum of the individual component attributes should be accepted as a resulting set of attributes for the design system.

Paradox of methodology says, that a design system can be described with a methodology developed distinctly for that particular type of system. Without knowing the system in advance, however, it is practically impossible to develop the requested defining methodology. It shows that the process of system definition and the substance of a design system are interdependent. So we may conclude that there is no sense in defining a general design methodology that could be applied to any design system invariantly. Furthermore, in developing the problem solving strategy for different objects (ie. design systems) characteristic features have to be taken into account.

AN APPROACH TO AND THE POSTULATES OF AICD

Evidently, with a view to automating the design process a formal description of conceptual design is highly requested. Yoshikawa (1981 and 1983) made a very remarkable contribution to it by setting the design onto a set theoretical basis. Unfortunately, the practical implementation of this very promising approach is more or less limited by the lack of the ideal information processing complexes and the time requirement that can be envisaged in developing the needed methodology and practice.

For the near future a very practical solution can be the one that is based on knowledge processing design expert systems (DES) with specialized design methodology. Their capabilities can be exploited even in the support of conceptual design. Methodology for DES can be developed partly based on the experiences gained with human design activity, and partly on the development of computer oriented design problem solving strategy and methods.

The need for formalised description of design entails the need for axiomatizing. The ultimate aim of axiomatizing is to provide a knowledge structure that covers all aspects of design (e.g. activities, tools, features, etc, which can be simply named categories). An important perception of Yoshikawa's extended general design theory is that design can be formulated based on set theoretical approach. However, system theoretical paradox have to be kept in mind too.

Design axioms are difficult to formulate for at least two reasons. First, they are not evident, and secondly, they are or must be abstract, because of their expected general applicability. Because of the general context, using a design axiom calls for a multi-level mapping process. Knowledge structure developed by axiomatizing must

- (a) supply guidance to the execution of the design process,
- (b) show how to arrive at a good design,
- (c) inspire design creativity and innovation.

Our approach, presented below, starts out of the contextual model of design science, Figure 2. It takes into consideration the fact, that the knowledge space of design comprises of the synergic unity or concord of theory, methodology and practice. The point of this approach can be summarized as follows. Based on the accepted concepts of design it is possible to postulate those axiomatic assertions, the postulates, that describe the fundamentals both of conceptual and detail design with the needed abstraction. Besides these postulates, conjectures are to be presumed for those supplementary facts, which can not be declared with absolute certainty. With the contents and interrelations of the postulates theorems can be proved. Based on these, the problem solving strategy and methods of conceptual design can be worked out. Capturing a

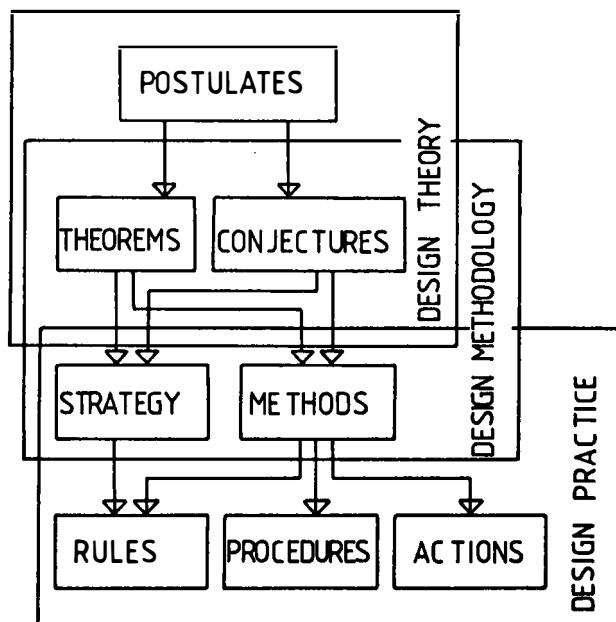


Figure 2. Model of design science

sufficient number of propositions an enabled set of rules, procedures and actions can be generated. These altogether will result in the practical knowledge base of a DES.

Without striving for completeness, we specify the conceived postulates of design. Note that there is no way of proving the postulates in a mathematically exact manner since they are hypothetical and/or empirical.

Postulate of design knowledge space

Subsets $P_i(x)$ of universal set of design categories $U(k)$ form a power set $T(x)$, that is the knowledge space of conceptual and detail design.

Postulate of finiteness

Subsets $P_i(x)$ of knowledge space $T(x)$ are finite sets.

Postulate of correspondence

Binding the elements and subsets of search spaces $P_i(x)$ (the mapping) is an abstract form of conceptual and detail design.

Postulate of comprising

There exists a rule of selection by which the possible set of segments (physical entities) S' of a design system G can be assigned on $T(x)$.

Postulate of definition

A design system G is a related set of the subset S of segment space S' and the relevant topology R .

Postulate of decomposition

There exists a morphological decomposition m of segments s that results in a set of segment-elements e providing elementary operations depending on the accompanying physical effects (PE), material properties (MP) and degrees of freedom (DoF).

Postulate of constancy

In case of a concrete design system G the number of the elements of set $S(s)$ must be constant.

Postulate of functioning

Functioning of a design system G is based on harmonized physical phenomena PE.

Postulate of totality

A design system **G** interacts with its environment as a totality.

Postulate of procession

Operation of a design system **G** is a result of a related change in the state of its material, energy and information flow.

Postulate of morphology

Morphological features **MF** of a design system **G** reflect the topology $r(F'(S'))$ specified on the functional space $F'(S')$.

Postulate of invariance

It is possible to specify functionally invariant topologies $r(D)$ on the metric subset $D(a,b)$ of design knowledge space $T(x)$.

Postulate of selection

Operational functions **F** of a segment **s** or a segment-element **e**, belonging together physically, are aspects of selection.

Postulate of regularity

Regularity of interlinks of segments and segment elements are determined by their functional and morphologic features **MF** together.

Postulate of prototype

Conceptual model of a design system **G** is the set of its generic prototypes **GP**.

Because of the limited extent of the paper discussion of theorems and conjunctions should be neglected.

PROBLEM SOLVING STRATEGY OF AICD METHODOLOGY

Design synthesis is aimed at creating design systems that have not existed earlier (or at least not in the required form). This activity needs creativity which is still more or less mystified human capability. We actually don't know the motive power behind engineering creativity. These aspects have been discussed in detail elsewhere. Creativity is partly an inherited, partly an acquired capability. Undoubtedly, getting acquainted with human creativity helps us to reproduce an artificial version of it.

Significant forms of creativity are brainstorming, relaxation, mutation, reduction, analogy, abstraction, inversion. Besides these, there are some that are definitely computer oriented, eg, trial and error, generate and test, random combination and systematising. In our research controlled systematic combination was applied as the ground of artificial creativity.

Scanning and manipulation of the solution element space that is broad enough to be fertile, generally leads to combinatorial explosion. The chance of it can hopefully be avoided, or decreased at least, if a really DES oriented problem solving methodology can be developed. At the development of the design methodology for DES it was presumed, that problem solving in design does not need the reproduction of the entire human knowledge, general problem solving capability or artistic ability. Instead of these, a potential for guided problem solving on a given field of application and a DES oriented methodology were considered as primarily important issues. The DES oriented methodology rests on four pillars, ie.:

- (a) Design process is accepted as a series of mappings amongst related search spaces (design category sets),
- (b) Activities of (conceptual and detail) design are formalized as operations,
- (c) Design problems are categorized based on their features and the set of operations needed to solve them,
- (d) Specialized problem solving methods are integrated in a knowledge-based environment.

The key to success in the application of DES for mechanical engineering tasks is the proper formalization and description of the problem. Solving mechanical engineering design problems with DES is not easy because the system must achieve a result in a rather mechanical way while designers tend to create potential solutions based on intuition, heuristics and analogies. The principle behind the problem solving strategy for DES is to be the well-known 'divide and conquer'. It means that design problems are not treated in general, but they are categorized based on their features and the set of operations needed to solve them. The systematical generation of concepts that may be expected of DES can be made more effective by appropriate control strategies. Thus our task is to better understand and formalize that, which has been thought of as the creative activities in mechanical engineering design.

We have found that the apparently different activities of design may be led back to a limited number of common roots. Consequently, formalized DES oriented operations based on their effects, can be grouped as creative operations (Σ) and investigating operations (Γ). Here, activities needed to generate and manipulate models are called operations. There are two typical occurrences of creating operations, namely combining synthesis (Σ_c), that is for topological configuration of the structural elements of an assembly, a unit and a part respectively, and matching synthesis (Σ_m), that is for specifying functional and constructional quantities (parameters) among the structural elements. Investigative operations can also be divided into two subclasses. An investigation may be orientated towards finding the most suitable entities based on the qualitative comparison of attributes. This is a selective analysis (Γ_q). Another type of investigation may be for finding the most appropriate parameter values based on a quantitative evaluation. Therefore, it can be called evaluative analysis (Γ_s).

Our experiments have demonstrated that with the suitable combination of these typified DES operations it is possible to formalize engineering problems at a level which can be a great support to their solution with complex knowledge

processing. After adapting the DES operations to the problem classes concerned they serve as the activity elements of the applied artificial design creativity. In addition to that, these operations can be used to typify the design problem classes themselves. This way, it is not an exaggeration to say that a product model, i.e. an instance of the design prototype, can be a by-product of the design process generation by formalized operations. Operations, however, must be adapted to a given product type or category.

From the point of view of solving design problems by DES, the target problems can be categorized from the aspect of the needed typified design operations. In this respect four problem classes can be identified.

(1) Qualitative instance selection

It means selecting a member from a set of known parts, units and assemblies that meets the given functional conditions and demands.

(2) Design prototype evaluation

Finding the most appropriate set of design values for a parametrized object prototype, which can be an assembly, a unit or an individual part, to meet functional requirements.

(3) Configurative synthesis

Selecting appropriate items from a given set of kindred or diverse (i.e. paradigm type) segments, coupling them together to form a design prototype and evaluating them for functional requirements.

(4) Generative synthesis

Selecting segment-elements to built up segments (parts and units) and combining them with known segments to develop a design prototype for functional, physical and geometrical evaluation.

Complexity of the specified problem classes grows as listed. In classes (1) and (2), the design prototype exists either in a form of an instance or a parametrized description, respectively. In classes (3) and (4), however, existence of design prototype is not assumed. Order application of formalized operations is definitive as a rule, that is neither the law of associativity, nor commutativity are applicable. Actually, problem classes introduced above mean different entries to the general mapping process of design presented on Figure 3.

METHODS AND PROCEDURES OF SEGMENT-BASED DESIGN

To implement an automatized DES oriented design process first we have to separate those tasks that are to be solved outside of the DES environment in advance and that are expected to be solved by the DES automatically. Therefore it has been accepted that a DES oriented design methodology should not cover the tasks of initial problem specification, problem clarification, requirement analysis, investigation of the different environment, studying the feasibility and functional decomposition. These remain human tasks. A real entry point can be the specification of functions and editing the topology of functions for the design system. And the final selection among the solution variants is a task for the DES user too.

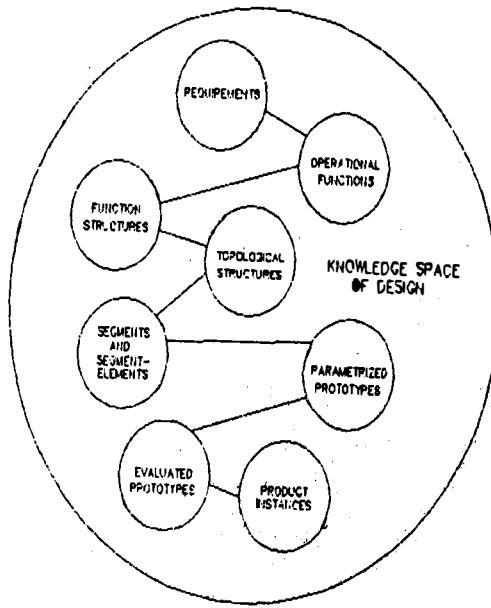


Figure 3. Design process as mappings

In this interpretation a design expert system must support and/or solve the following general tasks:

- (a) specification of requested functions,
- (b) constructing functional structure variants,
- (c) generating structure topology
- (d) definition of segments and segment-elements,
- (e) geometrical representation of segments and -elements,
- (f) searching for segments and -elements based on pattern matching,
- (g) systematic combination of segments and segment-elements,
- (h) interfacing segments and -elements based on parameters,
- (i) presentation of design prototypes,
- (j) numerical evaluations of design prototypes,
- (k) ranking evaluated design prototypes.

For these tasks a pilot design expert system **PANGEA** is being developed (Horváth and Takáts, 1988). At the beginning of a design session with **PANGEA** the user must specify the requested functions. We adopted the functional concept of Roth (1982). With the generalized symbols of the selected functions the user can construct function structure variants, Figure 4.

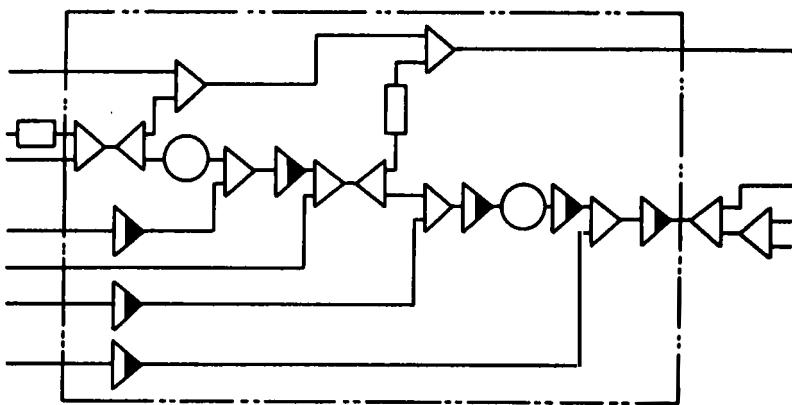


Figure 4. Function structure diagram

The next step is to assign the limits or functional scope of the design system. Then the assigned system domain is to be decomposed into potential subsystems. It can be done by the user intuitively or by the DES based on pattern matching

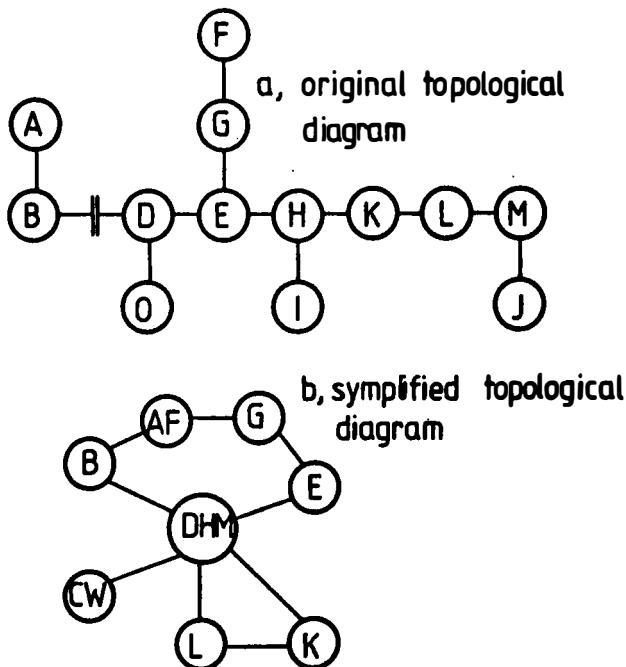


Figure 5. Manipulation on structure topology diagram

(ie. trying to cover the system domain by the function patterns of known segments. It results in a structure topology that can be represented as a bi-graph. Nodes of the graph are the needed function-carriers and the edges are the needed

```
*****
*           * SEGMENT CODE:          *
*           * . NAME:             *
* IDENTIFIER * . TYPE:            *
*           * LIBRARY ID:          *
*           * JOINT CODE:          *
*           * GEOMODELCODE:        *
*****
*           * CHILDRENLIST:        *
*           * PARENTSLIST :        *
* INTERFACE  * PARTNERSLIST:      *
*           * JOINT PARAMS:       *
*****
*           * EXPRESSION#1:        *
*           * :                  *
* METHODS   * EXPRESSION#I:       *
*           * :                  *
*           * EXPRESSION#N:       *
*****
*           * INPUTDATA #1:        *
*           * INPUTDATA #I:        *
* WORKING   * INPUTDATA #V:       *
* PARAMETERS * :                  *
*           * :                  *
*           * OUTPUTDATA#1:        *
*           * OUTPUTDATA#J:        *
*           * OUTPUTDATA#W:        *
*           * :                  *
*****
*           * MAINFNCTN #1:        *
*           * :                  *
* FUNCTIONS * MAINFNCTN #K:       *
*           * AUXIFNCTN #1:        *
*           * :                  *
*           * AUXIFNCTN #M:        *
*****
*           * PLACEMENT X :        *
*           * PLACEMENT Y :        *
* LOCATION  * PLACEMENT Z :      *
*           * POSITION XY :       *
*           * POSITION YZ :       *
*           * POSITION ZX :       *
*****
*           * ATTRIBUTE #1:        *
* ATTRIBUTES * :                  *
*           * ATTRIBUTE #R:        *
*****
*           * OPERATION #1:        *
* MANUFACTURE * :                  *
*           * OPERATION #Q:        *
*****
```

Figure 6. Frame prototype describing segments and segment-elements

functional relations that will be characterized by operational quantities after selecting the possible function-carries. Having recognized the possibilities the user may simplify the topology graph, Figure 5. If all else fails, individual functions must be covered by segment-elements (Horváth, 1990).

In principle functions (group of functions) and function carriers may be bound directly. However, because selection and combination of functions are influenced partly by the context of the functions and partly by their possible connections, constructing the topological structure is important. In case of a generative synthesis (ie. when developing a brand new design) we are forced to start with the individual functions. The user may select the function entities from the library of all functions that is compiled automatically based on the sets of functions of the segments and segment-elements known by the DES system.

Segments and segment-elements, being stereotyped, are described by attribute frames. Every frame is an object and represents a design prototype. Slots of the frame contain parameters, values, function definitions and expressions, Figure 6. The frame even contains an identifier of the B-rep model of the segment or segment-element, Figure 7. Frames are organized into libraries.

At the execution of combining the DES system, based on the set of assigned functions, picks out all of the possible segments or segment-elements. Meanwhile pre-matching is done by the help of interface parameters. This way geometric models of segments and segment-elements can be merged. Then parameters of the synthesized object variants are evaluated to find the best fitting operational and geometric values. At the end of this matching process an evaluated geometric model can be displayed. Note, to avoid combinatorial explosion filtering is done on three levels, ie. on interface, operational and geometrical parameter levels. Because of the computer limitations, although it would be advantageous, the methodology may not be prepared for parallel generation and evaluation of solutions variants.

SOME ASPECTS OF GEOMETRY REPRESENTATION

The primary goal of automatized design based on segment combination is the production of parametrized geometric model of the object. By evaluating this geometric prototype instances of the object can be generated. This conception poses special requirements against the used modelling methods, namely, they must be able to

- (a) cover all of the objects belonging to the problem classes,
- (b) ensure the validity and unambiguity of the geometric models,
- (c) handle geometrically non-complete geometries,
- (d) ensure joining and matching of arbitrary geometric substructures,
- (e) describe objects parametrically,
- (f) evaluate objects for their engineering quantities,
- (g) specify micro-geometry.

As a base of geometric modelling an extended B-rep scheme was selected. Segments are, in general, real objects that are or can be assembled into structures. Segment-elements, however, represent certain parts of a segment that have functional or morphological significance. Each segment-element consists of

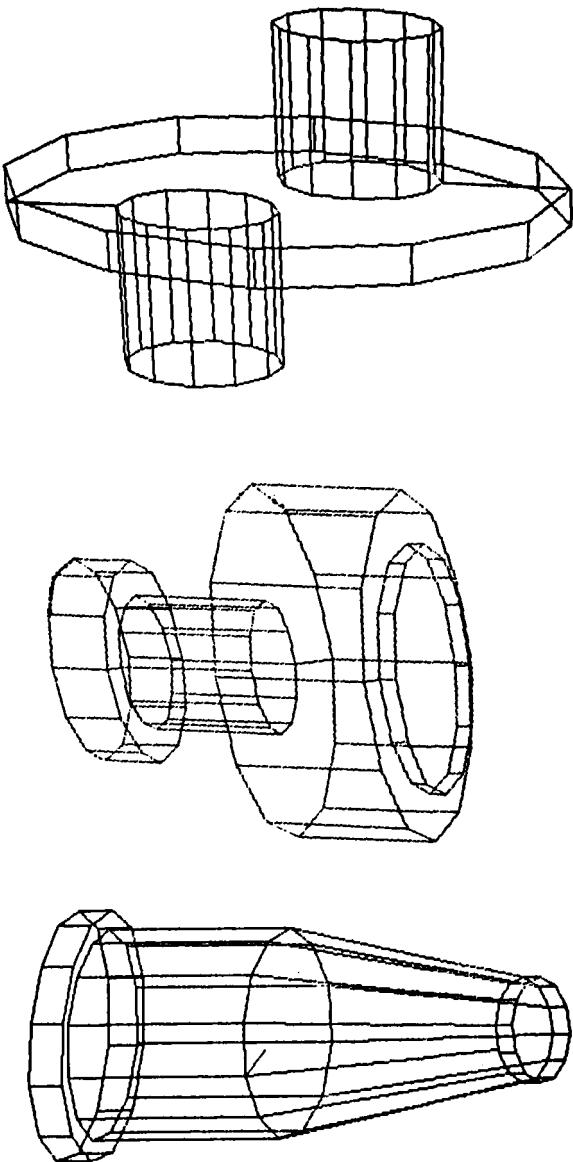


Figure 7. Segment-elements for shaft design

a limited number of surfaces and physical effects acting on these group or pairs of surfaces lend themselves for functioning. Because segment-elements can be derived by a physical dissectioning of existing segments or defining them as surface entities to certain physical effects, resulting in useable functions, they are geometrically non-complete objects. With the terminology of boundary representation we may say that segment elements are open-shell-objects in contrast to segments that are closed-shell-objects in general.

To describe objects with closed shells face-oriented approach of boundary representation is advantageous. The used modelling approach must provide tools both for initial description of segments/elements and for their merging. Boundary construction is traced back to stepwise construction. This method makes it possible to treat topological elements individually with correspondence to the relevant geometric primitives. Well-known Euler-operators are accepted tools for describing object shells, however, they are not able to describe an object with opened shell, Furukawa et al. (1985). If the concept of topological gap is introduced it is easy to develop operators that are able to open shells or to make them closed. Generating three new operators, that are $KfsMg \setminus MfsKg$, $KfMhg \setminus MfKhg$ and $MeKg \setminus KeMg$, makes it possible to follow any modification of the topological genus of the object designed even through the open-shell-state. It also facilitates for the DES to find out how to combine segment-elements to arrive at a valid topology, Figure 8. Theoretical and implementational issues are discussed in detail, among others, in Horváth (1988). Thus boundary modelling needs less geometric construction and makes it possible to further automate the prototype generating process.

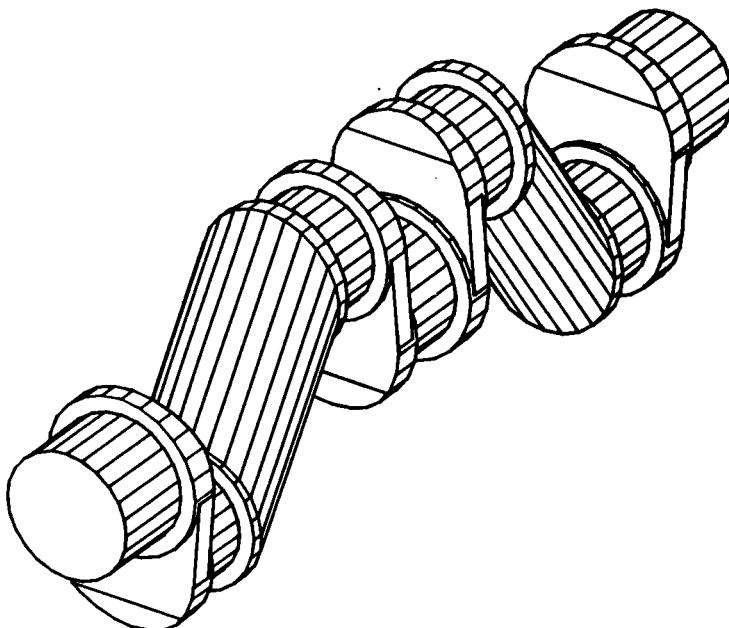


Figure 8. Crank-shaft made of segment-elements

CONCLUSIONS

It has been confirmed, that knowledge-based expert system paradigm can be used for developing intelligent design systems, however, a comprehensive DES oriented methodology is needed. This methodology may be derived based on a fundamental design theory and must give answers to the questions of problem solving strategy, solution generating process and design prototype representation.

At the Institute for Machine Design of BTU a pilot design expert system, PANGEA, is being developed to support the research activity in the field of DES methodology. Attempts were made to use this system for solving the four basic problem classes of design, ie. qualitative selection, prototype matching, configurative synthesis and generative synthesis. Because of the awareness that DES methodology need not necessarily be an emulation of that which is used by human designers, as a fundamental methodology for synthesis, systematic combination is applied. It is a possible alternative of the metamodel concept proposed elsewhere. Segments and segment-elements are defined as objects and they are selected and combined based on their operational functions and interfaces. Then parameter matching on three levels is applied. Geometric models can be generated automatically by a smart boundary representation algorithm.

After this first stage of research and development further questions have arised in connection with the complexity of the problems to be solved, possibilities of improvement of the control, the evaluation process, and the effectiveness.

REFERENCES

- Allen, R. H. (1986). *Design Guidelines for Expert Systems, in Applications of Artificial Intelligence in Engineering problems*, Springer Verlag, Berlin, pp. 651-658.
- Akman, V., ten Hagen, P.J.W., Tomiyama, T. (1990). *A fundamental and theoretical framework for an intelligent CAD system*, Computer-Aided Design, 22 (6), pp. 352-367.
- Chen, C. S. (1988). *Developing a Feature Based Knowledge System for CAD/CAM Integration*, Computers in Industrial Engineering, 15 (1-4), pp. 34-40.
- Coyne, R., Rosenman, M., Radford, A., Gero, J. (1987). *Innovation and Creativity in Knowledge-Based CAD*, in *Expert Systems in Computer-Aided Design*, ed. by Gero, J., Elsevier Science Publishers, Amsterdam, pp. 435-465.
- Dym, C. L. (1985). *Expert Systems: New Approaches to Computer-aided Engineering*, Engineering with Computers, 1 (1), pp. 9-25.

- Furukawa, M., Kakazu, Y., Okino, N. (1985). *Shell Geometric Modelling with the Euler Topology Characteristics*, in Design and Synthesis, ed. by: Yoshikawa, Elsevier Science Publishers B. V., Amsterdam, pp. 163- 168.
- Gero, J. S. (1989). *Prototypes: a Basis for Knowledge Based Design*, in Knowledge-Based Systems in Architecture, ed. by Gero, J.S., Oksala, T., Acta Polytechnica Scandinavica, Helsinki, pp. 11-17.
- Grundspenkis, J. (1983). *The synthesis and analysis of structures in computer aided design*, in Computer Applications in Product and Engineering, ed. by Warman, E.A., North-Holland Publishing, Amsterdam.
- ten Hagen, P.J.W., Tomiyama, T. (eds) (1987). *Intelligent CAD Systems*, Part I, Springer-Verlag, Berlin.
- Hoeltzel, D.A., Chieng, W-H. (1989). *Factors that Affect Planning in a Knowledge-Based System for Mechanical Engineering Design Optimization with ...*, Engineering with Computers, 4 (5), pp. 47-62.
- Horváth, I. (1987). *Térbeli lemezszerkezet tervező CAD rendszer tervezésmódszertani alapjai és komplex tervezői modellje*, Műszaki doktori értekezés, Budapest, (in Hungarian).
- Horváth, I. (1988). *Towards a More Intelligent Boundary Modelling*, in Intelligent Manufacturing Systems II, ed. by: Milacic, V.R., Elsevier Science Publishers, Amsterdam, pp. 109-120.
- Horváth, I. (1990). *Selection and Combination of Part Segments by Function*, in Proceedings of the 1990 International Conference on Engineering Design, Volume 1, ed. by: Hubka, V., Kostelic, A., Heurista-JUDEKO, Zagreb, pp. 91-98.
- Horváth, I., Takáts, I. (1989). *Szakértőrendszerdzs-fejlesztés géptervezési feladatokra*, Automatizálás, XXII (9), pp. 2-9, (in Hungarian).
- Hubka, V., Eder, W. E. (1988). *Theory of Technical Systems*, Springer Verlag, Berlin.
- Jain, D., Maher, M. L. (1988). *Combining Expert Systems and CAD Techniques*, in Artificial Intelligence Developments and Applications, ed. by Gero, J.S. and Stanton, R., Elsevier Science Publishers, B.V., pp. 65-81.
- Koegel, J.F. (1988). *Planning and Explaining with Interacting Expert Systems*, Second Eurographics Workshop on Intelligent CAD, The Netherlands, pp. 19-32.
- Krause, F.-L. (1989). *Wissensverarbeitung für die rechnerunterstützte Produktgestaltung*; Produktionstechnisches Kolloquium PTK '89, Berlin, pp. 112-122.

Márkus, A. (1989). *Új eszközök és módszerek a számítógéppel segített készüléktervezésben*, Kandidátusi Értekezés, Budapest, (in Hungarian).

Ohsga, S. (1983). *A New Method of Model Description*, in CAD Systems Framework, ed. by Bo, K., Lillehagen, F. M., North Holland, Amsterdam, pp. 285-309.

Ohsga, S. (1989). *Toward Intelligent CAD Systems*, Computer Aided Design, 21 (5), pp. 315-337.

Papalambros, P.Y., Chirehdast, M. (1990). *An Integrated Environment for Structural Configuration Design*, Journal of Engineering Design, 1 (1), pp. 73-96.

Rosenthal, C.W. (1990). *Computer Aids for the Systems Design Process*, Computers in Industry, 14 (1-3), pp. 51-57.

Roth, K. (1982). *Konstruieren mit Konstruktionskatalogen*, Springer- Verlag, Berlin.

Szadovszkij, V.N. (1976). *Az általános rendszerelmélet alapjai*, Statisztikai Kiadó Vállalat, Budapest, (in Hungarian).

Tomiyama, T. (1988). *Object Oriented Programming Paradigm for Intelligent CAD Systems*, Workshop on Intelligent CAD Systems, CWI, Amsterdam, pp. 5-11.

Tomiyama, T., Kiriyma, T., Takeda, H., Xue, D., Yoshikawa, H. (1989). MetaModel: *A Key to Intelligent CAD Systems*, Reserach in Engineering Design, 1 (1), pp. 19-34.

Tomiyama, T., Yoshikawa, H. (1988). *Extended General Design Theory*, Report CS-R8604, Centre for Mathematics and Computer Science, Amsterdam, pp. 1-30.

Yang, C. C. (1989). Deduction Graphs: *An Algorithm and Applications*, IEEE Transactions on Software Engineering, 15 (1), pp. 60-67.

Yoshikawa, H. (1972). *Kikai-no toporoji*, Seimitsu-kikai, Japan, 38 (12), pp. 30-35.

Yoshikawa, H. (1981). *General Design Theory and a CAD System*, in Man-Machine Communication in CAD/CAM, ed. by Sata, T., Warman, E., North-Holland Publishing, pp. 35-58.

Yoshikawa, H. (1983). *CAD Framework Guided by General Design Theory*, in CAD Systems Framework, ed. by Bo, K., Lillehagen, F.M., North-Holland Publishing, Amsterdam.

Building a model for augmented design documentation

A. C. B. Garcia and H. C. Howard

Department of Civil Engineering
Stanford University
Stanford CA 94305-4020 USA

Abstract. Project-specific knowledge is the rationale behind the project data and specifications, including the design decisions that link elements of basic data, design data, project-specifications, domain knowledge, and general knowledge to explain the design. This information should be available in design documentation, but usually it is missing. The paper describes an approach for improving design documentation in which the computer acts as an apprentice to the designer to capture the rationale during the design process. The initial focus of the work is on HVAC (Heating, Ventilation, and Air Conditioning) design. We are using videotape analysis of design sessions along with structured interviews to develop a model of design rationale in this domain.

INTRODUCTION

The life cycle of the civil engineering facilities can be measured in decades, a long period during which the facility may undergo substantial changes. Moreover, most of the facilities are highly complex and require substantial time from initial planning to the final construction. The agencies involved in the realization of these facilities are also numerous, making information exchange and communication during the facility development life cycle very vital. Huge amounts of data and knowledge are produced and consumed during the various project phases. However, the final design documentation for the facility contains primarily drawings and numbers, with very little, if any, knowledge of planning, design and other decisions (and their bases) which went into making the facility the way it is. Generally, the documentation does not record the intermediary decisions nor the reasons behind those decisions. This information is what we call the *design rationale* behind a project or *project-specific knowledge* (Howard 89, 91). The lack of design rationale in project documentation hinders the understandability of projects and the transfer of expertise.

The project documents are developed by the designers, but they are consulted by many different users. During the development of a project, the documentation is used as a communication medium among the different trades involved in the project. Plan checkers use the documentation to verify whether the project is in accordance with the codes and

standards. Contractors use the documentation as a plan for realizing the artifact. Designers use the documentation to support redesign and also as a repository of experience.

In the era of automation, computers offer an attractive environment in which the designer can develop and document a project. A number of research projects have started in this direction. Some propose to build an integrated environment able to offer all the tools needed by the designer, including analysis programs and e-mail communication, in order to record all actions that took place in a specific design case, working as a notebook. They suggest that this historic data would allow design rationale reconstituting (Lakin 89). Other researchers are trying to create environments in which designers could explicitly enter their decisions and the rationale for them, but again in a noninterpreted form, e.g., (Conklin 88, Lee 91, Fischer 89). Still other researchers are interested in explaining all the decisions by going to first-principle knowledge, e.g., (Gruber 91, Baudin 89). On the other hand, some researchers emphasize the difficulty of capturing design rationale in a computer due to the complexity of the way people explain design, e.g., by using gestures (Tang 89).

As our approach for capturing the design rationale, we have developed a conceptual architecture for an intelligent design interface which acts as an *assistant* and *apprentice* to the designer. Whenever the designer proposes a design action that differs from the apprentice's expectations, the interface will ask for the designer for justifications to explain the differences. Subsequent queries for design rationale are answered using a combination of the interface's domain knowledge and the designer-supplied justifications. The conceptual architecture, to be implemented in a prototype called *ADD* (Augmented Design Documentation model), is described in Section 2.

The implementation of the ADD prototype will focus on the preliminary design of HVAC systems (Heating, Ventilation and Air Conditioning) for commercial office buildings. The task consists of choosing elements (such as ductwork, pipes, heaters, and chillers) and configuring them to provide the proper environmental control. By focusing on preliminary design, we are concentrating on the more significant decisions made early in the design process. In the case of HVAC system, the decisions being documented include the selection of a main HVAC system type, the equipment for providing cooling and heating, the duct and pipe selection, and finally the rough location of those components into the floor plan. Section 3 describes the process we are using to learn about project-specific knowledge in the HVAC domain by videotaping design sessions and interviewing designers. Section 4 discusses some of our initial observations from the interaction with designers.

THE ADD MODEL

The ADD (Augmented Design Documentation) model is based on the metaphor of having a computerized apprentice following the designer in the design process. The role of the designer is to create the specification of an artifact using his or her knowledge and to answer questions the apprentice might ask. The role of the apprentice is to follow the designer's decisions and ask whenever they are not clear. The apprentice should try to come with its own hypotheses in order to verify whether it is actually understanding the designer's reasoning. The objective is to minimally disturb the designer, but maximally understand what the designer is doing. By understanding a decision, we mean being able to check

whether the decision is consistent with the knowledge that the apprentice contains about design in a specific domain.

When the designer and the apprentice propose the same design action, the apprentice records the decision with connections to its own rationale. All the decisions and their supporting rationale are stored in a case knowledge base. As soon as an inconsistency appears, the apprentice activates a knowledge acquisition agent to request more knowledge for its case knowledge base until it can reproduce the designer's decision. When another user wants to query the case knowledge base to understand more about a design, a case interpreter creates a browsable form that the user can examine to see the decisions and explore the associated knowledge. This information flow is depicted in Figure 1.

Figures 2 and 3 depict the two views of design rationale that ADD must support. The first view is the chronological progression, in which the design process is strictly a sequential set of decisions. Each decision helps to set up the scenario for the next one. The second view emphasizes the logical relationships between design rationale, the design decisions, and the design data. Every decision reflects or is caused by a change in the device form. A decision is a selection or elimination of alternatives that should be carried in the design process. The decision is made in a specific situation of the design process, what we call design scenario, using essentially experiential knowledge. The design scenario contains the history of the decisions already made, the design parameters already instantiated, and the decisions that have been deferred. Each alternative, direct or indirectly, represents a proposed change over the device form.

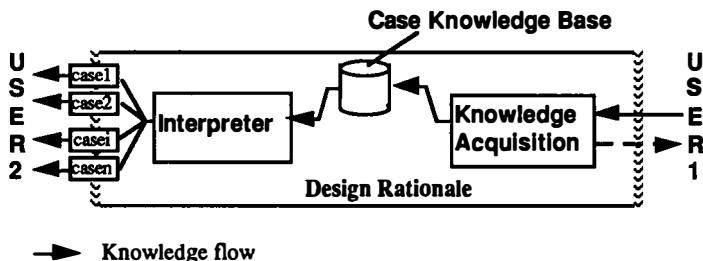


Figure 1: The ADD model

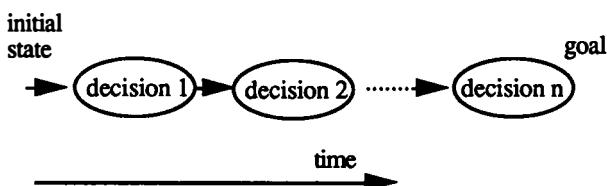


Figure 2: Chronological View of Design Rationale

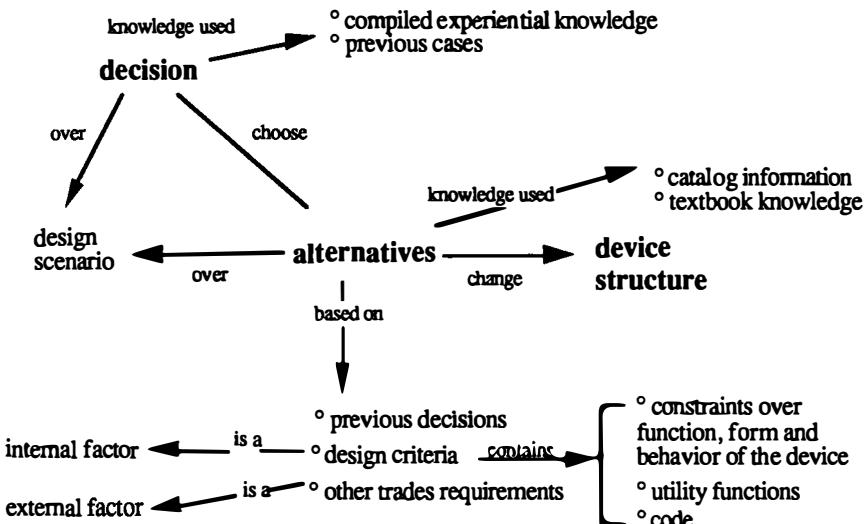


Figure 3: Logical View of Design Rationale

Figure 4 describes the apprentice process as implemented in ADD. The designer posts an action (i.e., a design decision) based on the current state of the design case. Meanwhile, ADD's hypothesis generator proposes its own action based on its design knowledge and on the current design scenario. The reconciler is responsible for comparing those design versions, and if they are the same, the reconciler updates the design by incorporating the hypothesis generator's knowledge as the explanation for the user's action. If the design decisions are different, the reconciler activates the acquisition mode to elicit knowledge that justifies the differences. The new knowledge is inserted, and a new cycle of hypothesis generation starts. This cycle continues until ADD is able to reproduce the user's decision.

We are assuming that if the system is able to reproduce the designer's actions, it is because the system has enough knowledge to explain those actions. We do not believe that the entire set of decisions that leads to a design can be matched by coincidence. We are also assuming that the user is able to explain the difference for its actions in an operational way. ADD offers a menu driven interface in which the user can use pre-existing concepts or create new concepts for explaining an action. The knowledge acquisition is done by presenting the user with the system's hypothesis and supporting rationale. The knowledge acquisition model includes two-way knowledge flow. The system is teaching the user about its knowledge and how it processes that knowledge, while the user is explaining his or her reasoning processes. Therefore, ADD does not have to start with a complete and powerful design knowledge base, but rather one that provides basis for dialogue with the designer.

In order to develop a decision hypothesis, ADD relies on three sources of information: an HVAC system design knowledge base (domain KB), a code knowledge base (code KB), and a catalog data base. All three sources of information are biased towards the HVAC designer view; i.e., only information that is relevant for the HVAC system design will be represented. In the domain KB, we find the previously created building description and the evolving artifact specification. Both pieces of information define the scenario for the design

task. This specification reflects the way designers look at the building by describing the components, the relations among the components, and the attributes characterizing each object. For example, a building is described composed of floors, which are composed of zones, which in turn are composed of rooms. Each of these objects needs an attribute to represent ceiling height. The domain KB will probably represent this information in a network of frames to attain the desired semantic expressiveness.

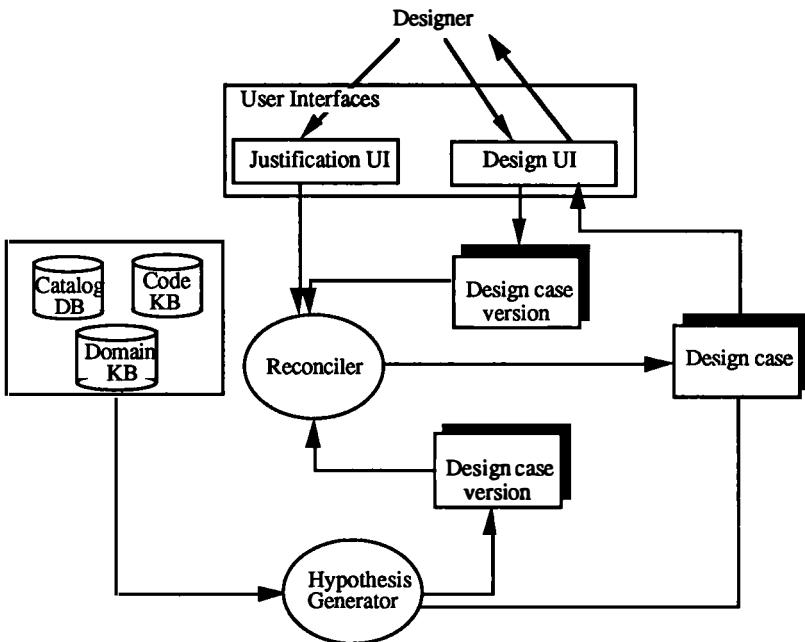


Figure 4: Knowledge Flow in ADD

Also in the domain KB, we find the design constraints, including both internal and external constraints. Internal constraints are imposed by either the function or the behavior of the artifact in the environment such as "the dependency relation between number of ducts and ceiling space." External constraints are imposed by other design agents, such as the owner's constraint on cost. These constraints can be implemented using dependency relations among objects, as illustrated in SALT (Marcus 88) a knowledge acquisition system for configuring elevators. The domain KB will also contain heuristic rules to support the decision-making process. From our observations of design sessions, preliminary design decisions are heavily supported by heuristic reasoning.

Figure 5 illustrates a small portion of the domain KB's contents. Figure (5.a) illustrates the environment in which the design will be developed including the values for the artifact parameters, (5.b) illustrates the classification model of the HVAC system, (5.c) external and internal constraints are presented and finally (5.d) a design heuristic rule example.

The code KB contains the restrictions imposed by law; these constraints can not be relaxed by ADD. The catalog DB represents the available standard components. Our goal is to connect ADD with available manufacturers' databases, but initially we will create our own catalog data base. The components are described in terms of their attributes such as capacity and price. Since we are using frames to describe the artifact and environment, we will build the catalog using frames too.

In addition to domain knowledge, ADD needs to understand the user's needs in requesting design explanations. Five types of users access design documents: the original designer, another designer, the owner, the owner's representative, designers of other design trades and the plan checkers. Their needs are different, depending on their background knowledge and their goals in accessing the documents. These differences are observed in the objects and relations contained in the design explanations requested by each one of them. We are currently identifying those needs for designing the explanation.

METHODOLOGY FOR DEVELOPING ADD

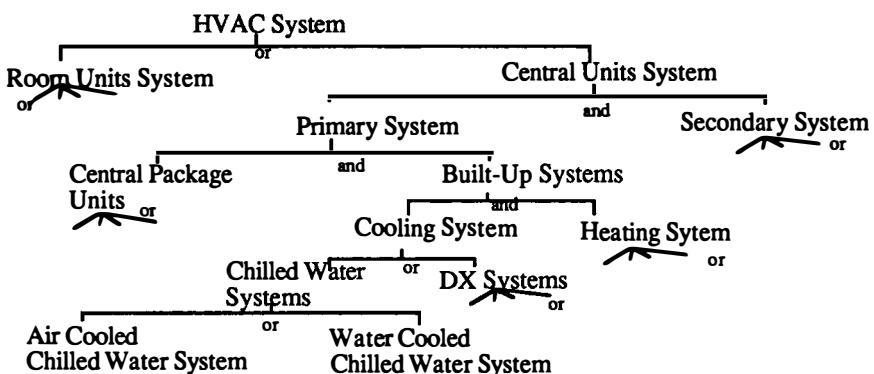
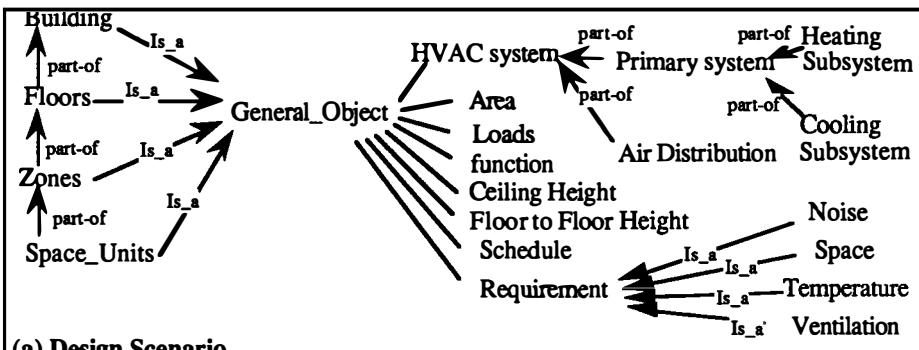
Our starting point for implementing the ADD model is observing how designers develop designs for HVAC systems. We are trying to identify the important aspects of the design process and the rationale generated. The methodology we are using for this knowledge engineering work includes videotape analysis and structured interviews. We are working with three different companies (two contractors and one consulting firm). The contractors are known as cost oriented companies while the consulting companies are quality oriented. We have videotaped design sessions and meetings during the conceptual design stage. Those sessions are showing us which type of knowledge is important for justifying a design; i.e., which objects, attributes and relations should be present in design explanations in order to satisfy the goals of the users in accessing the documentation. We are looking for evidence of rationale normally used and provided by designers. The structured interviews with designers are providing additional background on the HVAC design process and the designers' strategies.

Videotape Analysis

We are videotaping four types of sessions:

1. The designer explains a previous project to the knowledge engineer.
2. Two designers develop the main concepts of the design.
3. The designer develops the design with interruptions by the knowledge engineer.
4. The designer develops the design in the presence of designer of other trades.

Initially, we viewed the tapes without having any expectations of what would be important. After some initial observations, we built an initial framework for analyzing those taped sessions. We tried to clarify how much information should be acquired from the tapes, the characteristics of that information, and the user interface needs. The main objective of this framework is to define design rationale for the domain of HVAC system design. The framework is divided into three levels: analysis by session, analysis by decision (within a session), and analysis by alternative (within a decision).



<p>Zones -- dependency --> Air Distribution</p> <p>Air Distribution -- Fit? --> Zone_Ceiling_Space >= Ductwork_Required_Space</p> <p>ceiling space = floor to floor height - beam height - (fixture height)</p> <p>HVAC subsystem Alternative</p> <p>(c.1) Internal Constraint: dependency relations</p>	<p>Ductwork required space</p> <p>Activated Evaluation Functions:</p> <p>Min First Cost(Alternatives)</p> <p>(c.2) External Constraint</p>
<p>If number of floors ≤ 3 & building area $\leq 1000 \text{ sq ft}$</p> <p>Then Building is a mid-size building</p> <p>(d) Heuristic Rule for choosing the system concept</p>	<p>If building is a mid-size building & no "strong" aesthetics constraints on roof</p> <p>Then Package systems are "strongly" recommended</p>

Figure 5: Domain Knowledge Base

1. Analysis by Session

- Type of session**

Participants. The session participants (HVAC designers, other design specialists, owners, contractors, etc.) determine the type of interactions that take place within the session.

Session format. This parameter indicates the level of consciousness in the designer's reasoning process. For example, a designer working alone frequently takes big steps in the decision process without annotations. If the knowledge engineer interferes to ask for an explanation, the design process is a bit clearer, but the design process altered. If the designer is discussing a design with a peer from the same company, there are frequently reasoning jumps that are difficult for the uninitiated to follow. If the designer is explaining the project to other trades, the amount and type of information used are different than would be used in discussions with a peer. In summary, this parameter measures the changes in the type and granularity of information used during the session.

- Purpose**

The purpose of a design session is very subjective. It is related with the session output, which sometimes is indefinite. For the intention of this research, the purpose of the session is classified in three main categories: (1) generating an initial design concept, in which at least the type of HVAC system is selected; (2) identifying constraints that will necessitate changes in the initial design; and (3) producing the schematic design (how everything fits in the building). Purpose (1) and (3) fits the synthesis type of session, while (2) fits the analysis sessions.

- Session summary**

The summary includes a synopsis of what happened during the session, the design strategy, the amount of background knowledge used, the understandability of the explanation raised during the session, the jargon used, and the types of inference needed.

- Research output**

This parameter identifies the type and content of the design information conveyed in a specific design session, for example, the designer's strategy, the designer's heuristics, and the structure used to explain the design. By structure, we mean the objects and relations contained in the explanation.

- Designer behavior facing new knowledge or constraints.**

The idea here is to record the effects of new factors as they are discovered. Does the designer simply check whether previous decisions are still valid or he will start again his

analysis? How much does the new factor influence previous design? What type of adjustment is required?

- **Interface needs**

Discussion media. This illustrates the environment required for design development during a given session. This parameter defines the tools necessary for integrating the design interface. For example, we observed that designers need to have a calculator available for them. We also noticed that they need to look at working drawings during the design process to understand the design.

Language needs. Does the discussion remain inside the domain? What type of common sense knowledge is invoked?

- **Number of decisions taken and the names of those decisions.**

These are links to the decision analysis records described below.

- **Final comments**

2. Analysis by Decision

- **Transcript of the design session portion that leads to the decision.**

The detailed representation of the decision includes a transcript from the videotape of the designers' discussion.

- **The decision**

This parameter describes the type of decisions that were taken, which may include no decision and eliminating or adding a set of alternatives.

- **What triggers the decision?**

This parameter identifies the type of knowledge that triggering the decision including a recognized pattern, the building data, the sequence of the decisions, heuristic rules, or a set of constraints that need to be satisfied. It helps to identify the domain design strategy.

- **Sub-decisions**

This parameter points to the intermediate decisions that are needed for building the specific decision that is being analyzed.

- **Related decisions**

This parameter links the current decision to other decisions that triggered this decision or are triggered by it. The relation may be chronological or logical (causal). This parameter together with the sub-decision parameter helps to organize the decision graph of a particular design case.

- **What different types of knowledge are used for generating the decision?**

The knowledge may include such things as designer heuristics, constraints from building codes, or solutions from previous similar cases. If the designers draw on their experiences in specific cases, we need to record the criteria for similarity and the methods for transferring solutions.

- **Number of alternatives considered and names of those alternatives**

These are links to the alternative analysis records described below.

- **Final comments.**

3. Analysis by Alternative

- **Transcript of the portion of the design session that leads to the identification of the alternative.**

The detailed representation of the alternative includes a transcript from the videotape of the designers' analysis.

- **What is the alternative?**

This parameter describes the type of decisions that were taken including no decision and eliminating or adding a set of alternatives.

- **What triggers the alternative?**

This parameter identifies the type of knowledge triggering the decision including a recognized pattern, the building description, the natural order of the decisions, heuristic rules, a set of constraints that need to be satisfied. It helps to identify the domain design strategy.

- **Sub-decisions**

This parameter points to the intermediate decisions that are needed for making this candidate decision feasible.

- **Related decisions.**

This parameter links the current decision to other decisions that triggered this decision or are triggered by it. The relation may be chronological or logical (causal). This parameter together with the sub-decision parameter helps to organize the decision graph of a particular design case.

- **What is the change proposed by this alternative?**

Does it add or relax constraints, add or relax preferences, cause the solution to converge to that of a previous design, etc.?

- **Use of previous solutions**

If information from a previous solution is used, does that previous case provide positive or negative examples? Similarly, what triggered the recognition of similarity—matching elements in the building descriptions, previous experiences with other project participants (especially owners), etc.? What type of knowledge is used for selecting the similar case? How was the solution transferred?

- **How does the design change proposed by this alternative influence other trades?**

- **Final comments**

Even though this videotape analysis consists the major technique for knowledge acquisition, it is not enough. Many design sessions bring new concepts and discussions that are only superficially mentioned at that time. In order to investigate in more details what actually happens we make use of another knowledge acquisition technique: the structured interviews.

Structured Interviews

The purpose of the structured interviews is to give a better understanding about the design sessions and also about the domain model. The building (where the HVAC system fits), the artifact (HVAC system), the design process and codes are examples of information that composes the HVAC system design domain.

In our first meetings, we elicited some general information about the field. Those interviews helped us to define the scope of our project as well as to provide us literature reference. The second round of interviews has helped us to clarify the design session videotapes. We have been meeting with the designers and asked for clarification over the design session. We propose a scenario to the designer and ask for his or her reactions. Then, we fix all variables, but one, in that scenario. We change the value of the free variable and observe the designer's reaction in order to get a deep understanding in the role of such variable.

Figure 6 presents a sample scenario and the reaction of one of our designers. First, the designer examined the building initial condition and proposed a conceptual HVAC system for it, in our case, a built-up floor by floor system. Maintaining all design parameters fixed except for the height constraint, we observed that the designer changed his choice from floor by floor to central system. Repeating the exercise with varying numbers of floors, we verified that the change goes from built-up system to package units. Based on those observations, we verified with the designer the degree to which each isolated variable influenced his decisions.

Design Initial Conditions	
rectangular building	
cost constraint	
height constraint	
7-story building	
building square footage	
no basement	

Designer's initial choice:	BUILT-UP SYSTEM.FLOOR BY FLOOR SYSTEM
Change to design specification	Designer's new choice
Elimination of height constraint	BUILT-UP SYSTEM.CENTRAL SYSTEM ON ROOF
Increase building size	BUILT-UP SYSTEM.FLOOR BY FLOOR SYSTEM
Decrease building size to up to 3 floors	PACKAGE UNITS SYSTEM
Elimination of height constraint +	BUILT-UP SYSTEM —> FLOOR BY FLOOR SYSTEM
High level of flexibility	

Figure 6: Sample Scenario from a Structured Interview

INITIAL OBSERVATIONS FROM THE DESIGN SESSIONS

The initial analyses of the design sessions emphasized the importance of having the floor plan visually represented as the medium for developing the HVAC system design. The designers use simple electronic calculators, rulers for taking measures from the floor plan, and parts catalogues to aid in component selection. In later stages of the design (but also during conceptual design phase), they run heating load analysis programs to make sure that their estimates of the cooling and heating loads are realistic. The numbers, the decisions,

and the explanations get more precise depending on the size of the project and the client's demand.

The designers we studied rely heavily on their experience from previous cases. In every videotape session, the designers have recalled at one previous case. They recall those cases by either according project description similarities or by contact with the same project participants (especially owners). The negative examples (poor design alternatives) are recalled more often than the positive examples. Our initial explanation for this phenomenon is that the positive examples are compiled in the designer's generalized rules of thumb, while the bad experience remains primarily connected with the case.

Another observation is concerned with the different level of abstraction and also different type of explanation depending on the type of design session. When explaining for the knowledge engineer, designers explain in a professorial format. When talking among each other (i.e., between designers of the same trade), they speak at a much higher level. When explaining the design for other trades, they try to explain the design by addressing only the portion that affects the other designer.

A last observation on those design tapes is that designers do take notes of their design process. Those notes are very concise and rely upon the designer's power to recall the entire situation in which a decision was made. Those notes work as pointers for their memory.

FUTURE WORK

The first steps of our research were towards understanding the design knowledge capture problem, as described in this paper. We are proposing a descriptive model of the design process in a specific domain, HVAC system design, for supporting the documentation of the rationale in design cases. The next steps are towards the implementation of this computational model and further development of the ADD model.

Acknowledgements. We would like to thank Jim Bishop, Steve Poe, Linda Wilkins and Grant Wong for participating in our experiments. We also would like to thank Paul Chinowsky for his comments on drafts of this paper. This work has been supported by grants from the National Science Foundation (MSM-8958316) and the Stanford Center for Integrated Facility Engineering.

REFERENCES

- Baudin, C., Sivard, C., and Zweber, M. (1989). A Model Based Approach to Design Rationale Conservation, technical report, No. M.S. 244-17, NASA AMES Research Center.
- Conklin, J., and Begeman, M. L. (1988). gIBIS: A Hypertext Tool for Exploratory Policy Discussion, *Computer-Supported Cooperative Work*, Association for Computing Machinery, pp.140-152.
- Fischer, G., McCall, R., and Morch, A.(1989). Design Environments for Constructive and Argumentative Design, *CHI' 89*, Austin, pp.269-276.

- Gruber, T. R., and Russell, D. M.(1991). Design knowledge and design rationale: A Framework for representation, capture, and use, to appear in the special issue of *Human Computer Interaction on Design Rationale*, Summer 1991.
- Howard, H. C., Wang, J., Daube, F., and Rafiq, T.(1989). Applying Design-Dependent Knowledge in Structural Engineering Design, *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 3(2): 111-124.
- Howard, H. C. (1991). Project Specific Knowledge Bases in the AEC Industry to appear in the *Journal of Computing in Civil Engineering*, 5(1): 25-41.
- Lakin, F., Wambaugh, J., Leifer, L., Cannon, D., and Sivard, C. (1989). The electronic design notebook: performing medium and processing medium, in *The Visual Computer*, 5: 214-226.
- Lee, J., and Lai K.(1991). What's in Design Rationale?, to appear in the special issue of *Human Computer Interaction on Design rationale*, Summer 1991.
- Marcus, S. (1988). Salt: A Knowledge-Acquisition Tool for Prpose-and-Revise Systems, in *Automating Knowledge Acquisition*, Kluwer Academic Publishers, Boston, pp.81-123.
- Tang, J. C. (1989). Listing, Drawing, and Gesturing in Design: A Study of the Use of Shared Workspaces by Design Teams, PhD thesis, Department of Mechanical Engineering at Stanford University, April 1989.

Representing and reasoning with design intent

R. Ganeshan,[†] S. Finger[‡] and J. Garrett[†]

[†]Department of Civil Engineering
Carnegie Mellon University
Pittsburgh PA 15213 USA

[‡]Robotics Institute
Carnegie Mellon University
Pittsburgh PA 15213 USA

Abstract. An engineering drawing contains the results of the decision-making process in a design, but does not record how and why the decisions were made. We call this information about the decisions *design intent*. A record of the design process may be used for: 1) verifying the design, 2) explaining the design, 3) determining the effect of modifications to the design, and 4) reusing all or part of a design in a different context. Our goals are to develop a model of design decision-making that explicitly identifies the intent and to develop a representation for the record of the design process. We take the view that design is the process of refining high-level specifications, called objectives, into a physical description of the artifact. Examples of types of objectives are: functionality, aesthetics, economy, constructability, maintainability, and disposability. Objectives may be refined into other objectives or into design decisions. The intent behind a decision is the set of all the objectives starting from the highest-level objectives, including all their refinements, leading to the decision. Our design paradigm is characterized by five kinds of activities: 1) determining design focus, 2) refining the objectives 3) evaluating design alternatives with respect to objectives, 4) selecting an alternative based on the results of evaluations, and 5) resolving conflicts among various design objectives. Operators perform each of these activities in a given design context. The current state of the design is the description of the artifact plus the objectives, both satisfied and unsatisfied. The design record is a tree whose nodes represent design states and whose arcs represent the activity that caused the transformation from one state to the next.

INTRODUCTION AND MOTIVATION

Civil engineering facilities are large, one-of-a-kind, and last for many years, and many diverse perspectives participate in their design. The duration of the design process itself is often measured in years. During this period many decisions are made, affecting many parts of the design. The motivations for these decisions are usually recorded informally in the designer's notebook, if they are recorded at all. When changes to the design must be made, it is often difficult to reconstruct why certain decisions were made and to predict the consequences of proposed changes.

A design may be altered for many reasons. During the design process, previous decisions may need to be altered or retracted because of changes in specifications or assumptions. Events during the construction of a facility may require design changes. Renovations involve making design changes to completed facilities. A new design may be generated by modifying the design for an existing structure. To make changes in a design, the intent behind the original decision must be known.

For example, the choice of a material for a structure is usually based on service conditions (exposure, temperature, humidity, chemical attack etc.). Architectural requirements may leave certain elements of the structure exposed. As a result a material (say A242 steel) with specific properties (corrosion resistance) may be chosen for these elements. Such information is not documented as part of design drawings. The designers responsible for the specifications may or may not be able to recollect the reason behind a particular decision. It is this information that we call *design intent*.

We explicitly represent and reason with design objectives. Objectives are requirements at different levels of abstraction that drive the design process. Examples of types of objectives are: 1) functionality, (2) aesthetics, (3) economy, (4) constructability or manufacturability, (5) maintainability (the artifact must be maintained during its service life), and (6) disposability (the artifact must be disposed of at the end of its service life). Another class of design objectives is concerned with planning the design process. For example, process objectives might be to minimize the time to complete a design and to minimize the number of design iterations. These objectives guide the decision process during design.

We view design as a process of refinement of objectives (see Figure 1). The design begins with a set of objectives that represent the highest level specification (level n in Figure 1). Objectives are refined into other objectives and/or into design decisions. At any stage in the process, the design is represented by the decisions that have been made so far. The refinement of an objective may be thought of as the process of meeting the objective. Thus an objective may be achieved by decomposing it into a set of new objectives, by making a design decision, or by making a decision and creating a new set of objectives. This process of refinement continues until all current objectives in the design are met.

Given this definition of the design process, we can now define the intent for a decision, D, as follows. Let S_0 be the set of objectives that were directly responsible for D. The set S_0 is a subset of the objectives in level 0 (Figure 1). For each O_i in S_0 , let s_{0i} be the set of objectives that were directly involved in the refinement that created O_i . Define S_1 to be the union of all sets s_{0i} i.e. $S_1 = \bigcup_{O_i \in S_0} s_{0i}$. Similarly, for each objective O_i in S_1 , let s_{1i} be the set of objectives that were involved in the refinement that created O_i . Define S_2 to be the union of all sets s_{1i} . The intent for

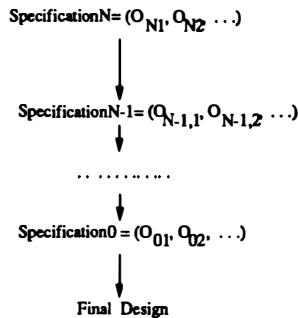


Figure 1: Design as refinement of objectives

decision D is the union from level 0 to n, that is, $\text{intent}(D) = S_0 \cup S_1 \cup S_2 \dots \cup S_n$.

Our approach to capturing design intent is to maintain a record of the design process starting from the highest-level objectives, including all their refinements, leading to the decision. A documentation of the design process is useful for:

- (a) Explanation - to explain how and why a particular decision was made;
- (b) Verification - to determine if characteristics of the final design are consistent with the intended characteristics;
- (c) Modification - to predict the effect of making changes to the design; and,
- (d) Re-use - to synthesize a design from a previous design with a similar specification.

No formal methods are available for documenting the design process for constructed facilities. The design process may be recorded informally in the designer's notes, but only the final results of this process are documented in design drawings and other specifications. Informal notes are unstructured and often incomplete and hence, are not useful. Our goals are to develop a model of design decision-making that explicitly identifies the intent and to develop a representation for the record of the design process.

BACKGROUND

Much of the research on capturing design intent has been done for software design. The motivation behind this work is to make it easier to maintain large programs and to reuse existing software. The record of the decisions made in the development of a program from the specifications to the final implementation may be replayed to determine how changes to the specification affect the program (Wile, 1980).

Some of the earliest work in capturing design intent was in the development of the XPlain system (Swartout, 1983). The XPlain system presents an approach for capturing the reasoning that goes into the design of an expert system. The goal is to generate explanations of the system's behavior. An automatic programmer is used to generate the expert performance program. As the program is generated, a refinement structure is created which gives the explanation routines access to the decisions made

during the creation of the program. The refinement structure can be thought of as a trace of the process that created the expert system.

Most of the work in capturing design intent has been motivated by the notion of redesign in which a new design is created by modifying a previous design solution. Redesign is the underlying methodology used in design by derivational analogy (Mostow, 1987) and in case-based reasoning (Kolodner et al., 1985). Redesign, a system for circuit design, uses the solution to a previous design to obtain a new design for a similar set of specifications (Mitchell et al., 1983). It uses a representation called a *Design Plan* to capture the purpose for each module of the circuit. The design plan describes how the specifications were decomposed to develop a circuit (that is, the rules that were fired) but does not capture why a particular decomposition was used.

Another motivation for capturing design intent is to determine the effect of modifying design decisions. Modification is particularly important in the maintenance of complex software. Wile developed a language that allows the user to express the design decisions made in the development of a program (Wile, 1980). The language, called Paddle, has definition facilities for transformations, strategies, and plans capable of expressing different types of goal structures. The language does not include justifications for the decisions made during the development of a program. The history of the development of a program from specification to final implementation is captured in a development structure that can be replayed to determine the effect of changes to the original specification.

In the areas of civil or mechanical engineering design, little effort goes into capturing the design intent. The only records of such information are the informal notes in a designer's notebook. We describe two recent efforts that begin to address this problem. Rossignac et al. present an approach in which the designer interactively designs a geometric model of a mechanical part. The commands are captured as un-evaluated specifications of operations (Rossignac et al., 1988). The system provides tools for editing, parameterizing, and combining these sequences so they can be applied to a family of models. The sequence of operations explains how the final model was determined but does not answer why the particular set of operations were used. Thompson et al. describe a design environment that allows designers to express design strategies and tactics used in developing the final product description by successively refining the specification (initial product description) (Thompson and Lu, 1990). The design decisions made during the refinement process are recorded.

The main limitation of most of the work described above is that the objectives motivating decisions are not explicitly represented in the design process. Recording the rule that fires is not sufficient because the rule does not contain information about the motivation for the rule. For example, while designing the columns and girders of a rigid-frame for seismic loading, a common design rule is to keep the column-girder stiffness ratio at around 2.5. The objective behind this rule is to ensure frame stability. It is preferable to have plastic hinges form in girders before they form in the columns. This kind of information is not available in the rules of existing design expert systems.

A system for capturing design intent must represent high-level objectives (e.g. stability) and the manner in which they are considered in the design process. In the example given above, stability in a rigid-frame is refined into an objective requiring that plastic hinges form in girders and not in the columns. This objective is then refined into the heuristic constraint that the column-girder stiffness ratio be greater

than 2.5. We propose to represent the high-level objectives and the refinements that they undergo during the design process. This raises the issue of representing both the evolving design artifact and the design objectives. In the remainder of this section, we look at the representations used in the research described above and in other relevant research.

Computer programs are expressed in well-defined programming languages. Editors are used to manipulate these programs. However, programming strategies such as divide-and-conquer are expressed using English-like descriptions in Paddle (Wile, 1980). In Mamour (Rossignac et al., 1988), the design is represented as a list of features defined on a boundary element model. As the authors themselves note, features are useful insofar as the system developer, the user, and the application programs agree on the semantics of the feature. Thompson *et al.* (Thompson and Lu, 1990) describe the design product in terms of assemblies and components. They allow design decisions to be expressed in mathematic, algorithmic, temporal, and textual formats. The textual formulations cannot be interpreted by the system.

While well-defined representations for describing completed design artifacts exist, the representation of the evolving design at different levels of abstraction is an ongoing research problem. In several engineering domains, research has been devoted to defining a formal language for expressing the design. For example, VHDL (Lipsett et al., 1986) is a language for describing digital computer systems. The goal for developing the language was to provide precise syntax and semantics for hardware components, enabling design transfer both within and among organizations. The language allows the description of hardware components at different levels of abstraction ranging from the logic gate level to the digital system level. It provides a well-defined interface for the development of various design tools. Another effort in this direction is PDES/STEP (Bloom, 1989). The definition of this language is an international standards effort.

THE DESIGN MODEL

There are four types of entities in the design model: objectives, decisions, alternatives and operators:

- (a) **Objectives** are functional, aesthetic, economy, constructability, maintainability and disposability requirements for a design that exist at different levels of abstraction.
- (b) **Decisions** constitute the descriptions of the design artifact (e.g. shape-of-element = wide-flange). They are the result of decision-making activities.
- (c) **Alternatives** are sets of decisions that represent the current state of the evolving design artifact. A choice is usually made about which of the alternatives to pursue.
- (d) **Operators** contain the knowledge to make design decisions, i.e. they use objectives to make design decisions.

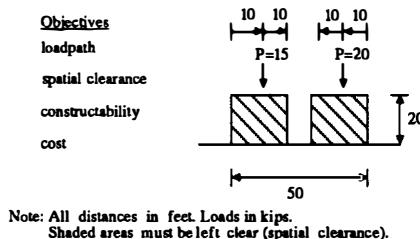
We distinguish between following types of **activities** in our design model:

- (a) determining *design focus*
- (b) *refining* a decision or an objective
- (c) *evaluating* design alternatives with respect to objectives
- (d) *selecting* an alternative based on the results of evaluations
- (e) *resolving* conflicts among various design objectives

This notion of the types of activities is derived from an approach used in a system called SPEX (Standards Processing EXpert) (Garrett, 1985).

Operators are invoked when specific combinations of decisions and objectives are present in the design state. The design is completed when a level of detail sufficient to construct (or manufacture) the artifact has been reached and the current set of objectives are achieved. This model is conceptually similar to the approach of Pahl and Beitz (Pahl and Beitz, 1984). In their approach the problem specification identifying the functional requirements and constraints is broadened systematically by focusing on the important functions and identifying solution principles to fulfill those functions.

The activities in the design model are better described with the help of an example. Consider the design specification in Figure 2. In this example, we use the following



Note: All distances in feet. Loads in kips.
Shaded areas must be clear (spatial clearance).

Figure 2: Specifications for Design Example

definitions:

A *line element* is any long structural element whose cross-sectional dimensions are small with respect to its length, e.g. beams, columns etc. An important structural property associated with line-elements is their rigidity. They may be rigid (small deformations under load) or non-rigid (substantial deformations under load as in a cable) (Schodek, 1980).

A *load path* is a collection of line elements that are spatially organized to transfer loads to the ground.

Stability means that a structure, or any part of it, must be in equilibrium under all possible loading conditions. In the planar domain, stability means that the structure and its parts must satisfy the three equations of equilibrium: $\sum F_x = 0$, $\sum F_y = 0$ and $\sum M = 0$ for all loads.

Strength means that every part of the structure must be able to withstand the structural actions to which it is subjected without reaching any of its limit states.

Constructability means that it must be possible to construct the design with available construction technology.

Cost is the cost of materials and labor used in the construction process.

In this example, the design goal is to carry the given loads, i.e. to provide a load path, subject to the requirements of spatial clearance, constructability, and cost.

Focusing. The first step in the design process is to determine which objectives to refine. Often one objective is chosen before another due to an information dependency between objectives. The focus decision may have implications for the efficiency of the design process, i.e. it may affect the time to arrive at a solution. The current version of the model does not address this issue. We intend to address this in future versions of our model. In the example, the initial focus is to provide a load path that satisfies the spatial clearance objective. Thus both the load-path and spatial clearance objective become the current focus.

Refining. In our model, an *objective refinement* results in an objective being decomposed or specialized into other objectives or a decision decision. Here, the objectives are achieved by making refinements to the design and to the load-path objective. This results in a number of alternatives as shown in Figure 3. Also, the load-path objective is refined into objectives for stability and strength.

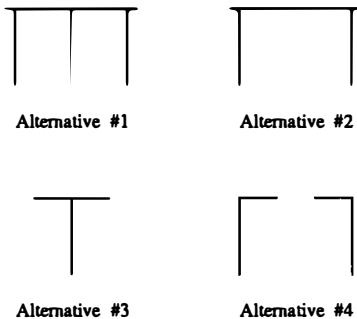


Figure 3: Alternative Load Paths

Evaluating. At this point the designer choose which alternative to pursue first. The evaluation of the alternatives must be based on the information available at the current level of abstraction. Considerations of economy might eliminate elaborate load-paths. Evaluation of a set of alternatives with respect to an objective might result in a ranking of the alternatives, i.e. some alternatives may be preferable to others. Also, different objectives might give preference to different alternatives.

Selecting. The process of selecting involves choosing the alternative that best satisfies all of the objectives considered in the evaluation. The selection process makes use of knowledge about the relative importance of the objectives. Sometimes, it may

not be possible to make a clear choice at this point, and the designer might arbitrarily select any one of the alternatives.

Resolving. All of the alternatives may be eliminated in the process of evaluation. In other words, none of the alternatives may satisfy all the relevant objectives, so no alternatives are available for selection. We refer to this state as a *deadend*. This interaction among objectives may be resolved by modifying one of the alternatives or one of the objectives. This is done during a resolution activity.

The use of this methodology in a computer-based-environment raises a number of problems that need to be addressed:

- (a) A representation language needs to be defined for expressing objectives and design decisions so that the design operators can manipulate them. Work on design representations (Fenves and Baker, 1987), (Lipsett et al., 1986) is relevant here.
- (b) A library of design operators is needed to perform the kinds of design activities for the specific domain in which the design is occurring.
- (c) The appropriate operator to use in the current design context must be found. A classification scheme for the operators is needed to allow a user or the system to browse through the library to find the appropriate operator.
- (d) The granularity of the operators, and hence of the design decisions, must be determined. The appropriate level of granularity will depend on the domain.
- (e) The intent for focus activity must be represented.
- (f) The design decisions, objectives, activities and intermediate states in the design process must be recorded.
- (g) The problem of scaling to realistic design problems, with a large number of decisions and many intermediate design states, must be addressed.

However, the focus of this research is on documenting the design process, so not all of these problems need to be resolved for our purposes. In particular, we assume that the designer will provide the design focus and substitute for design operators when needed.

Design Objectives

Objectives are statements of the desired attributes of a design, dealing with functionality, aesthetics, constructability, maintainability etc. We include constraints in our definition of an objective.

Representation of objectives. An objective may be expressed in one of the following ways:

- (a) a string of text representing an abstract concept. An abstract concept is a word or phrase with which certain semantics are associated that are understood by those familiar with the domain. For example, stability implies that the structure as a whole must satisfy equilibrium conditions.

- (b) a mathematical expression involving design variables, e.g. $\text{deflection} \leq \text{span}/240$
Min Cost, where Cost = Lbh etc.
- (c) qualitative expressions such as proportionality, e.g. cost is directly proportional to length. There is considerable research in developing qualitative models to simulate the physical behavior (Bobrow, 1985). While the overall goals of our research may be different, the representation issues are similar.
- (d) predicates in first order logic, e.g. direct-access(kitchen,diningroom) states a requirement that the diningroom be directly accessible from the kitchen.
- (e) schematic diagrams

Refinement of an objective. Objectives are refined based on the current design context. The refinements may be thought of as a way to achieve the objective in the current design context or as a specialization of the objective. The refinements are strategies for achieving the parent objective. We provide the following constructs to express dependencies among refinements of an objective:

- (a) “AND” - all the refinements must be satisfied (in any order)
- (b) “OR” - either of the refinements may be satisfied
- (c) “SEQ” - the refinements must be satisfied in the given order

This list of constructs is not necessarily complete.

The language for expressing objectives may be described in BNF syntax as follows. The items in parentheses are optional.

```

<objective> ::= (AND <objective><objective-list>)
                | (OR <objective><objective-list>)
                | (SEQ <objective-list>)
                |<obj>
<objective-list> ::= <objective><objective-list>
                    |<obj>
<obj> ::= <symbol> (<decision>)
           |<expression>

```

A symbol is simply a string such as stability, spatial-clearance etc. An expression states a relationship between design decisions using arithmetic and logical operators.

Satisfaction of an objective. Some objectives can be either satisfied or violated (e.g. stability). Others may have levels of satisfaction (or violation). When objectives are expressed quantitatively it may be possible to compute a value for the objective function. The degree of satisfaction is a measure of the difference between the optimum value of the objective and its current value. If it is possible to arrive at a uniform measure for all objectives, this kind of information may be useful in analyzing trade-offs. An issue that needs to be considered is how the degree of satisfaction is reflected in the refinement of an objective.

Importance of an objective. Another concept closely associated with satisfaction is the importance of each objective. Some objectives, such as stability, must

be satisfied in the design of a structure. Information on importance will be useful in resolving conflicts between objectives.

Every design objective has the following information associated with it:

- (a) the parent objective of which it is a refinement
- (b) the dependencies with other objectives
- (c) the value (a word or phrase like stability or an expression like $\text{deflection} \leq \text{span}/240$)
- (d) the status of the objective
- (e) the importance of the objective
- (f) the version

The status of an objective indicates whether the objective is the *current focus* (it is currently being addressed) or it has been *satisfied* (it is satisfied in the current state of the design) or it is still *active* (it is not satisfied in the current state of the design) or it is *refined and active* (it has been refined but its refinements are not yet satisfied). The status of an objective may be changed during the design process as described in the following section. Objectives may be modified, or a constraint may be relaxed, during the design. The version of an objective is used to keep track of these revisions.

Design Decisions

A design decision describes an aspect of the design artifact at some level of abstraction (e.g. material = A36). Note that we use the term decision to describe the result of the decision-making activity. In the structural design domain, the values of design decisions can be materials like A36 steel, reinforced concrete, or glued-laminated timber; points and lines; structural properties like rigid or non-rigid; load-transfer mechanisms like bending, shear or axial force; and shapes like wide-flanges, rectangular sections etc.

We represent design decisions in terms of entities and attributes. An entity is a grouping of attributes. Entities with attributes may be predefined, or the designer may create entities and attributes.

A decision refinement adds an attribute or set of attributes to newly-created or existing design entities. For example, an entity representing an element may be refined to have centerline-geometry (line), a rigidity (non-rigid or rigid), load-transfer-mechanism (bending, shear, or axial), and a shape (wide-flange) and material (steel). See Figure 4.

A set of design decisions constitutes a **design alternative**. When a decision within an alternative is refined a new alternative is created. A decision refinement can create a number of alternatives. During the design process one of these alternatives is selected for further refinement. See Figure 5.

Every design decision, has the following information associated with it:

- (a) entity with which it is associated (e.g. element)

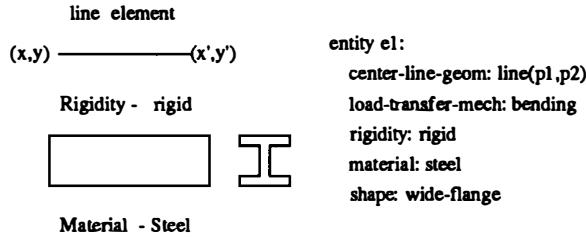


Figure 4: Refinements in design of an element

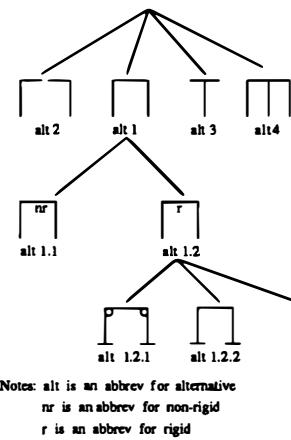


Figure 5: Representation of the evolving design

- (b) type of attribute (e.g. load-transfer-mechanism)
- (c) value (e.g. bending)
- (d) version
- (e) validity

The version indicates whether the decision is the current version. The problem of tracking design versions is an ongoing research issue (Katz, 1990). The validity indicates whether the current value for the design decision is valid. A value may be invalidated by changes made to other parts of the design. The dependency information is maintained by the design record.

Design Operators

Design operators provide the knowledge for performing the activities (focus, refine, evaluate, select and resolve) in a design domain. Design intent is incomplete without capturing this knowledge. Explanations and replay of the design are not possible without this knowledge. Design operators can be thought of as a function mapping

one state in the design to another state. The design state includes both the objectives and the design decisions. Operators are triggered when an (*activation pattern*) exists in the current design state. The activation pattern may include both objectives and decisions. Operators may have preprocessing requirements that must be satisfied before they can be used in the design. These requirements involve the values of the design variables that the operator uses, for example, the results of an analysis to determine the magnitudes of moments, shear forces, deflections etc. In satisfying these requirements, assumptions are often made. For example, analyzing an indeterminate structure may require making assumptions about the relative stiffnesses of the individual members because the member sizes are unknown. They are recorded to check that subsequent decisions do not violate the assumptions.

OPERATION OF THE DESIGN MODEL

The design state contains objectives and alternatives. The objectives are divided into five sets: 1) O_a : set of active objectives, 2) O_s : set of satisfied objectives, 3) O_f : set of in-focus objectives, 4) O_{ra} : set of objectives that are refined but still active, that is, with unsatisfied refinements, and 5) O_i : the set of objectives that were considered and whose violations are being ignored.

The alternatives are divided into three sets: 1) A_c : the current alternative, a singleton set, 2) A_e : the set of eliminated alternatives (an alternative is eliminated when it fails to satisfy an objective with respect to which it is evaluated), and 3) A_{ne} : the set of alternatives that are not-eliminated. In any state one or more of these sets may be empty. The basic actions that may be performed in a given state are: focus, refine, evaluate, select and resolve. We describe each of these actions and their effect on the design state using the example described earlier.

The focus operation determines the objectives in focus (O_f) for a refine, evaluate, or resolve action based on the current design state.

Action: Focus $[(O_a, O_s, O_f, O_{ra}, O_i), (A_c, A_e, A_{ne})]$

After State: $[(O'_a, O'_s, O'_f, O'_{ra}, O'_i), (A_c, A_e, A_{ne})]$

where

$$O'_a = O_a - O'_f$$

In the example, focus applied to the initial state S_0 results in the state S_1 .

Action: Focus

Before state: $S_0 = [((\text{loadpath, spatial-clearance, constructability, cost}), (), (), (), (\text{(), ()}, (\text{(), ()}, (\text{(), ()})))]$

After state: $S_1 = [(O'_a, (), O'_f, (), (), (((), (), ()))]$

where

$$O'_f = (\text{loadpath, spatial-clearance})$$

$$O'_a = (\text{constructability, cost})$$

Based on the current alternative, the refine operation invokes an operator to refine the objectives that are in focus. When decision alternatives are generated by a refinement, some of the alternatives may violate previously satisfied objectives. Such objectives are automatically identified and included as part of the focus for the evaluate action.

Action: Refine $O_f A_c$

Before State: $[(O_a, O_s, O_f, O_{ra}, O_i), (A_c, A_e, A_{ne})]$.

After State: $[(O'_a, O'_s, O'_f, O'_{ra}, O_i), (A'_c, A'_e, A'_{ne})]$.

Let $O_f = O_{fr} \cup O_{fs}$

where

O_{fr} be the set of objectives that were refined

O_{fs} be the set that were satisfied

Let O'_{fr} be the set of refinements of O_{fr} .

Let $O_{ras} \subset O_{ra}$ be the set of objectives satisfied

because of objective satisfied in this action (i.e. O_{fs}).

then

$$O'_{ra} = O_{ra} - O_{ras}$$

$$O'_a = O_a \cup O'_{fr}.$$

$$O'_s = O_s \cup O_{fs}.$$

$$O'_{ra} = O_{ra} \cup O_{fr}.$$

$$O'_f, A_c, A_e \text{ are empty.}$$

A'_{ne} contains the new alternatives created in this step.

Referring back to our example, refining the objectives in focus in state S_1 produces a new state S_2 . The loadpath objective is refined into two sub-objectives: stability and strength. We can use the “SEQ” construct to specify that the stability objective must be achieved before the strength objective. The alternatives generated are shown in Figure 3.

Action: Refine loadpath spatial-clearance

Before state: S_1

After state: $S_2 = [(O'_a, O'_s, (), O'_{ra}, ()), (((), (), A'_{ne})]$

where

$$O'_a = (\text{constructability, cost, stability, strength})$$

$$O'_s = (\text{spatial-clearance})$$

$$O'_{ra} = (\text{loadpath})$$

$$A'_{ne} = (1,2,3,4)$$

The evaluate operation invokes operators that check whether the current alternatives satisfy the objectives that are in focus. For some objectives, an alternative either satisfies the objective or it does not satisfy the objective, for example, the stability objective. For other objectives, there may be degrees of satisfaction; one alternative may satisfy an objective more than others. For example, in the case of the objective to minimize cost - more costly alternatives may not be eliminated on the basis of cost alone because they may have other important attributes. The degree of satisfaction could be expressed as an ordering of the alternatives with respect to the objective. For example, evaluate $(O_2 O_1)$ (A_1, A_2, A_3) could result in $(A_2 > A_3 > A_1 \text{ w.r.t } O_1)$ and $(A_3 > A_2 > A_1 \text{ w.r.t } O_2)$.

Action: Evaluate $O_f A_{ne}$

Before State: $[(O_a, O_s, O_f, O_{ra}, O_i), (A_c, A_e, A_{ne})]$.

After State: $[(O'_a, O'_s, O'_f, O'_{ra}, O_i), (A'_c, A'_e, A'_{ne})]$.

Let $O_f = O_{fs} \cup O_{fa}$

where

O_{fs} is the set of objectives satisfied by all alternatives

O_{fa} be the set of objectives not satisfied by any of the alternatives

Let $O_{ras} \subset O_{ra}$ be the set of objectives satisfied

because of objective satisfied in this action (O_{fs}).

then

$$O'_{ra} = O_{ra} - O_{ras}$$

O'_f is empty.

$$O'_a = O_a \cup O_{fa}$$

$$O'_s = O_s \cup O_{fs}$$

A'_e = alternatives that fail to satisfy some objective

A'_{ne} contains the ordered set of alternatives

$$A'_c = ()$$

In our example, no objectives are available to evaluate the alternatives at this stage. In other words, all of the alternatives are equally viable.

The select action uses knowledge about the relationship between the objectives to choose an alternative to become the current alternative. If all of the alternatives are eliminated, that is, A_{ne} is empty, the resolve action is invoked. The objectives in focus are a subset of the set of objectives used in evaluation, i.e. those objectives with degrees of satisfaction. A special case may arise, when no objectives can be identified to evaluate the alternatives or it is not possible to state a preference for one alternative over another; i.e. the evaluation identifies more than one alternative as being equally viable as candidates for refinement. In such cases an arbitrary choice may be made. Such a choice is made by using the arbitrary objective in the select action.

Action: Select $O_f A_{ne}$

Before State: $[(O_a, O_s, O_f, O_{ra}, O_i), (A_c, A_e, A_{ne})]$.

After State: $[(O_a, O_s, O'_f, O_{ra}, O_i), (A'_c, A_e, A'_{ne})]$.

where

$$O'_f = ()$$

A'_c contains the current alternative.

$$A'_{ne} = A_{ne} - A'_c$$

Going back to our example, we may arbitrarily select one of the four alternatives available in state S_2 resulting in state S_3 .

Action: Select arbitrary (1,2,3,4)

Before State: S_2

After State: $S_3 = [(O_a, O_s, (), O_{ra}, ()), (A'_c, (), A'_{ne})]$

where

O_a = (constructability, cost, stability, strength)

O_s = (spatial-clearance)

O_{ra} = (loadpath)

$A'_{ne} = (2,3,4)$

$A'_c = (1)$

Evaluate and select activities must solve a multiple-objective decision problem. The problem may be stated as follows: A decision must be made. The possible

alternatives are $A_1, A_2, A_3, \dots, A_m$. Each alternative can be evaluated on the basis of a set of objectives, O_1, \dots, O_n . Suppose the result of evaluation of A_i on objective O_j is given by x_{ij} for $i=1..m$ and $j=1..n$. Thus for each alternative A_i we can determine a vector $x_i = (x_{i1}, x_{i2}, \dots, x_{ij}, \dots, x_{in})$. Selecting the best alternative involves comparing all such n-tuples. Several issues arise in solving such problems: 1) a uniform measure is needed to compare the different objectives (e.g. cost vs aesthetics), 2) the result of some evaluations may have probabilities or uncertainties associated with them, and 3) trade-offs must be made among objectives given the relative importance of objectives. Many techniques (Bell et al., 1977) are available that address the problem of multiple objective decision making.

The resolve operation is invoked when the evaluate operation results in the elimination of all the alternatives, i.e. A_{ne} is empty. Resolution may be handled in one of the following ways: 1) by relaxing or ignoring one of the objectives, or 2) by modifying the value of one or more design variables. The decision regarding which option to use and the choices within each option (i.e. which objective or variable to modify) is made by the designer. We do not have an automated mechanism for performing resolution at this time; however, the system aids the designer in identifying options and identifying objectives that may be affected by these decisions. Changing the value of a variable can violate other objectives. These violations can be identified using the recorded information. In resolving conflicts among objectives we need to recognize two situations: 1) when a cycle exists and 2) when no feasible solution exists for the set of objectives. While it is possible to identify cycles from the information in the record, it is much more difficult to recognize when the set of objectives has no solution. We are currently investigating ways of using the intent information in the record to identify and help resolve interactions among design objectives.

Action: Resolve $O_f A_e$

Before State: $[(O_a, O_s, O_f, O_{ra}, O_i), (A_c, A_e, A_{ne})]$

After State: $[(O_a, O'_s, O'_f, O'_{ra}, O'_i), (A'_c, A'_e, A_{ne})]$.

Let $O_{ras} \subset O_{ra}$ be the set of objectives satisfied because of objective satisfied in this action (O_f).

then

$$O'_{ra} = O_{ra} - O_{ras}$$

O'_f is empty.

$$O'_s = O_f \cup O_s$$

O'_i contains any objective that was ignored during resolve.

A'_c contains the current alternative.

$$A'_e = A_e - A_c$$

The design process may be viewed as a cycle of refining-evaluating-selecting or resolving operations. The following description of the process is not an algorithm, but rather a high level description of the actions in the design process. While there are active objectives in the design (O_a is not empty) do the following:

- From the set of active objectives (O_a) identify a subset (O_f) for refinement.
- Refine $O_f A_c$ (current alternative; for the initial state of the design this will be just the root).

- From the set of active objectives (O_a) identify a subset (O_f) for evaluation with respect to alternatives (A_{ne}).
- Evaluate $O_f A_{ne}$
- If A_{ne} is not empty, then
Select $O_f A_{ne}$
where the set O_f will be the set of objectives with respect to which alternatives A_{ne} were evaluated
Else Resolve $O_f A_e$
where the set O_f consists of the set of objectives used in the refinement and evaluation steps.

For each action above, the specific operators that are invoked depend on the objectives and decisions that were in focus when the action occurred. When objectives are expressed quantitatively, it may be possible to use optimization techniques or constraint satisfaction methods (Borning, 1979), (Freeman-Benson et al., 1990), (Krishnan et al., 1990), (Serrano, 1987), (Sussman and Steele, 1980).

THE DESIGN RECORD

The design process is documented in the design record. The purpose of the record is to maintain the history of the design process. The record can be used to answer questions about the design. The information in the record includes the high-level objectives that were considered, how they were refined, which objectives were considered before others and why, interactions between objectives, and how interactions were resolved to arrive at the final design.

The design process is recorded as a sequence of states. The transition from one state to another is caused by one of the following activities: focus, refine, evaluate, and resolve. For each activity, the operator invoked, any assumptions made in the application of the operator, e.g. self-weight of element=0.5 kips, and the dependencies, e.g. span of element is used to determine the material and shape, created by the operator are recorded. The record for the example that was considered in the earlier section is shown in Figure 6.

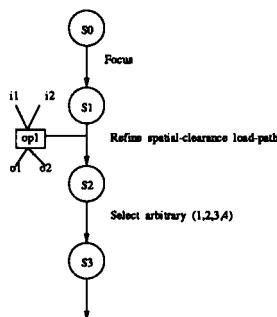


Figure 6: The Design Record

A number of operations may be supported by the information in the record. Changes in the values of design decisions and objectives can be propagated through the activity network to determine their effect on the design. Since the activities establish relationships between design decisions and objectives, decisions and objectives that are affected by changes in the values of other decisions or objectives can be identified.

The record can be used to explain the derivation history of a design decision. For example, how was the decision to use A242 steel made? At a certain point in the history, the user can change a design decision and explore an alternative solution to the design.

SUMMARY

We have presented a model of the design process in which the motivation behind design decisions are explicitly represented. When changes are made to a design, the motivation behind the original decisions must be known, so that the validity of the modified design may be verified. We call this information design intent. Our goal is to represent and record design intent.

Our design model consists of three basic entities: objectives, decisions and operators. We distinguish between five kinds of activities in our design model:

- (a) determining design focus
- (b) refining a design decision or objective
- (c) evaluating design alternatives with respect to objectives
- (d) selecting an alternative based on the results of evaluations
- (e) resolving interactions among various design objectives

All of these activities are performed by operators. Operators are invoked when specific combinations of decision and objective descriptions are present in the design state, which is the current set of decisions and objectives. In our approach, objectives may be refined into sub-objectives or they may refined into design decisions. The sub-objectives are specializations of the higher level objectives based on the current design context. The mapping from high-level concepts to design decisions through the process of refinement allows an incremental approach to satisfying the objectives, guided by high-level design knowledge. The design is complete when a level of detail sufficient to construct or manufacture the artifact has been reached and when the current set of objectives are met. The record of the design process may be viewed as a tree whose nodes represent the design state and whose arcs represent the decision-making activities that change the design state.

We are currently evaluating the model using design examples from different domains. We are investigating the representation of objectives and operators in a computer based environment. We are also looking at how the recorded information on high-level objectives can aid in resolving conflicts.

ACKNOWLEDGMENTS

This work has been sponsored in part by the National Science Foundation under the Engineering Research Centers Program, Grant CDR-8522616.

REFERENCES

- Bell, D., Keeney, R., and Raiffa, H., *Conflicting Objectives in Decisions*, John Wiley, New York, 1977.
- Bloom, H. M., "The Role of the National Institute of Standards and Technology as it Relates to Product Driven Engineering", no. NIST IR-89 4078, Washington, DC, 1989.
- Bobrow, D., *Qualitative Reasoning About Physical Systems*, MIT Press, Boston, 1985.
- Borning, A. H., "ThingLab- A Constraint Oriented Simulation Laboratory", Xerox Palo Alto Research Center, 1979.
- Fenves, S. J. and Baker, N. C., "Spatial and Functional Representation Language for Structural Design", in: *Spatial and Functional Representation Language for Structural Design*, by S. J. Fenves and N. C. Baker, edited by J. Gero, *IFIP 5.2*, Elsevier Science (North-Holland), 1987.
- Freeman-Benson, B. N., Maloney, J., and Borning, A., "An Incremental Constraint Satisfaction Algorithm", *Communications of the ACM*, vol. 33, no. 1, 1990.
- Garrett, J., *A Standards Processing Expert System*, Ph.D. Thesis, Department of Civil Engineering, Carnegie-Mellon University, May 1985.
- Katz, R. H., "Towards a Unified Framework for Version Modeling in Engineering Databases", *ACM Computing Surveys*, vol. 22, no. 4, 1990.
- Kolodner, J., Simpson, R., and Sycara, K., "A Process Model of Case-Based Reasoning in Problem Solving", in: *Proceedings of the International Joint Conference on Artificial Intelligence*, 1985, pp. 284 – 290.
- Krishnan, V., Navinchandra, D., Rane, P., and Rinderle, J. R., "Constraint Reasoning and Planning in Concurrent Design", no. CMU-RI-TR-90-03, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.
- Lipsett, R., Marschner, E., and Shahdad, M., "VHDL - The Language", *IEEE Design and Test*, April 1986.
- Mitchell, T., Steinberg, L., and Kedar-Cabelli, S., "An Intelligent Aid for Circuit Redesign", in: *Proceedings of National Conference on Artificial Intelligence, AAAI-83*, 1983.
- Mostow, J., "Design by Derivational Analogy: Issues in the Automated Replay of Design Plans", Technical Report, Department of Computer Science, Rutgers University, New Brunswick, NJ, 1987.
- Pahl, G. and Beitz, W., *Engineering Design*, The Design Council, Springer-Verlag, London, 1984.
- Rossignac, J., Borrel, P., and Nackman, L., "Interactive Design with Sequences of Parameterized Transformations", in: *Interactive Design with Sequences of Parameterized Transformations*, by J. Rossignac, P. Borrel, and L. Nackman, *Eurographics Seminar Book*, Springer-Verlag, New York, 1988.
- Schodek, D., *Structures*, Prentice-Hall, Englewood Cliffs, New Jersey, 1980.
- Serrano, D., *Constraint Management in Conceptual Design*, Department of Mechanical Engineering, Massachusetts Institute of Technology, 1987.
- Sussman, G. J. and Steele, G. L., "CONSTRAINTS- A Language for Expressing Almost Hierarchical Constraints", *Artificial Intelligence*, 1980, pp. 1-39.
- Swartout, W., "A System for Creating and Explaining Expert Consulting Systems", *Artificial Intelligence*, vol. 21, no. 3, 1983.
- Thompson, J. and Lu, S., "Design Evolution Management: A Methodology for Rep-

resenting and Using Design Rationale", in: *Proceedings of the Second International ASME Conference on Design Theory and Methodology*, 1990.

Wile, D., "Program Developments: Formal Explanations of Implementations", *Communications of the ACM*, vol. 26, no. 11, 1980.

A knowledge-based approach to the automatic verification of designs from CAD databases

M. Balachandran, M. A. Rosenman and J. S. Gero

Design Computing Unit
Department of Architectural and Design Science
University of Sydney NSW 2006 Australia

Abstract. This paper commences by discussing some of the difficulties involved in the process of automatic verification of designs from CAD databases. The main focus of the paper is to describe and demonstrate an approach to the development of a knowledge-based integrated system which is capable of verifying a design through its description in a CAD database for conformance with some specified requirements. IPEXCAD, an early version of such an integrated environment, is described in terms of its architecture, implementation and operation issues. The system is illustrated by an application in building design.

INTRODUCTION

In the various disciplines of design, such as engineering, architecture, drawings are the medium through which decisions are communicated. A designer uses drawings to represent symbolically designed objects and their configurations in space. Furthermore, graphical representations are used to illustrate the different alternatives available at any stage of a decision making process and to aid in the evaluation of the outcome of these decisions. Commercially available computer-aided drafting (CAD) packages are widely used by design professionals. Such computer-aided drafting packages provide designers with powerful and flexible facilities for creating and modifying a graphical representation of their design descriptions. A CAD system stores a design description produced by a designer in its database in terms of drawing elements. Such a database only contains syntactic relationships among those drawing elements. Therefore, one major limitation of such CAD systems is that they do not possess the capabilities to interpret the contents of a drawing as human designers do.

The verification of a design is the process of checking the completeness, consistency and correctness of the design against given sets of requirements. Such requirements may be of many different types, such as those given by design codes and may come in the form of clauses, heuristic rules, tables, diagrams, graphs and mathematical equations. Verification of a design from its computer representation in the form of a CAD database is a knowledge

intensive task which requires semantic interpretation of the CAD database and the representation and manipulation of the design verification knowledge. Advances in artificial intelligence have led to the emergence of expert systems which explicitly represent the knowledge of human experts in some specific domain and apply this expertise to the problem solving process (Bobrow et al., 1986; Waterman, 1986). Expert systems are capable of representing and using the design verification knowledge in an explicit manner. The semantic interpretation of a CAD database is the process of transforming the syntactic relationships in the CAD database in accordance with the semantic relationships in the expert system's knowledge base.

One major problem involved in this approach is the mapping of syntactic information stored in the CAD system's database to the semantic information contained in the expert system's knowledge base. The syntax to semantics mapping of a CAD database is a tedious and domain dependant task which requires extensive domain knowledge. We use the notion of design prototypes to represent the domain knowledge as a set of abstractions which are used by the expert system for conceptual knowledge and by the CAD database interpreter to search for information from the drawing database. Therefore in order to verify a design from its graphical representation, a system must be able to perform interpretations and knowledge processing tasks.

Based on the needs and the nature of the design verification process, a conceptual architecture of an integrated knowledge-based CAD environment has been developed. The remainder of this paper describes the motivation, issues and design of the system with an emphasis on the problems encountered and techniques used.

BACKGROUND

Interactive computer graphics has attained considerable importance in computer-aided design and manufacture. Designers make their decisions with the help of computer presentations in the forms of perspectives, plans, elevations, and so on. The task of preparing and modifying such drawings has been considerably simplified by various packages commercially available for computer-aided drafting. However, the major drawback with most such systems lies in the interpretation of the graphical entities created by the designer or their incorporation into other stages of the design process. Currently, in most systems, the mapping between the graphical representation and the designed artefact is made by a human designer. In such systems this is partly because the systems have no representation of the artefact created by the user other than its graphical manifestation on the screen and its syntactic model in a data structure in the database of the system. In order to expand the effectiveness of the relationship between the user and the computer, we need to incorporate knowledge about the underlying artefact so that the system can use that knowledge to derive meanings of design elements represented graphically. For instance, in building design, a building plan is graphically represented by a series of straight lines and some symbols. If the knowledge required to interpret that set of lines and symbols as walls, windows, doors, etc. were represented in the system, the system could enforce other properties such as topology and geometry during any operation.

In the past significant efforts have been expended in developing integrated computer

systems for design applications. For example Lee (1977) describes an integrated system, called ARK-2, for architectural design. Hoskins (1977) presents an integrated interactive computer-aided design system, called OXSYS, for building design. More recently many design researchers have carried out work aimed at developing integrated systems for design applications utilising knowledge-based approaches. Rehak et al. (1985) and Fenves et al. (1990) present some approaches to developing integrated knowledge-based software for design applications. Jain and Maher (1987) discuss a number of possible ways of combining expert systems and computer-aided drafting techniques. Dym et al. (1988) presents a knowledge-based system for checking the Life Safety Code of the fire regulations. Tyugu (1987) describes the importance of merging conceptual and expert knowledge in CAD. Balachandran and Gero (1988) describe a model for building knowledge-based graphical interfaces.

So far, existing integrated systems have been implemented using a tightly-coupled approach (Hoskins, 1977; Jain and Maher, 1987). That is, the systems know exactly what entities to expect from the graphics and the graphics system is tailored exactly to the knowledge in the system. Rehak and Howard (1985) put forward KADBASE as an approach to the integration of a distributed CAD system containing a variety of heterogeneous systems and design databases. This approach of advocating a flexible interface in which multiple expert systems and a variety of design databases communicate within an integrated system is the one pursued in this paper.

Current CAD Systems

Today's CAD systems can be broadly classified into three categories, namely general drafting systems, general modeling systems and domain-specific modeling systems. Drafting systems merely produce a graphic representation of whatever views of the design are drawn. There is no relation between elements in one view and those in another view. General modeling systems allow for the modeling of elements and assemblies of elements. The graphical representations are merely views of the one model. Information regarding the model can be entered or modified via any view (although some systems restrict this to the plan view only). Elements are created through drawings or macro-languages and are accessed through name-labels attached to them. In some systems other non-graphical properties, such as material, colour, and catalogue numbers may be associated with an element. In domain-specific systems the system has knowledge about a variety of types of elements within its domain as libraries of elements with various attributes. These elements may be of fixed geometry or parametric in nature.

Expert Systems in Design

In the past decade several expert systems have been demonstrated for design applications (Kostem and Maher, 1986; Gero, 1987a; Rosenman, 1990). While the advantages of expert systems are well documented for symbolic processing in logical inferences, they are not well suited to the computational processing involved in mathematics or graphics. They are also criticised for their shallowness and inability to reason beyond very limited scope. An important area of application of expert systems in design is that of engineering and building codes (Rosenman et al., 1986a; Garrett and Fenves, 1987; Sharpe et al., 1989). Complex

design codes are ideal application areas for expert systems since both users and developers often have difficulties with ensuring correct interpretations.

THE VERIFICATION TASK

Let us have a look at the problems, processes and knowledge involved in the verification task. In order to understand this task better we first describe the main properties of designed artefacts, namely function, behaviour and structure properties.

Function, Behaviour, Structure

A designed artefact may be described in terms of its function, behaviour, and structure. the structure description is a description of what the artefact is; the behaviour description, a description of what it does and how it does what it does; and the function description, a description of what it is for, i.e. its purpose. For example, the structure of a clock includes the parts from which it is made, their material, shape, dimensions, how they are joined, etc.; its behaviour (if it is an analog clock) is that its hands rotate with a periodicity matching the elapsing of time, pointing to marks on the dial to display this; and its function is to mark the time.

Structure properties describe the form and physical properties of an artefact in terms of its geometrical and physical attributes, such as shape, dimensions, material, colour, etc. The behaviour properties of an artefact are emergent properties of the artefact given its existence and can be directly derived from its structure properties given that we have the necessary knowledge regarding these relationships. The function of an object, however, is a human determined property related to some human needs. There is no clear link from behaviour to function except through a human interpretation. Having said that, design is about the fulfilment of required functions through the creation of structures which exhibit behaviours which are recognized to be beneficial in producing these functions.

Not only are these function, behaviour and structure attributes important in the description of designs, but also their relationships. Relationships exist between function and behaviour, between behaviour and structure, between function and structure and among the attributes themselves.

Design Analysis and Evaluation

Artefacts, through their existence, exhibit a multitude of behaviours, not all of which are relevant to some particular context. For example, we are not generally interested in the electrical resistance of a pencil or the fact that it makes a sound when we strike another object. However, we may indeed be interested in the latter behaviour in the context of tapping out a message. The selection of which behaviours to focus on depends on the context defined by the function to be satisfied.

Given that we are operating in a certain functional context we will be interested in certain behaviours of any proposed design and that these behaviours achieve satisfactory levels of performance. These required levels of performance of the behaviours may be derived from

the function, if we have such relationships or else they may be directly prescribed by codes, rule of design, etc. The process of interpreting a design description to derive levels of performance of behaviour is the process of *design analysis*. The process of comparing these derived performance levels to the required levels is the process of *design evaluation* while the process of determining the cause of an unsatisfactory behaviour is the process of *diagnosis* and the process in which structures are derived from behaviours and functions is the process of *design synthesis*. Design verification includes the processes of design analysis and design evaluation and, this paper will concentrate on these processes as we are interested in evaluating the performance of a proposed design whose description is given by modeling it graphically using a CAD system.

The Verification Task—An Example

Let us have a look at the sort of verification task we may want to do. As an example let us take a verification task dealing with compliance with a building code. The building code in question is the Building Code of Australia, referred to as the BCA Code (AUBRCC, 1988). We wish to ensure that a design, described graphically using a CAD system, complies with the provisions of this code. Let us take an excerpt from this code, namely some of the provisions dealing with masonry construction. Following is an excerpt from the BCA Code itself:

B2.3 MASONRY CONSTRUCTION

- B2.3(1) **External wall thickness: Interpretation** - For the purposes of this clause the combined thickness of the inner and outer leaves of a cavity wall shall be deemed to be the thickness of the wall.
- B2.3(2) **Minimum thickness of external walls** - The external walls of a building, if of masonry construction, shall be not less than 200 mm thick, except in the case of:
- a Class 7 or 8 building;
 - a single storey building or the topmost storey of a multi-storey building where cavity wall construction is used and the combined thickness of the inner and outer leaves is not less than 190 mm;
 - a Class 10 building or garage, laundry, tool shed, closet or the like forming part of a building of another Class; or
 - a building subject to subclause B2.3(7).

As we can see, Clause B2.3 applies to masonry construction; Subclause B2.3(1) is an interpretive provision explaining how to arrive at the thickness of cavity walls (double walls); while Subclause B2.3(2) deals with the required minimum thickness of external walls. Except for specified situations this minimum thickness is required to be 200 mm.

Having a design of a building, our verification task consists of checking whether the above provisions apply, and if so whether our building complies. That is we must first check that our building contains elements of masonry construction, then Clause B2.3 is applicable; that our external walls are of masonry construction, then Subclause B2.3(2) is applicable and

finally that all such external walls comply with Subclause B2.3(2).

The first problem that arises after having provided a graphical description of our design (and having determined that compliance with the BCA Code is required) is to determine which provisions of this code are actually applicable. This depends on the nature of the design. For example, a one storey building does not have stairs and therefore all provisions dealing with stairs are not applicable. We could thus start with the elements of the design and search the BCA Code for all relevant provisions. This entails three problems. The first problem is that of recognizing the elements of the design from the graphic database; the second problem is that of matching the description derived from the database to the descriptions used in the code; and the third problem is that of interpreting the BCA Code to ascertain the relevancies of certain references in the BCA Code to the elements found. For example, the above provisions deal with masonry construction. Most probably none of the elements interpreted from the CAD database will be identified as of 'masonry construction' but perhaps of brick, concrete, etc. In order to determine that Clause B2.3 is in fact applicable, there will have to be some recognition, that, a wall of brick or a floor of concrete does constitute masonry construction. So that, if we were to start from the elements themselves as obtained from the CAD database we could not directly identify the applicable provisions in the BCA Code.

Alternatively, we could start with the provisions of the BCA Code and determine their applicability. This would mean interpreting each provision so as to determine its relevancy vis-a-vis the elements derived from the CAD database. Because of the structural organization of the BCA Code (and most such codes) it is not necessary to examine exhaustively every provision but an implicit enumeration approach makes it possible to prune those portions of the BCA Code not relevant to the situation at hand. For example, since the BCA Code is structured into Parts; each Part into Clauses; and each Clause into Subclauses it will be possible to determine firstly if a Part is applicable. If not, then there is no need to investigate any of its provisions. Similarly if a Clause is found not to be applicable then none of its Subclauses need be investigated.

We do not want to design a system in which the description of a designed artefact is tailored to a particular description used in any particular application program. Therefore, the representations and the descriptions derived from a CAD system will be independent of the various application programs or knowledge systems which are needed to carry out the verification task. We will assume, however, a common domain, the domain of building designs, for example, so that both the descriptions in the CAD system and in the verification system refer to the same domain. We will continue using the BCA Code as an example of a verification application. It is obvious that both the descriptions of elements in the CAD system and those in the BCA Code are based on a much wider domain knowledge than the information stated. There are many assumptions made in the BCA Code about the relationships of building elements to each other, e.g. that columns are structural elements supporting such elements as beams, etc; about how to determine various information required, e.g. distance between exits. It is assumed that users of the BCA Code have a good deal of understanding of buildings. Similarly, when we interpret drawings we bring to bear this wide domain knowledge, e.g. walls connect to each other, they enclose spaces (together with ceilings and floors); doors and windows are located in or between walls, they form openings in walls, etc.

Therefore, an essential part of any integrated system for the verification of designs from

CAD databases will have to incorporate a domain knowledge base wherein knowledge exists about the elements in the domain and the relationships between these domains. It is not sufficient to include descriptions which are concerned only with the physical and geometrical aspects of these elements, the descriptions must include functional and behavioural aspects since a verification task involves the evaluation of the performance of designs. This domain knowledge base, while recognizing that CAD systems and verification applications will refer to it, needs to be independent of the particular CAD system and application program. It requires a representation suitable for the description of this domain knowledge in a generic fashion so that it can be useful in a wide variety of situations. However, in order to effect the necessary communications between the CAD database and this domain knowledge, there will be a need for a graphic database interpreter. Similarly, in order to effect the necessary communications between the verification program and this domain knowledge, there will be a need for an interpreter, which in the case of verification programs in the form of expert systems will be a knowledge base interpreter. Both these interpreters will, of necessity, be specific to the systems being used.

Interpretation

Graphical images are usually interpreted semantically by humans without any concern for the process used to produce the image. One of the main difficulties in using traditional CAD systems is that they store their models syntactically in the form of data and procedures. However, as was shown in the example above, verification programs deal in terms which require interpretation of the syntax produced by the CAD system. It means that a computable semantic model of the screen graphics needs to be generated in order to be used in other design activities. Such syntax to semantics transformations are fundamental to any automated design verification system using graphical representations.

Syntax and semantics in design

Syntax is the vocabulary and rules governing the composition or structure of such vocabulary. Semantics is the meaning derived from some syntactical description. The vocabulary varies according to the domain and the application. For literature, the syntax consists of words and the rules of grammar. The semantics consists of the ideas expressed by this composition. The collection of these ideas is called a 'book'. At a different level, the syntax consists of books and the rules governing their arrangement. The semantics is the interpretation of this collection as a 'library'.

Design is also concerned with vocabulary elements, such as gears, rods, transistors, struts and windows and how these elements are put together (Coyne et al., 1990). Design descriptions are in terms of syntax. The interpretation of design descriptions is an issue of semantics. Semantics is concerned with derived descriptions, either in terms of conceptual identification or in terms of performance. For example, a design description of four walls, a floor and a ceiling arranged in a given structure is interpreted to mean a 'room'; this same design description (given dimensions) can also yield such interpretations as: the area of the room is such and such; the size of the room is large; and the proportions are pleasing. The notions of syntax and semantics are not absolute. What is semantics at one level is syntax at another. For example in a geometrical context, given a syntax comprising points and their

locations, the semantic interpretation may be that of a set of lines. Taking the lines and their arrangement as syntax, at a higher level of description, the semantic interpretation may be that of some shape, e.g. a rectangle.

Type of knowledge inside a CAD system: syntax

Drawings are graphical representations of design descriptions usually representing only topological and geometrical properties of a design although other properties may be attached to the elements of the design. Drawings on paper are merely marks on paper while on a computer screen they are merely an arrangement of pixels. Drawings are therefore syntactical descriptions of a design. Interpretations of the meaning of such drawings is done by the human observer. Certain lines mean elements such as walls or doors; groupings of these elements mean rooms, spaces, etc.; groupings of these spaces mean buildings. Working drawings specify the type of elements, their material, dimensions and location. This is all that is necessary for contractors (or CAM systems) to assemble the artefact. There is no need for them to know about the intent of the designer as to the function of the artefact or the sort of effects the designers are trying to create as long as the syntactic descriptions are complete and unambiguous. Of course when this is not so a knowledge of intent can help resolve such problems.

As mentioned previously, there are three general types of CAD systems: drafting systems; modeling systems and domain specific modeling systems. Drafting systems, such as AutoCAD (Autodesk, 1988), store information as vectors of points or lines and/or may have a limited capability for representing 2D shapes and creating symbols to which element labels can be attached. There is no relationship between different views of the same object and such relationships have to be computed if required. A modeling system, such as EAGLE (Carbs, 1985), allows for the modeling of objects through graphic (and textual) input. These objects are identified by labels or names at the time of modeling. The modeling system stores information about objects—the database is organized as a series of objects indexed by their name and possessing certain properties, such as geometric and labels as to material. Drawings can be generated from any view and relate to the same object or set of objects. Elements such as walls which are represented by parallel lines at certain fixed distances apart but which may vary in length (and height) have to be interpreted as such. Elements such as rooms and other spaces must also be interpreted as these are not modeled explicitly but 'emerge' from the arrangement of their constituent elements. A domain-specific modeling system, such as ArchiCAD (Graphisoft, 1989), provides menus of parametric elements (such as walls, roofs, floors) about which the system has certain knowledge regarding their topology and representation. Users select the appropriate symbol representing the required element and locate it by defining its position as well as its dimensions in three dimensions. These elements usually come with certain default values for various properties, such as thickness, material and representation. Users may change these default values at any time and these values are then associated with all instances of that element. In addition, there usually exist libraries of domain elements, such as doors, windows, fixtures and fittings as well as the ability for users to create their own.

CAD databases usually have special formats that could be read and used by other computer systems. Two of the most common such file formats are:

- (a) DXF (Drawing eXchange Format) and
- (b) IGES (Initial Graphics Exchange Standard)

A DXF or IGES file contains both graphical and non-graphical information about a drawing formatted to be easily read by the user or a computer. Almost every CAD system provides some sort of facilities for reading and writing DXF or IGES files. An example of a DXF representation of an object inserted into a drawing is shown below. The indented numbers 0, 8, 2, etc. are some of the group code numbers used by the DXF format.

0		0	
INSERT	<i>Entity type</i>	ATTRIB	<i>Entity type</i>
8		1	
FLOOR	<i>Layer name</i>	JOHN	<i>The value of the attribute</i>
2		2	
ROOM	<i>Block name</i>	OCCUPANT	<i>The name of the attribute</i>
10		...	
5.0	<i>The X coordinate of the insertion</i>		
20			
10.0	<i>The Y coordinate of the insertion</i>		

Type of knowledge inside verification applications: semantics

The knowledge in verification applications, as was shown in the above example drawn from the BCA Code, deals with required behaviours of elements in various situations. That is, the knowledge deals with semantic information and requires the interpretation of the current situation so as to match the intentions specified in the Code. For example, the meaning of the term 'exit' is that of a doorway leading to a road or open space or a pathway leading from spaces in the building to the exterior through passageways, stairs or ramps. This needs to be interpreted from the available syntax.

DEVELOPMENT OF A SYSTEM FOR THE AUTOMATIC VERIFICATION OF DESIGNS FROM CAD DATABASES

Domain Knowledge—Design Prototypes

Knowledge about a particular domain needs to be represented in a generic way so that many situations pertaining to that domain can relate to this knowledge. The type of knowledge required is knowledge about artefacts within the domain from the point of view of their structure, behaviour, function and relationships between these and also to other artefacts. Such knowledge provides a deep structure useful for reasoning within the domain. This knowledge should be sufficiently comprehensive to allow for a wide variety of situations which may arise within the particular domain and for a variety of verification tasks which may be deemed necessary. Furthermore, this knowledge must be able to be augmented or modified as the need arises since it is practically impossible to determine every need that may

arise except for a limited set of applications. In general, all the objects that need to be described using the CAD system must exist in the domain knowledge for meaningful instances to be created which will be capable of being evaluated.

Previous systems for representing global domain knowledge have represented this knowledge using frames (Rehak and Howard, 1985; Dym et al., 1988). In the CODE system (Rosenman et al., 1986b), the model of the artifact described was extracted from the rule base of an expert system dealing with a building code and represented as frames. The EAGLE system was used to describe graphically an instance of a building and the elements thus described were modeled as instances of the model frames. The disadvantages in frames is that all slots representing attributes are equivalent in type. There is no capability to distinguish between the structure, behaviour and function properties of an object. Moreover, the frames describe only the attributes themselves and not the knowledge regarding the relationships between these attributes.

Design prototypes have been proposed as a means of representing design knowledge comprehensively (Gero, 1987b; Gero and Rosenman, 1989). Design prototypes represent a class of design elements and embody all the knowledge necessary to produce a particular instance of the class or to evaluate the performance of a given instance. Design prototypes are structured along function, behaviour and structure properties as well as relationships between these. The design prototype schema allows the syntactic and semantic knowledge about the above classes of properties and how they relate to each other to be represented and manipulated efficiently. Figure 1 shows the model of a design prototype. The knowledge in a design prototype allows the derivation of values for structure and behaviour attributes, and the interpretation of structure descriptions to derive behaviour and function properties. In addition, there exists knowledge about the relationships to other design prototypes, and knowledge about function and structure constraints.

Design prototypes may be at various levels of abstraction and may be related to each other by typological or structure properties. For example, a design prototype may have links 'a_type_of', 'an_element_of', etc. with some other design prototype, thus forming a hierarchy of design prototypes both from a taxonomic and an elemental view. Instances may inherit properties from other more generic design prototypes through appropriate links. The prototype base contains all the prototypes deemed necessary for a given domain. This prototype base must be able to be augmented and/or modified as necessary. A design prototype engine performs the tasks of manipulating the knowledge in the prototype base, creating an instance of a specific design prototype and deriving values to specific variables of the instances as needed.

It has been shown that an expert system based on such design prototypes has the capability to represent experience as a set of general concepts in which semantic and syntactical relationships are explicitly defined (Rosenman et al., 1989). This can form the basis of an expert system where different knowledge bases dealing with these concepts can be implemented to interpret structure descriptions to derive behaviour, to evaluate performances or to derive structure descriptions from function descriptions. Moreover, this representation allows for other applications, as well as graphical modeling systems to provide and receive information which can be interpreted as necessary.

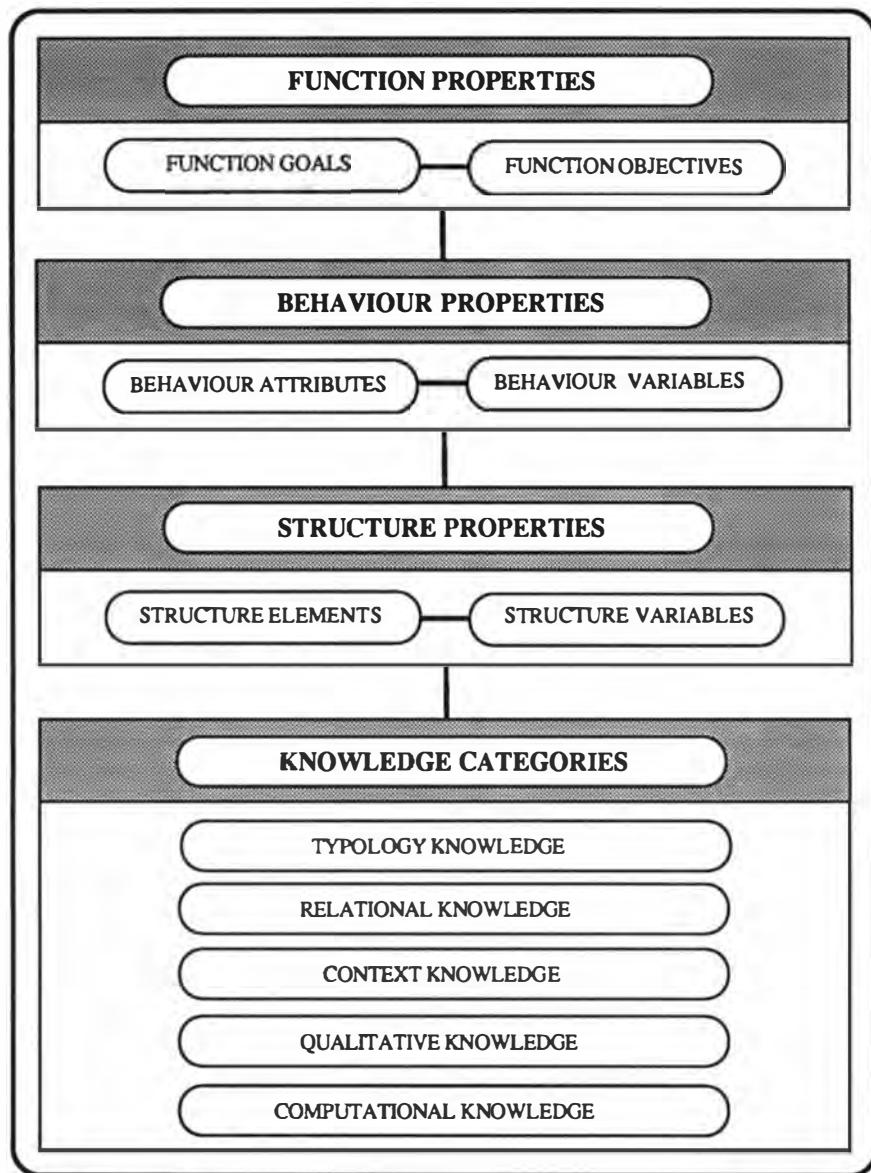


Figure 1. The model of a design prototype

Verification Knowledge—Production Rules and Procedures

The verification tasks may be quite varied depending on the domain, the particular design and also the design context. Moreover, the verification knowledge may be in different forms, e.g. procedures, tables, rules of thumb, prescriptive clauses etc. Rule-based expert systems have shown themselves especially capable of modeling situational and causal knowledge in a form which is easily comprehensible and relatively easy to formulate and maintain. In addition, they have the capability of providing explanations during the implementation process. However, for such systems, to incorporate some meaningful descriptions of the objects in their domain, an object-attribute-value representation is necessary to enable the extraction of these descriptions. This object-attribute-value representation thus allows for the mapping between the objects and their attributes within the scope of the expert system and that of the design prototypes. This knowledge representation allows for a wide variety of verification applications to be modelled in a uniform manner.

For example some of the rules formed from the Code example of Section 3.3 are as follows:

R50	IF AND THEN	element OF building IS E type of construction OF E IS masonry construction applicability OF clause B2.B2.3 IS determined.
R60	IF AND AND AND THEN	applicability OF clause B2.3 IS determined FOR_ALL subclause OF clause B2.3 IS SCL applicability OF SCL IS determined compliance WITH SCL IS satisfactory compliance WITH clause B2.3 IS satisfactory.
R70	IF AND THEN	applicability OF clause B2.3 IS determined FOR_ANY type of construction OF external wall IS masonry construction applicability OF subclause B2.3(2) IS determined.
R80	IF AND AND AND THEN	applicability OF subclause B2.3(2) IS determined FOR_ALL type of construction OF external wall IS masonry construction NOT exemption FROM subclause B2.3(2) IS applicable thickness OF external wall >= 200 compliance WITH subclause B2.3(2) IS satisfactory.
R90	IF OR_IF THEN	classification OF building IS class 7 OR class 8 OR class 10 number of storeys OF building IS 1 exemption FROM subclause B2.3(2) IS applicable.

In general, the above rules take the form of:

IF
THEN condition₁ AND ... AND condition_m
 consequence₁ AND ... AND consequence_n

where condition; and consequence; take the form:

<attribute> <PREP> <object> <VERB> <value>

where <PREP> and <VERB> are any user defined terms

The rules are formulated as a representation of the particular verification knowledge and, as such, follow the terminology used in such applications. This may not match exactly to terms used in the graphic description or existing in the domain knowledge. To be useful, this knowledge will have to be interpreted. Further, there may be knowledge regarding some aspect of the design both in the expert system knowledge base and in the design prototype base. Since the expert system knowledge is more specific to the problem at hand it takes precedence over the general domain knowledge.

Not all knowledge can be formulated as rules. For example, calculations may be required or table lookups required. In such cases procedures will be required. The necessary information will have to be passed to these procedures and results obtained in a form compatible with the rest of the knowledge.

Knowledge Base Interpreter

The role of the knowledge base interpreter (KBI) is to interpret the rules in the knowledge base and make explicit the terms used therein. Using the object-attribute-value format the KBI is able to extract the objects, attributes and range of possible values within the scope of the expert knowledge. Each object is then represented as a frame, its attributes as slots and facets provide for descriptions of various types of values for the attributes, including in which rules and which part of the rules the attribute is to be found. The KBI notes whether matches exist between the objects found in the expert system knowledge base and those in the domain knowledge.

Not all the knowledge present in the knowledge base of the expert system can be interpreted by the KBI automatically. The knowledge engineers formulating the verification knowledge rule base will, determine matches and non-matches between the objects constructed and those in the domain knowledge. They may modify the terminology in the rules to conform with the terminology in the design prototypes or they may modify the design prototype base to incorporate new information where it is deemed appropriate. This may include modifying existing design prototypes and/or adding new design prototypes. Else, they have the option to add links between frame objects and design prototypes. For example, instead of changing the term 'stairway' it is possible to add a descriptor ' same_as' as in 'same_as: stair' to notify the system that 'stairway' and 'stair' are synonyms.

In most cases, extra interpretative knowledge is required to make the necessary linkage between the knowledge base objects and the design prototypes. This knowledge will usually include procedures for deciding which information is required to identify such associations. To illustrate this kind of necessary interpretative knowledge, let us have a look at another example taken from the BCA Code:

C3.5 TYPE A CONSTRUCTION

C3.5(1) Requirements - In a building required to be of Type A construction, each part mentioned in Table C3.5, and any beam or column incorporated in it, shall (subject to the modifications set out in this clause and clause C4.2) -

- (a) ...
- (b) have an FRL not less than that listed in the Table, for the particular Class of building concerned; and
- (c) ...

A simplified form of this knowledge formulated as rules and procedures could be as follows:

R200 IF applicability OF Part C3 IS determined
AND type of construction OF building IS type A
THEN applicability OF clause C3.5 IS determined.

R210 IF applicability OF clause C3.5 IS determined
AND FOR_ALL element OF Table C3.5 IS E
AND DO identify-element(E)
AND DO table_lookup(C3.5, E, FRL, T)
AND FRL OF E IS REQD_TO_BE T
AND FRL OF E >= T
THEN compliance OF clause C3.5 IS satisfactory.

Once clause C3.5 is applicable, i.e. we are dealing with a building of type A construction, we check every element in Table C3.5 to see if it exists in the building. If so, we look up the table to get its required FRL and compare that to its actual FRL. The 'DO' keyword in the rules makes calls to the function following. The elements in Table C3.5 are described in terms that will not match directly to the design objects in the domain and interpretation is required. A procedure to identify one of the elements mentioned in Table C3.5, e.g. 'loadbearing internal wall bounding a public corridor' would have the following form:

```
identify_element(loadbearing internal wall bounding public corridor)
  if find      instance I
    such that I is an instance of internal wall
    and      I is an instance of loadbearing wall
    and      I bounds S
    where    S is an instance of public corridor
```

The above procedure will succeed if it finds an instance with the following type of information:

wall1 instance_of: internal wall instance_of: loadbearing wall bounds: hall2 ...	hall2 instance_of: corridor ...
--	--

In addition, since the domain knowledge will not have the concept 'public corridor' this will have to be defined in the following manner:

public corridor a_type_of: corridor membership_criteria: dependent_on R43
--

where rule R43 may make a call to a procedure for determining whether an instance (of a corridor or hall) is in fact an instance of a 'public corridor'. In this case this means determining such things as whether the instance 'hall2' provides means of egress from a part of a storey to a required exit.

Knowledge Base Objects—Frames

It can be seen from the example rule base that there are two types of objects referred to in the rules. The first type are objects that refer to design objects. The frames corresponding to these objects are shown below:

building (prototype) element affects: [BCA_R50] classification options: class7, class8, class 10,... affects: [BCA_R90] category: unknown number of storeys affects: [BCA_R90]	external wall (prototype) type of construction options: masonry construction, ... affects: [BCA_R80] thickness affects: [BCA_R90]
---	---

In this case, the above objects could be reasonably expected to exist in the domain knowledge as design prototypes. When a match is found in the design prototype base this is noted by adding '(prototype)' to the object frame's name. Not all design objects mentioned in the expert knowledge base will have corresponding matches in the design prototype base. This may be because this object is specific to the expert knowledge, e.g. the object 'exit' or because of terminology used, e.g. 'stairway' as against 'stair'. It is also possible for attributes in an object matched to a design prototype not to exist in the domain knowledge as it is specific to the particular expert knowledge, e.g. the attribute 'classification' of the object 'building'. In this case the attribute is marked with the value 'unknown' in the facet

'category' as the system has no way of telling the type of such attributes, i.e. if function, behaviour or structure.

The other type of objects are objects which are not design objects but are objects to the expert system. These are also formulated as frames but, obviously, there will be no reference to any design prototype. Examples of such frames are shown below:

clause	subclause
applicability	applicability
options: determined	options: determined
affects: [BCA_R60, BCA_R70]	affects: [BCA_R80]
subclause	dependent_on:
[BCA_R70]	
affects: [BCA_R60]	
compliance	compliance
options: satisfactory	options: satisfactory
[BCA_R80]	dependent_on:
dependent_on:	[BCA_R60]

The frame part of the system makes use of the usual properties of frame systems.

Design Description—CAD Database

Here we will assume a level of information in the CAD database corresponding to objects in the design. That is, we are not concerned with the interpretation of arrangement of pure geometrical features, such as lines, to derive features or objects as is the work of Nnaji and Kang (1990). We are therefore assuming that the CAD system allows the users to create 'objects'. We have also previously stated that all objects to be described using the CAD system must exist as design prototypes. Nevertheless, the representation of such objects in the CAD database will be in a format incompatible with that of the domain knowledge and must be interpreted to produce descriptions which are meaningful to the rest of the system.

CAD Database Interpreter

In the process of integrating CAD systems with expert systems, the information presented graphically must be converted to a form that can be understood by the expert system, that is to a form commensurate with that of the design prototype format. For this purpose a CAD database interpreter (DBI) must be used. Where the CAD system database uses a standard graphic database format such as IGES or DXF or higher level format the interpreter must convert these formats to that of the design prototype format while if the CAD system database uses some non-standard format the interpreter must be written specifically for the particular CAD system used. CAD systems consist of a graphical interface and a database. The database of a CAD system contains representations of drawing elements and numeric or alphanumeric information associated with those elements. The syntactical information, namely, in the form of dimensions, locations, shapes, etc. can be mapped onto the structure properties of a design prototype. The graphic database interpreter consults the appropriate design prototypes

and converts the information in the CAD system's database to the appropriate instances of design prototypes. Where required, the design prototype will have the necessary information for recognizing elements in the CAD database and forming the appropriate instances. For example, the design prototype 'room' will have the required cells to procedures for recognizing that a 'space' with a label bounded by walls (and/or windows, partitions, etc) is a room of a certain type. Most current CAD systems provide special database utilities to access and manipulate their databases. For example, AutoCAD and EAGLE have programming languages that let users write programs to manage and manipulate both graphic and nongraphic data.

Instance Base

All processes concerning an actual entity deal with an instance of that entity. Entities are generated through their description using the CAD system. The instance base contains the instances created during the running of an application. The instance base thus constitutes the working environment of the system. Instances are descriptions of elements derived from the CAD database interpreter or of knowledge base objects derived from the knowledge base interpreter with their relevant attributes instantiated to derived values.

System Implementation

A system named IPEXCAD (Integrated Prototype-based EXpert CAD environment) for the verification of designs from CAD drawings has been implemented on SUN workstations and is described below.

One of the major concepts utilised in the development of the system is a clear separation between the generic expert system module, which performs reasoning processes, the generic domain knowledge and the CAD package, which performs drawing operations. The system consists of seven subsystems. They are:

- (a) a user interface;
- (b) a stand-alone expert system shell, EXBUILD (Balachandran and Rosenman, 1990);
- (c) a stand-alone CAD system that provides standard drafting and modelling features;
- (d) a stand-alone design prototype system, PROTOKIT;
- (e) a knowledge base interpreter which interprets EXBUILD's knowledge base and allows the knowledge engineers to provide necessary interpretations;
- (f) a graphic database interpreter which interprets queries from PROTOKIT and directs them to the CAD database; and
- (g) the working memory.

Figure 2 illustrates the system architecture diagrammatically. IPEXCAD uses a Macintosh-like interface under the Sunview window system. The user interface provides facilities for creating, modifying, displaying and saving information associated with design prototypes, instances and knowledge bases. A mouse-based text editing facility allows any text to be modified easily and conveniently. An appropriate CAD system may be invoked by selecting from a sub-menu which is displayed when clicking on the CAD icon from the main menu.

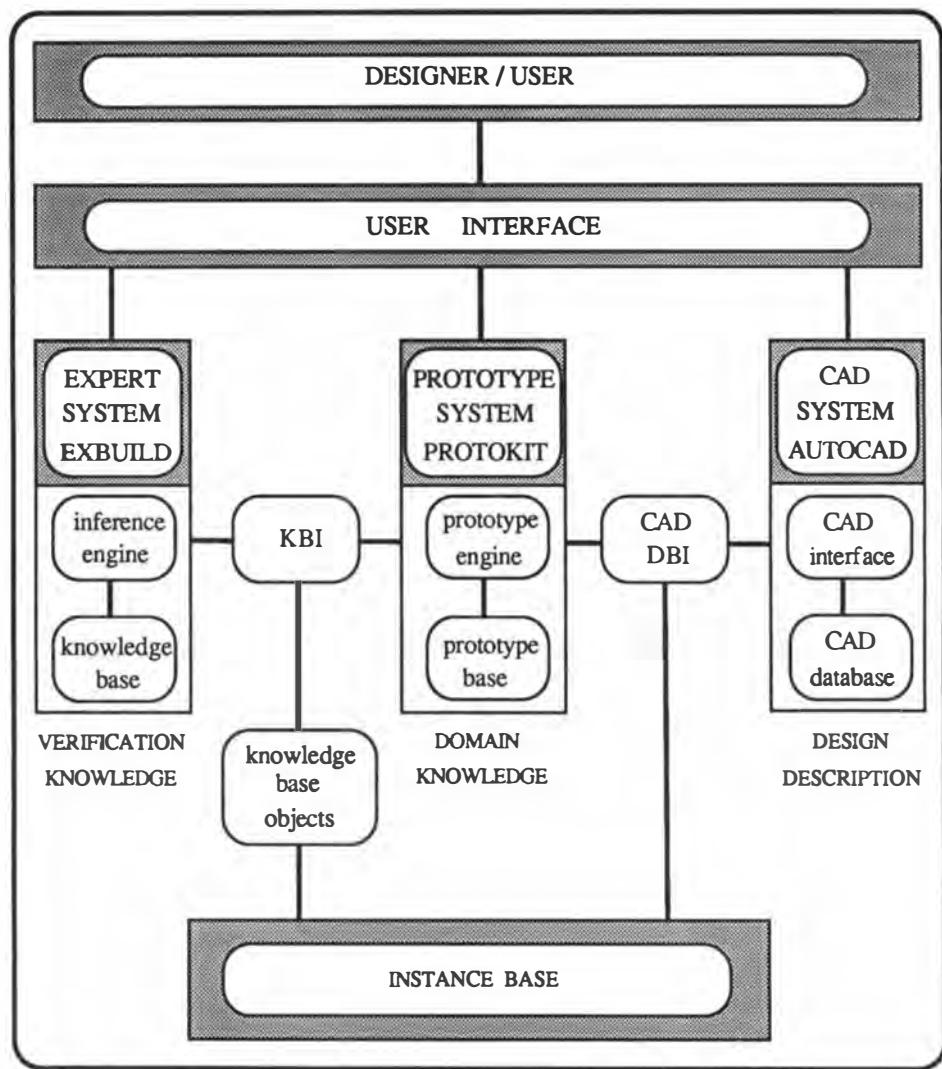


Figure 2. The system architecture of an integrated system for the verification of designs from CAD databases

The verification process is initiated by the users selecting the 'verify' icon. If no CAD database is loaded (e.g. no design description exists) the users will be asked to describe their design either by generating a new description or by loading an existing one. Similarly, if no knowledge is loaded the users will be asked to load one or more such bases. The process will then pose the necessary queries to each such knowledge base to activate the verification process. Alternatively, users may load knowledge bases and pose selective verification queries to IPEXCAD.

The expert system used is EXBUILD (Balachandran and Rosenman, 1990), a hybrid expert system development tool written in C. EXBUILD uses both rule-based and frame-based representations.

In this project we have chosen two of the popular CAD packages, namely EAGLE (Carbs Ltd, 1985) and AutoCAD (Autodesk, 1988). AutoCAD provides two ways to access and manipulate databases. The first is with AutoLISP, a programming language within AutoCAD that lets us write programs that will manage and manipulate graphical and non-graphical data. The second way is to extract the desired data from the DXF file using other external programs.

EAGLE is a general purpose three-dimensional CAD modelling system which includes database management facilities. EAGLE is capable of exchanging graphical and non-graphical data through a vast array of data-exchange formats including DXF and IGES file formats.

The design prototype system developed is PROTOKIT. PROTOKIT contains the design prototype manager, the design prototype base and the design prototype engine. The design prototype manager allows the user to create, modify, display, print or delete prototypes and instances. The design prototype system is capable of accessing information through inheritance using links such as 'a_type_of', 'an_instance_of', 'an_element_of' and 'same_as'.

The KBI performs the necessary interpretations on EXBUILD's knowledge bases and creates the knowledge base objects according to the format of PROTOKIT.

Currently a CAD database interpreter is being developed that performs the necessary mappings between AutoCAD's database and PROTOKIT.

The working memory contains all the instances generated by the CAD database interpreter during a session. The expert system is capable of accessing information from the working memory through the knowledge-base interpreter. It is also capable to post inferred information to the appropriate instances in the working memory.

APPLICATION TO BUILDING DESIGN

Building Design Representation

The AutoCAD drawing system is used to model and represent building designs in this example. Although AutoCAD is primarily used for 2-D drafting it provides features that are useful in modeling objects. A set of associated objects can be placed on layers or grouped together to form complex objects that can be manipulated as a whole. AutoCAD stores the locations, sizes and colours of the objects we draw for subsequent retrieval, analysis and manipulation. For example, objects such as walls, floors, stairs, windows and doors can be explicitly modelled and manipulated. The top window in Figure 3 shows a plan of an office building modeled using AutoCAD.

Building Code Representation

As noted previously, the requirements of the building code are represented in the form of IF/THEN rules. The major advantage of using the rule-based approach is that it allows us to keep the representation of any particular requirement at the same high level of abstraction

appearing in the original code. The representation of a part of the BCA Code, namely the Clause D2.13 is shown in the bottom window of Figure 3.

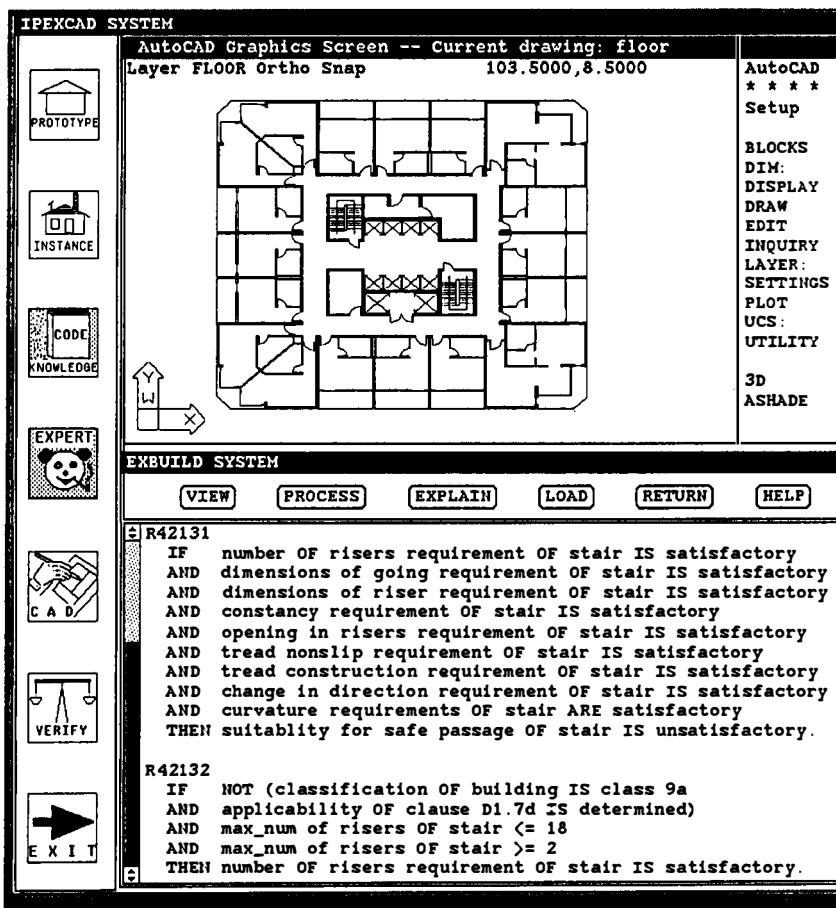


Figure 3. Example of a design description and verification knowledge

The rules shown are concerned with suitability and safety of stairways and deal with verifying the compliance with the given requirements as to provide safe passage.

Domain Knowledge Representation

As described previously, the domain knowledge is represented as design prototypes. The process of representing a design domain using the design prototype schema requires a large number of design prototypes in a network of hierarchies. Figure 4 illustrates the interface of the PROTOKIT system and the right window displays the prototype, named 'stair'.

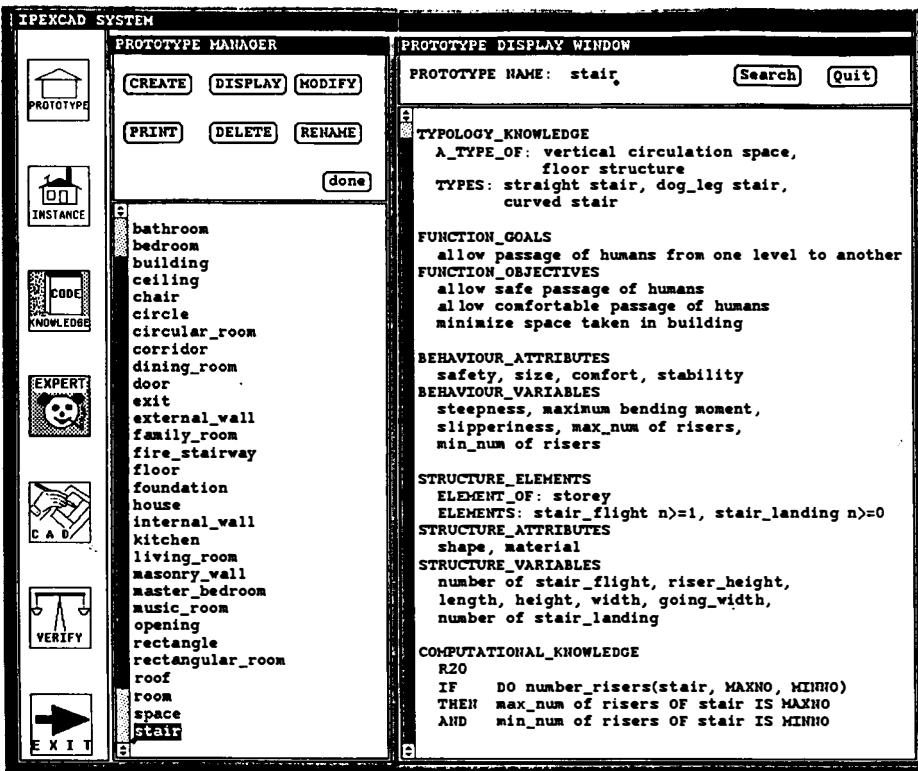


Figure 4. Example of domain knowledge represented as prototypes

Verification Process

We will now follow through the verification process for the example given in Figure 3. The rules are structured so that, firstly, the applicability of each clause is determined and, if found applicable, then its compliance is checked. A part of the verification knowledge used to check stairs for their compliance is shown in Figure 3. Rule R42131 states all the conditions necessary for a stair to allow safe passage while rule R42132 states the conditions necessary for the first condition of rule R42131, i.e. the requirement on the number of risers, to be satisfied (for certain buildings).

According to rule R42132, given that our building is a Class 5 building and the first condition is satisfied, each stair must have a maximum number of risers of 18 and a minimum number of risers of 2. The design prototype description of a stair, illustrated in Figure 4, shows that the values for the behaviour variables 'max_num_of_risers' and 'min_num_of_risers' can be determined from the computational knowledge in the design

prototype, namely rule R20. Note that the computational knowledge of the design prototypes in the form of rules should not be confused with the rules of the expert system representing the specific verification knowledge. Rule R20 makes a call to a procedure 'number_risers' which requires a given instance of a stair and returns the maximum and minimum number of risers for that instance. In this example, the stair under question is the stair instance 'stair1' as shown in Figure 5. Figure 5 also shows the instance 'stair_flight1' which is an element of 'stair1'. The information regarding the number of risers in 'stair_flight1', namely 9, is derived from the graphical database of the CAD system. The procedure 'number_risers' uses this information and instantiates the values of 9 and 9 for the maximum and minimum number of risers of 'stair1', since all the stair flights are equal. The expert system accesses the required information from the appropriate instances and updates the instances with all the facts inferred during the verification process.

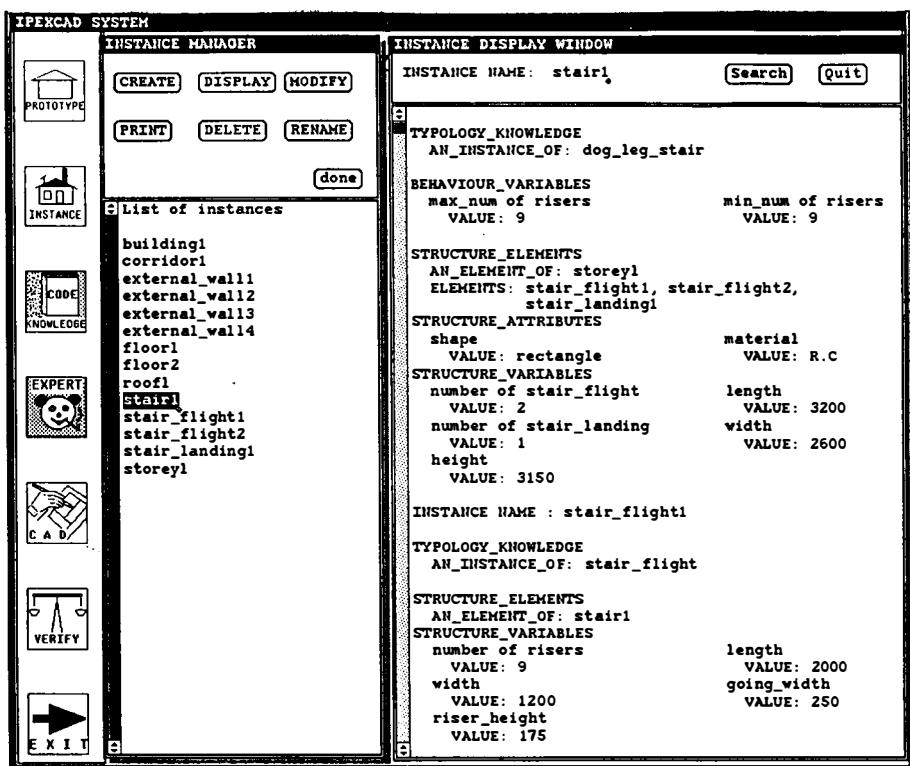


Figure 5. Example of stair instances generated during the verification process

DISCUSSION AND CONCLUSIONS

This paper has described an approach to the development of a knowledge-based system capable of checking a design through its description in a CAD database for conformance with some requirements. Conformance with the building code, BCA Code has been used as an

example of this process. Design prototypes have been used to represent the domain knowledge necessary to provide the necessary interpretations from semantics to syntax. The expert system, the domain knowledge and the CAD system are independent systems using knowledge representations suitable to their application. Communication is through the knowledge base interpreter and the CAD database interpreter. The domain knowledge is central in providing the unifying model through which communication is possible. The role of the knowledge engineer is to provide the specific CAD database interpreter depending on the particular CAD system used and providing the necessary interpretations to allow the expert system to access the relevant domain knowledge.

This flexible coupling of the elements in the system allows for a wide variety of CAD systems and expert knowledge applications to be used. Though this paper has used the BCA Code as an example of verification knowledge and AutoCAD as an example of a CAD system, in general, various other knowledge bases and CAD systems can be used. For each such different knowledge base and CAD system the task of integration will be to provide the appropriate interpreters.

While this paper advocates the production of a heterogeneous system, it is envisaged that in the future such systems will become more homogeneous as CAD databases become more standardised and especially as CAD systems begin to contain more domain knowledge, i.e. become more 'intelligent'.

ACKNOWLEDGEMENTS

This work is supported by continuing grants from the Australian Research Council. EAGLE is provided by courtesy of CADCOM Pty Ltd, and AutoCAD by courtesy of AutoDesk (Australia) Pty Ltd.

REFERENCES

- AUBRCC (1988). *Building Code of Australia*, Australian Uniform Building Regulations Co-ordinating Council, Department of Industry, Technology and Commerce, Canberra.
- Autodesk (1988). *AutoCAD Reference Manual*, Autodesk, Inc, Sausalito, CA.
- Balachandran, M. and Gero, J.S. (1988). A model for knowledge-based graphical interfaces, in J.S. Gero and R. Stanton (eds), *Artificial Intelligence Developments and Applications*, North-Holland, Amsterdam, pp.147-163.
- Balachandran, M. and Rosenman, M.A. (1990) *EXBUILD, Expert System Shell Users Manual*, Design Computing Unit, Department of Architectural and Design Science, University of Sydney, Australia.
- Bobrow, D. G., Mittal, S. and Stefik, M. J. (1986). Expert systems: Perils and promise, *Communications of the ACM*, 29(9): 880-894.
- Carbs Ltd (1985). *EAGLE, 3D Modelling System Command Reference Manual*, Clwyd, UK.
- Coyne, R. D., Rosenman, M. A., Radford, A. D., Balachandran, M. and Gero, J. S. (1990). *Knowledge-Based Design Systems*, Addison-Wesley, Reading, Mass.

- Dym, C.L., Henchey, R.P., Delis, E.A. and Gonick, S. (1988). A knowledge-based system for automated architectural code-checking, *Computer-Aided Design*, 20(3):137-145.
- Fenves, S. J., Flemming, U., Hendrickson, C., Maher, M. L. and Schmitt, G. (1990). Integrated software environment for building design and construction, *Computer-Aided Design*, 22(1):27-36.
- Garrett, J. H., Jnr and Fenves, S. J. (1987). A knowledge-based standards processor for structural component design, *Engineering with Computers*, 2:219-238.
- Gero, J. S.(ed.) (1987a). *Expert Systems in Computer-Aided Design*, North-Holland, Amsterdam.
- Gero, J. S. (1987b). Prototypes: a new schema for knowledge-based design, *Working Paper*, Architectural Computing Unit, Department of Architectural Science, University of sydney.
- Gero, J. S. and Rosenman, M. A. (1989). A conceptual framework for knowledge-based design research at Sydney University's Design Computing Unit, in J.S.Gero (ed), *Artificial Intelligence in Design*, CMP/Springer Verlag, Southampton and Berlin, pp.361-380.
- Graphisoft (1989). *ArchiCAD, Version 3.4, User's Manual*, Graphisoft.
- Hoskins, E.M. (1977). The OXSYS system, in J.S. Gero (ed), *Computer Applications in Architecture*, Applied Science, London, pp.343-391.
- Jain, D. and Maher, M.L. (1987). Combining expert systems and CAD techniques, in J.S. Gero and R. Stanton (eds), *Artificial Intelligence Developments and Applications*, North-Holland, Amsterdam, pp.65-81.
- Kostem, E and Maher, M.L. (eds) (1986). *Expert Systems in Civil Engineering*, ASCE, New York.
- Lee, K. (1977). The ARK-2 system, in J.S. Gero (ed), *Computer Applications in Architecture*, Applied Science Publications, London, pp.312-342.
- Nnaji, B. O. and Kang, T. S. (1990). Interpretation of CAD models through neutral geometric knowledge, *AI EDAM*, 4(1):15-45.
- Rehak, D. R. and Howard, H. C. (1985). Interfacing expert systems with design databases in integrated CAD systems, *Computer-Aided Design* 17(9): 443-454.
- Rehak, D.R., Howard, H.C. and Sriram, D. (1985). Architecture of an integrated knowledge-based environment for structural engineering applications, in J.S.Gero (ed), *Knowledge Engineering in Computer-Aided Design*, North Holland, Amsterdam, pp.89-117.
- Rosenman, M. A. (1990). Application of expert systems to building design analysis and evaluation, *Building and Environment*, 25:(3):221-233.
- Rosenman, M.A., Gero, J.S. and Oxman, R. (1986a). An expert system for design codes and design rules, in D. Sriram and R. Adey (eds) , *Applications of Artificial Intelligence in Engineering Problems*, Springer-Verlag, Berlin, pp. 745-758.
- Rosenman, M.A., Manago, C and Gero, J.S. (1986b). A model-based expert system shell, *IAAAI'86*, pp.c:1:1-15.
- Rosenman, M.A., Balachandran, M. and Gero, J.S. (1989). Prototype-based expert systems, in Gero J. S. and Sudweeks, F. (eds), *Expert Systems in Engineering, Architecture and Construction*, University of Sydney, Sydney, pp.179-200.

- Sharpe, R., Marksjo, B. and Ho, F. (1989). Wind loading and building code expert systems, *in* Gero J. S. and Sudweeks, F. (eds), *Expert Systems in Engineering, Architecture and Construction*, University of Sydney, pp.223-242.
- Tyugu, E. (1987). Merging conceptual and expert knowledge in CAD, *in* J.S. Gero, (ed.) *Expert Systems in Computer-Aided Design*, North-Holland, Amsterdam,pp.423-431.
- Waterman, D. (1986). *A Guide to Expert Systems*, Addison-Wesley, Reading, MA.

A preliminary structural design expert system (SPRED-1) based on neural networks

X. Liu[†] and M. Gan[‡]

[†]Department of Civil Engineering
Tsinghua University
Beijing 100084 PR China

[‡]Beijing Architecture Design Institute
Beijing 100045 PR China

Abstract. A prototype for the preliminary design of space grid structures is introduced. In the present prototype several neural networks are treated as blocks, which can be used not only as knowledge representation modules but also as simulation and optimization modules. The system architecture is an integration of three components: prediction, evaluation and intelligent controller. Considering the lack of general availability of parallel computers, the back propagation algorithm is modified. Finally, some practical examples are given.

INTRODUCTION

Although computers have been employed in civil engineering for more than thirty years, and essentially all aspects of structural analysis have been highly computerised, many experienced engineers still question the usefulness of computers in the design task. To some extent, it is true that although designers have their own rich experience, they have difficulty expressing their own thinking processes. Different kinds of characteristic tables have already been stored in their minds, but the mapping process is just a black box which is very hard to explain, even by themselves. The exploration of computer aids to assist this stage of structural design is in its infancy (Fenves, 1991). The concepts of knowledge-based expert systems (KBES) have begun to emerge over the last years but, strictly speaking, there have been few contributions in artificial intelligence by KBES researchers. The most difficult issue is that of knowledge formalisation. Mostly, the knowledge representation is based on rules, semantic nets, frames, etc., which are all based on the logic thinking mode. From the authors' point of view, thinking in logic is derived from thinking in images. If human intelligence could be placed in a box, for example, then the logic thinking mode would be the top layer in the intelligence box. If we want to model human intelligence, which means taking the whole stuff out of the box, we have to touch the top layer first. In this case, using some appropriate tools (such as rules, semantic nets, frames) to move the top layer out is not

very difficult. However, if we want to use the same tools to move the deeper layer, such as the thinking in images, it may be impossible. As many scholars have mentioned, there are several important limitations of knowledge-based technologies, such as static and brittle characteristics, which are just shortcomings of the logic thinking mode. In structural design, especially in the preliminary design, both thinking in logic and thinking in images are needed.

In China, the entire structural engineering design process can be divided into three stages—the conceptual design, the preliminary design, and the construction design. In most areas of structural engineering, particularly in building and plant design, the conceptual design only provides the spatial layout decisions affecting the structural system, which are usually made by architects. This stage seems to be essentially unsupported by computers. Based on the information provided by the conceptual design, the preliminary design begins. In this stage, the structural engineer first performs a schematic design, generating one or more structural configurations that may provide the intended structural function. This stage is considered as the most creative phase of structural design. In this case, both thinking in terms of images and some necessary calculations cannot be avoided. Finally, some important structural parameters will be given. It should be mentioned that, in the preliminary stage, high calculation accuracies are not as necessary as in the construction design stage. It is often to provide feedback on structural configurations or originally assumed key parameters to satisfy the architectural decisions. In the construction design, however, almost each step has been either fully or partially computerised. Although a large number of CAD software packages are available, few of them are for the preliminary design stage. The key point, as mentioned previously, is how the knowledge can be formally represented, especially to include both thinking in logic and thinking in images.

Artificial neural networks, as a new knowledge-representation paradigm, is receiving a lot of attention and represents advancements that will eventually find their way into mainstream knowledge-based system development (Garrett, 1990). The past six years or so have seen a substantial amount of work being done in this area. Neural networks are ideally suited for solving certain types of engineering problems that require a mapping from a large distributed set of input features to a large distributed set of output features, such as pattern association, etc. Thus, neural networks may be treated as a new representation mode of knowledge, especially related to thinking in terms of images. The first expert system for the preliminary design of high-rise buildings (HIDE-1), which is based on neural networks, was developed at Tsinghua University (Zhang and Liu, 1989). A new version (HIDE-2), which is based on a combination of neural networks with a rule base, was developed in 1990 (Li and Liu, 1990).

In the present paper, a prototype for the preliminary design of space grid structures (SPRED-1) is introduced. In this prototype, each neural network is treated as a block which can be used not only as various knowledge representation modules but also as a simulation module (instead of the finite element calculation model) and an optimization module.

In the first part of this paper, the basic concept of neural networks and the back propagation algorithm are introduced briefly, followed in the second part by a description of SPRED-1. In the third part of the paper, modifications of algorithms of the present system are described. Finally, some engineering application examples and conclusions are given. Only steel structures are considered in the present prototype.

NEURAL NETWORKS

A neural network is a system composed of many simple processing elements operating in parallel whose function is determined by a network structure, connection strength, and the processing performed at computing elements or nodes (DARPA, 1988). A number is associated with each node, referred to as the node's activation. Similarly, a number is also associated with each connection in the network, called its weight. These are based on the firing rate of a biological neuron and the strength of a synapse between two neurons in the brain.

Each node's activation is based on the activation of the connected nodes and the connection weights. A rule that updates the activation of each node is typically called the learning rule (Zeidenberg, 1990). Obviously, all activation would be updated simultaneously. Thus a neural network is a parallel model. Following the physiological structure of a neuron shown in Figure 1(a), the simplest node of a neural network can be modelled as Figure 1(b), which includes: (i) a set of input units, X_i , that arrive through weighted pathways; (ii) W_i , the weight of a particular pathway i , which represents the synaptic efficacy; a single output signal, y , which is related to the input values X_i by

$$y = f \left(\sum_i W_i X_i + \Theta \right) \quad (1)$$

and (iv) a bias term, Θ , which acts as a form of threshold. In equation (1), the function f can be nonlinear.

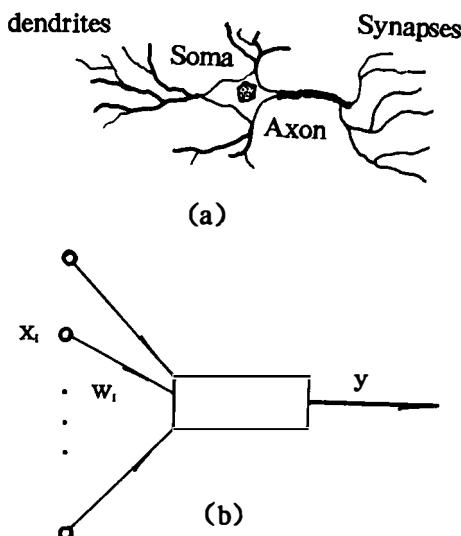


Figure 1. A simple neuron model

A neural network usually is defined by its connection scheme, its nodal characteristics, and the learning rule. The goal of most neural network models is to learn relationships between inputs and outputs. Learning in a neural network typically occurs by adjusting weights with a learning rule. At the beginning of learning when the weights are not correct, the network performs badly. At the end of learning when the weights are adjusted, one hopes that it will perform well. Usually the learning rule does not change, only the weights. After learning, the weights are not usually adjusted further unless something new must be learned (Zeidenberg, 1990).

In recent research, the back propagation algorithm is the most widely used. It is usually used in a multilayer feedforward net as shown in Figure 2. The input is the lowest layer and the output is the highest, and all activation flows from lower to higher units.

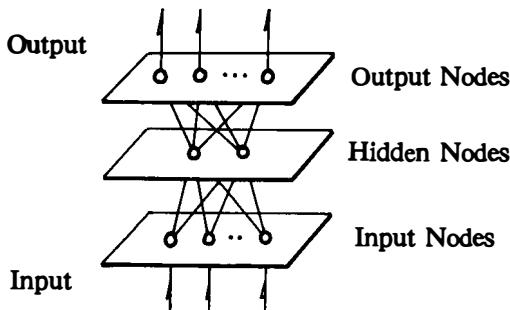


Figure 2. A multilayer feedforward net (three-layer perceptron)

The back propagation can be explained as follows (Zeidenberg, 1990). If the network is presented with input pattern p , assume the response of the network on output unit j to be O_{pj} and the desired target output be t_{pj} , then the difference is expressed as:

$$\delta_{pj} = t_{pj} - O_{pj} \quad (2)$$

If the i th unit of the input pattern has input I_{pi} , then the change in the weight connecting unit i and j is given by

$$\Delta_p W_{ji} = Z \delta_{pj} I_{pi} \quad (3)$$

where W_{ji} is the weight between i and j , and Z is some constant. For generalised cases,

$$\delta_{pj} = (t_{pj} - O_{pj}) f_j'(net_{pj}) \quad (4)$$

where $f_j'(net_{pj})$ is the derivative of the activation function f , evaluated at net_{pj} , which is the net input that unit j is receiving.

If j is not an output unit, the error signal is a function of the error signals of connected units higher in the network. While activation propagates upward, when computing the

output, errors propagate backward in order to adjust the weights. The error in a connection that is below the top is:

$$\delta_{pj} = \sum_k \delta_{pk} W_k \cdot f_j'(net_{pj}) \quad (5)$$

where k ranges over all the units to which unit j outputs.

Equation (3) can be applied iteratively, and may cause a convergence of the actual and target output. If there are n output units, the weights connecting them with the input units form a n -dimensional space. Usually, the minimum value for the error in the space can be found by the steepest gradient descent method.

SPRED-1 SYSTEM

SPRED-1 is a prototype for the preliminary design of space grid structures (Figure 3). As mentioned previously, during preliminary design, the structural engineer has to interact and coordinate his work with architects, ventilation engineers, environmental engineers, and others. Discrepancies, discussion, feedback and redesign usually are all inevitable. In this case, time becomes more critical. From a knowledge engineering point of view, both the thinking in logic (such as necessary calculation and rules from codes and experts) and the thinking in images (such as patterns and topological ideas) are all involved. Therefore, a more friendly interface with designers, a quicker response from the system, and a more efficient knowledge representation module are very important.

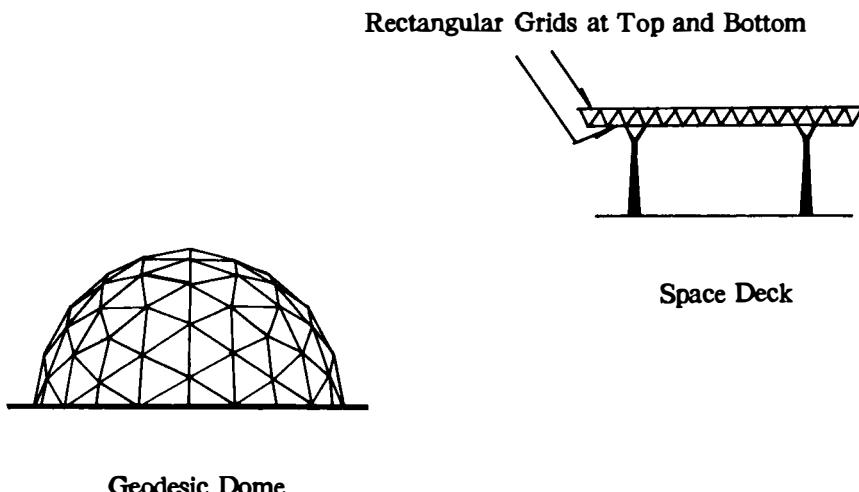


Figure 3. Typical space grid structures

The system architecture is shown in Figure 4. Each subsystem of the present prototype is introduced as follows.

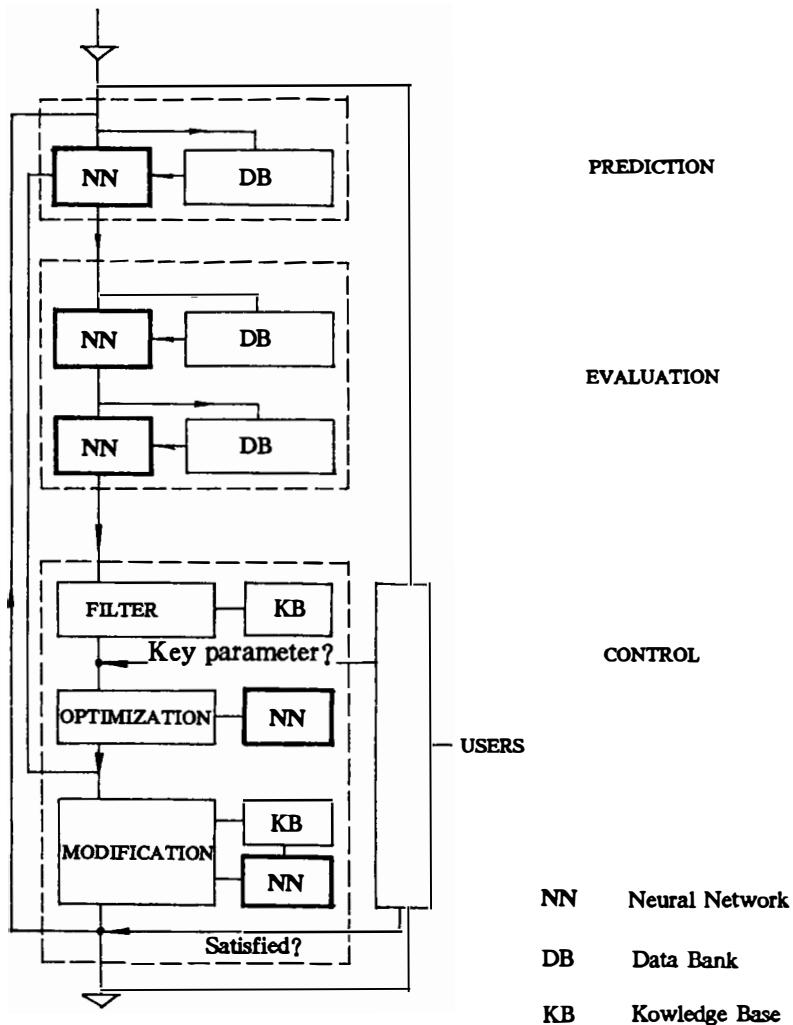


Figure 4. System architecture of SPRED-1

The prediction subsystem

According to the spatial layout decision from the conceptual design, the present subsystem predicts a set of basic parameters for a new space structure. In this case, a three-layer perceptron is used. It has been proven that any pair of arbitrary samples can be recorded by a three-layer perceptron with $K-1$ hidden nodes where K is the number of inputs (Ying, 1990). Therefore, in the present system, the number of hidden nodes always equals $K-1$. Actually, the prediction subsystem is a simulation module which can be used to model the calculation process with reasonable accuracy in the preliminary design.

Similarly, as shown in Figure 2, the input nodes on the lowest layer are:

- (a) plan dimensions,
- (b) structural depth,
- (c) grid lines,
- (d) the number of different member types,
- (e) support condition, and
- (f) applied loads for the new structures.

The output nodes, which are on the highest layer, are:

- (a) total amount of steel for the new project,
- (b) maximum internal forces,
- (c) maximum deflection,
- (d) maximum reaction, and
- (e) maximum size of the joint.

From the input and output terms, it can be seen that this is a typical calculation simulation module. After sufficient learning, the outputs can provide enough accuracy without any structural analysis program.

Various space grid structures, such as grid domes, grid decks, etc., can all be considered in the subsystem. For a particular space structure, a related data bank, which includes almost thirty previously designed projects, can be used for subsystem learning. From a knowledge engineering point of view, this is also called example-based reasoning (Wang and Tai, 1990). Often, designers would like to prepare a number of input sets, which seems reasonable for a new project, and check the outputs provided by the prediction subsystem.

The evaluation subsystem

There are two evaluation blocks in the subsystem. In each block, the three-layer perceptron is also used for different level evaluations. For the first block, the input nodes are only the inputs and outputs of the prediction subsystem, which means eleven terms in total. The output nodes of the block are five evaluation values, each of which indicate the credit of the corresponding output from the prediction subsystem. Credits are expressed by a defined linguistic variable, such as 'excellent', 'good', 'fair', 'bad' and 'very bad'. After learning from a data bank in which a great number of evaluation credits are collected from a group of

experienced engineers, the weights in the perceptron are determined.

The second block gives the overall evaluation based on the outputs of the first block. It means that the inputs of the present block are the five previous evaluation values and the output is an overall evaluation only. During the learning process of both blocks, the back propagation algorithm is also adopted.

An engineer may have given an unusual credit for a particular prediction which makes the learning convergence slow. After a certain number of oscillations, this example can be deleted automatically (Shi and Liu, 1990). It means that some contradictory samples will be filtered.

The control subsystem

In the present subsystem, a number of suggested parameter sets (in our case each set has six parameters) are checked, filtered and modified, which are either sent to users as final results or fed back to the prediction subsystem. The control subsystem, as an intelligent controller, consists of three blocks such as the filter block, the optimization block and the modification block, which are described as follows.

The filter block is used for checking the suggested sets of five basic parameters from the evaluation subsystems. There is a rule base for checking; this contains many rules from design codes and experts. The threshold of the overall evaluation is given. In general, if the overall credit is worse than 'fair', the suggested set of basic parameters will be deleted. For a suggested set A, if each term is worse than the corresponding term in another set B, then A is defined as an inferior solution. After comparison, inferior solutions are deleted. Finally, a set of solutions, or several so-called satisfied sets of eleven basic parameters, remain and are sent to the optimization block.

In the optimization block, one or two parameters can be chosen as the objective(s) by an interface with users. Usually, the total amount of steel is the most important factor in China. In this block, another three-layer perceptron is used for optimization. The output of this perceptron is the chosen objective and the inputs are the same as the inputs of the prediction subsystem; that is, the plan dimension, the structural depth, etc. After learning from the satisfied basic parameter sets, the connection weights of the perceptron are all fixed. Following the Hopfield idea (Zeidenberg, 1990), a new energy function is defined as

$$E(q) = \Sigma(g - g')^2 + \Phi(\Sigma \ln(Cq)) + \Phi_1(\Sigma q - F_{tot}) \quad (6)$$

where q is the same as the inputs of the prediction subsystem. $E(q)$ is the energy function which is borrowed from thermodynamics, g is the ideal value of the objective, g' is the output from the present perceptron, Φ is the entropy constraint function, Φ_1 is an additional function, F_{tot} is the possible range of q , and C is a constant shown as

$$C = \frac{N}{e^{\Sigma(1/q)}} \quad (7)$$

where N is a constant.

Since the constraints of optimization in the preliminary design are very soft and thus the algorithm ill-posed, equation (6) needs to include the third term. The second term of equation

(6) is the entropy constraint function. Using equation (7), the entropy is well distributed. When the minimum E is reached, the optimal q can be obtained and transferred to the prediction system again to obtain the optimal outputs.

The third block of the control subsystem is the modification block. After optimization, only one set of eleven basic parameters is taken and can be transferred to the modification block. In this block, the feasibility of the eleven basic parameters are checked by a rule base. According to the comments from the rule base, if some parameter is modified, one more three-layer perceptron is needed for sensitivity analysis. In this perceptron, the output is the parameter which should be modified and the inputs are the ten remaining parameters. After the sensitivity analysis, several rules can be obtained for modification, such as 'If the maximum force of the ball joint is larger than 78 tons, then the structural depth should increase.' Finally, the most satisfied feasible design parameters are obtained.

If it is necessary, the block can interact with and coordinate the chosen parameters with parallel design activities by an interface. If something should be changed, the changed parameters can be fed back to the prediction subsystem.

MODIFICATION OF THE BACK PROPAGATION ALGORITHM

As mentioned previously, the neural network is a parallel model. Because of the general unavailability of parallel computers, neural networks are performed on conventional serial computers. The back propagation and related procedures, therefore, may be prone to the poor convergence problem in some cases, and often lead to a local minima. In the present system, two methods are adopted to improve the back propagation algorithm.

1. Using the Daridom-Fletcher-Powell (DFP) method instead of the steepest descent method in the back propagation algorithm. It is well known that the steepest descent method is a very simple and robust method, but its convergence speed is very slow, especially when the results are close to the optimal point. In this method, only the first order derivative is used to determine the descent direction. In the serial computing environment, more CPU time for convergence is needed. However, the optimal strategy of the DFP method uses the gradient information from previous steps. It builds a direction matrix, which is very close to the Hesse matrix. This direction matrix is used to find the descent direction and to calculate the descent scale of the next step. This method, therefore, is also called the variable scale method. It can be proven that the DFP method has second-order convergent speed. Table 1 is a comparison of the DFP and the steepest descent methods. The testing function is taken as a conditional failure rate function of structures (sometimes called the 'bathtub curve'), for example, $y = 100(x_2 - x_1^2) - (1 - x_1)^2$, where x_1 and x_2 are variables.
2. Classify and carefully select the sample sets for the neural network learning. It is well known that, in order to increase the convergent speed of the back propagation algorithm, classification of the sample sets before network learning is necessary, especially when the more efficient DFP method is used. If a grid deck is to be designed, for example, a related data bank which collects almost thirty previously designed grid decks has to be prepared for network learning in the prediction subsystem. This means

that only those structures similar to the new deck can be considered in the data bank. Conversely, in order to avoid the local optimal point in back propagation, the possible ranges of each parameter (i.e. the terms in q , see equation (6)) are considered. The orthogonal tables are used for selecting the previously designed samples.

Table 1. Comparison of the DFP and the steepest descent method

Method	Max. error (%)	Iteration times	CPU (hrs)*
Steepest descent	21	1100	4
DFP	18	137	0.128

* On a Vax 780

After the learning process, another previously designed project which is a space grid structure similar to those collected in the data bank, is used to check the network ability. It is found that network learning from the selected samples, which are chosen by orthogonal tables, has much higher efficiency than that from randomly set samples (Table 2).

Table 2. Comparison of the efficiency of different sample selections

Sample selection	Error (%)	Iteration times	Number of samples	CPU (hrs)*
Random	43	1405	21	3
Orthogonal tables	15	148	8	0.4

* On a Vax 780

APPLICATION EXAMPLES

Preliminary design of the Madagascar Gym

The plan dimensions of the gym are 60 x 60 m. Its roof is a typical space deck shown in Figure 5, which is supported on just 12 columns to give long, clear, unobstructed spans. The roof is double-layered with a large structural depth. Many spatial orthogonal pyramid trusses are used to compose the space grid deck.

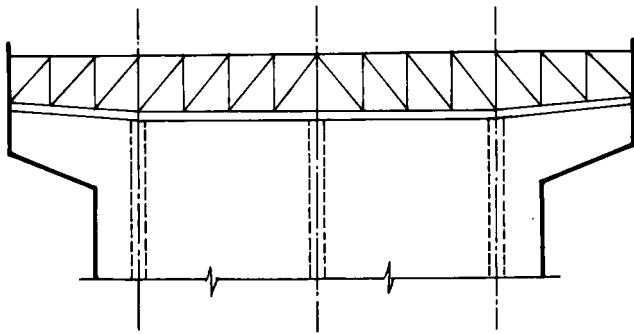
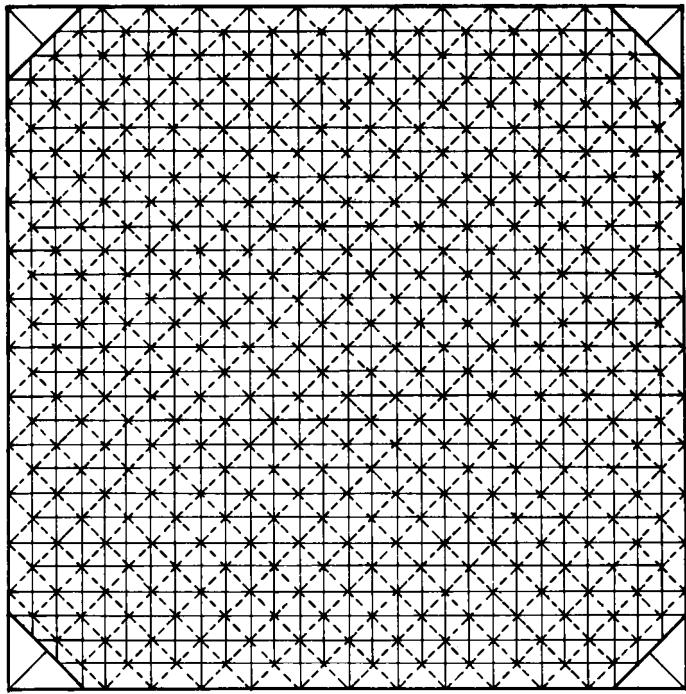


Figure 5. Madagascar Gym

In the control subsystem, optimal parameters are obtained by the Hopfield algorithm. The suggested structural depth and grid dimension are 4.74 m and 4.61 m respectively. However, in the modification block, architects are able to give comments via an interface. They hope to change the structural depth to 3.00 m and the changed depth is fed back to the prediction subsystem. Then, the total amount of steel, the maximum forces, deflection, reaction, and joint size can be obtained from the prediction subsystem and transferred to the subsequent subsystems. Subsequently, it is found that the maximum deflection exceeds the limitation of the China Design Code and the total amount of steel is too much. Therefore another suggestion has to be given and the new depth is fed back to the prediction subsystem again. After several cycles, the final scheme is obtained in which the structural depth and grid dimension are 3.71 m and 4.28 m respectively (Table 3).

Table 3. Comparison of different schemes and methods

Method	Structural depth (m)	Max. internal force (KN)	Maximum displacement (mm)	Total amount of steel (ton)
Present system	4.74	500	40	162
	3.00	780	110	202
	3.71	650	70	184
Conventional method	3.71	620	76	176

In Table 3, the results of the conventional method are also shown which are obtained by the finite element method and the given depth, 3.71 m, is treated as one of inputs. It can be seen that the accuracy of the simulation function of the present system is satisfactory compared with the finite element method.

Preliminary design of geodesic domes in Beijing Zoological Garden

The configuration of the geodesic dome is shown in Figure 6. In the network dome there is no ring and rib, and the structural deflection is considerable. So the geometric non-linearity has to be calculated. When the critical load of this kind of structure is calculated by the finite element method, it takes too much CPU time which is not convenient in the present system. Conversely, in the preliminary design, increased accuracy is not a major requirement. For the analysis method, the typical level of accuracy associated with parameters that determine the success of a design idea may change from 2% to 30% (Smith, 1991). It seems that the extra accuracy, especially in the preliminary design stage, is not needed. In the prediction subsystem of the present system, the critical load is treated as one of the outputs of the three-layer perceptron. After the learning process, the network presents very satisfactory results for the preliminary design (Table 4) (λ_{cr} is the ratio of the critical load to the standard load).

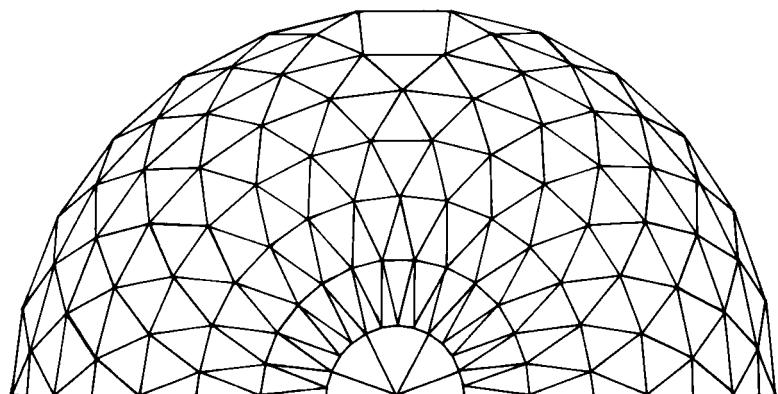
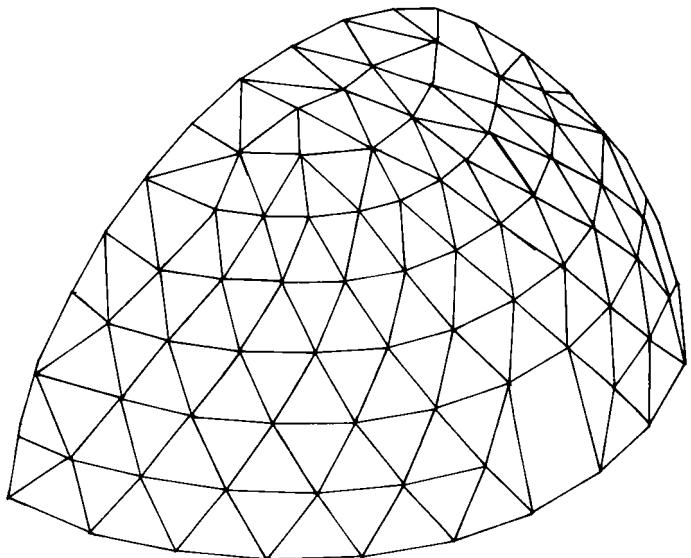


Figure 6. Geodesic dome in Beijing Zoological Garden

Table 4. Comparison of critical loads

Plan diameter (m)	Dimension of steel tube (mm)	λ_{cr}	
		By present system	By finite element method
18	112 x 5	9.8	12.7
80	180 x 4	8.85	10.7
28	60 x 4	9.92	11.3

CONCLUSION

1. A prototype for the preliminary design of space grid structures (SPRED-1) is introduced in which neural networks are treated as blocks for knowledge representation, simulation and optimization.
2. The system architecture is an integration of three components, such as prediction, evaluation and intelligent control. It has a friendly interface, a quick response, and is a more efficient knowledge representation module.
3. The DFP method is introduced in the back propagation algorithm instead of the steepest descent method to increase convergence. In this case, the classification and selection of samples have to be considered.
4. The exploration of artificial intelligence in design is in its infancy. We have to do something but the conventional tools in artificial intelligence cannot help us very much.

Acknowledgements. The assistance of Fay Sudweeks in improving the English presentation is greatly appreciated.

REFERENCES

- DARPA (1988). *Neural Network Study*, AFCEA International Press.
- Fenves, S. J. (1991). Status, needs and opportunities in computer assisted design and analysis (prepared for *Structural Engineering International IABSE, VI*).
- Garrett, J. H. (1990). Knowledge-based expert systems: past, present and future, *IABSE Periodica* 3: 21–40.
- Li, J. M. and Liu, X. L. (1990). An expert system for the preliminary design of high-rise buildings based on a coupling of a neural network and a rule base, *Technical Report CE-ESS-90-01*, Department of Civil Engineering, Tsinghua University, Beijing (in Chinese).
- Shi, C. G. and Liu, X. L. (1990). Learning strategy of a neural network and its application to earthquake engineering, *Proc. Research of the Application of AI-Aided Decision-*

- Making Systems in Civil Engineering*, Electronic Industry Publishing House, Beijing, pp.160–165 (in Chinese).
- Smith, I. F. C. (1991). Research tools or design aids (prepared for *Structural Engineering International IABSE*, VI).
- Wang, J. and Tai, R. W. (1990). An approach to build a knowledge system with an artificial neural net, *Proc. Neural Networks and Their Application*, Vol.1, Institute of Automation, Academic Sinica, pp.22–27 (in Chinese).
- Ying, X. R. (1990). The relation between sample pairs recording and hidden units of three-layer neural networks, *Proc. Neural Networks and Their Application*, Vol. 4, Institute of Automation, Academic Sinica, pp.33–38 (in Chinese).
- Zeidenberg, M. (1990). *Neural Networks in Artificial Intelligence*, Ellis Horwood.

Structuring an expert system to design mineshaft equipment

G. J. Krige

Department of Civil Engineering
University of the Witwatersrand
Private Bag 3, PO Wits 2050 South Africa

Abstract. The design of the equipment in any mineshaft is a complex combination of knowledge in different domains, analytical methods and heuristic information and rules. The complete design is typically carried out by two or more domain experts, which often leads to independent design of components whose designs influence each other. This results in less than optimum final products. In many cases, several conceptual designs are carried out as feasibility studies to investigate mining of new ore bodies. This requires a major time commitment on the part of experienced senior engineering personnel.

The provision of an expert system is proposed as a solution to these difficulties. The specific needs of the users are investigated. In summary these are that the engineer must feel that he retains all responsibility for the design, there must be sufficient flexibility to accommodate changing practice and design philosophy, the system must have clearly identifiable benefits, it must provide proven results and be accessible and it should call on as much other design software as possible.

A multi-module system, controlled by the user through a control module, has been developed to meet these requirements and is described in the paper. Aspects of this development covered are the structure of the expert system, some comments on the knowledge acquisition, and the structure of the design process.

INTRODUCTION

The design of the equipment in any typical mine shaft is a complex and often time-consuming operation. The many different factors involved mean that a great deal of experience, as well as clearly understood and up-to-date technical information, is necessary in order to achieve a good result. This implies that the work must be carried out by highly skilled and experienced engineering personnel. In the early planning and feasibility studies for a new mine, the conceptual design of the mineshaft equipment is typically performed many times, with different initial parameters. Computer software capable of providing assistance with this task would be extremely valuable.

The design may also be roughly categorised into three different aspects, each of which would typically be dealt with by different domain experts, but each of which may have a significant impact on the others. In the first place, engineers and planners

responsible for the overall layout and design of the mining operation determine what mining methods will be used, which in turn determines the ventilation and shaft size requirements. These same engineers would usually assess what shaft hoisting equipment is necessary for the transport of men and materials and the removal of ore. Then a conceptual and detailed design of the equipment itself must be done. The design of the hoisting machinery and ropes is typically carried out by mechanical engineers, the design of the headframe is carried out by structural engineers, and the design of the conveyances is carried out by either the mechanical or structural engineers.

The mining processes and equipment used have been developed largely by trial-and-error over many years, so that much of the technology resides in heuristic knowledge. In recent years, however, the desire for more consistently reliable shaft equipment and a sufficient understanding of the mechanics of the hoisting operation so as to allow the confident introduction of innovations, has led to the development and acceptance of more rational design procedures for certain parts of the shaft equipment. Implicit in future enhancements and current application of these more rational design procedures is the need to link the heuristic and analytical knowledge. A further factor has been recognised as contributing to problems in mineshafts, and it must thus be addressed in the quest for improved design methodologies. This is that different components of the mineshaft equipment are commonly designed independently of each other, often overlooking the vital interaction between these various components. There is thus clearly a need to provide tools so that the various domain experts may have improved access to each other's domain knowledge.

With these factors in mind it appears that the development of an expert system to cover the conceptual design of mineshaft equipment is likely to offer great benefit in this complex design area. The author is well acquainted with the domain knowledge for the structural design of mineshaft equipment, so that the expert system described is partially self-authored, as discussed by Cohn (1988), but important contributions are also derived from other domain experts.

An initial difficulty was that the total domain considered could extend almost indefinitely, either into the area of planning and layout of the mine, or on the other hand to the detailed design of all the components of the shaft equipment. This clearly had to be limited, so it was decided to assume that the required monthly production requirement, shaft diameter and shaft depth are already fixed and known to the user, and also that no detailed design would be included.

STRUCTURING THE EXPERT SYSTEM

The structuring of an expert system for the design of the equipment in mineshafts must take cognizance of several factors if it is to be accepted in a practical working environment. Primary among these factors are the following:

Definition of the Tasks

It has been recognised (Ortolano 1987) that successful expert systems are based on knowledge which is narrowly focused, and on tasks which should not require more than several hours of an expert's time. The design of the equipment in mineshafts is thus an interesting problem because it consists of several fairly distinct, and yet not

completely independent tasks. Most of the tasks taken in isolation would satisfy the requirements for likely success of an expert system, whereas grouped together they constitute too diverse and too large a design domain. Certain of the tasks are also primarily analytical, so that they can best be undertaken by more conventional software which is currently available and is widely and confidently used in mineshaft design. Examples of such software and its use are described by Touwen, Agnew and Jougin (1973), Krige (1986) and Greenway and Thomas (1989).

Integration of Rules and Existing Cases

Much of the design carried out is case based, but three significant difficulties with case based design of mineshaft equipment are raised. First is the fact that the design parameters vary so widely that not all new design cases are sufficiently similar to some existing case that they can be solved using case based design. Second is the trend towards increasingly mechanised mining methods with implicit changes to the numbers of men and types of equipment conveyed in shafts, as well as a trend to deeper shafts, and third is a regularly expressed concern that the application of case based design will dampen any creative innovation. These difficulties suggest the use of production rules in addition to a database of cases, and the provision of user overrides of any solution achieved by the expert system.

Satisfying User Requirements

In discussion with many experienced engineers in the mining industry, it has become very clear that there are a number of basic requirements if they are to use an expert system for shaft equipment design.

(a) The engineer must retain responsibility for the design, and must always control the design process. This responsibility is a specific legal requirement, and discussion with many engineers who are potential users of such a system, shows that a psychological responsibility, in the sense of feeling responsible, is also vitally important. It would thus not be acceptable for the expert system to perform the entire design function. It must operate as a tool or prompt, performing only those tasks which the engineer specifies, and even then allowing constant interaction so that the engineer can follow all design stages. The engineer must then be given the opportunity to accept or alter all decisions made by the expert system. This expressed desire is perhaps a development of what Garrett (1990) refers to as transparency of the knowledge applied and the solution developed.

(b) The expert system must have identifiable benefits to the user. In the case of mineshaft equipment design there are four primary benefits. First is retention of knowledge, which is implicit in the development of rules and databases. This is particularly important in the South African context where the pool of competent engineers is small, and senior engineers are not easily replaced. Second is the opportunity to introduce standardised design philosophies and procedures. A uniform approach to mineshaft design, which can then be tested over an extended period of time in order to have its reliability confirmed, is very desirable as the mineshaft is a crucial component of the transportation system on which the entire production of the

mine depends. Third is the possibility of experienced senior engineers delegating design to inexperienced younger engineers, while still being assured that the design will be executed properly, and that important factors will not be overlooked. Fourth is the ability to quickly and effectively perform good conceptual studies in order to assess the viability of new mines or the implications of changes which become apparent during the construction or operation of shafts. This would offer a significant time and cost saving, particularly if it were possible to quickly access and search databases containing key information, such as legal safety requirements, information regarding rope construction and strengths, as well as existing cases.

(c) The expert system must be developed and tested in such a way as to provide confidence in its conclusions. The mineshaft is the lifeline of any mine. The safety of all underground workers, as well as the entire production of the mine depends entirely on the continued reliable operation of the equipment in the mineshaft. This leads to a very strong resistance, on the part of most engineers, to using any procedures or design tools in which they do not have the utmost confidence.

(d) The various mining companies have design philosophies which differ in certain aspects. It is also possible that basic design parameters, such as the legal safety requirements, may alter in the future. It is thus important that the expert system is flexible wherever differences or alterations can be anticipated.

(e) Accessibility of the expert system in the sense of operating on a computer which is physically located in the engineer's office, and "user friendliness" in the sense of having attractive and clear input and output, are also most important if the system is to be extensively used. This factor, together with the desire to develop an expert system in a reasonably short time period, led to the decision to use an expert system shell which runs on a desktop computer, of which the most common in South Africa are IBM compatible PCs. LEVEL5 OBJECT (1990) was selected for this purpose.

The System Structure

Wang (1988) proposes an integrated system for structural design which is structured as a rule based designer and a case based designer operating on a design blackboard. The case based designer in turn accesses a design knowledge base. Such a structure would appear to offer many advantages for the mineshaft design task at hand. It allows for the introduction of case knowledge through databases, as well as heuristic knowledge through rules and/or default values. The blackboard concept allows for the development of different modules incorporating the domain knowledge in the different areas of the mineshaft equipment design. LEVEL5 does not, however, offer a blackboard capability, so the expert system is structured as a suite of sub-systems, or modules, each of which may be accessed as necessary by a control module. A further benefit which has transpired from this approach has been the ease with which each sub-system may be tested and modified, or indeed further sub-systems added. Figure 1 shows the structure of the system schematically.

The expert system is sub-divided into three different portions.

(a) The engineer accesses all the different parts of the system via the control module which operates as a link, providing ruled-based guidance as to which further module

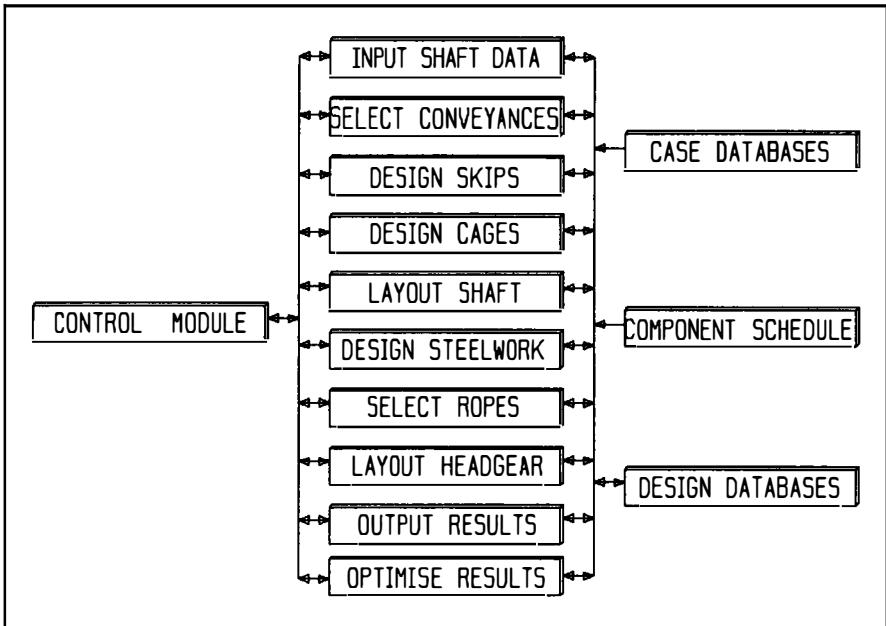


Figure 1: Structure of Expert System

should be accessed next. One of the current drawbacks of LEVEL5 OBJECT is that it cannot load sub-systems while retaining the control module, so that access to all other modules of the system is by chaining onto them, which deletes the current module. However, on completion, all sub-systems chain back to the control module.

(b) The sub-systems themselves form the second portion of the system. Each sub-system is set up to be self-contained in the sense that it can operate and reach conclusions without the need to access information from any other sub-system, provided the relevant information has been defined in the design databases. This enables the easy alteration, development and checking of each sub-system. Such a structure also enables the domain expert, or other domain experts, to expand the system by adding further sub-systems. One significant drawback of this approach is that most of the sub-systems require information which is determined within other sub-systems, or determine information which may influence the conclusions of other sub-systems. In order to provide a certain superficial level of truth maintenance and retain consistency in the final conclusions, the procedure adopted in this expert system is to use WHEN CHANGED methods to update the values of dependent variables or to recommend chaining to other relevant sub-systems. These methods are attached to defined attributes. A recommended design procedure which defines the order in which sub-systems are accessed is also useful in reducing the need for cross-access between different sub-systems. Consideration was given to whether such an ordered procedure should be incorporated as a necessary procedural use of the expert system, but this was discarded as it would have severely limited the flexibility of use of the system. The structure of these sub-systems is described in detail under the structure of the design process below.

(c) The final portion of the structure is three sets of databases. The first set of databases contains information relating to existing cases, to be accessed as necessary for case based reasoning. The second set of databases consists of all the necessary component information and legal safety requirements. Both of these sets of databases are confined to read only access, primarily for two reasons, both relating to the user requirements. First, it is important that these databases are not altered in any uncontrolled fashion if the expert system is to be assessed over some period of testing and feedback. The development of confidence in its use is dependent on careful monitoring and a detailed study of the implications of any changes. Second, inexperienced young engineers must not be able to introduce any changes into the databases. Senior engineers must at all times control what information is used for the case based reasoning. It is however, important that senior engineers are able to make changes to the database information, to ensure that it reflects their own company's design philosophy and to reflect any changes to legal regulations. All databases thus use the dBASEIII format and can be altered as necessary by the use of the dBASEIII software. The third set of databases contain the current design information. Access to each of the sub-systems results in the current information being updated or further design information being stored in these databases as final results of the session, or to be used by other sub-systems.

KNOWLEDGE ACQUISITION

In acquiring the knowledge it was recognised in particular that there was a need to exclude any bias, whilst allowing incorporation of the different company philosophies, as well as having a complete and reliable knowledge base. The procedural strategy used in knowledge acquisition for the mineshaft equipment problem thus closely followed the fairly standard procedures suggested by Cohn (1988), and thus will not be dealt with in much detail. However, certain adjustments had to be made because of specific difficulties encountered. The most important of these was that different experts were available for the different tasks and all had little time available. There was not one, widely recognised domain expert in any of the domain areas covered by this expert system. Many of these experts also had little or no knowledge of the potential or operation of expert systems. The enthusiasm of the experts thus required some prompting in several instances.

It thus appeared that knowledge acquisition would be very much enhanced by the early production of the skeleton of an expert system. This had three major advantages. The first was that a high level of enthusiasm was generated, mainly from the fact that experts in the area of certain of the tasks could immediately see how their performance could be enhanced by the ability to use the expert systems in the other tasks. The second was the ability, even given the very sketchy outline provided by the use of the skeleton system, for users to assess the benefit to their tasks, of the full expert system, and to gain an understanding of the procedures involved. The third benefit was the clearer assessment of the scope to be included in the expert system, an assessment made possible by seeing the early skeleton system working.

The knowledge acquisition scheme which was developed to address these factors, and used for all modules, was thus:

Identify all relevant sources of knowledge

The three major sources of knowledge for this expert system were published literature, as for example Soutar (1973), Greenway (1990) and Napier (1990), domain experts within the mining industry, and domain experts from support industries, such as rope and hoisting equipment manufacturers. As there was no single recognised domain expert in any of the areas covered, it was regarded as important to interview a wide range of senior engineering experts.

Develop skeleton expert system

A skeleton expert system was developed, primarily using data extracted from published literature. This was found to provide very few complete cases, but most of the important factors were covered to the extent that it was possible to define simple preliminary rule sets. Some published information also related to older mineshafts, so care had to be exercised in recognising what data would have changed by virtue of technological and other developments. It was however possible to develop a skeleton system which was complete enough so as not to appear facile, and which could be used to demonstrate the potential of such a system at the same time as acquiring more detailed data from the domain experts.

Extract full knowledge from various sources

The extraction of knowledge from the domain experts and its conversion into database information and rules is currently being carried out by interview and demonstration of the skeleton system. It has been extremely useful to develop the interviewing procedure on the basis of the most recent domain information, and to have this already included in the skeleton expert system. Concurrently with this interviewing, the mining companies are assembling schedules of detailed numerical information relating to their mines.

Define the full scope of the expert system

The scope of the expert system is being defined dynamically in response to user comments. The sub-system structure has proved to be extremely useful in this regard, allowing a good deal of flexibility.

Completion and Evaluation of Prototype System

The earlier use of a skeleton system in interviews with the various domain experts has facilitated a measure of early evaluation of the knowledge base quality. However, with the full definition of the scope of the expert system its quality and the performance of the expert system is still to be evaluated comprehensively.

THE STRUCTURE OF THE DESIGN PROCESS

Expert systems are best structured to follow the experienced engineer's design process as closely as possible, and as mentioned above it is important that a flexible user

interaction is provided so that the expert system is a tool and not a "black box". Gero (1988) uses the concept of previous known cases, which may be structured into what he calls prototypes, to reflect this process. Thus a prototype may be refined if the current situation is very similar, or it may be adapted if the current case is similar but requires significant changes. A new situation requires the generation of a new prototype.

The structure of design data is modelled on Gero's concept. A complete prototype, or set of design data, is encompassed by several databases, where each database is treated by the expert system as a class, and each record of the database as an instance of that class. Separate databases are used for existing cases and the current design. Four databases are used to form a complete prototype. One is the class containing all the basic information relating to the shaft, such as its size and depth, its transport requirements, the number of skips and cages, and the utilisation of the shaft in terms of days per month and hours per day of operation. The other three contain the conveyance classes, ie information relating to skips which transport rock, cages which transport men and materials and dual purpose conveyances which may be used for the conveyance of either rock or men and materials. The attributes of these classes include the conveyance capacity, the material of which it is made, a definition of its type, its selfmass, the hoisting velocity and loading times and fittings.

The complete design process consists of entering the control module and running each of the design modules from that platform. Each design module adds to, or adjusts, information in the prototype classes contained in the various databases.

Gero's case based approach is followed, but introducing certain modifications, in order to follow the experienced engineer's design process and allow flexibility. The process employed in this expert system is described as five aspects below, with specific reference to the sub-systems for skip design and for rope selection. These are shown schematically in figure 2.

Access to Databases

The design process is initiated by accessing relevant case databases to establish prototype classes or available components.

Thus, for example, the skip design sub-system accesses and searches databases of existing cases to locate a prototype class. A prototype class is located if defined similarity criteria are met. These similarity criteria essentially set upper and lower limits or cut-offs, on one or more user-specified key variables. Thus the user may specify that a similar skip requires that the payload, the shaft depth and the hoisting velocity of the case, must all be within, say, 75% to 150%, of the required values, and that the skip must be of the same type. Should several similar cases be located, a similarity factor will be calculated, by the summation of the value $(1 - \text{Required value}/\text{Case value})$ for all the key variables. The user may then select a preferred case, or use the optimisation sub-system. If a similar case is located, or one of several is selected by the user, it is then refined by the use of a different set of rules to comply with the requirements of the current design.

If no similar case is located, rules are employed to generate a new prototype class. A "new prototype class" as used here, does not necessarily mean conceptually new. It may, for instance, mean that in order to satisfy rules regarding the maximum combined weight of skip plus payload, a new light weight material is used. This would however, only be possible, if the user had previously included such light weight

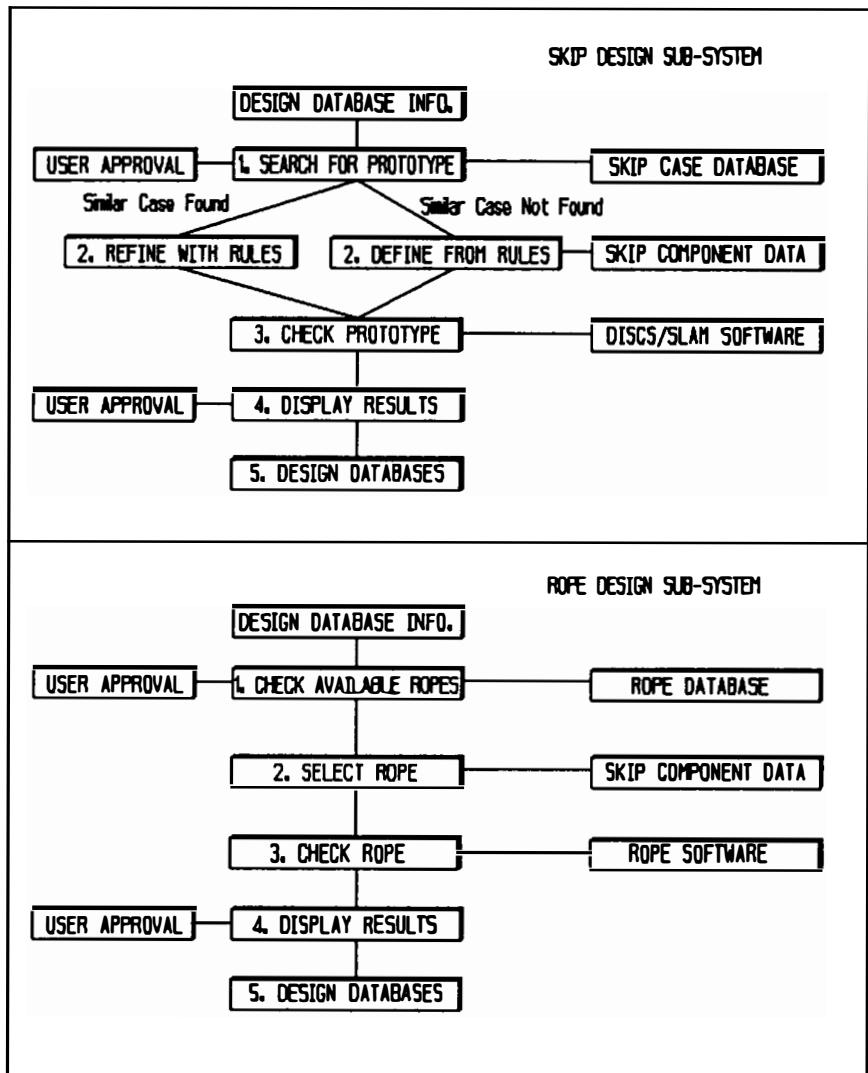


Figure 2: Skip and Rope Design Sub-Systems

material in the materials database, as these rules include searches of other databases to locate appropriate materials and components. Or, it may mean that wear resistant liners are omitted in order to increase the capacity of the skip, but with a concurrent reduction in its life expectancy. This again would only be possible if it was specifically allowed by the user. Thus any "new" aspect of any prototype class, must have been previously defined as a possibility by either the developer or the user of the expert system.

As a further example, the rope selection sub-system searches the database of available ropes, identifying all possible rope types, including information as to where each rope has been successfully and/or unsuccessfully used. A set of rules is then used to establish which rope types are appropriate for use in the current design situation. These rules include the influence of such factors as type of winder used, depth of the shaft and likely corrosion. Appropriate and inappropriate ropes are then indicated to the user, who may then accept or override this categorisation of the ropes. A rope is then selected, such that it has an appropriate type and is sufficiently strong. If no sufficiently strong rope is available, then a different winder type which utilises more ropes, must be specified as an adaptation of the prototype being considered. This may alter the appropriate rope types, and it requires a new search of the ropes database to select a sufficiently strong rope.

Use of Rules Alone

The user is also free, at any time, to specify the use of the rule base in preference to the database search, for the definition of a new prototype class. In this case no reference is made to any of the case database information. The values of all relevant attributes of the prototype class are defined by processing the rule set. The rule sets used include both forward and backward-chaining rules as well as defaults and methods attached to attributes of the various classes.

Optimisation Hierarchy

Within the design there is allowance made for optimisation in some circumstances, notably for material selection. The optimisation used is a single variable optimisation, but within an optimisation hierarchy. Thus the user is requested to define the optimisation goal in terms of minimum weight, minimum cost or minimum maintenance. In the mineshaft environment, certain rules governing which of these criteria is most likely to predominate can be developed. For example, in a deep shaft, the minimum weight criterion is likely to be primary, because if a conveyance is too heavy there may be no sufficiently strong rope. However, provided the weight is below a particular threshold (which is determined by a satisfactory rope size), the minimum maintenance may become primary because production depends on low downtime.

It is thus the intention to introduce this type of optimisation hierarchy, where the weight is optimised, attempting to achieve a threshold value. If it cannot be achieved, no further optimisation takes place. However, if this threshold is achieved, further optimisation is in terms of minimising maintenance. Cost can then become the optimisation variable, if a specified maintenance threshold is attained. The actual basis of optimisation is again user defined.

Dealing with Unknowns

Any required values which are unknown in the context and cannot be established from rules are either requested from the user or obtained from default values within the sub-system. The user may enter requested values if they are known. If they are not known and thus not entered, they will then be established in one of two ways. Certain values may be established in other sub-systems. In this case the current sub-system will chain to the relevant sub-system which will establish the values and then chain back. Other values are established by the use of the default values. These default values are attached to attributes of the prototype classes and are specified in the sub-system, having been obtained from a careful evaluation of current design practice and expert knowledge, as part of the knowledge acquisition process.

Accessing other Software

Once the refined or new prototype is fully defined conceptually, it may sometimes be desirable to access other conventional software in order to more accurately assess the performance of the prototype. The experienced engineer knows when this is necessary and when it is not. The skip design sub-system models this process by a set of rules which define the conditions under which a better knowledge of their rigid-body vibration characteristics is required in order to determine the magnitude of any lateral forces present. If the result of this rule set is positive, then the sub-system runs purpose written software, described by Krige (1986) and Greenway and Thomas (1989). The results of this run are then used in modifying the previous prototype. Similarly, the rope selection sub-system consults a group of rules to determine whether a rope vibration package should be run and its results employed to modify the rope selected or any other parameters.

Results processing

All results within any sub-system are displayed at the end of the session in that sub-system. When they are displayed, they may be accepted by the user, or alternative values may be entered. The accepted or alternative values are then stored in the relevant database as a further part of the emerging new design, and for use by later modules.

CONCLUSIONS

By employing this modular approach, with a fairly large overall expert system domain broken down into several smaller sub-system domains and a control module, it has been possible to develop an expert system for the design of mineshaft equipment, which meets the primary requirements of the intended users.

The expert system allows flexibility of use, so that it is obvious to the user that it is a design tool, and it is not trying to do the engineering. The engineer is thus in control of the design process through all stages, and at all times the progress of the complete design, as well as the current stage, are transparent to the engineer. All final decisions rest with the engineer, who thus clearly bears all responsibility for the design which has been produced.

The expert system also remains expandable, because of its modular nature. Different domain experts are able to add different sub-systems, which can be developed and evaluated with a minimum of interference with any other domain sub-systems.

Where the design philosophies of the different mining companies differ in any way, this can be accommodated by the introduction of databases which reflect their individual practices.

The extent to which the identifiable benefits to the mining companies will be realised, and to which this expert system will be used with confidence as a design tool, cannot be assessed at present, as it is still too early a stage. A very positive reaction has however been forthcoming from the senior engineers who will be the users of the system, and their feeling at this stage is that the benefits and confidence will develop with use of the system.

REFERENCES

- Cohn L.F., Harris R.A. and Bowlby W. (1988). Knowledge Acquisition for Domain Experts, *J. of Computing in Civ. Eng.*, Vol 2 No 2, pp.107-120.
- Garrett J.H. (1990). Knowledge-Based Expert Systems: Past, Present and Future, *IABSE Periodica* 3/1990, Zurich.
- Gero J.S., Maher M.L. and Zhang W. (1988). Chunking Structural Design Knowledge as Prototypes, in Gero J.S. (ed), *Artificial Intelligence in Engineering: Design*, Elsevier, Amsterdam, pp.3-21.
- Greenway M.E. (1990). An Engineering Evaluation of the Limits to Hoisting from Great Depth, *International Deep Mining Conference: Technical Challenges in Deep Level Mining*, SAIMM, Johannesburg, pp.449-481.
- Krige G.J. (1986). Some Initial Findings on The Behaviour and Design of Mineshaft Steelwork and Conveyances, *JSAIMM*, vol 86, pp.205-215.
- LEVEL5 OBJECT (1990). Object-Oriented Expert System for Microsoft Windows, Information Builders Inc, New York.
- Napier L.G.D. and Stones D.W. (1990). Men and Materials Handling in a 2700 m Deep Shaft System, *International Deep Mining Conference: Technical Challenges in Deep Level Mining*, SAIMM, Johannesburg, pp.845-862.
- Ortolano L. and Perman C.D. (1987). Software for Expert Systems Development, *J. Computing in Civ. Eng.*, Vol 1 No 4, pp.225-240.
- Soutar B.N. (1973). Ore Handling Arrangements in Shafts of the Republic of South Africa, *International Conference on Hoisting - Men, Materials, Minerals*, SAIME, Johannesburg, pp.54-84.

- Thomas G.R. and Greenway M.E. (1989). Shaft Steelwork and Conveyance Dynamics, *International Conference on Hoisting of Men, Materials and Minerals*, pp.1156-1180.
- Touwen F.H., Agnew C.L. and Joughin N.C. (1973). Computer Simulation of Hoisting Operations, *International Conference on Hoisting - Men, Materials, Minerals*, SAIME, Johannesburg, pp.51-53.
- Wang J. and Howard H.C. (1988). Design-dependent Knowledge for Structural Engineering Design, *Artificial Intelligence in Engineering: Design*, Elsevier, Amsterdam, pp.267-278.

Intelligent real time design: application to prototype selection

S. R. Bradley and A. M. Agogino

Department of Mechanical Engineering
University of California at Berkeley
Berkeley CA 94720 USA

Abstract. An intelligent real time problem solving (IRTPS) methodology is developed for design applications. The goal of the approach is to select sequences of actions that balance the cost of the limited resources consumed during the design process, particularly the designer's time, against the benefit to be derived from the utilization of those resources in terms of expectations of an improved design. Examples from design prototype selection are presented to clarify the theory.

INTRODUCTION

Intelligent Real Time Problem Solving (IRTPS) is an emerging area of Artificial Intelligence (AI) research that addresses problems which require balancing limited time or computational resources against the resulting quality of the decision made or action taken. Erman et al. (1990) defines an RTPS system as: *an intelligent system that is sensitive to real time constraints on the utility of its behavior*. Such systems attempt to perform AI tasks in the face of real time resource limitations and employ dynamic strategies for coping with such limitations during the course of computation or reasoning. Some examples of RTPS applications currently under investigation include:

Game playing. For games such as chess or backgammon, the game tree defines an enormous search space which a player can hope to only partially explore in the finite amount of time available. The problem facing a player is then to determine that part of the space to search that yields the most useful information about the best move to be made, so that limited search time may be focused on exploring that part (Russell and Wefald, 1989).

Medical diagnosis and treatment. Doctors are often called upon to make time critical decisions under conditions of uncertainty. Horvitz et al. (1989) explore the tradeoff between taking more time to perform diagnostic inference, which might yield a more appropriate course of treatment, versus the risk to the patient that the delay might cause. The RTPS problem is then to determine the appropriate amount of diagnostic reasoning to perform before making a decision. Hayes-Roth et al. (1989) present the Guardian system under development to perform real time monitoring and control of life support systems, where limited computational resources must be allocated between processing incoming

sensor data, performing diagnostic inference and selecting treatment strategies in a manner that yields the most appropriate life support control actions for the patient.

Other IRTPS problems under investigation include resource allocation and planning (Fehling and Breese, 1988, Dean and Boddy, 1988), state space search (Hansson and Mayer, 1989a, 1989b), and machine monitoring and control (Agogino 1989, 1990).

Simon (1955) introduced the concept of *bounded rationality* to describe practical resource limitations to rational human decision making, limitations which apply equally well to automated intelligence (Agogino, 1989). In Simon's words: *some constraints that must be taken as given in an optimization problem may be physiological and psychological limitations of the organism (biologically defined) itself. For example, the maximum speed at which an organism can move establishes a boundary on the set of its available behavior alternatives. Similarly, limits on computational capacity may be important constraints entering into the definition of rational choice under particular circumstances.* Good (1971) makes the distinction between "type I rationality," the maximization of expected utility, and "type II rationality," the maximization of expected utility less the cost of "deliberation." Horvitz et al. (1989) report that Good (1968) was the first to propose a decision-analytic approach to controlling reasoning to achieve "type II rationality."

There are a number of approaches to addressing IRTPS problems, including domain dependent approaches which use heuristics to develop a computational strategy that best utilizes the available resources, and decision-analytic approaches. Heuristic approaches are limited to applications where the decision making context is well defined, and expertise is available which can provide the needed heuristics. The decision-analytic approach does not suffer from these limitations, but is often mathematically and computationally too complex for real time applications. The decision-analytic approach may be distinguished from the heuristic and other domain dependent approaches in that it requires reasoning explicitly about the value of time or computational actions in the same, uniform manner as value of the "object level" actions of the problem domain. It is distinguished from the satisficing approach (Hayes-Roth, 1990) in that no pre-defined acceptable level of utility or resource consumption need be specified, although hard constraints can be. This paper explores the application of the real time intelligence perspective to the development of computational models for the design process using a decision-analytic approach to the control of design reasoning.

Design problems, in general, are characterized by a very large space (potentially infinite) of possible solutions. In addition, when attempting to assess how "good" a design is, the designer is presented with the problem of trying to predict the value to the customer of an, as yet, nonexistent device. Because exhaustive search of the space of possible designs and exact characterization of their behavior when built are not possible, designers must cope with incomplete and uncertain information and make the best possible decisions under these circumstances. In the IRTPS approach to controlling design reasoning, the goal is to focus the designer's limited resources on searching those portions of the space which yield the information most relevant to selecting the best design, and eliminating or reducing uncertainty where it has the greatest impact on the utility of the final design developed. The first section of this paper elaborates on that portion of design which can be modeled as a search process, and clarifies the nature of the resource constraints facing designers. One

type of design search decision, prototype selection, is also defined. The remainder of the paper then develops a decision-analytic approach to prototype selection. The final section discusses conclusions and future research directions.

MODEL OF THE ROUTINE DESIGN PROCESS AS SEARCH

Much of the routine design process may be modeled as search through a space of possible designs, as illustrated in Figure 1. In this model, design commences in an initial state described by some set of requirements, or constraints and objectives for the artifact to be designed, and some set of initial commitments to the form of the artifact, such as, for example, interface features that must be present. States are transformed to new states by making additional commitments that progressively constrain the design form, such as, for example, "the artifact will be a beam," or "the length of the beam will be 3.0 m." Note that some states admit an infinite number of possible successor states, since the available commitments may include the selection of values for one or more continuous variables. The goal of the search process is to find the fully specified artifact (an artifact which is sufficiently specified by the design commitments that a plan for its production can be developed) with the highest expected utility to the design stakeholder, which we will assume in this paper is the customer (but could be anyone on whose behalf the design is being developed). The objectives supplied in the initial requirements are an attempt to define criteria which describe such a utility maximizing design. We rely on the *expected* utility to the customer because it is impossible to predict with certainty the utility of the, as yet, unbuilt artifact.

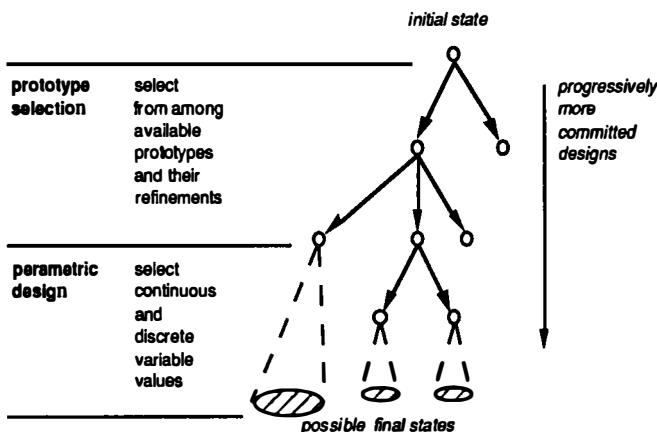


Figure 1. Routine design as search

Each search state in Figure 1 may be assigned a search value which represents the highest expected utility of any successor state. The goal in searching the space is then to

determine a sequence of commitments which leads to the design with the highest expected utility, or, for each state, to select the successor with the highest search value. The search value of the direct successor states of the initial state cannot be evaluated without searching the entire space, which is typically an intractable task. Thus, some means of computing an estimate of the search value of successor states must be employed. There is a tradeoff between the quality of the search value estimates computed and the quality of the decision made as to which design commitment to make. We will assume that developing such search value estimates can be broken down into a number of computations, each of which provides information that contributes to the accuracy of the estimate. The designer is thus presented with the choice of making a design commitment or performing more computation to improve the quality of the estimates, where performing more computation increases the *design process costs*. This process of searching the tree of Figure 1 can then be represented by a flowchart as shown in Figure 2.

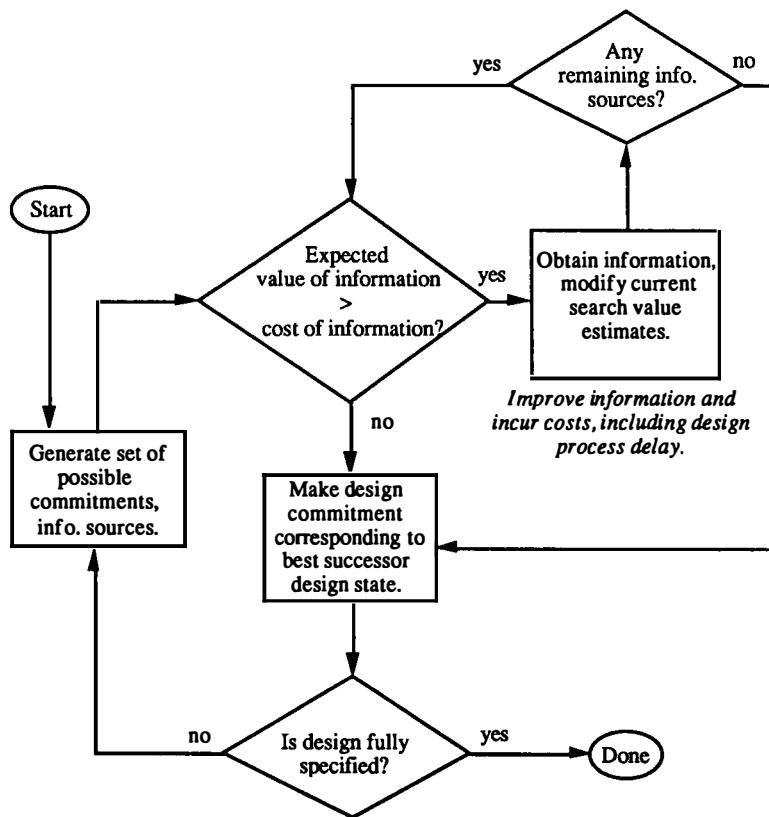


Figure 2. Design decision making under resource constraints

So far we have said nothing about how the set of possible commitments for a given design state can be derived. The model assumed in this paper is shown in Figure 3, and is

adapted from Gero et al. (1988) and Jain and Agogino (1990). Requirements are used to select candidate *prototypes*. Gero et al. (1988) define a prototype as “a conceptual schema for the representation of generalized design knowledge” that unifies an intended interpretation, a vocabulary or design elements, and knowledge. A prototype represents a potentially infinite variety of possible artifacts or design entities; for example, a prototype representing a beam captures the commonality of all beams with an infinite variety of possible cross sections, lengths, materials, and so on. Prototypes participate in a hierarchy, and may be *refined* to develop more committed designs. From each prototype an associated parametrized design description can be developed. Some iteration between the parametric design phase and prototype selection/refinement may be necessary. The final parametrized design is then used in the detailed design phase to make commitments to specific design variables. At any point in this process the designer may determine that it will not be possible to refine the original prototype into an acceptable product and new prototypes will need to be generated through some innovative or creative process.

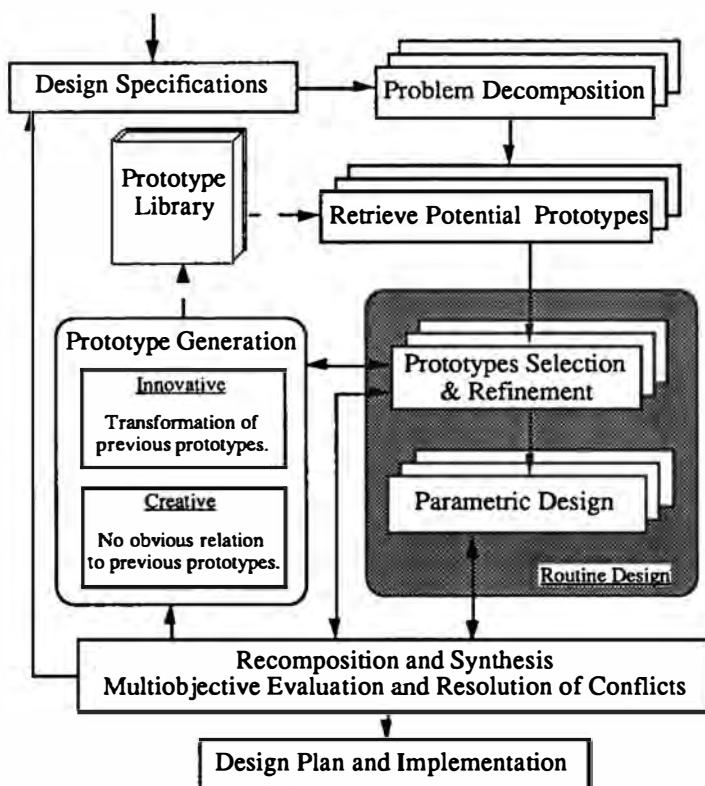


Figure 3. Role of prototype selection in the design process

For the problem of designing a mechanical element to transmit power from one shaft to another, examples of these tasks might be: searching for all known means of transmitting rotational motion and enumerating these; selecting, say, a belt drive system from the possible options; and sizing the belt and sheaves appropriately. By using decomposition or abstraction, the performance of these tasks may be interleaved, so there need not be a strict ordering or temporal separation of the tasks, but the tasks must, in some fashion, be performed.

This paper deals with one part of this search process, *prototype selection*. Gero et al. (1988) describe how prototypes may be selected based on whether they possess certain attributes or have attribute values within certain specified ranges. Such methods "weed out" those prototypes that are of no possible utility in the current design context. For example, in the domain of power transmission elements, one design requirement might be that the artifact to be designed transmits power between two parallel shafts. This requirement would rule out a number of classes of devices, or prototypes (such as a bevel gear pair, a rack and pinion, or a slider-crank) but would still admit several solutions (a spur gear pair, a V-belt drive, a cable drive, etc.). These remaining prototype options represent the next level in our search tree. A domain independent approach to efficiently select from among such alternatives is the topic of the remainder of this paper.

In performing prototype selection the search value of the alternative prototypes available must be estimated, where the search value in the prototype selection problem is defined here to be the expected utility of the designed artifact that will result from selection of that prototype. To perform such estimation requires "looking ahead" in the search space of possible designs that might be developed from that prototype. In the process of estimating the expected utility of a prototype option it is therefore possible (but not necessary) that other optimal design commitments (choices of refinements of the prototype or design variable choices) might be determined. Thus, it is often advantageous to perform prototype selection and some part of parametric design simultaneously (shaded area of the design process in Figure 3). This point will be elaborated further below.

A DECISION-ANALYTIC APPROACH TO IRTPS FOR PROTOTYPE SELECTION

Figure 4a shows a simplified influence diagram that describes the prototype selection problem. The square nodes in such a diagram represent *decisions*, the ovals *variables*, the rounded rectangle the *goal* or expected value to be optimized, and the arcs *influences*. During prototype selection, one goal might be the maximization of the utility of the designed artifact to the customer. This utility is influenced by the values of a number of design attributes. For example, returning to the problem of designing a mechanical element to transmit power between two parallel shafts, the utility of the designed artifact will be influenced by such design attributes as cost, usable life, load capacity and inertia. The values of these attributes will in turn be influenced by the prototype selection made; the same combinations of cost, life, load capacity and inertia will not be possible with a pair of gears as with a V-belt drive system, for example.

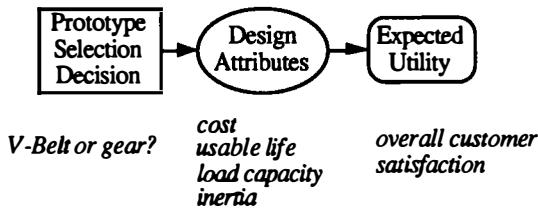


Figure 4a. Influence diagram for the prototype selection problem (power transmission example).

At the time a prototype is selected the exact value of the design attributes and their impact on utility will not be known precisely, motivating the use of a multivalued calculus, such as probability (Siddall, 1983) or fuzzy theory (Wood, 1989). The degree of certainty of the values of the design attributes will depend on the amount of effort the designer spends on modeling, analyzing, prototyping (in the fabrication sense) and testing the design. Even if the design attributes for competing designs were known with certainty (for example, the devices had been built, and their values measured) the value or utility to the customer would likely still not be known precisely. The extent of this uncertainty will depend on the amount of effort expended by the design and marketing team to model the preferences of the customer.

Thus there is a trade-off between the benefits of obtaining a certain level of precision in understanding the implications of prototype options and the design process costs required to obtain this precision, such as labor, computation, and time-to-market (Figure 4b). It is often not advantageous for the designer to refine and define a prototype option to the point that each of the design variables that are under the designer's control are precisely modeled and optimized before making a design decision. Approximate models for prototype evaluation can often be cost effective, including use of simplistic handbook guidelines (Rothbart, 1985) and knowledge-based programs (Mittal and Araya, 1986, Waldron et al., 1986, Brown and Chandrasekaran, 1989), each with its own merits and relative level of precision.

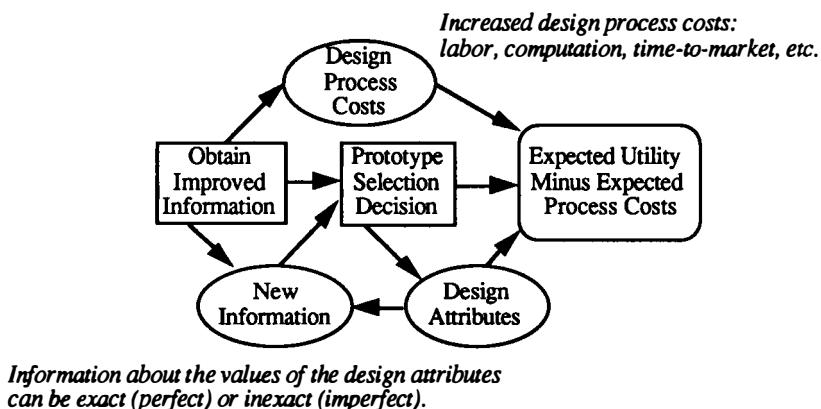


Figure 4b. Influence diagram for the prototype selection problem with option to "buy" information

Thus design not only involves decisions concerning the design specifications themselves but also meta-decisions concerning the design of the design, or decision process. Adding the cost of process to a designer's decision responsibilities places design into the realm of intelligent real time problem solving. The designer is faced with making two types of decisions, as shown in Figure 4.b; the designer must decide on the appropriate amount of information to acquire and the best prototype selection to make. In this paper we model uncertainty and imprecision with probability theory, taking a decision-analytic approach.

Let us represent the utility of a design developed with prototype option i as U_i , and the maximum possible utility for any design developed with that prototype (the "prototype utility") as U_i^* . As noted above, it is typically not possible to determine this value with certainty, and it will therefore have an associated probability distribution, $\text{pr}(U_i^*)$, as illustrated for two prototype options in Figure 5. It is not required that the statistical probability of the prototype utility be obtained through repeated experimentation; rather, the designer must only be capable of expressing the current state of belief with regard to the prototype utility value in terms of a probability distribution (see (Pratt et al., 1965) or (Raiffa, 1968) for a discussion of subjective probability estimation).

The designer's goal is to select the prototype option that will lead to the design with the highest expected utility, that is, the option k such that:

$$k = \underset{i}{\operatorname{argmax}} E[U_i] \quad (1)$$

Where the "argmax" function returns the value in the domain of i that maximizes its argument, and $E[\cdot]$ is the expected value operator. Figure 5 shows the expected utilities for two prototype options, U_1^* and U_2^* , where $U_i^* = E[U_i]$.

In performing design with limited resources, the designer seeks to acquire exactly that amount of information that maximizes the difference between the utility of the design developed with the benefit of the information and the cost of acquiring such information. More information results in a more accurate assessment of $\text{pr}(U_i^*)$ and therefore $E[U_i^*]$ for some i , and thus a better or more informed choice of the best option k . Referring to Figure 2, the designer is typically presented with the following alternative actions:

- Develop at some cost (in time or some other measure of resources) a more accurate assessment of $\text{pr}(U_i^*)$ for some i .
- Go ahead and select that option i that appears best in the present state of information.

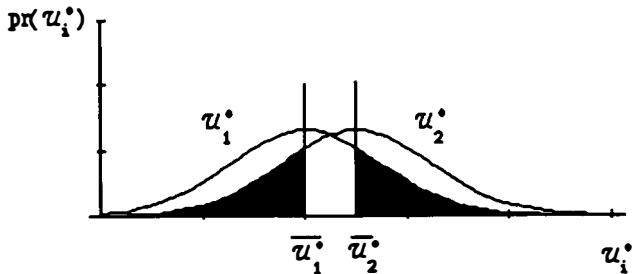


Figure 5. Example prototype utility distributions (adapted from Russell and Wefald, 1989)

Figure 6 gives us an intuitive sense of when it is worthwhile for the designer to acquire more information. In the situation in Figure 6a, it is highly unlikely that the utility of option 1 will exceed that of option 2, and therefore further information will be of little value. In Figure 6b, option 1 and 2 appear to be so nearly equivalent and the probability that they will prove significantly different in utility is so small that further effort spent trying to distinguish the utility of one from the other would be wasted. Figure 6c depicts the situation where further information will be valuable; there is substantial uncertainty as to whether one prototype option is significantly worse or better than the other.

To develop the optimal design given resource limitations the designer must determine the *sequence* of actions that maximizes the difference between the expected utility of the design developed after performing the sequence, and the expected cost of performing that sequence. After each action is taken, the designer must consider all possible sequences that might unfold in the future, and choose the next action that is expected to be the first step in the sequence with the highest utility. In this paper we will greatly simplify the problem by considering only individual actions, and not sequences, and assume that the sequences that result from selecting actions one at a time are a reasonably close approximation to an optimal sequence. This assumption has been termed the “one step horizon assumption” (Pearl, 1988) or “single step assumption” (Russell and Wefald, 1989).

Qualitatively, the decision rule we will use for selecting an appropriate action will be: select that design option that appears best given the present state of information if the cost of acquiring more information would outweigh the expected benefit to be had using that information; otherwise, “buy” that information whose expected value most exceeds its cost. Here, “benefit” means improvement in the expected utility the chosen option. For prototype selection, then, we define the expected value of information, E , as:

$$E = \text{the expected utility of the prototype option chosen with information} \\ \text{minus the expected utility of the prototype chosen without information.} \quad (2)$$

We will choose to “purchase”, or acquire information, only when

$$E > C \quad (3)$$

where C is the expected cost of such information. This expected cost is the price we expect to have to pay in order to acquire the information, measured in the same units in which we

are measuring utility. Several IRTPS researchers (Russell and Wefald, 1989, Horvitz et al., 1989) have discussed how one might determine this cost when the decision making agent is a computer program operating under time constraints.

If more than one information source is available, we will select that with the largest value for $E - C$, that is, the action k given by:

$$k = \underset{j}{\operatorname{argmax}} (E_j - C_j) \quad (4)$$

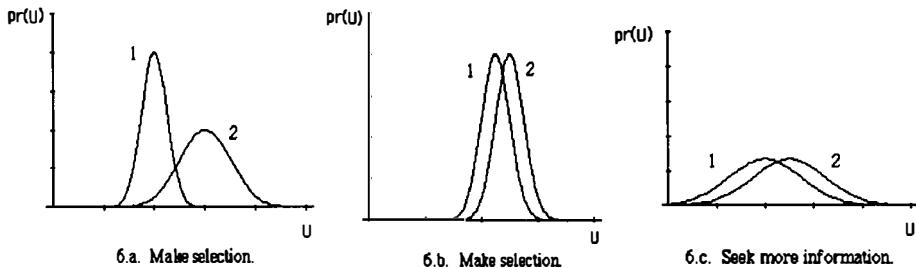


Figure 6. Three prototype selection situations (adapted from Russell and Wefald, 1989)

Figure 5 shows the probability distributions for the utility of two prototype options. As may be noted from the figure, $\bar{U}_2^* > \bar{U}_1^*$, and therefore, with no further information, the designer would choose prototype 2. Let us suppose that, for a cost C_1 or C_2 , the exact value of U_1^* or U_2^* could be determined. What would this information be worth to the designer? Let us look at the value of precise information of U_1^* , E_1 . If the designer were told $U_1^* < \bar{U}_2^*$, the designer would select prototype 2 as before, and nothing would have been gained by acquiring the estimate. This is an example of the general principle that a piece of information is worthless if it cannot change the decision or course of action that will be taken. However, if the designer were told $U_1^* > \bar{U}_2^*$, the designer would choose prototype 1 over 2, with an improvement in utility of $U_1^* - \bar{U}_2^*$. The values of U_1^* for which the designer expects this to be the case in the present state of information (before purchasing the estimate) are shown by the rightmost shaded region in Figure 5. Thus, surveying all the possible values of U_1^* and their probabilities, we conclude that E_1 is given by:

$$E_1 = \int_{\bar{U}_2^*}^{\infty} pr(U_1^*) (U_1^* - \bar{U}_2^*) dU_1^* \quad (5)$$

By similar reasoning, it can be shown that the value of precise knowledge of U_2^* , E_2 , is given by:

$$E_2 = \int_{-\infty}^{\overline{u_1^*}} \text{pr}(u_2) (\overline{u_1} \cdot u_2) du_2 \quad (6)$$

which is the integral over the leftmost shaded region of Figure 5.

Equations 5 and 6 give the *expected value of perfect information* (EVPI) (Howard, 1966). Any estimate of U_1^* or U_2^* which might actually be acquired will never exceed E_1 or E_2 in value. The more informative the estimate is, the nearer its value approaches EVPI. EVPI is useful in decision making situations in which the information that is available has much less uncertainty than the present state of information. More mathematically complex expressions for the expected value of (imperfect) information (EVI) may be derived (Bradley and Agogino, 1990) and, for a limited class of estimates, tractable methods have been identified to evaluate EVI (Russell and Wefald, 1989). The following example illustrates how equations (5) and (6) can be applied to design decision making.

Example: Gears or Belts?

Let us suppose that an agent is designing an element to transmit power between two shafts in a machine of which 100 units will be built. Suppose the agent has found two candidate prototypes in its library, a gear pair and a belt drive, and has used empirical relationships to estimate the probability density function for the prototype utility in monetary units for the two options available, as shown in Figure 7. Beta distributions were used to describe this initial state of information (the distribution parameters are $q = r = 3$ for the belt distribution, $q = 2, r = 4$ for the gear distribution; see (Siddall, 1983) for definitions of the parameters). Given these distributions, the expected value of the cost for each option may be determined (see (Siddall, 1983) for an analytical expression for the mean value of a beta distribution), giving $U_{\text{belt}} = \$550$ for the expected value of the utility per unit for the belt design and $U_{\text{gear}} = \$450$ for the gear design. Thus, without further information, the belt option appears superior.

If the design agent estimates that a more accurate preliminary design and utility estimate can be performed for an amount of design process time worth $C_{\text{gear}} = \$400$ for the gear design and $C_{\text{belt}} = \$300$ for the belt design, and assuming that these utility estimates will have little uncertainty compared to our present information, should the agent go ahead and select the belt option, or perform one of the utility estimations?

Figure 8 presents the problem in terms of a *decision flow diagram* (Raiffa, 1968). The square nodes in the tree represent decisions to be taken by the designer, while the circular nodes represent chance events. In this case, the initial decision facing the designer is to choose between the four alternative options shown in the figure; select the gear option, select the belt option, or purchase an estimate for the utility of either of the two options for the price indicated in the figure. If one of the estimates is purchased, "chance" decides which value for the estimate will be returned; this could be any of an infinite number of values, as represented by the arc extending from the chance node. After the estimate has been

revealed, the designer is once again faced with selecting the belt or gear option, as shown by the decision nodes that follow the chance nodes.

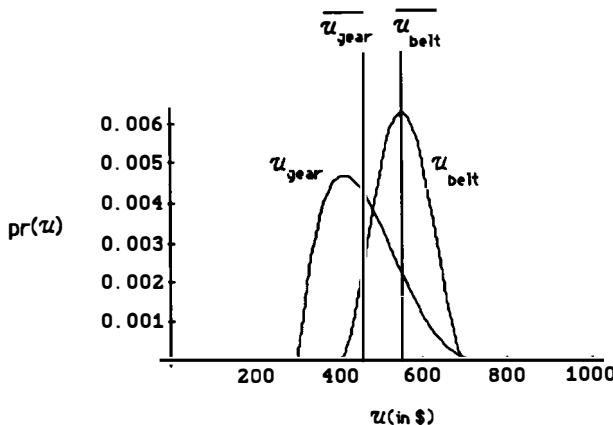


Figure 7. Probability density functions for the utility of the gear and belt design options

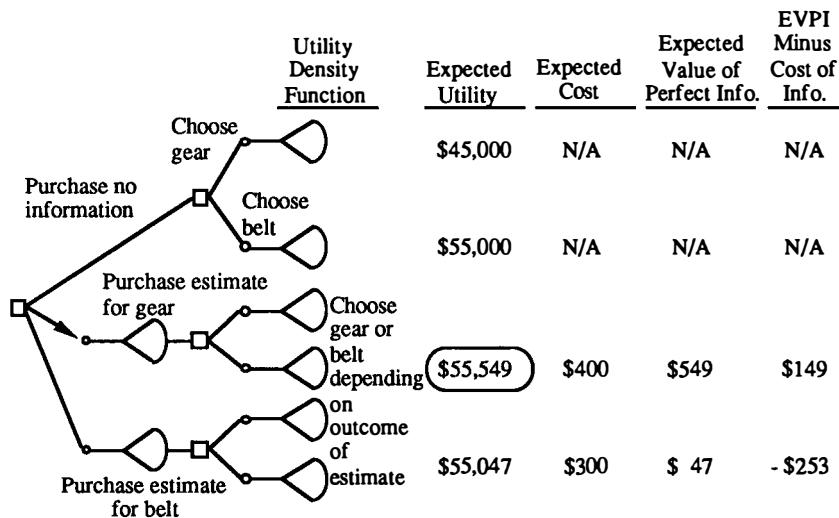


Figure 8. Decision flow diagram for the belt vs. gear prototype selection problem
(With no information, choose the belt prototype. Optimal choice is to purchase information, estimate for the gear utility, and choose the belt or gear prototype accordingly).

Since we expect each cost estimate to be quite accurate, the expected value of perfect information (equations 5 and 6) will give a good measure of the value of each estimate. Applying equations (5) and (6) to determine the value of perfect information of the cost of the gear and belt drives gives:

$$E_{\text{belt}} = 100 \int_{-\infty}^{\bar{U}_{\text{gear}}} \text{pr}(U_{\text{belt}}) (\bar{U}_{\text{gear}} - U_{\text{belt}}) dU_{\text{belt}} = \$ 46.9 \quad (7)$$

$$E_{\text{gear}} = 100 \int_{-\infty}^{\bar{U}_{\text{bear}}} \text{pr}(U_{\text{gear}}) (U_{\text{gear}} - \bar{U}_{\text{bear}}) dU_{\text{gear}} = \$ 549.2 \quad (8)$$

The above definite integrals were evaluated using Mathematica™. Given these results, it would appear that the next action the designer should take is to determine the cost estimate for the gear, since $E_{\text{gear}} - C_{\text{gear}} = \$149.2 > 0$, while $E_{\text{belt}} - C_{\text{belt}} = -253.1 < 0$. The expected utilities of the various possible outcomes from which the designer may select are summarized in Figure 8.

PROTOTYPE SELECTION WITH AVAILABLE PARAMETER ESTIMATES

So far, we have investigated prototype selection when estimates of the utilities of the prototype options are available. In this section, we investigate the common situation in which the utility estimates developed are functions of some random parameters p , and some portion of the prototype design variables x . In this case, prototype selection may involve partial selection of the design variables as well. Let us represent this utility function as $U_i(p, x)$ for prototype option i , where U is a deterministic function of the random parameters p and variables x . All of the uncertainty in our estimation of U_i is then due to that associated with the parameters p . The domain of the design variable vector x may be constrained, in which case we will assume that suitable mathematical constraints in terms of x and p can be formulated. Below, whenever maximization with respect to x is specified, we will assume that such constraints are considered without actually writing them out.

If there are "n" prototype options available, and the designer is powerless to change the uncertainty in the parameters p , the optimal solution would be the option i for which the expected value of the utility is a maximum. The expected value of the utility of an option is dependent on the order in which decisions will be made, and we will treat two cases here:

Full commitment case: The designer will select not only the prototype option, but the values of the design variables "now," that is, perform parametric design given the current state of information as well. In this case, the designer should choose the option i^* such that:

$$i^* = \underset{i}{\operatorname{argmax}} \left\{ \underset{x}{\max} \left\langle \underset{p}{E} [U_i(p, x)] \right\rangle \right\} \quad (9)$$

$E_p [\cdot]$
where E_p is the expectation taken over the random parameters p . Solution methods for mathematical programming problems of the form:

$$\max_x \left\{ E_p [\max_x U_i(p, x)] \right\} \quad (10)$$

have been treated by a number of authors (e.g. Siddall (1982, 1983)).

Partial commitment case: In this case, the designer is committing to the choice of design option only, and will select the design variables at some time in the future when the parameters will be known with relative certainty (note that, in the case that the parameters will not be known with any greater certainty at some time in the future before the design variables are selected, the designer gains nothing by postponing the selection of the design variables, and the full commitment case is the logical treatment for the problem). We will explore the case in which the parameters will be known with certainty at the time that parametric design takes place. For this partial commitment case, the designer should select the option i^* such that:

$$i^* = \arg \max_i \left\{ E_p [\max_x U_i(p, x)] \right\} \quad (11)$$

This case arises when performing “back of the envelope” calculations during conceptual design. For example, during the conceptual design of a satellite, when deciding whether to use a solar or some other power system, a designer might estimate, say, the required solar collector area in order to estimate the cost of this option, but would not wish to commit to this estimated value, rather preferring to wait until more precise information about the required power consumption were available. The distinction between the cases is then whether the designer will commit only to a particular successor prototype at this time (partial commitment), or will commit the design to some subspace of the space defined by the design variables (full commitment to prototype selection/refinement and some portion of one of the shaded regions in the search representation of Figure 1).

Returning to the problem with limited resources where estimates of the parameter values are available for purchase, we would like to develop expressions for the relationship (2) for each of the two cases examined in equations (9) and (11) above. We will assume that information is available in the form of a parameter estimate of some parameter p_j , \hat{p}_j , whose “accuracy” is characterized by the conditional probability $Pr(\hat{p}_j | p_j)$, the probability the estimate will have a particular value given the true value of the parameter being estimated. For the full commitment case, the relationship (2) can be shown to be:

$$E(\hat{p}_j) = \int_{\hat{p}_{j1}}^{\hat{p}_{j2}} pr(\hat{p}_j) \left\{ \max_{i, x} \int pr(q) pr(p_j | \hat{p}_j) u_i(p, x) dp \right\} d\hat{p}_j - \max_{i, x} \frac{E}{p} [u_i(p, x)] \quad (12)$$

while, for the partial commitment case, (2) becomes:

$$E(\hat{p}_j) = \int_{\hat{p}_{j1}}^{\hat{p}_{j2}} pr(\hat{p}_j) \left\{ \max_i \int pr(q) pr(p_j | \hat{p}_j) \max_x u_i(p, x) dp \right\} d\hat{p}_j - \max_i \frac{E}{p} [\max_x u_i(p, x)] \quad (13)$$

where

$$\hat{p}_{j1} \leq \hat{p}_j \leq \hat{p}_{j2} \quad (14)$$

$$p_{j1} \leq p_j \leq p_{j2} \quad (15)$$

$$pr(\hat{p}_j) = \int_{p_{j1}}^{p_{j2}} pr(\hat{p}_j | p_j) pr(p_j) dp_j \quad (16)$$

$$pr(p_j | \hat{p}_j) = \frac{pr(\hat{p}_j | p_j) pr(p_j)}{pr(\hat{p}_j)} \quad (17)$$

$$q^T = [p_1, p_2, \dots, p_{j-1}, p_{j+1}, \dots, p_m] \quad (18)$$

Note that the above derivation assumes each parameter p_i is independent of an estimate of another parameter j , that is

$$pr(p_i) = pr(p_i | \hat{p}_j), \quad i \neq j \quad (19)$$

(although it would present no difficulty to modify the above equations for the case when this were not so). Equations (12) and (13) are difficult to apply to nontrivial problems. Because of the computational complexity of the above expressions, it is often attractive to use the expected value of perfect information, $E(p_j)$, as a metric for making decisions. As long as the uncertainty of the estimate being purchased, as described by $pr(\hat{p}_j | p_j)$, is much less than the uncertainty in the current state of information characterized by $pr(p_j)$, $E(p_j)$ will be a reasonable approximation to $E(\hat{p}_j)$. The formulas for $E(p_j)$ may be obtained from those for $E(\hat{p}_j)$ by setting $pr(\hat{p}_j | p_j)$ to the unit impulse function, giving, for (12),

$$E(p_j) = \int_{p_{j1}}^{p_{j2}} pr(p_j) \left\{ \max_{i, x} \left(E_q [u_i(p, x)] \right) \right\} dp_j - \max_{i, x} \left(E_p [u_i(p, x)] \right) \quad (20)$$

and for (13),

$$E(p_j) = \int_{p_{j1}}^{p_{j2}} pr(p_j) \left(\max_i \frac{E}{q} \left[\max_x u_i(p, x) \right] \right) dp_j - \max_i \frac{E}{p} \left[\max_x u_i(p, x) \right] \quad (21)$$

As long as the total number of variables and random parameters considered in the problem is not excessive and the utility functions are not too cumbersome (do not require extensive mathematical optimization), the integrals in equations (20) and (21) can be evaluated numerically. For more complex utility functions with associated constraints, a local approximation to the constrained utility function may be used to simplify computation. A particularly simple local approximation may be developed using sensitivity analysis (Vanderplaats, 1984, Luenberger, 1984).

Applying equation (20) or (21) in conjunction with the decision rule of equations (3) and (4) provides a domain independent basis for allocating computational resources in a design automation system, directing effort toward computing more accurately only those parameters that may have significant impact on decisions effecting the utility of the final design. The following simple example illustrates the application of equation (21) to a prototype selection problem.

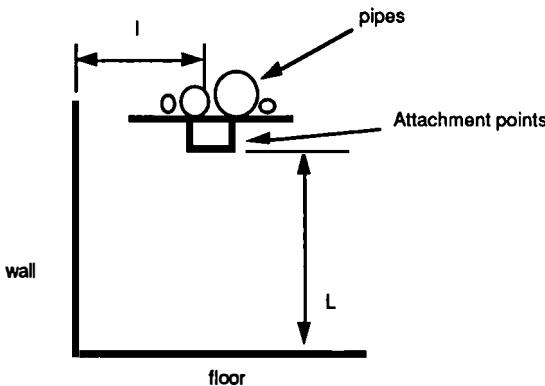


Figure 9. Geometry of the beam vs. column problem

Example: Pipe Rack Design

Ueno et al. (1987) discuss the problem of selecting chemical plant piping support configurations, and note the importance in this problem of choosing the correct level of accuracy at which to analyze the attributes of each of the competing alternatives. Let us explore a much simplified version of the problem they describe to demonstrate how information value theory may be used to select the appropriate level of analysis detail.

The problem we will explore is that of supporting a structure, a bundle of pipes, that is above ground and adjacent to a wall (see Figure 9). Suppose that the piping will be

supported at 20 attachment points. At this point in the conceptual design of the structure, the mass of the pipe, the deflection it can tolerate, and the precise location of its center of mass are uncertain, but will be known accurately before the final design of its support will be developed. Our design agent would like to develop a conceptual design for the pipe support with the currently available information. This is the “partial commitment case,” where the parameter values will be known before the final values for the design variables are selected, but are not known with certainty at the time the prototype must be selected. Two possible design options present themselves; the use of cantilevered beams attached to the wall, or the use of columns attached to the floor. Both options would be fabricated from round steel rod with a yield strength $S_y = 720 \text{ MPa}$, density $\rho = 76.5 \text{ kN/m}^3$, and modulus $E = 207 \text{ GPa}$. The designer assesses the probability distributions for the location of the load specified by the two dimensions L and l , the weight of the piping per loading point P , and the tolerable deflection δ_{\max} as shown in Figure 10. We assume that these parameters are independent. The utility of the design will be maximized when the weight is minimized and failure does not occur. Should the designer choose the beam option, the column option, or incur a cost associated with performing the necessary calculations to determine a better estimate of the location, mass, or tolerable deflection of the pipe? Each Newton of weight savings is worth \$1.

The mass of the beam option shown in Figure 11a is given by its volume times its density, or:

$$W_b = \pi r^2 l \rho \quad (22)$$

where r is its radius. In order for the beam to not yield under loading or deflect excessively, it must satisfy the following conditions:

$$S_y \geq \frac{l P r}{I}, \quad \delta_{\max} \geq \frac{P l^3}{3EI} \quad (23), (24)$$

where the cross sectional moment of inertia is given by:

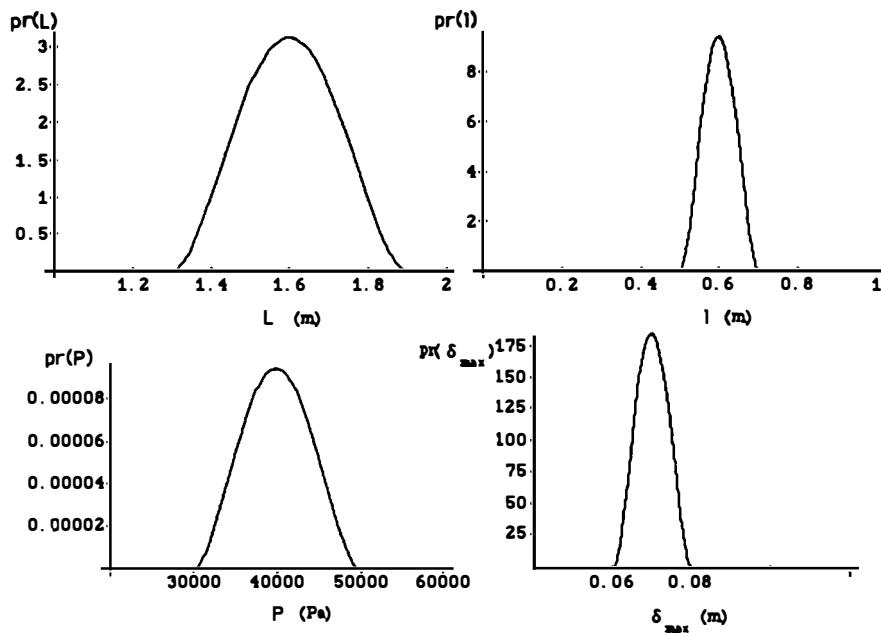
$$I = \frac{\pi r^4}{4} \quad (25)$$

Similarly, for the column of radius R shown in Figure 11b, its weight is given by:

$$W_c = \pi R^2 L \rho \quad (26)$$

where the column should not yield, deflect excessively, or buckle, and must therefore satisfy:

$$S_y \geq \sigma_a, \quad \delta_{\max} \geq \frac{P L}{\pi r^2 E}, \quad \frac{E \pi^2 R^2}{K L^2} \geq \sigma_a \quad (27), (28), (29)$$



All beta distributions, $r = q = 3$

Figure 10. Probability distributions for the random parameters

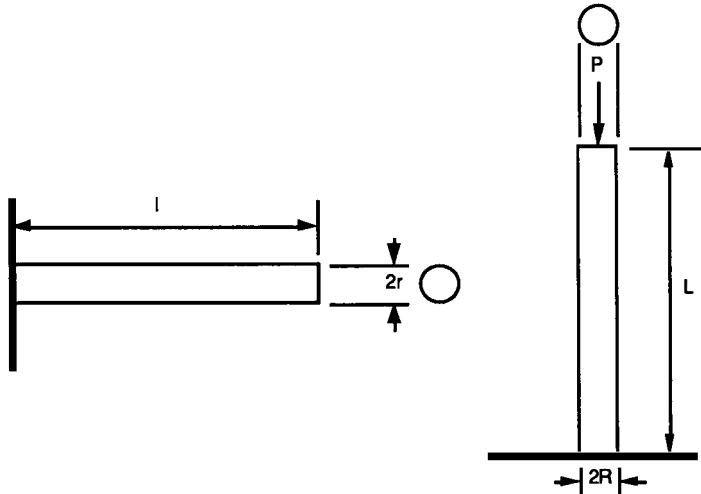


Figure 11. (a) Beam geometry and (b) column geometry

where axial stress in the column is given by:

$$\sigma_a = \frac{P}{\pi R^2} \quad (30)$$

and where K is a constant that depends on the boundary conditions for the column (taken to be 16 for the fixed-free condition here). Applying monotonicity analysis (Papalambros and Wilde, 1988), the minimum weight for each option can be found to be:

$$W_b = \max \left[\pi l \rho \left(\frac{4 l P^2}{\pi S_y} \right)^{\frac{1}{3}}, 2 \pi l^2 \rho \sqrt{\frac{P l}{3 E \pi \delta_{\max}}} \right] \quad (31)$$

$$W_c = \max \left[\frac{L P \rho}{S_y}, \frac{P L^2 \rho}{E \delta_{\max}}, L^2 \rho \sqrt{\frac{C P}{\pi E}} \right] \quad (32)$$

Substituting $u_1 = -W_b$, $u_2 = -W_c$, with $n = 2$, into equation (21) gives:

$$E(p_j) = \min \left[\frac{E}{p} [W_b], \frac{E}{p} [W_c] \right] - \int_{p_{j1}}^{p_{j2}} p r(p_j) \left\{ \min \left[\frac{E}{q} [W_b], \frac{E}{q} [W_c] \right] \right\} dp_j \quad (33)$$

Where p_j is one of the random parameters (l , L , P or δ_{\max}) and q is the vector composed of the remaining other three parameters.

The expected value of perfect information (equation 33) may be evaluated for each parameter numerically, giving $E(P) = 0.0$ N, $E(l) = 4.8$ N, $E(L) = 8.2$ N, and $E(\delta_{\max}) = 0.0$ N. Thus, with 20 attachment points, we expect perfect information of the lateral position of the pipe to generate a weight savings of 96 N, and perfect information of the height to yield 164 N of weight savings. At a value of 1 \$/N, the design agent should expend no more than \$164 worth of effort to determine the height of the pipe more accurately, and \$96 to determine the lateral position of the pipe more accurately. Note that better information of the load and maximum total deflection are clearly of no or negligible value, despite the significant uncertainty in both parameter values.

Let us compare this result with that which might be derived using a local approximation method for the utility function. Vanderplaats (1984) describes sensitivity analysis, a method by which a value for $\frac{d u_i'(p_0)}{dp_j}$ may be derived at a local optimum for any problem of the form:

$$\begin{aligned} u_i'(p_0) &= \max_x u_i(x, p_0) \\ \text{s.t. } h(x, p_0) &= 0 \\ g(x, p_0) &\leq 0 \end{aligned} \quad (34)$$

where p_0 is some (deterministic) choice for the parameter values.

Our approach to determining the information value for each p_j is then the following; we solve (34) for the mean value of each of the random parameters (set p_{j0} to the mean value of p_j) then approximate locally the optimum of the utility function by:

$$U_i'(p) = U_i'(p_0) + \frac{dU_i'(p_0)}{dp_1} (p_1 - p_{10}) + \frac{dU_i'(p_0)}{dp_2} (p_2 - p_{20}) + \dots \quad (35)$$

Equation (35) may then be substituted into equation (21) to give a much simplified expression for the expected value of perfect information. In our example problem all the probability distributions are symmetrical, and therefore equation (21) reduces to:

$$E(p_j) = \int_{\widehat{p}_{j1}}^{\widehat{p}_{j2}} pr(p_j) \left\{ \max_i \left(U_i'(p_0) + \frac{dU_i'(p_0)}{dp_j} (p_j - p_{j0}) \right) \right\} dp_j - \max_i U_i'(p_0) \quad (36)$$

The mean values of the parameters in this example are: $P_0 = 40,000$ N, $l_0 = 1.6$ m, and $\delta_{max0} = 0.07$ m. The optimum weight for these mean values is found from equations (31) and (32) to be 217.1 N for the beam and 227.8 N for the column. Applying sensitivity analysis at p_0 gives the following results:

$$\begin{aligned} \frac{dW_c}{dP} &= 2.07 \cdot 10^{-3}, \quad \frac{dW_c}{dL} = 284.8, \quad \frac{dW_b}{dP} = 2.63 \cdot 10^{-3}, \quad \frac{dW_b}{dL} = 607.7, \\ \frac{dW_c}{dl} &= \frac{dW_c}{d\delta_{max}} = \frac{dW_b}{d\delta_{max}} = \frac{dW_b}{dL} = 0.0 \end{aligned} \quad (37)$$

Using these results in equation (36) gives $E(P) \sim 0.012$ N, $E(L) \sim 8.59$ N, $E(l) \sim 4.92$ N, and $E(\delta_{max}) \sim 0$. The evaluation was performed by first analytically reducing equation (36) to split the integrals and remove the minimization operator, then using numerical integration capability of Mathematica™. Inspecting the above, we see that sensitivity analysis approach in this case gave good estimates with significantly reduced computational effort. This approach is quite general, being applicable to any problem for which derivatives of the objective and constraints are available or may be estimated. Sobieszczanski-Sobieski et al. (1981) give some guidelines for when an approximation based on sensitivity analysis, and therefore this approach, will be accurate.

CONCLUSIONS AND FUTURE DIRECTIONS

This paper presents a novel approach to the control of design reasoning based on the perspective of design as Intelligent Real Time Problem Solving (IRTPS). This perspective leads to a new model of the design process in which the design agent takes into account design process costs explicitly as part of the decision making process. Because this approach is based on first principles and a general model of the design process, it is applicable to any design domain that may be modeled in the general framework presented. It

offers a particularly attractive approach for controlling reasoning in systems faced with design problems for which the problem solving context cannot be established in advance, and thus heuristics or domain dependent expertise may not be available. Of course, heuristics and domain dependent knowledge can always be added, when available, to provide evaluation estimates and uncertainty mappings. The basic decision-analytic approach explored here can be applied as well to controlling reasoning during prototype generation, parametric design, and catalog selection. It can also be used as one type of design agent in a diverse multi-agent architecture like the one proposed by Hayes-Roth (1990).

The primary challenge in applying the decision-analytic approach to the control of reasoning lies in reducing the complex expressions to which it leads to a tractable form by making appropriate assumptions and simplifications. In this paper, we have limited ourselves primarily to the use of the expected value of perfect information, which provides an upper limit on the value of imperfect information. Developing other easily solved and reasonably accurate approximations to the expected value of information is an important area for future work.

The decision-analytic approach to the control of design reasoning holds great promise not only as a practical tool for developing strategies for controlling reasoning, but as a foundation for a theory of Intelligent Real Time Design and a standard against which other heuristic and domain dependent methods can be measured and understood.

Acknowledgements. The authors would like to thank Prof. Stuart Russell of U.C. Berkeley for the stimulus he has given this work, and Prof. Jon Cagan of Carnegie-Mellon University for his helpful comments. Funding for this research was provided by a grant from Rockwell International, Project MICRO grant #90-004 and an equipment gift from the Digital Equipment Company.

REFERENCES

- Agogino, A.M. (1990). Real Time Influence Diagrams for Monitoring and Controlling Mechanical Systems, in R.M. Oliver and J.Q. Smith (eds), *Influence Diagrams, Belief Nets and Decision Analysis*, John Wiley & Sons, Chap. 9, pp. 199-228.
- Agogino, A. M. (1989). Real Time Reasoning about Time Constraints and Model Precision in Complex Distributed Mechanical Systems, *Proceedings of the AAAI Spring Symposium Series on AI and Limited Rationality, Stanford University*, March 28-30, pp. 1-5.
- Bradley, S. R. and Agogino A. M. (1990). *Supplemental Appendix to Intelligent Real Time Design: Application to Prototype Selection*, Berkeley Expert Systems Technology Laboratory Working Paper #90-1101-0, Nov. 25.
- Brown, D. and Chandrasekaran, B. (1989). *Design Problem Solving: Knowledge Structures and Control Strategies*, Morgan Kaufman, Los Altos, CA.
- Dean, T. and Boddy, M. (1988). An Analysis of Time-Dependent Planning, *Proceedings of AAAI '88*, Morgan Kaufmann, Los Altos, CA, pp. 49-54.
- Erman, L. D. (ed) (1990). Intelligent Real-Time Problem Solving (IRTPS) Workshop, Report TTR-ISE-90-101, Cimflex Teknowledge Corp. (1810 Embarcadero Rd., P.O. Box 10119, Palo Alto, CA 94303), January, p. 7.

- Fehling, M. and Breese, J. (1988). A Computational Model of Decision-Theoretic Control of Problem-Solving under Uncertainty, *Proceedings of the Fourth AAAI Workshop on Uncertainty in AI*, Minneapolis.
- Gero, J. S., Maher, M. L. and Zhang, W. (1988). Chunking Structural Design Knowledge as Prototypes, in Gero, J. S. (ed), *Artificial Intelligence in Engineering: Design*, Elsevier, Amsterdam, pp. 3-21.
- Good, I. J. (1968). A Five Year Plan for Automatic Chess, *Machine Intelligence*, 2.
- Good, I. J. (1971). Twenty-seven Principles of Rationality, in V. P. Godambe and D. A. Sprott (eds), *Foundations of Statistical Inference*, Holt, Rinehart, and Winston, Toronto.
- Hansson, O. and Mayer, A. (1989a). Heuristic Search as Evidential Reasoning, *Proceedings of the Fifth Workshop on Uncertainty in AI*, Windsor, Ontario, pp.152-161.
- Hansson, O. and Mayer, A. (1989b). Decision-Theoretic Control of Search in BPS, *Proceedings of the AAAI Spring Symposium on Limited Rationality*, Palo Alto, pp.44-48.
- Hayes-Roth, B., Washington, R., Hewett, R., Hewett, M. and Seiver, A. (1989). Intelligent Monitoring and Control, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, Morgan Kaufmann, pp. 243-249.
- Hayes-Roth, B. (1990). Architectural Foundations for Real-Time Performance in Intelligent Agents, *The Journal of Real-Time Systems*, 2: 99-125.
- Horvitz, E., Cooper, G. and Heckerman, D. (1989). Reflection and Action Under Scarce Resources: Theoretical Principles and Empirical Study, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, Morgan Kaufmann, pp. 1121-1127.
- Howard, R. (1966). Information Value Theory, *IEEE Transactions on Systems Science and Cybernetics*, vol. SSC-2, no. 1, pp. 779-783.
- Jain, P. and Agogino, A. M. (1988). Theory of Design: An Optimization Perspective, *Journal of Mechanisms and Machine Theory* 25(3): 287-303.
- Luenberger, D. (1984). *Linear and Nonlinear Programming*, Addison-Wesley, Reading, MA.
- Mittal, S. and Araya, A. (1986). A Knowledge-Based Framework for Design, *Proceedings of AAAI '86*, Morgan Kaufmann, Los Altos, CA, pp. 856-865.
- Papalambros, P. Y. and Wilde, D. J. (1988). *Principles of Optimal Design*, Cambridge University Press, NY.
- Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, Morgan Kaufman, San Mateo, CA.
- Pratt, J., Raiffa, H. and Schlaifer, R. (1965). *Introduction to Statistical Decision Theory*, McGraw-Hill, NY
- Raiffa, H. (1968). *Decision Analysis: Introductory Lectures on Choices under Uncertainty*, Random House, NY
- Rothbart, H. (1985). *Mechanical Design and Systems Handbook*, 2nd edn, McGraw-Hill, NY
- Russell, S. and Wefald, E. (1989). On Optimal Game-Tree Search Using Rational Metareasoning, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, Morgan Kaufmann, pp. 334-340.

- Siddall, J. (1982). *Optimal Engineering Design: Principles and Applications*, Marcel Dekker, NY.
- Siddall, J. (1983). *Probabilistic Engineering Design: Principles and Applications*, Marcel Dekker, NY.
- Simon, H. (1955). A Behavioral Model of Rational Choice, *Quarterly Journal of Economics*, no. 69.
- Sobieszczański-Sobieski, J., Barthemlemy, J. and Riley, K. (1981). Sensitivity of Optimum Solutions to Problem Parameters, *Proceedings of AIAA/ASME/ASCE/AHS 22nd Structures, Structural Dynamics and Materials Conference*, Atlanta, GA.
- Ueno, T., Ohtake, Y., Nakayama, H., and Inoue, K. (1987). Multiple Criteria Decision Making System for Steel Structure in Chemical Plant, in Sawaragi, Inoue, and Nakayama (eds), *Toward Interactive and Intelligent Decision Support Systems*, vol. 1, Springer-Verlag, NY, pp. 313-322.
- Vanderplaats, G. (1984). *Numerical Optimization Techniques for Engineering Design: with Applications*, McGraw-Hill, NY
- Waldron, K., Waldron, M. and Wang, M. (1986). An Expert System for Initial Bearing Selection, ASME Design Engineering Technical Conference, Columbus, OH.
- Wood, K.L., and Antonsson, E.K. (1989). Representing Imprecision in Engineering Design: Comparing Fuzzy and Probability Calculus, *Research in Engineering Design*, vol. 1, no. 3/4, Springer-Verlag, NY, pp. 187-203.

A study on multicriteria structural optimum design using qualitative reasoning

M. Arakawa and H. Yamakawa

Department of Mechanical Engineering
Waseda University
3-4-1 Ohkubo, Shinjuku
Tokyo 169 Japan

Abstract. In ordinary optimum design process, designers are often required to express a design problem as a mathematical model with a single objective problem. However, several numbers of objectives are found in structural designs, then it may be tedious and difficult for designers to choose just one among them. In such cases, it will be natural and reasonable to formulate it in a multi-objective optimization problem. Recently, many studies have been done on the interactive multicriteria optimum design problems. In conventional methods, designers have been forced to make important and difficult decisions concerning the "trade-off ratio", "marginal rates of substitution", "surrogate worth trade-off" and so on, which may give much information in the optimization. To reduce the load of the designer forced to evaluate such quantities and to grasp the given optimum design problem in a qualitative way, we modify the existing qualitative reasoning which has been studied in the field of artificial intelligence: we first modify the conventional qualitative reasoning to be available for many design variables and then introduce new concepts of a qualitative trade-off ratio and a fuzzy language to multicriteria optimum design. Through these modifications, we can develop a new method for multicriteria design problems. From numerical examples of the optimum designs of a stepped cantilever beam and a parabolic antenna, it is shown that the proposed method is applicable and effective for optimizations of complex structural systems by qualitative understanding and can support much the decision making by designers.

INTRODUCTION

Recent rapid development of efficient computer systems have made it possible to solve the problem of its computational cost and time and to integrate optimization procedure into the practical process of engineering designs. Moreover, the number of well-trained designers, so called experts, decreases year by year. Consequently, demands for optimum design, which enable ordinary designers to do their designs just as experts, have become higher. And it will

be major concern for research in this field to meet these requirements.

In an ordinary optimum design process, designers often have to set a single objective mathematical model at the first stage, and during the optimization, the prescribed problem usually cannot be changed or examined until optimization results have been finally obtained. However, in general engineering problems, the number of objectives is not single. So it will be a hard task for the designer to choose only one among them and fix maximum or minimum limits on the rest of them as constraints. In such a case, it will be rather natural, and reasonable for a designer to formulate the problem as a multi-objective optimum design problem. But methods for multi-objective optimization are not simple and take quite long time for the computation generally, and there usually exists plenty of solutions called non-inferior solutions (Pareto solutions). In engineering, as we need to determine one design, we have to choose one among non-inferior ones. To choose one out of all non-inferior solutions which the designer thinks best solutions, we have to solve multicriteria decision making problem trially. In these days, many studies have been developed on interactive multicriteria decision making problems. In those methods designers are required to make such sophisticated decisions as "trade-off ratio", "marginal rates of substitution", "surrogate worth trade-off" and so on before the calculation.

In such circumstances, it will be important and meaningful to reduce a load of the designer. There will be many possible methods to overcome those problems. Qualitative reasoning, which has been mainly studied in the field of artificial intelligence, is one of powerful method among them. Some new directions for applications of qualitative reasoning on modeling and simulation are shown in reference(Widman,1989). As for the optimization problem, there are studies by authors(1989,1990a,1990b) and Agogino's(1987). In this study, we modify qualitative reasoning and use fuzzy language in order to reduce ambiguities of qualitative directions of design variables, which have been major problems in conventional studies by authors (1989,1990a,1990b), in inferring the qualitative directions of objective functions. These modification make qualitative reasoning possible to infer design values in semi-quantitative space to meet the qualitative directions of objective functions with a fewer candidates than the conventional way. And we also develop a new concept of qualitative trade-off ratio to determine qualitative direction of objective functions. The proposed method enables the designer to carry out the interactive multicriteria optimum design with only qualitative grasp of the modeled system and can reduce the load of the designers and the computation time in computer much more in comparison with the conventional one. The proposed method are demonstrated through applications to the design problems of a simple beam problem and a modeled parabolic antenna.

OPTIMUM DESIGN IN ENGINEERING

First, we make a brief formulations of multicriteria decision making problem which will be used to explain the proposed method.

General Definition in Multicriteria Decision Making

When there are no significant differences in importance among structural requirements,

it will be a great load for designers to choose only one among them. In such a case, it will be natural and reasonable to formulate a problem as a multicriteria optimum design, in another word vector optimization. A general expression of the multicriteria optimum design is following.

$$\min \mathbf{F}(\mathbf{x}) \quad (1-a)$$

subject to

$$g_j(\mathbf{x}) \leq 0 \quad j=1,2,\dots,M \quad (1-b)$$

where $\mathbf{F}(\mathbf{x})$ is a vector objective function:

$$\mathbf{F}(\mathbf{x}) = \{f_1(\mathbf{x}), f_2(\mathbf{x}), \dots, f_L(\mathbf{x})\}^T$$

where $\mathbf{x} = \{x_1, x_2, \dots, x_N\}^T$ is a vector consists of design variables.

The representative for all Pareto solutions, which is the one of optimality standards, is due to implicit preference of the decision maker (designer in the design problem). The multicriteria optimum design will be developed into interactive procedures which integrate the decision making process into optimization algorithms. These procedures consist of sequences of decision making phases and a computation ones. The general expression of multicriteria decision making problems may be expressed by the following.

$$\min \Phi(\mathbf{F}) \quad (2-a)$$

subject to

$$g_j(\mathbf{x}) \leq 0 \quad j=1,2,\dots,M \quad (2-b)$$

where $\Phi(\mathbf{F})$ is a preference function, and usually it is not known explicitly.

The method by Haimes is called surrogate worth trade-off method (SWT method) and it recognizes that an optimization theory is usually much more concerned with the relative value of additional increments of various non-commensurable objectives, at a given value of each objective function, than it is with their absolute values. This method is based on the concept that it is much easier for the decision maker to assess the relative value of the trade-off of marginal increase and decrease between any two objectives than it is for him to assess their absolute values. In the SWT method, the decision maker (designer) is assumed to have their implicit preferred trade-off ratio m , and in each step, he is asked to estimate the difference between the implicit preferred trade-off ratio and the gradients of hyperplane on the current point in the form of a surrogate worth function W . The preferred solution in the SWT method means that W is equal to zero. The trade-off ratio m in this case can be written as follows:

$$m = \left\{ \frac{df_1}{df_p}, \dots, \frac{df_{p-1}}{df_p}, 1, \frac{df_{p+1}}{df_p}, \dots, \frac{df_L}{df_p} \right\}^T \quad (3)$$

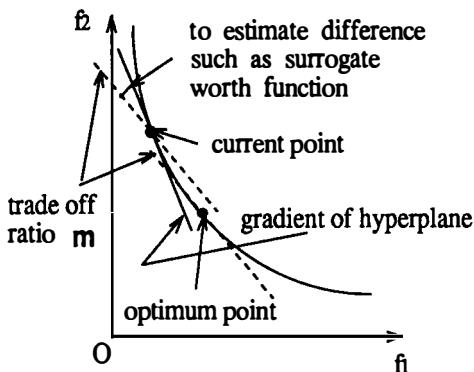


Figure 1 SWT Method

where f_p is a nominal function. Fig.1 shows the process of SWT method in the attainable space of two-objective functions.

EXTENSION OF QUALITATIVE REASONING TO OPTIMUM DESIGN

In this chapter, we treat a problem of the extension of the qualitative reasoning to engineering design, especially to the engineering optimum designs problems. To apply the qualitative reasoning to multicriteria optimum design problem the following will be pointed out:

- 1) It is difficult to express qualitative differential equations when the systems become complicated.
- 2) It is not easy to treat a multi-variables' problem. Mostly, a single variable in qualitative reasoning is correspond to nominal "time" in conventional manner.
- 3) Ambiguity in the "qualitative arithmetics" leads explosion of combinations of qualitative states.
- 4) Even if we obtained qualitative optimum solutions, we have to find quantitative solutions in the engineering problems so as to meet the qualitative solutions. In such cases, we have to solve the complicate non-linear problems.

Thus, it will not be helpful in engineering, when we use the qualitative reasoning in conventional manner.

Following extensions are added to the conventional qualitative arithmetic operations which can be found in the Kuipers(1986).

Extension to use multi-variables

The possibility of useful application of qualitative reasoning in the engineering designs may be stated that it can be inferred the qualitative direction (variation) of function, not its value. In order to use multi-variables as in usual engineering design, we introduced "qualitative partial derivatives" and chain rule as following(Arakawa 1990a):

$$[df_j] = \sum_{i=1}^N \left[\frac{\partial f_j}{\partial x_i} \right] \otimes [dx_i] \quad j=1,2,\dots,L \quad (4)$$

Where $[\partial f_j / \partial x_i]$ is a qualitative partial derivatives of function f_j , and $[dx_i]$ is a qualitative direction of a design variable x_i and \otimes is product operation among qualitative variables.

Extension to avoid ambiguity of qualitative arithmetics

In Eq.(4), we can treat multi-variables easily in the qualitative reasoning. And qualitative partial derivatives can be obtained from information of quantitative values in the process of an optimum design. However, there are qualitative adding in Eq.(4), ambiguity will occur if there are conflicts in each summation. In order to reduce these ambiguity, we introduce linguistic information of relative values of the sensitivities in all qualitative values. To qualify quantitative data, we use fuzzy languages. As there is no standard in definition of fuzzy language, we set the following membership function of the relative value of two absolute numbers and give the meaning for instance.

small: small means the absolute of x_1 is relatively small with respect to the absolute of x_2 .

$$\mu(\text{small},y) = \frac{1}{1+40y^4} \quad \text{for } y \geq 0 \quad (5-a)$$

same: same means the absolute of x_1 is relatively same with respect to the absolute of x_2 .

$$\mu(\text{same},y) = \frac{1}{1+20(y-1)^4} \quad \text{for } y \geq 0 \quad (5-b)$$

large: large means the absolute of x_1 is relatively large with respect to the absolute of x_2 .

$$\mu(\text{large},y) = \begin{cases} \frac{1}{1-500(y-2)^5} & \text{for } 0 \leq y \leq 2 \\ 1 & \text{for } y \geq 2 \end{cases} \quad (5-c)$$

In Eq.(5), "y" is the absolute of the ratio of two numbers:

$$y = \left| \frac{x_1}{x_2} \right| \quad (5-d)$$

Figure 2 shows the relationship between relative value y and membership functions of each fuzzy language graphically.

We also introduce "terns" to express absolute relative values more precisely.

very--it means μ^2 , and it makes the meaning of fuzzy language more meaningful.

absolutely A--it means $-A$ or $+A$, where "A" is fuzzy language

We will show the rules of multiplying and adding of two fuzzy numbers in Table 1, Table 2 and Table 3.

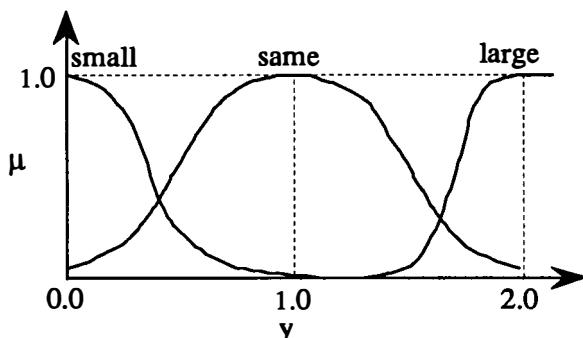


Figure 2 Membership function of each fuzzy language

Table 1 Multiplying of two fuzzy number

multiply	small	same	large
small	small	small	same
same	small	same	large
large	same	large	large

Table 2 Adding of two fuzzy number

add		+		
		small	same	large
+	small	small	same	large
	same	same	large	large
	large	large	large	very large

Table 3 Adding of two fuzzy number

add		+		
		small	same	large
-	small	absolutely very small	+	+
	same	- same	absolutely small	+ large or same
	large	- large	- large or same	?

In Table 3, we can see that possibility of ambiguity will decrease, because absolutely very small may be neglected , and absolutely small can be also neglected if there is another adding with same or large.

Algorithm of inferring directions of design variables

We will explain shortly how to infer the directions of design variables in this section.

- 1) Set or calculate desired directions of objective functions.
 - 2) Choose nominal objective function f_p and nominal design variable x_q .
 - 3) Calculate relative value of $\partial f_j / \partial x_i$ to $\partial f_j / \partial x_q$ and convert to fuzzy language as $RF(f_j, x_i, x_q)$.
 - 4-1) Case $[df_p]$ is steady ('std').
 - a) If $RF(f_j, x_i, x_q)$ is 'small' then set $[dx_i]$ 'any' else set it 'std'.
 - b) If there are any 'any' in $[dx]$ list then remove f_p and design variables other to 'any' and repeat from the beginning, else go to group revision stage.
 - c) The end of this stage follows that whether all $[dx]$ list is determined or settled by the decision maker or there is no function list left.
 - 4-2) Case $[df_p]$ is not steady ('dir').('dir' will be either 'dec' or 'inc')
 - a) Select $[dx]$ list to steepest change for $[df_p]$.
 - 5) Check whether $[dx]$ list match another $[dF]$ list. If they match then finish inferring else go to group revision stage.
- Group revision stage.
- 1) Choose new nominal function f_{p_1} other to f_p .
 - 2) Generate direction $[dx_i]$ and $[dx_q]$ as to match the direction $[df_{p_1}]$.
 - 3) Choose the one from generated list of 2) which most likely keep $[df_p] = 'std'$.

ALGORITHM OF PROPOSED METHOD

In this section, we will explain algorithm of the proposed method. Before that, we will introduce a new concept "qualitative trade-off ratio".

Qualitative trade-off ratio

In ordinary multicriteria decision making problems, "trade-off" means some units increment of objective functions f_i with respect to a unit increment of nominal objective function f_p , and it can be expressed as Eq.(3). In Eq.(3), if we consider unit increment of nominal objective function f_p , corresponding qualitative derivative $[df_p]$ equals $[+]$ (or inc.). Thus Eq.(3) can easily expressed in the following qualitative way without no ambiguity.

$$[m] = ([df_1], \dots, [d f_{p-1}], [+], [d f_{p+1}], \dots, [d f_L])^T \quad (6)$$

In multicriteria decision making problems, we generally consider to minimize implicit preferred function Φ . Thus, if we give the direction of nominal objective function $[df_p]$, we can calculate desired direction of objective functions $[dF]$.

$$[dF] = [m] \otimes [df_p] \quad (7)$$

Algorithm for implementation

The proposed method is a hybrid one of both qualitative and quantitative method. Qualitative partial derivatives $[\partial f_i / \partial x_j]$ are obtained by quantitative informations. In usual design, design variables of structural members are not always determined continuously, they are determined according to some discrete values prescribed by the standards ISO, ANSI, JIS and so on. Thus we can condense the problem as to determine the optimal design variables among prescribed semi-quantitative space. The proposed method consists of two stages, a trial optimization stage which is formulated in the single objective optimization problem and a qualitative inferring stage among prescribed semi-quantitative space. An algorithm of the proposed method is following:

- 1) Formulate and solve a single objective optimum design problem.

$$\begin{aligned} & \min f(\mathbf{x}) \\ & \text{subject to} \end{aligned} \tag{8}$$

$$g_j(\mathbf{x}) \leq 0 \quad j=1,2,\dots,M$$

where $\mathbf{x} = \{x_1, x_2, \dots, x_N\}^T$.

- 2) Re-formulate the problem as multicriteria optimum design problem according to the information from the trial optimum design problem 1).

$$\mathbf{F}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \dots, f_L(\mathbf{x}))^T \tag{9}$$

- 3) Set a desired direction and carry out qualitative inferring of directions of design variables according to the method explained in the section "Algorithm of inferring directions of design variables".
- 4) Search the database and determine the qualitative values of design variables \mathbf{x}^* and calculate the objective functions \mathbf{F}^* . If the decision maker satisfies the result \mathbf{F}^* as Pareto solution then go to 5) else go to 3).
- 5) Show the result and compare it with other Pareto solutions with fuzzy language. If the decision maker thinks \mathbf{F}^* as preferred solution then end, else if he thinks \mathbf{F}^* as Pareto solution then go to 6) else go to 3).
- 6) Ask the decision maker the nominal objective function f_p and the nominal design variable. And ask him qualitative trade-off ratio $[m]$ and direction of $[df_p]$.
- 7) Calculate a desired direction and carry out qualitative inferring of directions of design variables according to the method explained in the section "Algorithm of inferring directions of design variables".
- 8) Search the database and determine the qualitative values of design variables \mathbf{x}^* and calculate the objective functions \mathbf{F}^* . If the decision maker satisfies the result \mathbf{F}^* as Pareto solution then go to 5) else go to 3).

In this study, we combine a program written in FORTRAN for the quantitative calculation to LISP program for the qualitative reasoning. Figure 3 shows a flow chart of algorithm explained

just before. It corresponds to the 3) to 8). Interaction has done in a part "Indicate 1" in Figure 3, where we define qualitative trade-off ratio. In a part "Indicate 2", qualitative behavior of design variables has been obtained by "algorithm of inferring directions of design variables". And in a part "Indicate 3", designers have to make decisions whether calculated results will satisfy the implicit or explicit goals of objective functions through qualitative information. As shown in Figure 3, most parts of algorithm are programmed in LISP.

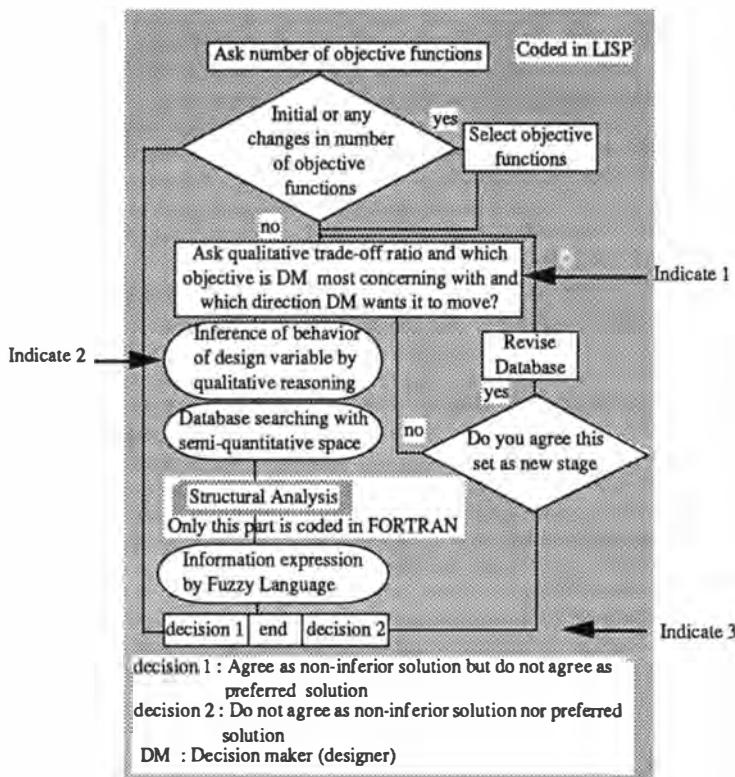


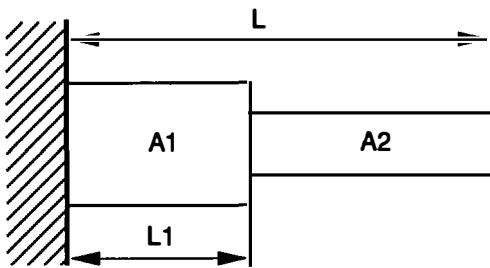
Figure 3 Proposing Flow of Algorithm

NUMERICAL EXAMPLES

We will show two numerical examples using the proposed method in order to show its effectiveness.

Example 1(Two objectives with two design variables)

We will show a simple two objective optimum design problem with two design variables defined as following.(Figure 4)



initial setting of
design variables
 $A_1^0 = 1.60 \times 10^{-4} [m^2]$
 $L_1^0 = 2.50 \times 10^{-2} [m]$

constants
 $L = 5.00 \times 10^{-2} [m]$
 $A_2 = 4.00 \times 10^{-5} [m^2]$
 $E = 2.05 \times 10^{11} [Pa]$
 $\rho = 7.70 \times 10^3 [kg/m^3]$

Figure 4 Model of example1

Find design variables $x = \{A_i, L_i\}^T$ such that
Minimize

$$\begin{aligned} \text{Obj(1)} &= \frac{M - M^0}{M^0} \\ \text{Obj(2)} &= \frac{(F_2 - F_1) - (F_2^0 - F_1^0)}{(F_2^0 - F_1^0)} \end{aligned} \quad (10)$$

subject to

$$\text{Const(1)} = \frac{5.00 \times 10^{-4} - V_{\text{tip}}}{5.00 \times 10^{-4}} \geq 0$$

$$\text{Const(2)} = \frac{6000 - F_1}{6000} \geq 0$$

$$\text{Const(3)} = \frac{A_1}{A_1^0} \geq 0$$

$$\text{Const(4)} = \frac{L_1}{L_1^0} \geq 0$$

$$\text{Const(5)} = \frac{3/4L - L_1}{L_1^0} \geq 0$$

Where

- M : mass of model
- F_1 : the first natural frequency of model
- F_2 : the second natural frequency of model
- V_{tip} : deformation of tip of model

Results of optimization are shown in Table 4 and the behavior of design variables in the design space is shown in Figure 5, Figure 6 demonstrates a way of the decision making by the designer. The result "GPM" in Table 4 was the one obtained by a trial optimization, whose

objective function was calculated by simply adding Obj(1) and Obj(2) with same constraints. From the result "GPM", the designer added Const(1) to objective functions and hoped to decrease Obj(2) with qualitative trade-off ratio {any,[+],std}. Where "any" means, he can take an arbitrary direction. After he got "Pareto 1", he understood there would be less hope to decrease Obj(2) and tried to decrease Obj(1) with qualitative trade-off ratio {[+],any,std}. Figure 5 shows semi-quantitative space of design variables, and each plot shows the candidate of design variables inferred by modified qualitative reasoning. In Figure 5, squired plots meant that the candidate design variables were not accepted by designer, and circled masked plots meant that those were not accepted by designers and suggested that there would be no more candidates of the directions for given directions of objective functions by computer.

Table 4 Results of Optimization

Contents	Initial	GPM	Pareto1	Pareto2
Obj(1) $\times 10^{-2}$	0.00	-2.86	-2.86	-3.04
Obj(2) $\times 10^{-3}$	0.00	-6.20	-6.34	-5.01
Const(1) $\times 10^{-3}$	30.3	0.00	-0.64	-3.89
A1 $\times 10^{-5}$	16.0	9.82	9.90	8.90
L1 $\times 10^{-2}$	2.50	2.70	2.66	2.79

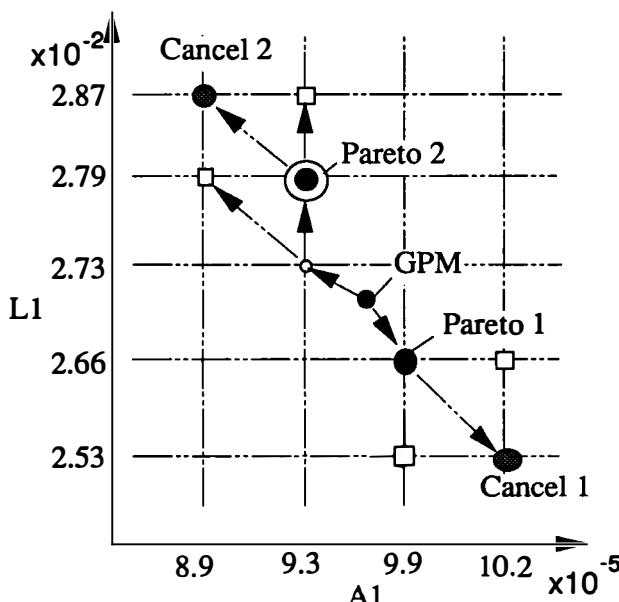


Fig.5 Behavior of design variables

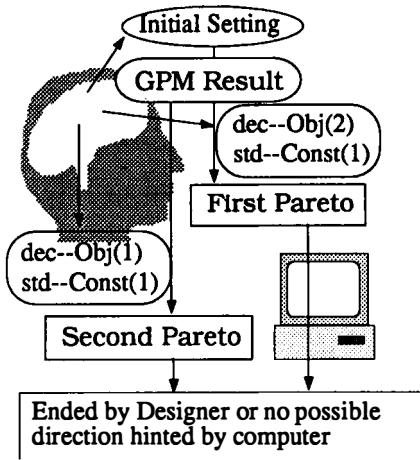


Fig.6 Decisions by Designer

From comparison of Pareto solutions in Figure 7, the designer selects Pareto 2 will be preferred solution.

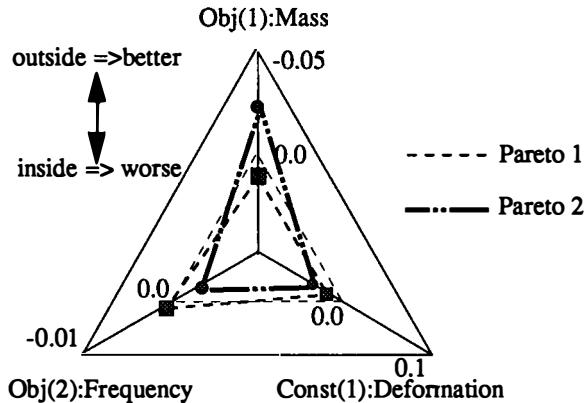


Figure 7 Comparison of Pareto Solution

Example 2 (Two objective optimum design with seven design variables)

We will show an optimum design problem of much more complicated model (supporting structure of modeled parabolic antenna) shown in Fig.8 and Table 5 and Table 6.

Table 5 Material Properties and Allowable Values

Contents	Values
Density	ρ [kg/m ³] 7.860×10^3
Young's Modulus	E [GPa] 2.050×10^2
Allowable Displacement	δ_{max} [m] 5.000×10^{-5}
Allowable Tensile Stress	σ_a [MPa] 2.000×10

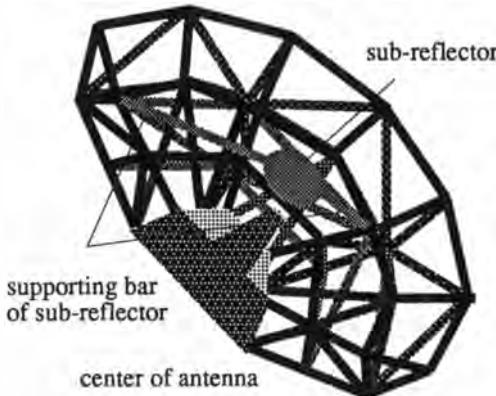


Fig.8 Modeled Antenna Supporting Structure

Table 6 Nodes and Coordinates

node	x	y	z	node	x	y	z
1	0.000	0.405	0.294	15	1.069	0.617	0.202
2	0.000	0.687	0.756	16	0.614	0.583	-0.198
3	0.000	0.969	1.219	17	1.500	-0.245	0.337
4	0.000	1.128	1.850	18	2.000	-0.490	0.674
5	0.000	1.363	1.119	19	0.433	-0.202	-0.147
6	0.000	1.116	0.565	20	0.866	-0.527	-0.125
7	0.000	0.869	0.010	21	1.299	-0.852	-0.104
8	0.750	0.806	0.010	22	1.732	-1.299	0.086
9	1.000	0.911	1.692	23	1.525	-0.773	-0.433
10	0.433	0.202	0.147	24	1.069	-0.382	-0.524
11	0.866	0.282	0.462	25	0.614	0.009	-0.615
12	1.299	0.362	0.778	26	0.750	-1.296	-0.426
13	1.732	0.319	1.262	27	1.000	-1.891	-0.344
14	1.525	0.651	0.602	55	0.000	-0.735	1.011

Table 7 Groups of Design variables

Group	Member
1	1-2,2-3,5-6,6-7 and their relatives
2	3-4,4-5,6-15 and their relatives
3	7-2,2-6,2-5,3-5,8-9 and their relatives
4	7-8,8-16 and their relatives, and 3-15,15-21,21-33,33-39,39-51,51-3
5	5-8,8-14,5-9,9-14 and their relatives
6	2-11,3-8,8-12,4-9,9-13,5-14 and their relatives
7	3-55,21-55,39-55

A trial single objective optimum design is as follow.
Find \mathbf{X} such that

$$\mathbf{M} \rightarrow \min$$

$$(11)$$

subject to

$$\begin{aligned}\delta &\leq \delta_{\min} \\ \sigma_{\max} &\leq \sigma_a \\ -\sigma_{\min} &\leq \sigma_{\text{Euler}}\end{aligned}$$

where

- M : mass
- δ : r.m.s deviation from best fitted parabolic surface
- δ_{\max} : maximum allowance of r.m.s deviation
- σ_{\max} : maximum tensile stress
- σ_a : allowable stress
- σ_{\min} : maximum compressive stress
- σ_{Euler} : Euler's buckling stress

We divide the members of the structure in 7 groups (Table 7) and take each of their cross-sectional area as a design variable. And these design variables may be selected from JIS standards, which means that semi-quantitative space is constructed by JIS standards. Results of trial optimization are shown in Table 8.

Table 8 Results of Trial Optimization

Stage		Initial	GPM results
design variables [m ²]	group1	2.616x10 ⁻²	1.683x10 ⁻²
	group2	2.616x10 ⁻²	1.547x10 ⁻²
	group3	2.616x10 ⁻²	1.358x10 ⁻³
	group4	2.616x10 ⁻²	9.467x10 ⁻³
	group5	2.616x10 ⁻²	8.506x10 ⁻³
	group6	2.616x10 ⁻²	3.664x10 ⁻⁴
	group7	2.616x10 ⁻²	8.298x10 ⁻³
Mass [Kg]		2.876x10 ³	9.582x10 ⁴
RMS Dev. [m]		3.262x10 ⁻⁵	4.996x10 ⁻⁵
Tensile Stress [MPa]		2.384	2.720
Compressive Stress Ratio		2.600x10 ⁻⁴	8.254x10 ⁻²

"Compressive Stress Ratio" in Table 8 defines the ratio of maximum of compressive stress to Euler's buckling stress of those element. From the trial optimization results, the r.m.s deviation became active constraints. Adding it to objective functions, we defined new objective functions as:

$$F(X) = \{M(X), \delta(X)\} \quad (12)$$

We carried out multi-criteria optimization with the proposed algorithm. At first, we tried to decrease r.m.s deviation, while we hopefully did not want to increase mass. As a result, we passed through local optimum solution and reached to "First Pareto" in Table 9. Then we tried to decrease mass while r.m.s. deviation keeps steady (hopefully "decrease") and we reached "Second Pareto" in Table 9. After we reached two Pareto optimum solution, we compared these two and decided "Second Pareto" would be optimum solution of the given problem. The final

results of the optimization are shown in Table 9 and the interaction with computer is explained in Figure 9. In Figure 10, we show a comparison of Pareto optimum solutions and an understanding in fuzzy language and the qualitative grasps on CRT.

Table 9 Results of Optimization

Stage		First Pareto	Second Pareto
design variables [m ²]	group1	1.698x10 ⁻²	1.840x10 ⁻²
	group2	8.736x10 ⁻²	6.913x10 ⁻²
	group3	6.913x10 ⁻²	5.891x10 ⁻²
	group4	4.040x10 ⁻³	3.479x10 ⁻³
	group5	4.927x10 ⁻³	3.961x10 ⁻³
	group6	2.919x10 ⁻⁴	1.583x10 ⁻⁴
	group7	5.727x10 ⁻³	4.603x10 ⁻³
Mass [Kg]		5.937x10 ⁻⁴	5.333x10 ⁻⁴
RMS Dev. [m]		4.152x10 ⁻⁵	4.663x10 ⁻⁵
Tensile Stress [MPa]		1.111	2.220
Compressive Stress Ratio		1.062x10 ⁻²	4.279x10 ⁻²

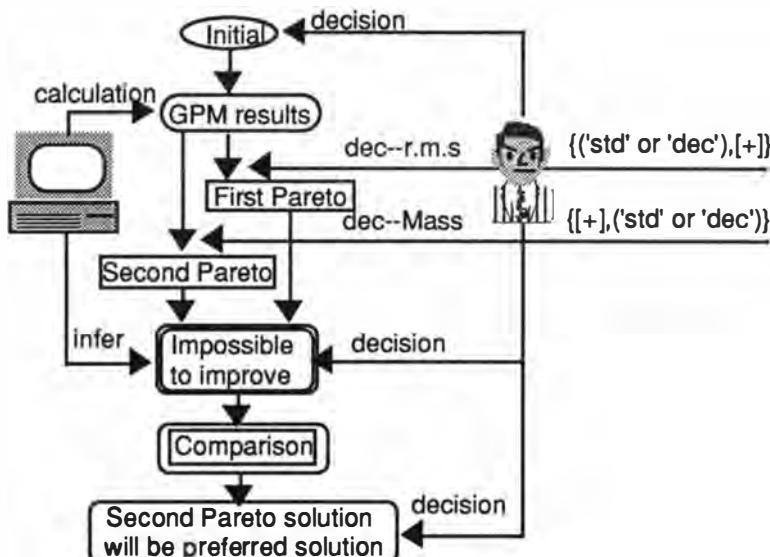


Figure 9 Interaction with Computer

In Figure 10, bold letters mean input by designer. In this case, we had two Pareto optimum solutions and we chose Pareto optimum 1 as a dominator of the relative number (for example $\{(\text{Mass}(\text{Pareto 2}) - \text{Mass}(\text{Pareto 1})) / \text{Mass}(\text{Pareto 1})\}$). Underlined letters are responses by computer and they mean a comparison Pareto 2 to Pareto 1. They are expressed as,

(DEC INC INC INC) ((MINUS LARGE) (EYE) (PLUS SAME) (PLUS SMALL))

which means qualitative directions and their relative amounts of change in "Mass", "r.m.s. deviation", "tensile stress" and "compressive stress ratio" with respect to the change of "r.m.s. deviation", relatively, and they meant that, for example, "Mass" had decreased "large" with respect to "r.m.s. deviation" increased.

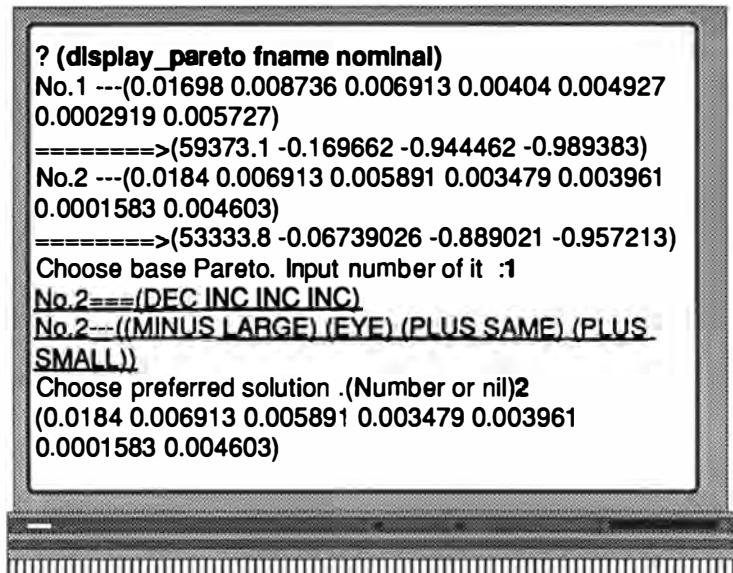


Figure 10 Comparison of Pareto Solution

Advantages of proposed method through the demonstrated numerical examples

- (a) As the proposed algorithm is interactive, such initial setting values of objective functions and constraints in the optimum design problem can be changed during optimization process. So the load of the designer in initial design will be greatly reduced.
- (b) As optimization process is proceeded with the information of qualitative expressions, the designer can grasp the behaviors of the system easily with the change of design variables and the load of decision making can be reduced.
- (c) In conventional multicriteria decision making problem, scalar optimization must be done as the change of preference of the designer. But in the proposed method optimum qualitative directions are inferred by the qualitative reasoning and the design variables are determined from semi-quantitative space in the database given by the standards. So the number of scalar optimization can be reduced much.

CONCLUSIONS

1. A new interactive multicriteria optimum design method using qualitative reasoning is proposed. We introduced qualitative differentials for multi-design variables and a qualitative trade-off ratio to be able to carry out an interactive multicriteria optimum design problem in qualitative grasp. And also we introduce fuzzy language to reduce ambiguities in qualitative values.
2. By the proposed method, we can infer the behaviors of design variables in semi-quantitative space more precise than in conventional ways. And the proposed method is turned out to

- be efficient method to reduce hard loads of the designers in multicriteria optimum design problems and to save the computational time for the optimization.
3. Through the applications of proposed method to modeled optimum design examples, it is found to be possible to do an optimization of complex systems even with qualitative grasp.
 4. Difficulty in implementation is to develop a more efficient inferring algorithm in order to avoid ambiguity in qualitative reasoning.
In order to solve those problems, we are now studying two approaches: One is to extend fuzzy languages and adopt some fuzzy addingrule, and the other is to make some intelligent data base systems to predict qualitative partial derivatives and to indicate some goals explicitly after some iterations.

REFERENCES

- Widman,L.E.,Loparo,W.A,Nielsen,N.R (1989). *Artificial Intelligence, Simulation & Modeling*, John Wiley & Sons
- Arakawa,M.,Yamakawa,H. (1990a). A Study on Optimum Design Applying Qualitative Reasoning,*Trans. of J.S.M.E* ,Vol.56 No.522:pp398-403(In Japanese)
- Arakawa,M.,Yamakawa,H.(1989). A Study on Multi-Objective Optimum Design Applying Qualitative Reasoning, *Proc. of the Int. Sympo. on ACD&D'89* :pp.267-272
- Arakawa,M.,Yamakawa,H.(1990b).A Study on Interactive Multicriteria Optimization Method Using Qualitative Reasoning, *Proc. of 1990 Dynamics and Design Conference* : pp.356-361 (In Japanese)
- Agogino,A.M.,Almgren,A.S (1987). Techniques for Integrating Qualitative Reasoning and Symbolic Computation in Engineering Optimization, *Eng. Opt.*,Vol.12:pp117-135
- Kuipers,B.(1986).Qualitative Simulation", *Artificial Intelligence*, Vol 29 :pp.289-338

A software architecture for design co-ordination

I. M. Carter[†] and K. J. MacCallum^{‡§}

[†]Engineering Design Research Centre
West of Scotland Science Park, Maryhill Road
Glasgow G20 0XA UK

^{‡§}CAD Centre
University of Strathclyde
75 Montrose Street
Glasgow G1 1XJ UK

Abstract. This paper considers the nature of design and how co-ordination of diverse resources is required to enable the consideration of the many different factors which affect the product throughout its life. A discussion of related work illustrates some of the methods for organising knowledge and controlling design activity. A software architecture based on a blackboard model with knowledge sources related in a hierarchy is presented as a means of supporting design co-ordination. Its use is demonstrated with an example application for the electromagnetic design of a turbine generator.

INTRODUCTION

Design in industry is a complex process generally undertaken by teams of people. The design team strives to produce the best possible design within constraints of time, given a set of requirements. This may mean producing a new concept, or reworking an old one to update it or make it more efficient; in any event, exploration of a variety of alternative solutions is desirable.

Design, however, does not exist in isolation. When viewed in global terms there is a lot more to engineering design than numerical and geometric modelling and meeting the technical specifications when designing an artefact, whether it is a minute electronic component or a major civil engineering construction. Many factors, often conflicting and in complex relationships, affect the total process and so must be considered (Derrington, 1987, Marechal, 1987). As well as the immediate technical design requirements that must be met, manufacturability (both for physical feasibility as well as for time and cost constraints), maintainability, environmental impact (of the product and its manufacture) and possibly decommissioning all have to be considered. In addition to these, marketability of the product will determine whether the design will be pursued, radically altered or dropped completely.

If these factors are considered only once a design is complete, much time and effort may be wasted either because the design has to be scrapped (too costly or impossible to manufacture) or because the design has to be extensively reworked (difficult to maintain or does not meet certain markets' legal requirements). The earlier that they are considered, the sooner any detrimental effects can be removed. This is important because much of the cost of a product may be committed in the very early stages of design (70% or more of the total cost is commonly committed during the first 3–4% of the actual spend (Andreasen, 1990), see Figure 1). Any subsequent redesign can have very costly knock-on effects in terms of technical or manufacturing complications, and hence in cost and timescale.

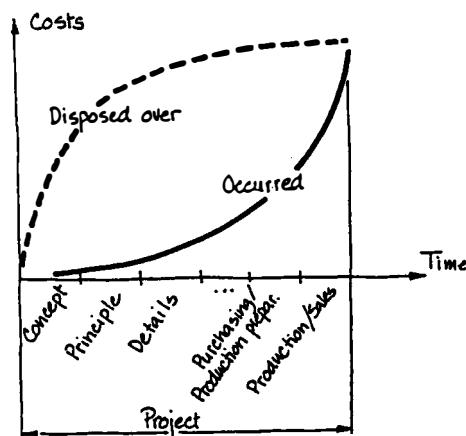


Figure 1. Project cost spend (Andreasen, 1990)

In order to succeed in design it is necessary to apply expertise from appropriate areas at the right time. In most organisations (even the smallest) sales, design, manufacturing and commercial knowledge will be vested in different groups of people. The problem is to organise and apply them and their knowledge in the most effective manner. This will be easiest in small companies, where communication is easier and structures more flexible. In large companies barriers tend to be created between departments, which slows down, or even stops, communication, and it then becomes difficult to apply manufacturing knowledge to the design stage or to get early cost estimates.

The environment in which engineering design is undertaken is often a commercial organisation with many people involved in the overall process (Harker, 1988). Each person involved has the problems inherent in design, but also has the problems of communication, interaction and co-ordination in an organisation.

Most companies or institutions have some form of management structure, often based on a hierarchy. Figure 2 shows a typical example, with directors responsible for major functional areas within the company and sector or department heads below them. This structure divides

the responsibilities across the directors and managers and defines the command and control channels. The various areas under a manager can be considered as the expert resources for which he is responsible and which he can apply to solve problems.

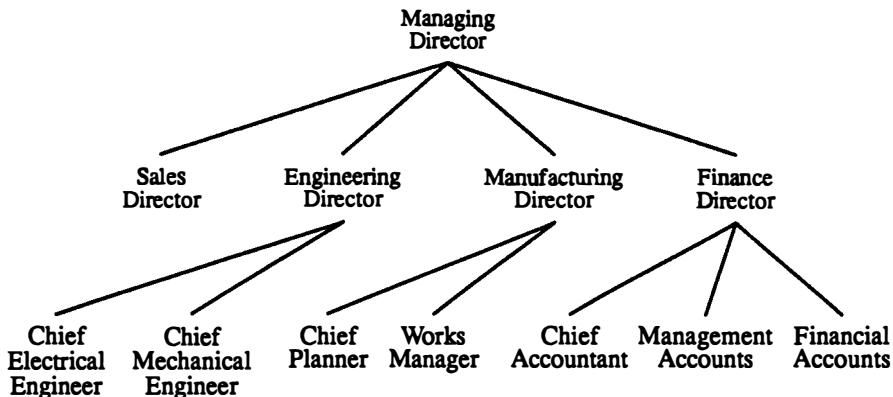


Figure 2. Typical company management structure

Small companies tend to have an informal structure, and one that is quite flexible in the way it can be used with greater interaction and less demarcation between the functions. Larger companies, on the other hand, tend to have more rigid and inflexible structures, primarily used as channels of communication, leading to an additional overhead of bureaucracy. Part of the function of such a structure is to control the activities that are taking place; part is to identify and organise the knowledge and information represented by the people, and hence allow access to that expertise by other parts of the structure. Communication and interaction in this structure can only take place effectively and efficiently when the people involved are talking the same language. However, the language often varies between departments and functions, causing confusion and inconsistencies.

Support for design is available in a number of different ways, and increasingly by computer systems. Those systems in place in industry are principally in the areas of computer aided draughting and design analysis. Generally these systems are reliant on the design being well advanced (i.e. they do not handle uncertainty or incompleteness) and do not cover all the business facets, outlined above. The latest generation of design systems emerging from research programmes are extending computer systems' abilities from analysis back towards conceptual design, thus giving the designer more support in the intellectually demanding tasks. They are, however, mostly derived from models of the individual design process, rather than the overall business process.

So we have a situation in which design, a complex process, must take place in conjunction with other processes, in limited timescales. While some computer based support is available, it usually provides help with particular aspects of the problem, normally in or towards the detailing stage. There is little assistance for either the designer or his management, in bringing together the efforts of many people and their resources, so that effective design can take place.

The type of computer support that is developing at present tends to concentrate on particular aspects of the design process (e.g. conceptual design systems), integration of a number of design programs or systems or looking at a slightly wider range of engineering factors (e.g. simultaneous engineering systems). There is, however, a significant requirement for design co-ordination which is not being addressed.

Co-ordination covers aspects of organisation and management of resources (people and systems), control of the use of resources and providing and oiling the interaction links that are required. There is an opportunity to support co-ordination using computer systems which assist in the process, aid interconnection of the resources (thus helping to break the barriers, both technical and personal) and provide the flexible environment in which interactive design can take place. In this way, design systems will begin to bridge the gap between narrow technical design systems and company management, and thus reflect the full impact of design on the business, and the business on the design process.

The aim of this paper is to present a computer based system architecture which has been developed to support design co-ordination (Carter 1990).

SUPPORTING DESIGN CO-ORDINATION

Studies of design processes and design organisation in industry have led to the conclusion that the co-ordination of design is centred around, and relies on, three main factors: organisation and control, integration, and modelling of products (Carter, 1990). The organisation of diverse expertise and information resources (i.e. the resource base) and the control of their use are essential in ensuring that corporate knowledge is used effectively, in the right place at the right time. Integration of resources supports the requirement for providing access to resources not in the immediate vicinity. Product modelling provides the common "language" by which designs will be described and their information distributed. This section looks further at these three aspects of design co-ordination, and how they have been addressed by other work.

Organisation and Control

Engineering involves the integration of significant resources on a large scale, and it is thus generally necessary to undertake it in organisations, where economies of scale can also be enjoyed (Harker, 1988). The structure of the organisation and the systems used in it then become major factors in the degree of success of the organisation. The problems of organising and controlling design have long been recognised, with the development of organisation theory and the introduction of the first structuring methods by Taylor and Fayol at the turn of the century (Harker, 1988).

A product's component structure can be used to structure information, as in the Air-Cyl system (Brown and Chandrasekaran, 1985, Brown, 1985), which performs parametric design of air cylinders. Here, the design expertise is contained in a hierarchy of design specialists which mirrors that of the components, so each component has a set of knowledge which is used

to design it. This approach works well for the parametric design performed by Air-Cyl, where only dimensions and material types are to be selected and varied, and for well-structured products, but does not suit design which involves parts configuration or innovation. Kitzmiller and Jagannathan (1987) have used this example to test their ideas for a comprehensive design environment in which the design problem is divided into a hierarchy of subproblems, each of which is then assigned to separate agents who have to co-operate to produce the final design. Their reason for this approach has been the need for a team of specialists and organisations to tackle large, complex design problems.

Often the organisation of knowledge within a system is by the tasks to be accomplished or the phases of the design. This can be seen with both Destiny (Sriram, 1986) and IBDE (Fenves et al, 1990), where different knowledge based systems are used for the different stages of the design process. Destiny uses a high level, strategy, knowledge source to establish the sequence of execution of tasks (and hence which knowledge source to use). A third level of knowledge sources are algorithmic systems, used by the knowledge based systems. IBDE has integrated seven separate systems, using a project data manager to pass data in the correct format between individual systems and the project store when the system control initialises the system. The VLSI tools in CADWELD (Daniell and Director, 1989) are organised in an abstraction hierarchy according to their type of operation (e.g. simulation or design capture). The selection of an appropriate tool then involves identification of the task and the rating and comparison of the tools able to meet that task.

Control of design activity can take place in a number of different styles. Jones (1980) gives five types: linear, cyclic, branching, adaptive and incremental. The type of control used will often depend on the organisation structure. In the case of a hierarchical structure, control of activity is usually achieved in a top-down manner, as with Air-Cyl (Brown and Chandrasekaran, 1985, Brown, 1985). In this case a decision is made at a high level of the component structure and tasks are passed down the hierarchy. As lower level solutions are developed, so the control passes back up the structure. The passing of control down a hierarchy, by passing selected tasks down to selected specialists, is very similar to company management.

An alternative method of handling control is shown by IBDE, Destiny and CADWELD, which are all based on a blackboard model (Nii, 1986a, 1986b). IBDE has a controller for the control of and communication between different processes, identifying the most suitable piece of knowledge to apply at any given time (usually depending on the state of the design and on blackboard messages) and then applying it. The controller is intended to be as generic as possible, so that different strategies can be applied. Destiny uses an agenda and inference mechanism to execute the knowledge source with the highest priority (the first on the agenda). CADWELD uses the blackboard model to handle communications but not as a global data space. Tools are requested to volunteer for the chance to operate, with the final decision being made by the designer or high level tools. By this means opportunistic tool selection is achieved, along with minimal domain knowledge being held in the framework for that selection.

Integration

A major concern in industry is to make better use of current facilities and resources, enhancing their capabilities rather than making them redundant through the introduction of new technology. David (1987) presents two possible ways of integrating design tools with a knowledge management system (KMS): i) direct access of the KMS by the tools; ii) isolation of the tools from the KMS, with data being transferred before and after execution. IBDE (Fenves et al, 1990) uses the second approach, by using a project data manager to pass the data between existing systems. CADWELD (Daniell and Director, 1989) integrates VLSI tools by treating them as objects (classified in an abstraction hierarchy) and wrapping them in a layered front end which defines the interaction and usage requirements.

Chalfan (1986, 1987a, 1987b) has investigated four alternative methods for integrating existing design programs used in preliminary aircraft design. These are: a procedural program which subsumed the original programs; a database management system; a procedural program that called the original programs as separate processes; and a knowledge system which executes the original programs as independent processes. The last system shows significant benefits from increased productivity and reduction of errors.

Product Modelling

Product modelling at one time meant a craftsman making an artefact. Now it means such things as three dimensional solid models of complex geometry and dynamic flow simulation. Many people working on different parts of the same product, possibly in different organisations, makes it important that the language and details of the product description are the same, so that confusion and inconsistencies are minimised or eliminated completely and so that communication and interaction can take place effectively and efficiently. A product description language can be used to identify the product's components and their attributes, providing the basis for systems such as design, costing, planning and stores or inventory (especially if they are structured, parametric systems). Examples of parts of a product description language are NEI parsons' Product Structure (Anderson, 1989), which defines the products functional breakdown, and the Brisch system (Hyde, 1981), which defines families of similar components types.

Both Corby (1986) and Eastman (1980) wish to provide designers, and other users, with the ability to view different sets of information about the design or different views of the same information, thus helping the communication process. The individual systems of IBDE (Fenves et al, 1990) use their own model representation schemas, with the central data store being based originally on a relational model and then on an object model. Destiny (Sriram, 1986) represents the building model as a set of structured objects organised in an abstraction hierarchy.

THE HIERARCHICAL OBJECT ORIENTED BLACKBOARD SYSTEM (HOBS)

Overview of the Architecture

The central concept in the HOBS architecture is that segments of design expertise or resources are organised into a hierarchy of objects (or Knowledge Sources), which are controlled through a blackboard mechanism (the Executive), and communicate with one another about the Product through a common area referred to as the Workspace. A Knowledge Source contains design knowledge, procedures and algorithms, the Workspace provides a global data area for the development of alternative solutions plus some generic modelling tools, and the Executive provides the general operating cycle. In addition to these are the User Interface, providing the user's view and method of access to the system, and the External Facilities, for connection to other systems.

The Knowledge Sources

A knowledge source is an autonomous piece of design expertise encompassing a specific function or scope of activity within the design process; it can be used at any appropriate time during the design process, and thus may be capable of application to both preliminary and detailed design. It can be a design rule or an analysis program, a piece of expertise or an expert member of a team. Thinking of a knowledge source as an analysis program reinforces the integrating notion of the architecture, while the role of a knowledge source as a member of a team encourages the co-ordinating nature.

It is possible to identify three types of knowledge source that can exist in a design environment: i) the fully automatic design module or self-contained analysis program which needs only to be integrated with the architecture; ii) the design support system, where the designer takes an active role; and iii) the human based activity, where no direct computer support is available. All three of these types can be incorporated within the architecture. Whichever sort of knowledge source is applying, it must be able to communicate and interact with other knowledge sources and architecture components.

Each knowledge source in HOBS comprises a "body" and a "shell", as shown in Figure 3. The body contains the actual design expertise, such as how to design or cost a component, or how to analyse a system's performance. The shell is used to couple the body to the system by defining the contents of the knowledge source and when and how it should be used. It communicates with the rest of the system through messages and data in the Workspace, and also through the user interface. It represents the "self-knowledge" of a knowledge source. This arrangement is similar to that used in CADWELD (Daniell and Director, 1989) to couple VLSI design tools into an environment, although in that case the shell does not seem to be so sophisticated.

Knowledge sources can be organised in a hierarchical control structure. This facilitates the decomposition of problems and tasks across a series of co-operating experts. The structure also allows the focusing of attention on a small number of knowledge sources at any one time, reflecting normal design management practice. The shell defines a knowledge source's parent

and children in this structure, with the body being able to interact and make use of the children as and when required.

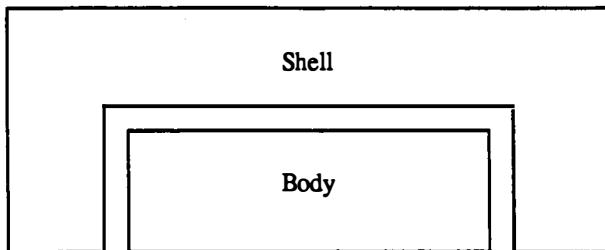


Figure 3. Knowledge Source Structure

The Workspace

In the standard model of a blackboard system (Nii, 1986a, 1986b), a global data area provides the data modelling and storage facilities of the system. In the case of HOBS, this role is met by a structured object data modelling environment with, in addition, some specific modelling tools. The Workspace thus becomes a general resource area for the system, handling data, tools and messages. It comprises items for each of these resource types: the Database, the Toolbase and the Message Board. As the main model development area, the Workspace has close interaction with the other parts of the architecture, in particular the knowledge sources, and the user. The Workspace provides the overall data management, such as handling the transfer to and from external systems and the storage and retrieval of the complete set of Database models and Message Board messages. The Workspace is thus more than the sum of its parts, having its own functionality.

In general, the Database stores the requirements and solution models for the problem, which provide a major forum for communication between the knowledge sources. The Database models are based on an abstraction structure of model objects, representing physical components and abstract concepts. The model objects have standard data structures which incorporate different relationships between the objects (e.g. part of, has features, connected to, etc.).

The Toolbase is a set of generic representation and manipulation facilities which can be used to manipulate specific aspects and concepts of different problems. Individual tools should be available for numerical modelling, geometric modelling, temporal modelling, etc.

The Message Board allows messages to be passed between components of the system (either specific messages or global broadcasts), so that actions can be requested or information passed.

The Executive

A blackboard system's control mechanism is concerned with deciding what action or operation is to be performed next. In many systems this is a case of collecting and scheduling

the responses of the knowledge sources to the current situation, and on some occasions the application of a strategy (Englemore and Morgan, 1988, Hayes-Roth, 1985, Erman et al, 1980, Kloth, 1988).

In the HOBS architecture the control mechanism is part of the Executive. The Executive contains the primary operating cycle, which solicits, receives and sorts bids for work from the knowledge sources, and selects the next one to operate. Once a knowledge source has finished operating, the control mechanism moves down to that knowledge source and solicits bids from its children, thus decomposing the problem. Once any of these have operated (and themselves decomposed), control moves back up the structure, and rebidding occurs to find the next most suitable knowledge source.

Interaction Specification

The system is composed of a number of components which interact with one another, with the fundamental mechanism for interaction being that of passing messages.

The system components are defined as objects which have a specified functionality and which respond to and send certain messages and message types. Each component can request actions or information from other components by using defined messages. The set of defined messages, and which components can use them, comprises the interaction control command set.

In addition to the interaction control set of messages, others can be placed on the Workspace Message Board. The Message Board can be used as a type of scratch pad or memo pad for passing unstructured messages between system components (particularly knowledge sources). In some instances these messages might be used to describe and formulate a design strategy, and thence the design progress and a design history.

In the operation of the system, the principal “active” components are the Knowledge Sources and the Executive. The information (or message) links between the components are as shown in Figure 4.

Taking each component in turn, the types of messages that they can send, and to whom, are listed below:

- (a) The Executive can send messages to a knowledge source asking for a bid or passing control to the knowledge source for it to operate. These form the basis of the control mechanism. It can also send messages to each of the system components requesting status information.
- (b) A Knowledge Source will respond to the Executive control messages by sending it a bid value message or an operation completion message. It can also send the Executive a message to decompose the current task to its children.

A Knowledge Source will send messages to the Database to create, store and delete the models and to access, update, create and delete the model objects which

form the models (consistency of these operations is handled by the Database). The Database will return values and any appropriate acknowledgement.

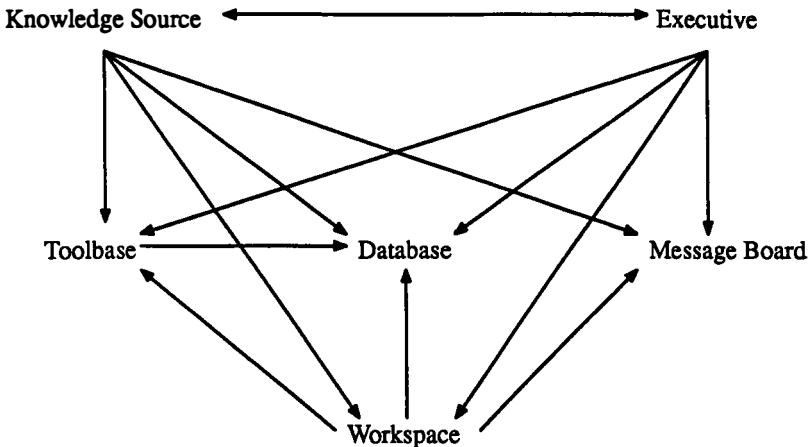


Figure 4. Component Information Links

It will send messages to the Toolbase for selection of individual tools and for tool command execution. The Toolbase will return values and other appropriate responses to these messages.

It can send messages to the Message Board to add, delete or find messages. All of these can be considered as modelling messages. The Message Board will return logical values relating these messages, indicating that a message has been found or that the operation was successful.

Finally, it can send system level messages to the Workspace. These are for a transfer of model data between the Database and an external system, and for the transfer of all the Workspace information (i.e. the Database models, the message information and the tools' model states) to "save its state".

- (c) The Workspace will send messages to the Database for the transfer of data with external systems and the Toolbase and Message Board for Workspace state storage.
- (d) The Toolbase will send access and update messages to the Database as part of the initialisation and shut-down process of a tool.

The user can send any of the messages to any of the components, and will also be able to send messages to the Executive to initialise or save an application system, and to the Workspace to save the complete Workspace state.

AN APPLICATION TO GENERATOR DESIGN

To illustrate and evaluate the use of the HOBS architecture, a number of aspects of steam turbine generator design were implemented and compared with more conventional approaches. In this paper, the particular aspect concerned with electromagnetic design is described.

The Application

The design and manufacture of steam turbine generators is a major project operation undertaken on a contract basis. In order to win contracts, manufacturers must produce a tender, comprising design, manufacturing, project management, financial and commercial information relating to the offered product, all meeting the specification laid down by the customer. The tender documentation contains a large amount of information and must be produced in a limited timescale (three months, for example). It is necessary to produce a compliant design as soon as possible, which limits the amount of optimisation which can take place. This can have serious effects when significant design changes are made late in the day, because major modifications will be required to other components, and knock-on effects will be produced in manufacturing requirements, delivery dates and costs.

It is therefore important to maximise the amount of related information that can be considered and the amount of expertise which can be brought to bear on the problem in the earliest stages. In addition, it is advantageous to consider alternative designs from both technical and commercial viewpoints, so that the best overall design can be chosen (i.e. the most profitable, most likely to succeed).

The electromagnetic design of a generator involves choosing some design parameters and developing the numerical model of the design based on those parameters. Either a design can be developed from scratch or, more usually, a previous design is modified to meet a new set of requirements. In both of these cases, the designer will develop the design to a certain degree, before making a decision about its suitability, and then either accepting it or modifying some parameters before redesigning. Account must be made at all decision points regarding the effects of any decision on cost or manufacturing. For example, increasing the number of stator slots may improve performance, but will also increase manufacturing costs; alternatively, decreasing the number of slots too much will create manufacturing difficulties in other respects. Current practice is to modify parameters and re-submit a computer program which produces the complete generator performance analysis.

Modelling the Design Problem

The structure for the example application was based on a Product Management Structure. A subset of this structure is shown in Figure 5. The structure initially consisted of the upper five knowledge sources, with the others being added as the generator design application was developed to reflect the subdivision of design expertise.

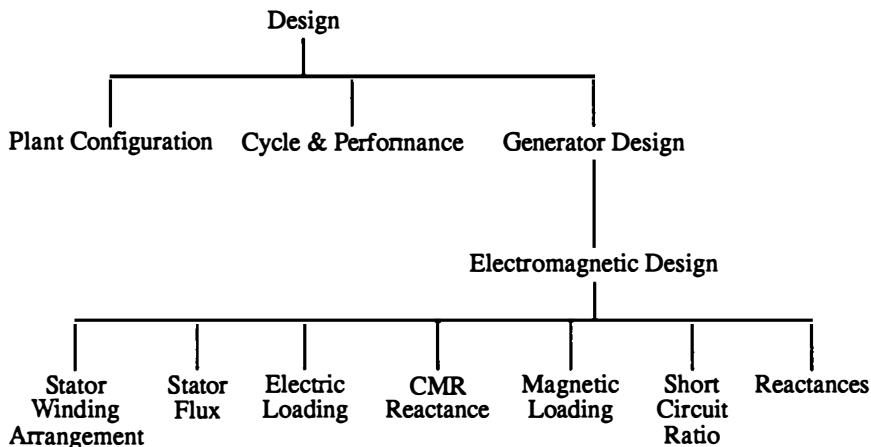


Figure 5. Knowledge Source Hierarchy

The problem for the generator design case study is to define one or more designs which meet technical requirements laid down by the designer, but which do not infringe current design practices, except where so decided by the designer. Having produced the designs, they must be evaluated for their suitability against multiple criteria, such as cost, manufacturability and customer requirements and evaluation conditions as part of the Generator Manager's role. Finally, a design must be chosen to submit to the customer. All operations should be via direct interaction with the designer, who will take the leading role in directing the design strategy. The methods and procedure required for the electromagnetic design of generators are documented (Parsons, a and b), but are written in a procedural and necessarily limited form. The documentation available allowed an adequate level of design knowledge to be represented, supplemented by discussions with designers. Details of existing designs are also available, allowing a similar design to be chosen as a starting point for the design process.

The general flow of the design tasks is shown in Figure 6 and Figure 7. Figure 6 shows the activities at the top level of the knowledge source structure, principally in Generator Design, whilst Figure 7 shows the activities in and under Electromagnetic Design, for designing and redesigning a particular generator. Figure 7 fits into the blocks "Design Generator", "Redesign Generator" and "Design New Generator" of Figure 6.

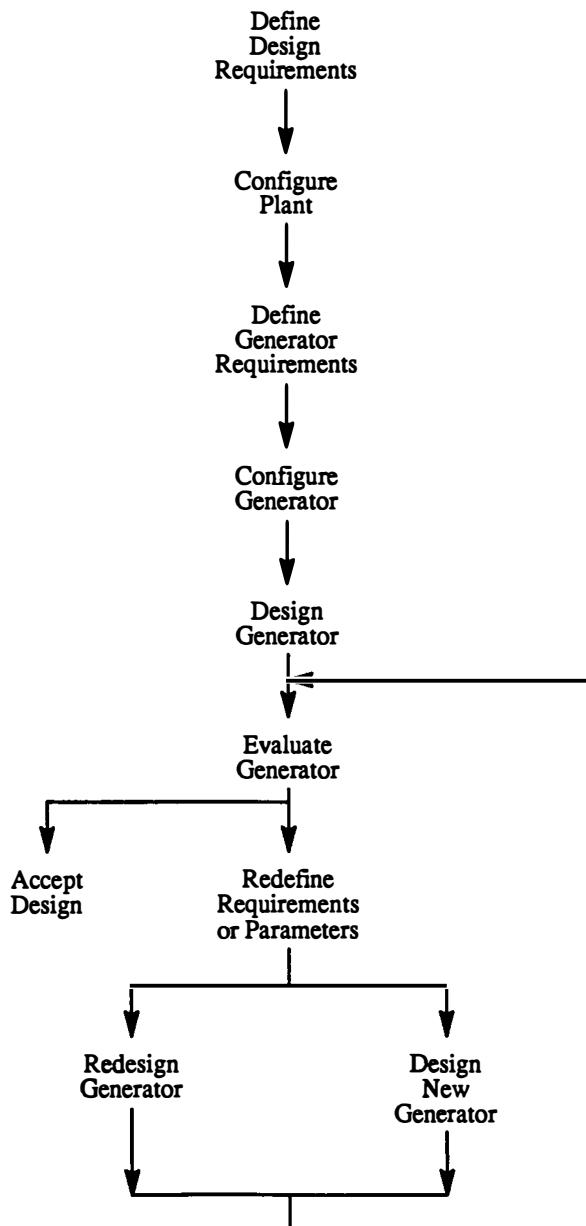


Figure 6. Generator Design Flow

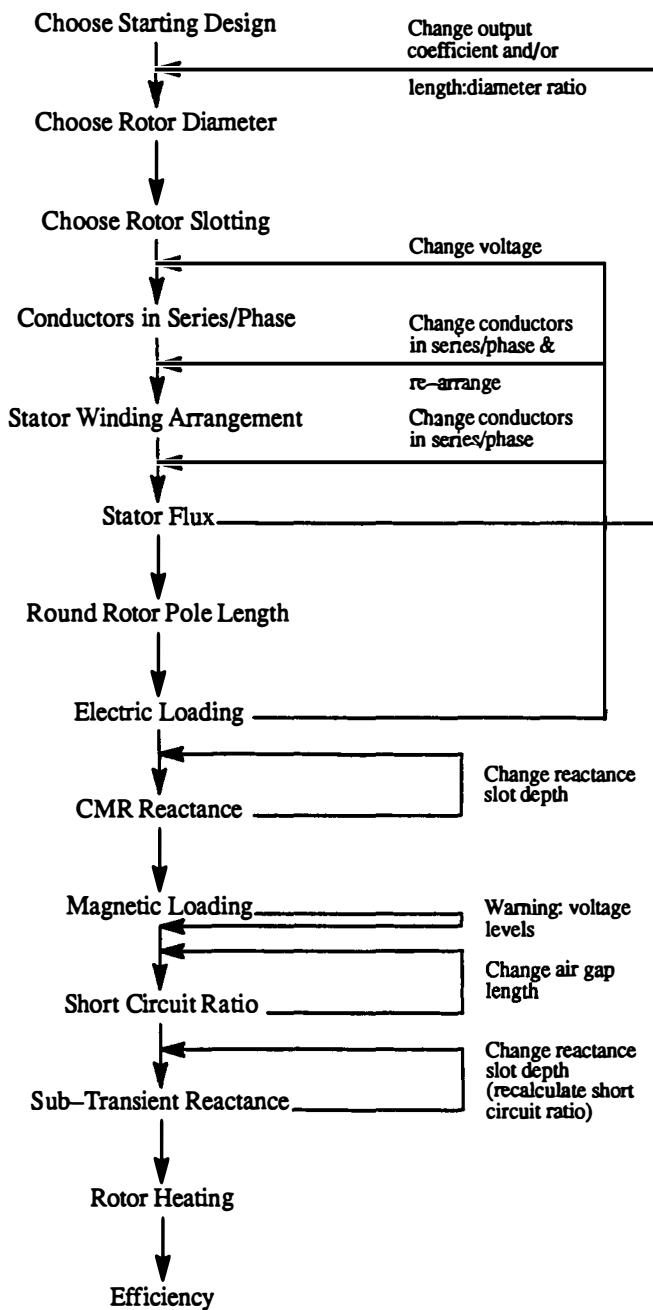


Figure 7. Electromagnetic Design Flow

Implementation and Operation of the Application

The implementation involved decomposing the design procedure into separate, child, knowledge sources and thus devolving responsibility, as shown in Figure 5. The approach used message passing (via the Message Board) as its means of moving from one task to another in Figure 7 and for communication between the knowledge sources. Messages are placed to indicate that an operation is required (e.g. "stator winding arrangement required") or has been completed (e.g. "pole length rounded") or that a condition has not been met (e.g. "stator flux unacceptable"). As the system asked the designer whether its choice of knowledge source was suitable each time, the designer can significantly alter the order of execution by specifying a different one.

The system has been developed so that Electromagnetic Design and its knowledge sources can design a single model whilst Generator Design can specify, evaluate and consider a number of designs. Thus design control is exercised by the manager (Generator Design) with designs actually produced by its staff (Electromagnetic Design).

All numerical calculations relating to the generator's performance were undertaken in the Designer tool (MacCallum and Duffy, 1985) using a suitable model of the generator (Carter, 1988). This permitted design changes (as shown by the feedback loops in Figure 7) to be made more easily than at present and the knowledge sources to contain more general design knowledge (e.g. how to change the model's performance). It also allowed investigation of the use of generic tools and the relationship they (and their models) have with the Database models.

A simple cost model was incorporated in the Database model. This functioned by automatically recalculating a component's cost when one of its critical parameters changed (the rotor's length or diameter, for example), and forcing a roll-up of costs from that point. In this way, as soon as major factors were changed in the Database model, the costs were recalculated. Although the costing formulae used were not accurate, they demonstrated the benefits of having a rapid rough costing mechanism available to the designer, getting a feel for the final price, but also being able to compare the relative prices of alternative designs. This was a feature that was particularly appreciated by practising designers.

The knowledge sources were defined by a generic class inherited by each individual knowledge source. As an example, Figure 8 shows the definition of Electromagnetic Design, in which three slots are used to define the bidding information (i.e. the shell) and one method forms the body (the method calls a procedure, which can be easily replaced, thus permitting modification of behaviour or upgrading without redefining the class). The bidding conditions are, in this example, five exclusive messages which are being awaited on the Message Board. The first and second react to the general message received from Generator Design, whilst the others occur during the operation of Electromagnetic Design and its children. The bidding mechanism (inherited from the generic KNOWLEDGE_SOURCE class) constructs the complete conditional statement from the individual conditions and the conjunctions. If the complete condition is true, it uses the bid values to calculate the final bid to be made to the

Executive. In this example, the bid value could (in theory) vary between 0 and 250, depending on how many of the individual conditions are true. In this way the relevance of the knowledge source to the current situation will vary and can be measured, thus providing the method by which adaptive and opportunistic behaviour is produced. As the complete conditional statement is only constructed when it is actually required and all of the bidding information is stored as lists in slots of the object, the information can be changed easily and the effects exploited immediately.

```

class electromagnetic_design isa KNOWLEDGE_SOURCE;
    slot bid_conditions
        default
            [[MESSAGE_BOARD~find([design generator ==])]
             [MESSAGE_BOARD~find([redesign generator ==])]
             [MESSAGE_BOARD~find([stator_flux unacceptable])]
             [MESSAGE_BOARD~find([stator_flux acceptable])]
             [MESSAGE_BOARD~find([electric loading acceptable])]];
    slot bid_conjunctions
        default
            [[or] [or] [or] [or]];
    slot bid_values
        default
            [50 50 50 50 50];
    method operate;
        “active”->self~status;
        emd_operate(self);
        “ready”->self~status;
    endmethod;
endclass;

```

Figure 8. Electromagnetic Design Class Definition

This application can be used by a designer to develop a number of alternative design models, each of which will be evaluated. At the highest level, the overall task is subdivided, as shown in Figure 5, into three separate tasks (plant configuration, cycle modelling and generator design), the first two of which are not considered in detail in this example. The designer is first asked about project level definitions, such as project title and customer, which may (ideally) cause accessing of relevant company information (e.g. known company preferences). These general definitions allow the overall plant to be configured, after which the operation moves down a level (in both managerial and system’s terms) to Generator Design, where more specific design requirements are defined and the generator itself configured. This shows the first level of task decomposition and subsequent distribution to

different knowledge sources (and hence different designers). At this point, Generator Design will have completed its initial tasks, and will request a design to be produced. It does this by posting an appropriate message ("design generator model 2") on the Message Board and decomposing the current situation to its children. Electromagnetic Design reacts to the message, bids accordingly and is picked by the Executive (and by the designer). It first looks to see if there are any existing designs which may be suitable, requesting the design to confirm the choice before loading the default values into the Database model. The next step is to initialise Designer, transferring the Database model values and calculating the first few characteristic values to get a rotor diameter (chosen from a list of standard sizes). This is used by the knowledge source to select an existing rotor cross section, which is offered to the designer, as shown in the dialogue box in Figure 9. If this accepted (which is likely in the first pass), Electromagnetic Design requests some design actions (i.e. defines some tasks that need to be done) by placing two messages on the Message Board ("stator arrangement required" and "stator flux required"), to which its children will react. The design process moves through the stages of Figure 7, with the designer making decisions, evaluating the design and iterating as much as necessary. The ordering of the knowledge sources is determined by messages placed by the knowledge sources on the Message Board and by the designer, who can override the choice of knowledge source made by the control or can modify the messages.

```

For this size of machine the 50 MW design default values are
recommended.

Is this choice Ok (y/n)? y
The default values have been set
#####
Loading model into Designer
#####

#####
Model loaded into Designer
#####

#####
Estimating the rotor diameter, and choosing the nearest
standard size.
#####
estimate of kVA_d1 is 37500.0 kVA
estimate of d21_d1 is 213150000.0 mm
estimate of diameter_d1 is 889 mm

#####
For this design, the chosen rotor cross section has the
following values:
    1. Rotor Wound Slots      = 32
    2. Rotor Slots per Circle = 49
    3. Rotor Conductors per Slot = 21

Note: reducing the number of conductors per slot reduces
manufacturing costs, but means a higher rotor current,
which will be more suited to a static exciter than a
brushless exciter.

Do you want to change any of these?
(Enter the choices, separated by spaces, e.g. 2 3, or
press return if no changes are required):

```

Figure 9. Suggestion of a Rotor Cross Section to the Designer

Once a design has been completed, Electromagnetic Design passes it back (in a control sense) to Generator Design where evaluation takes place, involving both the knowledge source, which gives the general information required, and the designer, who makes the decisions. Figure 10 shows the end of the evaluation process, and the point at which the designer decides the fate of the design. The design can be accepted, and hence released to the higher level of the design system, in which case it is copied to the general plant model and its ownership changes to the project level (as a primitive demonstration of version control). Alternatively, the design's requirements or certain of its parameters can be changed and either the design submitted for redesign or a new (alternative) design requested from Electromagnetic Design. This option has been chosen in Figure 11, in which a new design has been created and the designer is being prompted to modify some of the key parameters. Change of the requirements' definition permits more flexibility in the process, and is often required in order to consolidate a hazy original or to react to new or modified customer requests. The parameters which might be changed would principally be those about which decisions had already been made in the previous design. The process can then continue, producing a different design, which will also be evaluated. The application allows this to happen, giving some guidance, especially on typical parameter boundaries (through which the designer can pass if he wishes).

```

amount of copper on the rotor or by reducing the
electric loading.
Typical value = 0.0075
Calculated value = 0.000505
*****  

*****  

The cost = 937708.8
The evaluated cost = 1855098.0
The cost basis = 400000
Target nominal cost = 1200000
*****  

*****  

The evaluated cost is below the nominal cost for this.
size of machine.
*****  

*****  

At this point a technical suitability assessment should take
place, and recommendations of design modifications would be
made. For example, cost can be reduced by reducing the number
of stator slots.
*****  

The design has been evaluated, with the pass/fail status of
each test as follows:  

Short Circuit Ratio: passed
Sub-Transient Reactance: passed
Efficiency: failed
Magnetic Loading: passed
Electric Loading: failed
Rotor Heating: failed
Cost: passed  

You can now accept the design, redesign or create a new design
model. Alternatively, you can change back to another design
(if there is one).
1. Accept the current design, sign it off and copy it to
   the target design model.
2. Change the requirements and redesign the model.
3. Change the requirements and design a new model.  

Enter your choice: 3

```

Figure 10. Generator Design Evaluation

```

;; DECLARING VARIABLE stator3
;; DECLARING VARIABLE rotor3
;; DECLARING VARIABLE stator_conductors3
;; DECLARING VARIABLE stator_core3
;; DECLARING VARIABLE stator_slots3
;; DECLARING VARIABLE stator_teeth3
;; DECLARING VARIABLE rotor_conductors3
;; DECLARING VARIABLE rotor_slots3
;; DECLARING VARIABLE rotor_teeth3
;; DECLARING VARIABLE rotor_poles3

which of the following parameters do you want to change?

1. Power
   (current value = 38)
2. Power Factor
   (current value = 0.8)
3. Voltage
   (current value = 11)
4. Short Circuit Ratio Required
   (current value = 0.55)
5. Sub-Transient Reactance Required
   (current value = 0.12)
6. Output Coefficient
   (current value = 285)
7. Rotor Length to Diameter Ratio
   (current value = 3)
8. Rotor Diameter
   (current value = 869)
9. Rotor Pole Length
   (current value = 2700)
10. Stator Slots
   (current value = 54)
11. Wound Rotor Slots
   (current value = 32)
12. Rotor Slots per Circle
   (current value = 43)

Enter the numbers of those parameters which you wish to
change. Enter them in a single line, with spaces between,
e.g. 3 8 4: 10 7

What is the new number of stator slots?
(It should be divisible by poles*phases)
(current value = 54)
? 

```

Figure 11. Redefining Key Parameter Values

DISCUSSION

The interactions between knowledge sources take two forms: manager-staff and staff-staff. Generator Design acts as the manager for Electromagnetic Design, specifying the requirements of a design which Electromagnetic Design then produces, and GeneratorDesign checks and evaluates. Amongst Electromagnetic Design and its knowledge sources, the interactions are similar to those in a design team, requesting design tasks and reporting completions (via message passing). The order of execution of the knowledge sources is not predetermined, as it depends on the current situation as well as on the designer's final choice. In addition, modifications can be made to the bidding conditions of the knowledge sources, so that they respond differently.

The ability of the designer to halt execution and select a knowledge source to operate, whether in isolation or to delegate, allows greater flexibility for the designer. Putting different messages on the Message Board allows the designer to request operations without needing to know in detail where these happen. The effect of changing the "normal" sequence of operations depends largely on the level of knowledge in the knowledge sources. In this application, the system was able to cope, producing modified designs, although in some

circumstances, when an operation was requested by the designer before all necessary information was available, side effects could occur, although these are not terminal. Currently the designer has to recognise and deal with these.

Interactions at the lowest level mean that the designer has considerable flexibility in deciding, at an early stage, on features of a design, and having these decisions checked through, before committing to the design's full development. It is important to note the division between developing the design and being able to stand back, evaluate all the designs at a high level and then respecify the requirements. In this way, the benefits of having alternative design models can be attained, without confusing the actual design process or making it too complex. A further benefit from this arrangement is that the development of alternative designs could be distributed to various designers (working with identical versions of the system).

The decomposition of design knowledge into related modular blocks shows benefits in the way in which the knowledge can be used and also maintained. Knowledge division allows better identification of the capabilities of individual sets of knowledge, and hence permits better application of it. Also, the division allows the possibility of parallel application of sets of knowledge. Independence of operation of the knowledge sources permits a much more flexible response from the system than might otherwise be possible, and means that ordering is not predetermined and can even be modified during execution. Additionally, modularity of the knowledge sources means that they can be replaced and maintained more easily.

The knowledge sources respond to the messages placed on the Message Board when they bid or while they are operating and place them during their execution. This permits the messages to be formed and "written" at a late stage, thus avoiding a rigid sequence of operation, and hence giving more flexibility. Furthermore, the senders of the message do not have to know who the receiver will be (because the message is simply placed on the Message Board), thus removing this burden and hence freeing the process.

A final aspect to consider is the best way of developing the knowledge structures needed for an application. It is not always easy to identify the specific knowledge sources right at the beginning of development, but it is easier to describe a general procedure. Thus, as the procedure becomes more developed and better understood, knowledge sources can be defined for certain aspects and interactions made more complex. This type of learning and development style is something which might be supported during system execution, to extend the system's capabilities by learning from the actions of a designer.

Further developments of this application include extending the design capabilities of the knowledge sources to cover a wider range of features and checking, and to extend both the knowledge sources and the numerical model to include other types of generator design (notably, different cooling types). In addition, further modelling tools can be added to allow development of detailed and geometric models of the designs, as well as cost and manufacturing models.

CONCLUSIONS

The objective of this paper was to present a computer based system architecture for the support of design co-ordination. A number of conclusions can be drawn:

- (a) Design activity in an engineering business requires co-ordination that is different from the design process and is separate from the normal management planning and control activities, with the three factors of organisation and control of diverse resources (expertise and information), integration of those resources and product modelling, being central to design co-ordination;
- (b) The general blackboard model can be extended to provide a support architecture for design co-ordination. The extension involved the development of a blackboard system with knowledge sources related in a hierarchy which have self-knowledge. The revised architecture can be used to mimic the company organisation resource structure which is suited to inter-functional design co-ordination. This increases the flexibility of the model and encourages parallelism along with integration. Message passing provides a powerful and flexible method for interaction within the model;
- (c) An application of the architecture to electromagnetic design of generators has shown that the model works and that significant benefits over current practice can be gained. The application shows that the architecture has potential for further development into an advanced design system;
- (d) The architecture demonstrates significant potential for improving the effectiveness of designers within a business, but also having an impact on the operations and profitability of the business itself. This is through the identification, availability and application of company resources to design (and other) problems, so that greater accuracy of design and increased confidence in results are achieved.

REFERENCES

- Andreasen, M and Olesen, J. (1990). The Concept of Dispositions, *Journal of Engineering Design*, 1(1): 17–36.
- Anderson, A.F. (1989). Product Structure Guide, NEI Parsons, ISED/PS/89/1, 1989.
- Brown, D.C. (1985). Capturing Mechanical Design Knowledge, *Proceedings of the ASME International Computing Engineering Conference*, pp 121–129.
- Brown, D.C. and Chandrasekaran, B. (1985). Expert Systems for a Class of Mechanical Design Activity, in J.S. Gero (ed) *Knowledge Engineering in Computer-Aided Design*, Elsevier, pp 259–282.

- Carter, I.M. (1988). The Use of a Numerical Design Tool to Model an Electrical Generator, NEI Parsons, ISED/AI/88/1.
- Carter, I.M. (1990). Engineering Design Co-ordination and its Support by Co-operating Knowledge Resources, Ph.D. Thesis, CAD Centre, University of Strathclyde, submitted November 1990.
- Chalfan, K.M. (1986). A Knowledge System that Integrates Heterogeneous Software for a Design Application, *AI Magazine*, 7(2):80–84.
- Corby, O. (1986) Blackboard Architectures in Computer Aided Engineering, *International Journal for AI in Engineering*, 1(2):95–98.
- Daniell, J. and Director, S.W. (1989). An Object Oriented Approach to CAD Tool Control Within a Design Framework, EDRC, Carnegie–Mellon University.
- David, B.T. (1987). Multi–Expert Systems for CAD, *ICAD Systems I*, pp 57–67.
- Derrington, P. (1987). Management of the Design Process, in D. Sriram and R.A. Adey (eds), *KBES in Engineering: Planning and Design*, Computational Mechanics Publications, pp 19–33.
- Eastman, C.M. (1980). A Prototype Integrated Building Model, *IBS Research Report No. 3*, Institute of Building Sciences, Department of Architecture, Carnegie–Mellon University.
- Englemore, R. and Morgan, T. (1988). Blackboard Systems, Addison–Wesley.
- Erman, L.D., Hayes–Roth, F., Lesser, V.R. and Reddy, D.R. (1980). The Hearsay–II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty, *ACM Computing Surveys*, 12(2):213–253.
- Fenves, S.J., Flemming, U., Hendrickson, C., Maher, M.L. and Schmitt, G. (1990). Integrated Software Environment for Building Design and Construction, *CAD*, 22(1):27–36.
- Harker, K. (1988). The Evolution of Organisation Design, *IEE Review*, July/August 1988, pp 271–275.
- Hayes–Roth, B. (1985). A Blackboard Architecture for Control, *Artificial Intelligence*, 26(3):251–321.
- Hyde, W.F. (1981). Improving Productivity by Classification, Coding and Database Standardisation, Marcel Dekker, New York.
- Jones, J.C. (1980). Design Methods: Seeds of Human Futures, Wiley.
- Kitzmiller, C. and Jagannathan, V. (1987). Design in a Distributed Blackboard Framework, *Intelligent CAD*, I, pp 223–233.
- Kloth, M. (1989). Some Ideas on a Blackboard System for Design Tasks, *Workshop on Blackboard Systems, IJCAI '89*, Detroit.

- MacCallum, K.J. and Duffy, A. (1985). A Knowledge Based System for Handling Quantitative Information, *Engineering Software IV*, Springer Verlag.
- Marechal, G. (1987). The ARCADE and OSA Contribution to the Theory of Integrated CAD Systems, in IFIP *Design Theory for CAD*, pp 407–437.
- NEI Parsons. (a). Generator Handbook (internal document).
- NEI Parsons. (b). Design Sheet Calculation Manual (internal document).
- Nii, H.P. (1986a). Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures, *AI Magazine*, 7(2):38–53.
- Nii, H.P. (1986b). Blackboard Systems: Blackboard Application Systems, Blackboard Systems from a Knowledge Engineering Perspective, *AI Magazine*, 7(3):82–106.
- Sriram, D. (1986) DESTINY: A Model for Integrated Structural Design, *International Journal for AI in Engineering*, 1(2):109–116.

Knowledge-based engineering assistance

H.-J. Held, K.-W. Jäger, N. Kratz and M. Schneider

FAW-Ulm

Forschungsinstitut für anwendungsorientierte Wissensverarbeitung

W-7900 Ulm

Germany

Abstract. In this paper we propose a knowledge-based enhancement of conventional CAD that meets the requirements of complex mechanical design tasks. This enhancement is based on the explicit representation of design knowledge. Special attention is directed towards representing, evaluating, and inferring dependencies restricting design alternatives. The second major aspect is the determination and classification of relevant knowledge within a design process. The derived concept will be discussed by using the design of rotational parts as an example. Above all, we go into the integration of the knowledge-based systems with conventional software as, e.g., CAD, calculation and simulation modules in more detail.

1. INTRODUCTION

The development of manufacturing processes and the idea of a Computer Integrated Manufacturing (CIM) imposes many additional requirements on the process of mechanical design activities (Spur, Germer, and Lehmann, 1989). The main reasons for this development are based on the combination of decreasing product life cycles with increasing complexity of products, new materials, and an increasing number of alternative manufacturing methods and processing equipment.

Within these terms of reference, the CAD-engineer has to consider the requirements of subsequent processes. The main issues are, e.g., design for manufacturability, design for assembly, and cost estimation.

It is obvious that the engineer can hardly consider all these different views on his artifact in an adequate way without getting computer-based assistance, that is by far more effective than the support he gets by conventional CAD primarily supporting the geometric modelling. On the other hand, it does not seem adequate to propose a completely new environment for design processes.

This leads to a quite important issue in evaluating developments of knowledge-based systems. As knowledge-based systems are meant to assist a person in performing some tasks within a potentially by far more complex process, the system has to be integrated into the conventional procedures and organization. Because many activities are already based on electronic data processing activities, it is essential not only to have an administrative integration, but also to build a software integration which provides the possibility to use results being obtained by a knowledge-based system in software packages that support subsequent activities. For the concrete context of engineering, this implies the integration with CAD-systems as well as calculation and simulation modules.

Based on these considerations, the FAW-project CAD-AI aims at the realization of the prototype of a system that meets the following requirements:

- * The engineer works within his usual CAD-environment that has to be enhanced by additional commands.
- * The knowledge-based support should be able to check a design interactively within the design phases layout and detailing.
- * The knowledge-based system should offer an interface to different (arbitrary) CAD-systems.
- * By exchanging the knowledge-base the knowledge based module has to be capable of supporting different applications.
- * The AI-module should support different views on the product within the design process.
- * The system should be able to support the integration of knowledge about subsequent production processes within the design process.
- * Different kinds of design-knowledge including design-rules, standards, design elements, and dependencies of parameters should be modelled in an adequate way.

If using a knowledge-based assistance that meets the requirements mentioned above, the engineer will be able to improve the manufacturing process, reduce production costs, and significantly improve the quality of the product. Moreover, a continuous flow of information between design and manufacturing processes will reduce the number of cycles needed to achieve a working prototype and thus the time to develop products of increasing complexity.

2. CLASSIFICATION OF DESIGN KNOWLEDGE

Talking about knowledge-based assistance in the process of mechanical design one has to start by identifying and classifying the relevant pieces of information including their logical structure, and for the purpose of knowledge acquisition the potential sources for the different kinds of knowledge.

The following three different aspects of knowledge have to be discussed (Figure 1):

- knowledge structure,
- knowledge contexts, and
- knowledge sources.

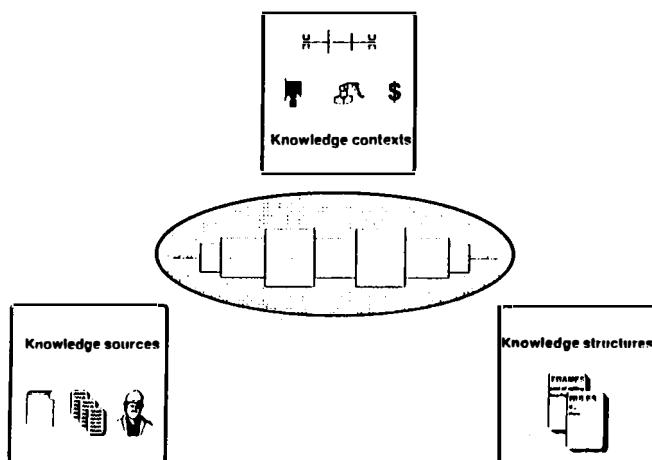


Figure 1. Different aspects of knowledge

The knowledge structures are the relevant criterium for selecting the mechanisms being used to represent knowledge in a knowledge-based system. In our application three different kinds of logical structures have to be considered :

- design objects (features) and their relevant attributes,
- design rules, and
- dependencies within one object (internal) or between two objects (external).

The knowledge contexts represent the possible different views on a product and thus specify the relevance of some pieces of information for a special purpose. As different views one may represent for example:

- manufacturing,
- assembling,
- functionality, or
- costs.

Especially for knowledge acquisition the determination of relevant knowledge sources is extremely important. One may distinguish for instance:

- literature,
- standards,
- material data sheets, or
- the experience of a human expert.

3. A CONCEPT FOR REPRESENTING AND OPERATING DESIGN KNOWLEDGE

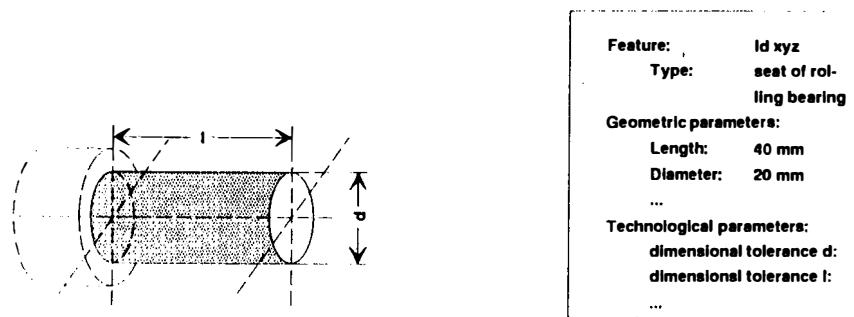
As stated in the last chapter, the discussion of knowledge representation includes three logical structures:

- design objects (features),
- dependencies, and
- design rules.

The expression *design object* or *feature* signifies a functional and geometrical object that is used as part of a concrete design (Roller, 1989). Talking about the design of a shaft, such elements are, e.g., a seat of a rolling bearing or an undercut. In addition to the representation of these elements in a CAD-system, their geometric description is

enhanced by giving their functionality and other additional data (technology, tolerances, etc.).

Features are represented by using frames (Figure 2) that include all relevant information on these design objects (Fikes and Kehler 1985).



Geometry

Element representation AI-module

Figure 2. Feature representation in CAD and AI

Consider dependencies between different parameters because of the potential complexity of the product, is one of the most difficult tasks for the engineer. Conventional CAD only support the consideration of known dependencies within the small and fix context of macros or programs for variational design. In some systems, the engineer may also explicitly define dependencies between arbitrary geometric parameters that are evaluated by the system. As an important aspect of our approach, we try to generate such dependencies automatically from the given context within the design process. Therefore, dependencies are represented together with the respective feature. If this feature is used as part of the concrete design object, the dependencies between its parameters and some other parameters will be activated dependent on the given context.

As mechanism for representing dependencies we use constraints, because the generated networks of constraints guarantee consistency on the one hand and on the other

hand may be used to calculate parameters that have not yet been specified by the engineer (Güsken, 1987; Richter, 1989).

Ending our discussion of knowledge representation, we deal with design rules. Design rules incorporate knowledge which alternatives should be used to layout a specific region of the product. Such design rules may be easily represented by production rules.

The actual syntax for expressing constraints and design rules is given by the examples in Figure 3.

*** Constraint for seat of
rolling bearing**

CONSTRAINT Greater_Equal_1,2
(Diameter,
All/OBJ (RELATION connected)
Diameter)

*** Production rule**

IF function = seat of rolling bearing
THEN surface quality = high

Figure 3. Constraints and production rules

In addition to the already specified mechanisms for representing design knowledge, one has also to specify the representation of dynamic knowledge, which means the representation of all information that has been derived as part of a concrete design task.

This knowledge includes the representation of the product in a *base-model*. This base-model may be seen as some kind of interface between the knowledge-based module and the geometric data-model of a CAD-system. The base-model is realized by a semantic network using the features as nodes and some specialized spatial relations that describe the topology by edges. Based on this base-model, additional specialized models that represent different views on the design-process may be generated (Figure 4). These models not only contain the relevant objects and the information associated with these objects, but also relations between these objects. These relations may connect objects within a specific model as well as objects from different models.

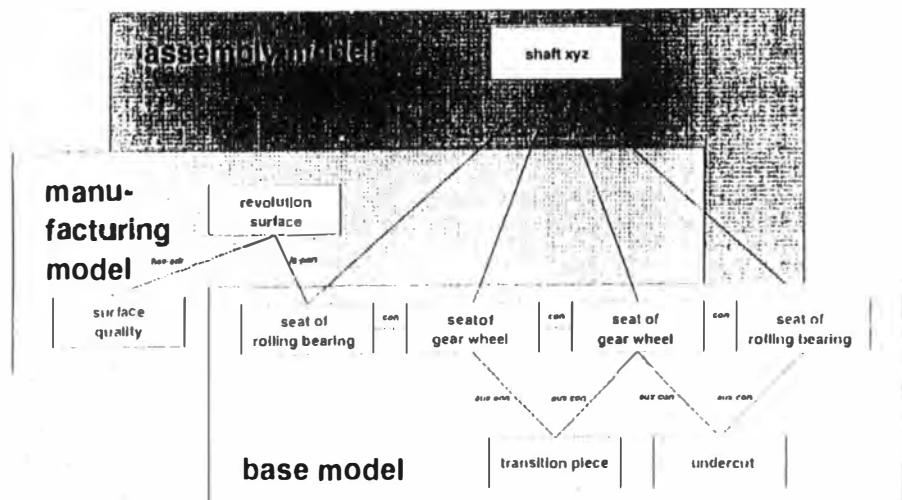


Figure 4. Model representation in the AI -module

On top of these models a constraint network exists that consists of all instantiated constraints in order to maintain and evaluate dependencies between parameters.

The inference mechanisms include:

- instantiation of objects,
- instantiation of constraints,
- constraint interpreter, and
- design rule interpreter.

The instantiation of an object in the base-model is triggered by a respective action (generation of a feature element) in a CAD-system. Using the parameters provided by the CAD-system, an instance of a specified object, e.g., the seat of a rolling bearing, is created as part of the base-model by the knowledge-based module. This object is associated with the respective element in the CAD-system.

Next, the context of the generated instance is represented by building up the appropriate spatial relations to other already generated elements. Objects in the different specialized models are similarly treated.

Instantiating constraints is automatically the basic difference to so-called parametric design in some CAD-systems. Instantiating an object in the dynamic database directly implies the instantiation and activation of all constraints that are relevant for the given context.

The constraint interpreter that is primarily based on local propagation, is capable of treating symbolic constraints as well as numeric ones. For numeric constraints, the interpreter provides special methods to solve equations (Güsken, 1987; Voß and Voß, 1987).

The interpreter for design-rules is a relatively simple interpreter for production rules.

A complete survey of the architecture of the knowledge-based module is given by Figure 5.

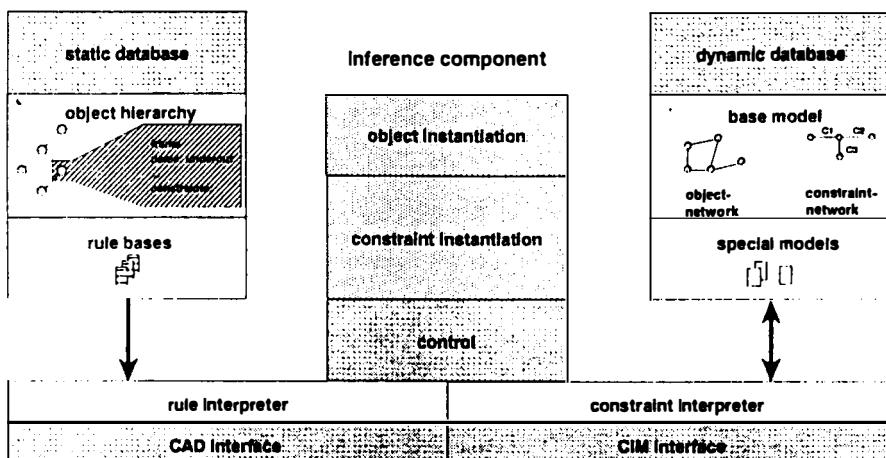


Figure 5. Architecture of the knowledge-based module

In order to give an impression on the cooperation of the different mechanisms described above, we will give a brief example of a concrete step in the process of

designing a shaft (Figure 6). Starting from the generation of a feature in the CAD-module the information is transferred to the AI-module to build up an external representation of the design object in the AI-module. Based on this representation different inferences can be executed thus leading to the generation of an additional feature in the CAD-module.

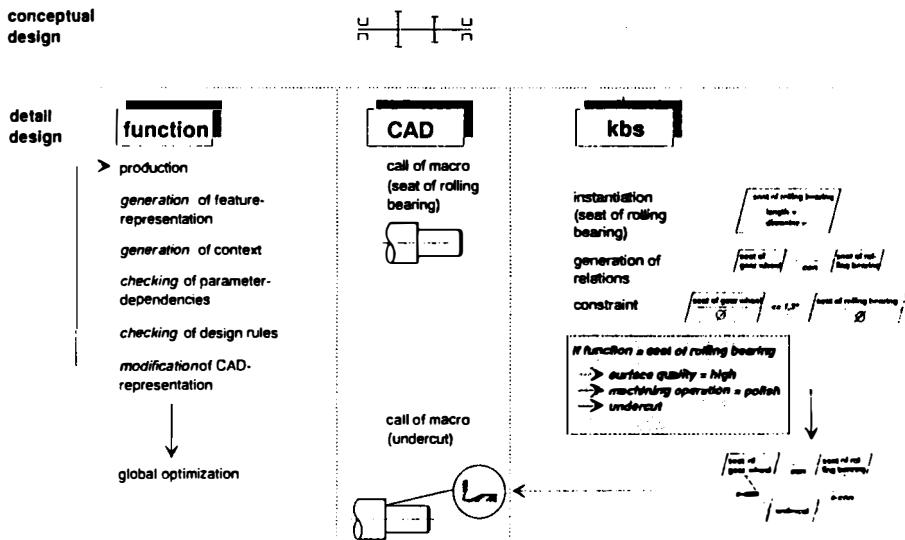


Figure 6. Example of a concrete step in designing a shaft

Based on the demand motivated above not to integrate data management, it is necessary to develop a mechanism for communication between CAD-system and AI-module, which guarantees the consistency of data in both system components.

A simplified diagram of the system architecture is given in Figure 7. To guarantee the decoupling of CAD-system and AI-module and by that the portability to other CAD-systems, the UNIX message concept IPC (Inter Process Communication) is used to connect the AI-module to the CAD-system. IPC includes special C-functions to send and receive messages. The messages will be transmitted as strings. Besides the functionality necessary for transmission, the IPC-procedures provide the mechanisms for synchronization as well.

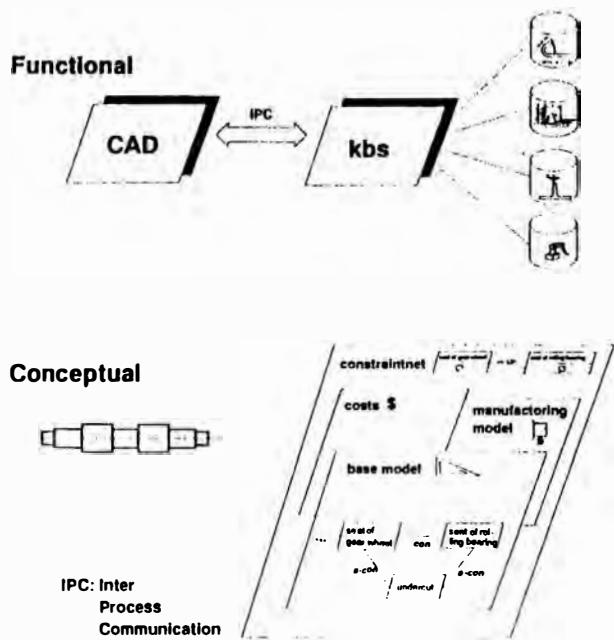


Figure 7. Architecture CAD-AI

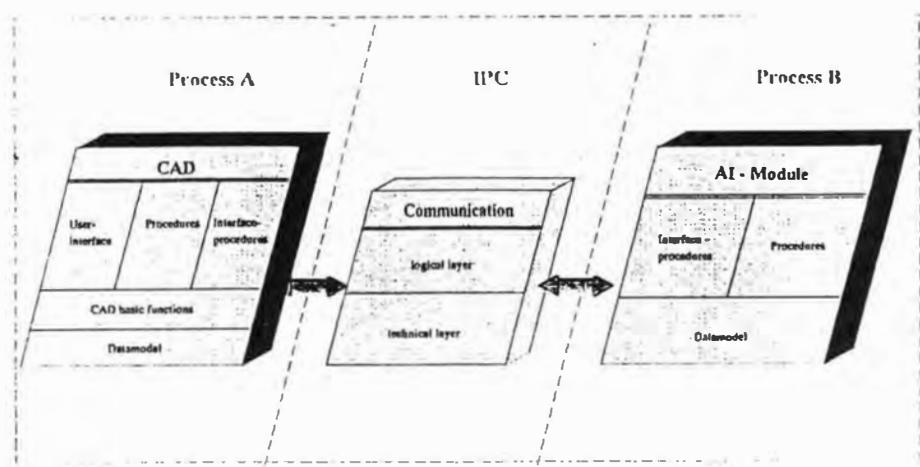


Figure 8. The different layers of communication

IPC only realizes the technical base for message transmission (technical layer). Based on this layer, the concrete syntax of the transmitted messages is to be specified within a so-called logical layer (Figure 8).

There are two different kinds of messages:

- commands and
- object data.

Commands have the following syntax:

```
<command name> "EOL" [<argument tripel> "EOL"]
```

The syntax of an argument tripel is equal to that of object data.

Example: setUserValue
 seat_of_rolling_bearing_1:diameter:20.0000

The syntax of object data is specified as follows:

```
<object name> ":"<attribute name> ":"<value>"EOL"
```

The colon separates the components within a tripel. End of line marks the end of the tripel. A message typically consists of several of these triplets.

Example: seat_of_rolling_bearing_1:diameter:20.0000
 seat_of_rolling_bearing_1:length:40.0000
 seat_of_gear_wheel_1:diameter:25.0000
 seat_of_gear_wheel_1:length:30.0000

The interface modules have to code and decode messages according to the specified syntax. The communication process for one command is shown in Figure 9.

The concept of integration described in this paragraph, guarantees consistency of data in CAD-system and AI-module on one hand, and on the other hand by using standard mechanisms (Inter Process Communication) it is possible to transfer this concept to other CAD-systems.

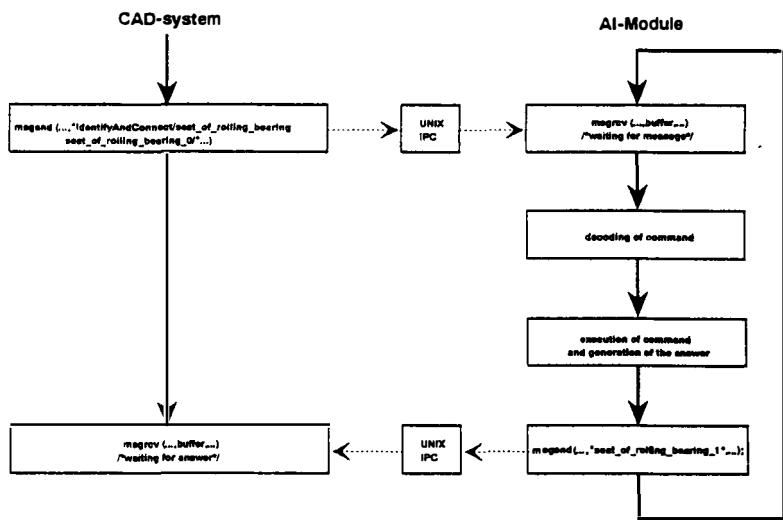


Figure 9. Example for a communication process

4. STATUS OF RESEARCH

For demo-purposes a first prototype has been realized using the C++ language and the CAD-system ME30 (HP). At this moment, a completely new prototype that is based on LISP is going to be realized.

5. CONCLUSION

In this paper an approach to knowledge-based assistance of design processes has been presented. The conception of the architecture of a prototype that is being realized within the project is discussed in detail. The description focusses the aspects of knowledge representation and inference.

Another important issue covered by this paper, is a concept for the integration of AI-module and CAD-system.

In contrast to systems using parametric design, dependencies are not only evaluated but also automatically instantiated dependent on the concrete design context based on the design knowledge that is represented together with the feature elements (Roller, 1990).

In the next phase of our research project, the functionality of the prototype will be improved by including additional feature elements and supporting additional specialized views on the design product.

6. REFERENCES

- Fikes, R. and Kehler, T. (1985). The role of frame-based representation in reasoning, *Communications of the ACM*, Vol. 28, No. 9.
- Finger, S. and Dixon, J.R. (1989). A Review of Research in Mechanical Engineering Design. Part I: Descriptive, Prescriptive, and Computer-Based Models of Design Processes, *Research in engineering design*, Vol. 1.
- Güsgen, H.W. (1987). *CONSAT - A System for Constraint Satisfaction*, Dissertation, GMD, Bonn.
- Neumann, B. (1988). Configuration Expert Systems: a Case Study and Tutorial, in Bunke, H.O. (ed), *Artificial Intelligence in Manufacturing, Assembly and Robotics*, Oldenbourg, München.
- Richter, M.M. (1989). *Prinzipien der künstlichen Intelligenz*, Teubner, Stuttgart.
- Roller, D. (1989). Constrained Features in Solid Modelling, in Radermacher, F.J. (ed), *FAW Workshop CAD and AI*, FAW Forschungsinstitut für anwendungsorientierte Wissensverarbeitung, Ulm.
- Roller, D. (1990). Advanced Methods for Parametric Design, in Hagen, H.; Roller, D. (eds), *Geometric Modelling: Methods and their Applications*, Springer, Berlin, Heidelberg.
- Spur, G., Germer, H.J., and Lehmann, C. (1989). Impact of Geometric Modeling for Computer Integrated Manufacturing, *Preprints of the International Symposium on Advanced Geometric Modelling for Engineering*, Berlin.
- Tong, C. (1987). Toward an engineering science of knowledge-based design, *Artificial Intelligence in Engineering*, Vol. 2, No. 3.
- Voß, A. and Voß, H. (1987). *Formalizing Local Constraint Propagation Methods*, Proceedings of the GMD, No. 248, Bonn.

A shorter version of this paper was presented on the ISATA 90 at Vienna, Austria.

The ICADS expert design advisor: concepts and directions

L. Myers, J. Pohl and A. Chapman

CAD Research Unit, Design Institute
California Polytechnic State University
San Luis Obispo CA 93407 USA

Abstract. The CAD Research Unit of the Design Institute at Cal Poly is engaged in a long term development of an Intelligent Computer Aided Design System (ICADS). Central to this work is the philosophy that the system should be a valuable assistant to the designer throughout the entire design activity. The current working ICADS prototype is a distributive system that can run with a variety of hardware processors in a Unix environment. It includes an Expert Design Advisor that has six knowledge based systems working as domain experts, a blackboard coordination expert, two knowledge bases and several sources of reference data. The Expert Design Advisor in the prototype interprets a drawing as it is being made by a designer working in a CAD environment. It reacts in real-time to monitor the evolving floorplan from the viewpoints of experts in the domains of Access, Climate, Cost, Lighting, Sound, and Structure. The system also provides for additional interaction with the designer outside the CAD system to help realize a resultant design that is free from conflicts in the six areas listed.

THE NOTION OF DESIGN PARADIGMS

The search for a model that adequately describes the design process for the purpose of devising a computer-aided design system has proven to be an elusive endeavor. The difficulty appears to lie in the translation of general patterns of design behavior into specific, repeatable sequences of designer actions that can be modelled on a computer.

The authors characterize design as an iteration process in which a great deal of information is analyzed and synthesized to form solutions. The iteration produces potential solutions which are evaluated on the basis of criteria established for the performance of the final designed artifact. This suggests that:

- (a) design proceeds in stages from information collection and broad concept development to detailed solution, within a cycle of continuous forward and backward tracking;

- (b) design is essentially a decision making endeavor, in which a significant portion of the progress toward each solution state can be made by the application of mathematically and/or heuristically based operations;
- (c) design is largely an information management activity and that much of the information is based on designer experience which can be predefined in the form of prototype knowledge bases;
- (d) designer expertise can be usefully supplemented by domain experts, although this advice may have to be significantly modified during the formulation of the design solution;
- (e) much of design is computable as a series of state representations comprising components that may be treated as objects with rich descriptions and clearly defined relationships.

This view of design as a human endeavor that can be modelled artificially has led to the fragmentation of what is essentially a holistic process into component tasks (Harfmann 1987). It has given credence to the concept of a design model of multiple subprocesses that incrementally transform a design problem statement into a solution. The subprocesses could be categorized as design methods for the generation, evaluation and selection of narrow solutions, to be integrated into broader solutions leading eventually to a final solution.

Much of the early work in this area has focused on the identification, application and adaptation of existing quantitative problem solving techniques as design methods for the systematic evaluation of narrow solutions and the combination of narrow solutions into broader solutions (Radford and Gero 1988). Such methods include: differential calculus (Au and Stelson 1969, Stark and Nicholls 1972, Page 1974, Bridges 1976); linear and non-linear programming (Mitchell et al. 1976, Dudnik 1977); dynamic programming (Gero 1976); multicriteria optimization (Nijkamp and Spronk 1981, Gero 1985a); and others. These systematic design methods were not intended to represent a complete model of the human design process, but rather to supplement the intuitive and creative contributions made by the designer (Cross 1984).

More recently, emphasis has been placed on the importance of design knowledge as the origin of the intuitive and irrational decisions made by the designer. The availability of knowledge to extend the design space in a computer-based design environment, coupled with effective tools for searching the design space for alternative solutions, has become the basis of a promising new direction in computer-aided design (Gero 1985b, Coyne et al. 1989).

Prominent among the first applications of knowledge-based systems in design are the use of expert systems as design analysis and synthesis tools (Dym 1985, Sriram and Adey 1986, Maher 1987, Rychener 1988). While much of the work in this area is currently focused on single domain stand-alone advisory systems, a small number of integrated CAD systems with coordinated multiple domain experts are emerging (Fenves et al. 1988, Maher 1988, Myers and Pohl 1989).

Expectations for knowledge-based design systems are particularly high in four areas:

- (a) systems that incorporate generalized design experience in the form of prototype knowledge bases (Gero et al. 1988, Pohl et al. 1989);

- (b) systems that incorporate expert design advisors involving multiple cooperating domain experts (Rychener et al. 1984, Hayes-Roth 1985, Myers and Pohl 1989);
- (c) systems that are capable of automatically generating new knowledge based on examples from the designer (Maher and Zhao 1987, McLaughlin and Gero 1988);
- (d) systems that integrate multiple resources (eg., prototype knowledge bases, domain experts, distributed databases, multi-media representation facilities and automatic knowledge acquisition capabilities) in a seamless and intelligent computer-based design environment.

Far from automating the design process, it is likely that the human designer will continue to play the primary role in knowledge-based design systems. As Laird (1986) has written, "AI systems have relied on subgoals created for specific difficulties in problem solving..[yet] no system has a scheme for creating subgoals for all types of difficulties, and for creating them automatically". The contributions of the designer in the areas of intuition and reflection-in-action (Schoen 1983) are necessary prerequisites for the evolution of a higher level of intelligence in the computer-based design environment.

Experience As Design Knowledge

Design decisions typically involve the evaluation of a large volume of information using a variety of algorithmic and heuristic design tools. In architectural design, even relatively small buildings may require consideration of more than a dozen major design issues (eg., user needs, climatic conditions, owner expectations, structural support, lighting, acoustics, economic feasibility, industry constraints and operational efficiency.) These design issues influence each other and must therefore be dealt with concurrently rather than sequentially. The concurrent requirement is an important element in design applications that places a severe burden on the designer.

Research involving case studies of design projects carried out in architectural offices found 'experience' to be the most important influence on design decisions (Mackinder and Marvin, 1982). Experience of the decision making process; knowledge and experience of materials, components, construction details, structural and mechanical systems, and occupancy needs; and, knowledge of typical solution strategies due to past experience with similar projects; were the most frequently quoted types of experience.

The reliance of the designer on experience gained in past similar projects, knowledge acquired in the general course of design practice, and expertise contributed by specialist consultants have been identified in the literature as major contributors to the design process (Mallen and Goumain 1973, Akin, 1978, Mackinder and Marvin 1982). This knowledge is the kind of information that is presented in explaining decisions and must include: the programmatic objectives and performance specifications of the design problem; information about prototype solutions of similar design problems; comprehensive site and neighborhood data that define the physical, environmental and sociological context of the design project; and, sundry reference catalogs (eg., manufacturers' data, codes, etc.).

The dominant emphasis on experience is confirmation of another fundamental aspect of the design function. Design seldom starts from first principles. In most cases, the designer builds on existing solutions from previous projects that are in some way related to the

problem under consideration. From this viewpoint, the process of design involves the modification, refinement, enhancement and combination of existing solutions into a new hybrid solution that satisfies the requirements of the given problem system (Gero et al. 1988). In other words, design can be described as a process in which relevant elements of past prototype solution models are progressively and collectively molded into a new solution model.

One way of capturing design experience is through the use of design prototypes. In architectural design the classification of 'building type' (eg., high school, courthouse, library, etc.) provides an appropriate encapsulation of one kind of prototype. In its fullest meaning a building type knowledge base should include not only the strategies that have been applied in the past to generate design solutions, but also the objectives, criteria and constraints that defined those design problems (Pohl et al. 1988).

Another type of prototype is represented by the more or less standardized solutions that have been developed over years of design practice to solve specific problems that are related to functional conditions or physical constraints, rather than building type. Standardized solutions exist for a broad horizontal spectrum of building design problems ranging from the technical considerations of curved sloping vehicular driveways to the aesthetic treatment of the intersection between circular and rectangular building wings. Such standard types are available in design guides.

The extent to which prototypic information can be used is exemplified by the ability in some cases to generate a grammar that defines the designs that can conform to a particular style. For example, such a grammar was developed for the villas of Andrea Palladio (March and Stiny, 1985). In fact it is possible to automatically generate examples within a given context (Flemming et al. 1986). This capability might eventually become sufficiently general to be useful in providing suggestions to the designer, when such assistance is requested.

In the CAD environment the computer system should aim to provide the designer with built-in experience that encompasses all significant design issues at a level of knowledge that would be expected of a competent consultant. Two types of support should be provided:

- (a) intermittent foreground responsiveness to requests for information and assistance initiated directly by the designer; and,
- (b) continuous background monitoring and evaluation of the evolving design solution.

AN EXPERT DESIGN ADVISOR

It is useful to think of the support that should be provided by the CAD system as an expert consultant serving as a spokesman and coordinator of a team of experienced design experts. In the computer-aided design environment, a team of human experts can be simulated by a set of expert systems in which each system has a special domain of expertise. A coordination facility can integrate these knowledge based systems such that the computer system as a whole will participate as an assistant during the development of a design solution. In functional terms, such an assistance facility could be referred to as an expert design advisor.

The basic expert design advisor model therefore consists of a set of domain experts, or Intelligent Design Tools (referred to as IDTs in Figure 1), whose operation in the automatic

and user-directed modes is controlled by a coordination expert that guides the evolving design solution within the holistic context of the design problem space.

Coordination of Domain Experts

In particular the coordination expert must be capable of resolving conflicts among the domain experts. A very common conflict condition will occur whenever two or more domain experts make different proposals for the same solution element. For example, at the same point in time the following advice may be given from some of the domain experts:

- (a) 'Thermal' domain expert suggests heavy-weight construction (ie., masonry or concrete) with one strategically placed opening to facilitate external and internal heat storage, for the external wall of the west side of the building. This decision may be based on a large diurnal temperature range.
- (b) 'Structure' domain expert suggests a light-weight timber structure, based on relatively short spans, programmatic cost criteria, and local construction practices.
- (c) 'Sound' domain expert suggests heavy-weight construction and no external openings for the west wall of the building, due to the existence of a saw mill west of the site.
- (d) 'Lighting' domain expert suggests clerestory windows with external vertical sunshading devices for a portion of the west wall of the building, based on daylighting and energy conservation criteria included in the design program.

The coordination unit, based on its own multi-domain knowledge of the design space, must invoke a procedure for resolving this conflict. The procedure should be consultative rather than dictatorial. In other words, while the coordination expert may suggest a compromise solution, all domain experts should be given an opportunity to consider the impact of this solution on their domains and suggest a new solution which may be different from the proposed compromise. The need for this kind of iterating consultation approach is based on observation of the characteristics of designer-consultant interactions in practice. The "reflective conversation with the situation" view of design (Schoen 1983) encompasses all resources that can be brought to bear on the problem by the designer, including the advice of consultants.

The mission of the coordination expert must be accomplished without encroaching on the role of the designer, as the orchestrator of the design process and the final adjudicator of any conflicts that might significantly impact the direction of the design solution. The assessment of whether or not a compromise decision will significantly change the current direction of the evolving design solution is an important observation for the coordination expert to make. At the most primitive level this capability can be based on a set of elementary indicators, such as: the occurrence of a deadlock between the coordination expert and one or more domain experts; repetition of a previous pattern of interactions among the domain experts and the coordination expert; or, a required change to a decision previously made consciously (eg., by direct intervention) by the designer. At a more advanced level, the coordination expert might be able to progressively develop its own knowledge of the

current state of the design solution by monitoring changes in the design space and comparing these to descriptions of prototypes contained in knowledge bases.

Having made the assessment that a deadlock exists or a change in direction of the solution strategy may be appropriate, the coordination expert must inform the designer of contentious issues and suggest alternatives for resolving the problem. The interaction between the designer and the coordination expert is essential to the partnership concept proposed in this model of an expert design advisor. It ensures the designer of a supervisory role in the automatic multi-domain evaluation of the evolving design solution.

THE ICADS EXPERT DESIGN ADVISOR

The first version of the ICADS (Intelligent Computer-Aided Design System) working model includes two knowledge bases, several sources of reference data, a CAD drawing system, a multi-function user interface, and an Expert Design Advisor that contains six domain experts and a blackboard coordination expert (Pohl et al. 1989).

The scope of the implementation environment has been restricted in terms of both the breadth of information available to the designer and the range of design functions supported. A subset of the domain experts necessary for a complete design environment has been developed and a specific design project has been selected for testing purposes. The test project is the design of a community center building for a small city on the central coast of California, Pismo Beach. The information resources provided by the working model are drawn from a more general architectural design application area and include Building Type and Site/Neighborhood knowledge bases, as well as a Reference database containing material and constructional information. The database information is sufficiently general so that the specific design project used to demonstrate the working system accesses about fifteen per cent of the available information.

The Expert Design Advisor is responsible for the evaluation of the evolving design solution and the resolution of conflicts that may arise when solutions in one domain interfere with solutions in another domain. It consists of advisory components, a coordination expert and operational components, as shown below:

operational components:	Attribute Loader Semantic Network of Frames
advisory components:	Geometry Interpreter Access domain expert Climate domain expert Cost domain expert Lighting domain expert Sound domain expert Structure domain expert
coordination expert:	Message Router Conflict Resolver

A schematic diagram of the ICADS working model is shown in Figure 1. The model is similar to that now being used in a variety of applications, including the coordination of

multi-sensor robots (Durrant-Whyte 1988). The primary difference is that the ICADS model does not use the Blackboard for planning and scheduling activities, which are primary to most other blackboard systems. Instead, the agents are allowed to execute in parallel. There is no external scheduling of the expert components. They execute immediately on distinct processors as values appropriate to them appear on the Blackboard.

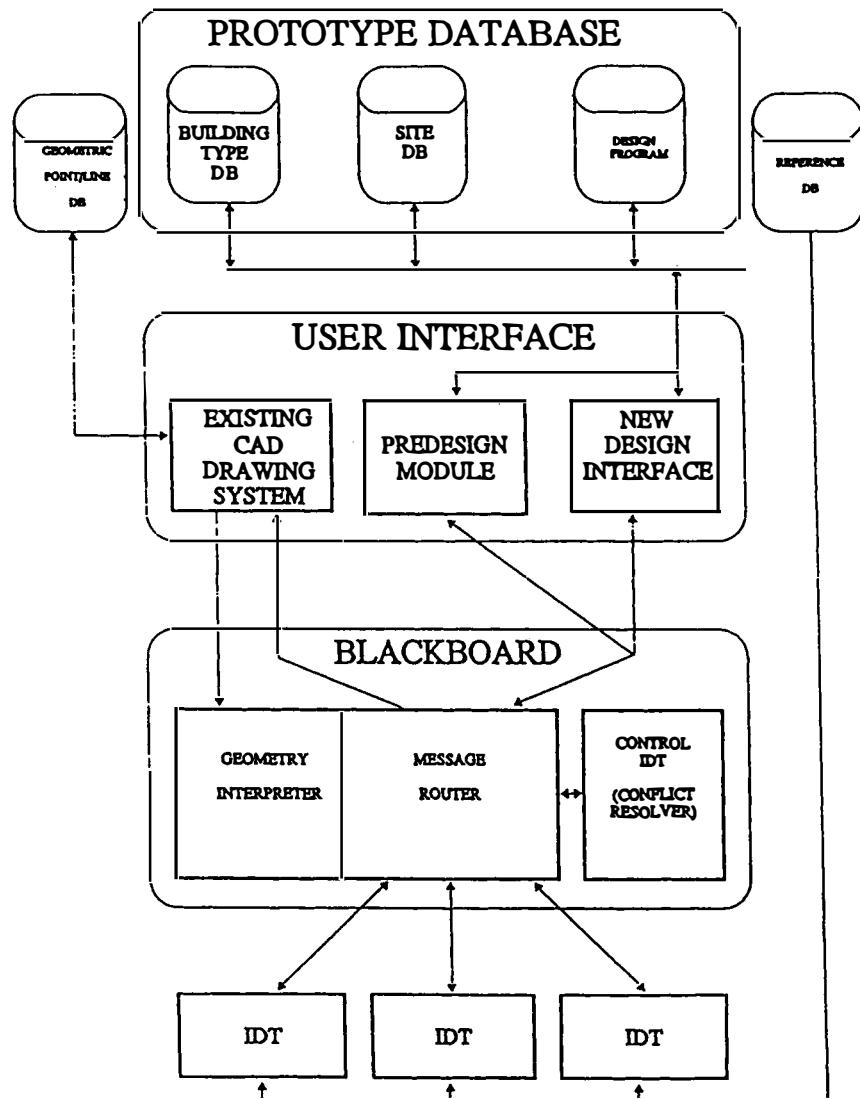


Figure 1. ICADS working model version 1

The Design Knowledge Bases

When a new design project is begun, a file of project specifications is established. This file is used by the Attribute Loader to initialize the full set of variables that are used by the system. For example, the file will identify the building type for the project. Prototypic information for the type of building will then be retrieved from the databases by the Attribute Loader to establish constraints and default values for the project.

The structure of the Building Type and Site/Neighborhood knowledge bases in the ICADS model have been reported previously (Pohl et al. 1988). These knowledge resources are intended to capture the experience and standard solution strategies associated with a given building type, and the specific conditions of the site and its surrounding environment, respectively. Collectively, they provide views of the design project from several vantage points represented by different interest groups (eg., owner, users, designer, community and government authorities).

The Building Type knowledge base is included in the Expert Design Advisor to provide prototype information relating to the type of building under consideration. In this context a prototype is defined as a body of knowledge relevant to the definition and solution of a related set of design problems. The prototype includes generalizations derived from specific instances, elements of previously tested solutions, descriptions of typical solution components, and solution boundary constraints. The boundaries within which the prototype is applicable is provided by the Site/Neighborhood knowledge base, in terms of the requirements and characteristics of the owner, and the physical, environmental, social and economic context of the project location.

The Frame Representation

All architectural information that is used for inference in the ICADS model, including the project specification information, is put into a specific frame representation (Assal and Myers, 1990). The inferences are made by expert systems written in CLIPS (NASA 1990), an expert system shell, so the frame representation is designed particularly for the form of CLIPS facts. The frame implementation consists of representation, generation, and manipulation features whose scope is beyond that which can be described here. However, the most important ideas can be easily seen.

Each frame will either hold a class or an instance of a class. If the frame holds a class, it will describe the basic characteristics of the class including: default values; demons that dynamically obtain values or perform tasks for instances of the class; the names of the slots in the class; and relations between the class and other classes.

Table 1 shows a simple declaration of a frame for a 'space' class. The frame is merely a collection of CLIPS facts that satisfy certain rules of form. The CLIPS facts themselves are parenthesized lists of fields, with each field being a word, a number, or a string. Note that the default value for the ceiling-height slot of any instance of a space is 8 feet. Other slots in the space frame are denoted as 'name', 'key', 'floor-height', and 'area'. Further, there are relations to 'wall' and 'symbol' classes, which are interpreted to mean that a space can have walls and symbols. Finally, there is a 'demon' that indicates that the perimeter of a space should be calculated whenever it does not already exist.

(FRAME	space)
(DEFAULT	space ceiling-height 8)
(VALUE	space name)
(VALUE	space key)
(VALUE	space floor-height)
(VALUE	space area)
(RELATION	space wall)
(RELATION	space symbol)
(DEMON	space perimeter if-needed)

Table 1. A simple 'Space' class

The class frame is used particularly to provide default values and inherited values for class instances, which represent the architectural objects in a design project. The architectural objects themselves are generated as instances of the classes. For example, the first drawing action of the designer might be to sketch a rectangle which is then labeled 'office'. As the lines that make the four walls for this space are drawn, the system generates facts that describe what is represented by the drawing, in terms of the frames that have been defined for architectural objects. When the connection of the four walls is completed, the system will determine that a 'space' has been created and the instance for a 'space' frame would have been generated as seen in Table 2.

(FRAME	space room1)
(VALUE	space name room1 office)
(VALUE	space key room1 12)
(VALUE	space floor-height room1 0)
(VALUE	space ceiling-height room1 8)
(VALUE	space area room1 160)
(RELATION	space wall room1 wall1)
(RELATION	space wall room1 wall2)
(RELATION	space wall room1 wall3)
(RELATION	space wall room1 wall4)
(RELATION	space symbol room1 chair)

Table 2. An instance of the 'Space' class

The Semantic Network

The frames used to represent architectural objects are referred to as design object frames and the collection of all such frames used by the Expert Design Advisor is called the Semantic Network. Thus, it is the values of the design object frames that represent the current state of the design solution within the context of the project.

As the designer continues to draw, additional frames are automatically and transparently generated by the system to describe the architectural objects being represented and the relations between them. The frame information is generated in a blackboard environment to which the domain experts and the coordination expert are connected. As a result, the rules in any domain expert can fire as soon as their requisite facts are posted to the blackboard. This in turn might result in the posting of advice, in the form of new frame information, that could instantiate rules within the coordination expert.

The architectural objects currently represented by the system include the following seven geometric design objects:

FLOOR, SPACE, WALL, DOOR, WINDOW, SEGMENT, and SYMBOL.

The SEGMENT object refers to any part of a WALL object that is demarcated either by the intersection of another wall or has been drawn by the designer as a distinct wall component. The SYMBOL object represents directly by name any closed shape or icon within a SPACE object (eg., column, chair, table).

Geometric facts describing the geometry of the current design solution are extracted by the Geometry Interpreter in terms of the nature, physical dimensions, and relative locations of the objects.

Attribute facts describing the context of the project and the non-geometric characteristics of the current design solution are derived from the Building Type and Site/Neighborhood knowledge bases, directly or indirectly through the extrapolation of several information items. Non-geometric attribute values are included in the blackboard in association with the following eight design objects:

PROJECT, NEIGHBORHOOD, SITE, BUILDING, FLOOR, SPACE, WALL, and OPENING.

The differences between the geometric and non-geometric design object sets are entirely consistent with the nature of the information they contribute. The design knowledge bases that support the design process in the ICADS model encompass a much wider view of the design space than can be represented by any instance of the geometric design solution. For example, although regional and neighborhood parameters are an important part of the design decision-making process they are no longer discernable as discrete information items in the drawing of the geometric model. At that level they are embedded under several layers of synthesis and are therefore an implicit rather than explicit part of the geometry of the artifact.

In the ICADS working model the distinction between design context and design solution has led to the separation of the Semantic Network of design objects into two logical sections.

PROJECT DESIGN OBJECT FRAMES: comprising one frame for each design object represented in the design program (ie., design specifications), which is a subset of the Building Type and Site/Neighborhood knowledge bases. Slots in these frames are used to store non-geometric attributes that have either direct equivalents in the Building Type and Site/Neighborhood knowledge bases, or are inferred from several values by the Attribute Loader.

SOLUTION DESIGN OBJECT FRAMES: comprising one frame for each (geometric) design object analyzed by the Geometry Interpreter. Slots in these frames represent the geometric descriptions of the particular design object identified by the Geometry Interpreter and the solution evaluation results generated by the domain experts under the coordinating role of the Blackboard.

In the current ICADS working model the slot values of the 'project design object' frames cannot be changed by the designer during the design process. They are established by the Attribute Loader at the beginning of a design session and remain as static members of the input templates of individual domain experts, and to a lesser extent the Conflict Resolver, throughout the design session. (In the full ICADS model the designer should be able to change information in the design knowledge bases, which would require the Attribute Loader to make the appropriate adjustments in the 'project design object' frames.)

Also, there is a duplicity of slots in the solution design object frames. It is necessary to distinguish the value for a solution design object slot that is currently accepted as part of the design solution from a value for the slot that is suggested by an Intelligent Design Tool (IDT). In order to do this, a standard naming convention is used to create a slot for the value suggested by each IDT which makes a suggestion. The convention uses the basic slot name and the name of the IDT so that both humans and machines can identify the association. The basic slot value is referred to as the 'current value' of the solution design slot, and the slot values which are suggested by the IDTs are referred to as 'suggested values'.

The Domain Experts

The current state of the design solution, represented as object descriptions (containing both geometric definitions and non-geometric attributes), drives six domain experts with evaluation capabilities in the areas of space access determination, construction cost projections, daylighting, sound control, structural system selection, and thermal behavior.

With the completion of each drawing action, the Geometry Interpreter will create all facts that are necessary to describe the results and transmit them to the Blackboard. The Blackboard immediately transmits the useful facts to the domain experts. There is no need for the output of the Geometry Interpreter to be examined; whatever is drawn by the user must be accepted by the system at that point in time. Each domain expert, executing continuously in background under a separate process, reacts to the information pertaining to each particular design object referenced and commences to evaluate the design solution based on its expertise (Figure 1).

For example, the Lighting expert will evaluate the degree to which each space in the current design solution satisfies the requirement for daylight. The evaluation consists of two components. First, the requirements are established. This may be a trivial task, requiring only the generation of a simple query to a Reference database to obtain recommended task and background illumination levels for the type of space under consideration. Or, it may be a much more complicated undertaking involving the analysis of qualitative and quantitative design criteria such as:

OWNER considers energy efficiency to be very important;
USER GROUP (A) considers energy efficiency to be optional;
USER GROUP (B) considers energy efficiency to be desirable;
DESIGNER considers energy efficiency to be important;
EXPERIENCE for each SPACE recommends:
x% of background illumination by daylight;
y% of task illumination by daylight.

Second, the Lighting expert will estimate the daylight illumination on the workplane at the center of each space in two parts. The Daylight Factor is estimated based on the geometry of the space, the geometry of windows in external walls and the reflectances of the internal wall, ceiling and floor surfaces. This Daylight Factor value is converted into an equivalent illumination level subject to an external daylight availability calculation for a particular month, day and time.

The results obtained by each domain expert are added to the appropriate design object frames, as suggested values. The Conflict Resolver, resident in the Blackboard, examines the values posted by the domain experts and arbitrates conflicts. For example, the Sound expert may have generated the requirement that the north wall of the conference room should have no windows. This is in conflict with the current design solution (based on the Geometry Interpreter) and the Lighting expert, who has determined that the % of background illumination by daylight' for this room is already 15% below the 'requirement'. Based on its own rules, the Conflict Resolver determines that the windows in the north wall should be reduced by 20% and double glazed to minimize noise transmission. Apparently, the reduction in the availability of daylight is warranted in view of the noise transmission problem. The Blackboard posts these new values to the appropriate frames and thereby initiates new evaluations by those domain experts whose previous results are now in conflict with the Blackboard's determination. If the Blackboard had decided that the windows must be deleted from the north wall of the conference room, then it would have requested permission for this radical action from the designer.

The interaction between the designer and the Blackboard is limited to extreme circumstances in the current ICADS working model. Such circumstances may arise:

1. If the decision of the Blackboard requires a modification of the drawing. In the above example, during the conceptual design stage a 20% reduction in the window area of the north wall can be accommodated without modification of the 2-D representation of the space. However, the deletion of all windows from the wall would require the drawing to be changed.
2. If the Blackboard cannot resolve a conflict set. Again, in the conference room example, it is conceivable that certain design specifications could mandate daylighting and sound control requirements that will not allow a compromise to be made. Under these conditions, the Blackboard will interrupt the designer and request guidance. In the full ICADS model the Blackboard would be expected to present the designer with a clearly argued set of alternative actions. Alternatives may include: changing one or more of the mandated design specifications; temporarily disabling one of the domain experts involved in the deadlock; or, revising the design solution to by-pass the deadlock condition.

At any time during this evaluation process the designer can request to review the current conflict state of the Expert Design Advisor (ie., the interactions of the Conflict Resolver with the six domain experts). This is accomplished through the Design Interface on a second monitor.

The Conflict Resolver

The coordination expert, or blackboard control expert, is primarily implemented as the Conflict Resolver in the ICADS working model. While it is envisaged that 'planning' will play an important role in future implementations of the ICADS model, at this time the resolution of conflicts appears to be sufficient to coordinate the activities of the advisory components in the Expert Design Advisor. The diminished role of planning in this blackboard system is largely due to the fact that the processes are running in parallel on multiple processes. This makes it less necessary to determine how the computational resources should be used.

The principal purpose of the Conflict Resolver is to assert 'current value' frame slots, representing the current state of the evaluation process performed by the domain experts, onto the Semantic Network resident in the Blackboard. To accomplish this, the Conflict Resolver receives from the Message Router all of the 'solution design object' frames which contain results generated by the domain experts. Current values fall into one of three basic categories: values which result from solutions proposed by a single domain expert; values which result from solutions proposed by several domain experts for a common current value; and, values which must be inferred from solutions proposed by several domain experts.

Structural expert:	'suggested roof material type'	= A
Thermal expert:	'suggested roof material type'	= B

Rule 1: if A == B then
current value 'suggested roof material type' = A

Rule 2: if A == timber and B == concrete then
current value 'suggested roof material type' = concrete

Rule 3: if A == concrete and B == timber then
and current value 'suggested roof material type' = concrete
and current value 'suggested roof struct.system' = concrete plate
and current value 'required roof struct. depth' = 4
and current value 'roof insulation thickness' = 3

Table 3. An example of Conflict Resolver rules

In the case of the first category, which represents solution values unique to a single domain expert, the Conflict Resolver does not change the values proposed by the expert. The proposed solution values are simply asserted as current values into the appropriate frame

slots. In the second category two or more domain experts propose differing values for the same solution parameter. In such direct conflict situations it is the responsibility of the Conflict Resolver to either determine which of the values is most correct or to derive a compromise value. The process of resolution may cause the Conflict Resolver to change several current values in addition to those in direct conflict, as shown in Table 3.

The Conflict Resolver incorporates resolution rule sets which determine the best current values from those proposed. There is a resolution rule set for each possible direct conflict. In the development of each rule an attempt has been made to achieve a desirable balance between the various design issues. At this level the Conflict Resolver can be considered an expert with knowledge of how this balanced integration can be achieved.

In the above example, let us assume that the Structural expert proposes 'timber' as the suggested roof material type, A, and the Thermal expert proposes 'concrete' as B. Under these circumstances the Conflict Resolver recognizes that:

- solutions for the roof structure proposed by the Structural and Thermal domain experts are substantially different;
- 'concrete' solution proposed by the Thermal expert suggests a need for thermal storage;
- 'timber' solution proposed by the Structural expert cannot be readily modified to provide thermal storage;
- in most cases a structural timber system can be replaced by a concrete system (there are exceptions to this rule of thumb (eg., seismic risk) and the Structural expert should be able to recognize such circumstances and, if necessary, refuse to agree with the Conflict Resolver's proposed compromise solution);
- energy conservation savings provided by a well designed passive thermal building are likely to exceed the higher capital investment costs associated with a concrete roof system.

In the third category the Conflict Resolver deals with proposed solution values that are indirectly in conflict with other proposed solution values and current values. The resolution rules for this category allow the Conflict Resolver to make the necessary modifications to any of the values involved. Under these conditions, in addition to changing proposed solution values, the Conflict Resolver may also change current values as shown in Table 4.

current value 'roof construction system'	= A
current value 'roof U-value (BTU/HR-SF-F)'	= B
current value 'roof insulation thickness'	= C
current value 'roof thermal lag (HR)'	= D

Thermal expert: 'suggested roof construction system' = E

Rule 1: if A \leftrightarrow E then

look up Reference database and change B, C, D to appropriate values for the suggested roof construction system, 'E'

Table 4. Conflict Resolver changes current values

In the example of Table 4, the Thermal expert has suggested a new roof construction system. The Conflict Resolver recognizes that several current values must be changed so that they match the new roof construction system. Similar to the direct conflicts discussed under the second category, each indirect conflict must also have a set of resolution rules.

Not all conflicts can be resolved by the system. In some cases, usually those requiring changes to the drawing or 'project design object' frames, the Conflict Resolver will ask for assistance from the designer. Under these circumstances, in the ICADS working model, operation of the Expert Design Advisor is suspended until the designer responds.

Some Implementation Notes

There are five areas of particular significance in implementing the system:

- database interface
- automatic configuration scheme
- Geometry Interpreter
- frame representation
- Blackboard interface

Developing the database interface for the working prototype is a major effort in itself. So much can be written on this important area that only one further remark will be made. The databases define most of the vocabulary of design that is resident in the system. Their organization is based both on concerns for efficient database management and natural understanding by the designer.

With respect to the other four areas listed, some brief notes can be made. A scheme to automatically configure the system is used to permit easy modification of the execution environment. The system normally executes on a network of five CPUs with three VDT display interfaces. However, the allocation of processes to CPUs and display interfaces to VDTs is not constrained by any particular number or location of units. Standard X-Window procedures for displays and UNIX sockets for communication make it possible to provide for custom configurations quite conveniently. After starting the X-Window server, the rest of the system is self starting. A configuration file is used to identify the processors and display units that each process will use. A key is used to identify each user, so it is possible to execute more than one ICADS system at the same time with different configurations of processors and display units. This also makes it easy to tune the system for performance and makes it unnecessary for new users to learn how to initiate processes on remote machines.

The Geometry Interpreter must determine such things as what constitutes a space, or room. It must do this by an examination of the adjacencies of walls drawn on a display unit. This may not seem difficult until one considers how to determine the spaces that remain when a wall is moved from an area, within one space, to a place where it divides another space. The CAD system is only aware of the movement of the lines that represent the wall. The correct representation of the new spaces requires a number of deletions and additions of previous information to describe the walls that form the new rooms and the new properties of those rooms. Moreover, it must be possible to guarantee that the interpretation of all drawing actions has been made correctly. This verification is provided by a scheme in

which a linked structure of the information, from the CAD database associated with the drawing before the action, is compared with a similar structure for the information associated with the resultant drawing (Taylor and Pohl 1990).

The frame representation developed for use in the CLIPS expert system shell has been sufficient to express all domain and control knowledge needed for the prototype and to execute quickly. All Blackboard information is kept in this common form.

The Blackboard interface was originally developed as functions, that were called only from CLIPS expert systems. They made it easy for the CLIPS programmer to generate a Unix socket between the Blackboard and an IDT. Then, those functions were duplicated in a C programming language environment. As a result, any process that could send and receive arguments to a C function could be added to the system and provided with communication links to the Blackboard. This made it possible to write agents in practically any software system (Taylor and Myers 1990). In particular it was possible to duplicate the CLIPS frame representation as a set of C++ classes, which were particularly useful in user interface facilities that link to the Blackboard as IDTs.

A few changes in the CLIPS source code were made to implement the distributed Blackboard. The primary addition was a function called 'bb_assert', which is used to broadcast facts from the Blackboard to all IDTs. To illustrate use of the actual code, the textual form of the first rule in Table 3 is shown in its CLIPS form in Table 5.

```
( defrule floor-suggested-roof-material-type-rule 1
  ( RELATION floor  floor-ceiling/roof-structural      ?floor  ?f-rel-struct )
  ( RELATION floor  floor-ceiling/roof-thermal        ?floor  ?f-rel-therm )
  ( RELATION floor  floor-ceiling/roof-BB            ?floor  ?f-rel-BB )
?A <- ( VALUE   floor-ceiling/roof-structural
          suggested-roof-material-type    ?f-rel-struct    ?material )
?B <- ( VALUE   floor-ceiling/roof-thermal
          suggested-roof-material-type    ?f-rel-therm    ?material )
?C <- ( VALUE   floor-ceiling/roof-BB
          suggested-roof-material-type    ?f-rel-BB      ~  ?material )
=>
  ( retract   ?A ?B ?C)
  (bb_assert (MODIFY VALUE   floor-ceiling/roof-BB
                      suggested-roof-material-type    ?f-rel-BB    ?material )
  )
  (bb_end_message)
  (assert (VALUE   floor-ceiling/roof-BB
                      suggested-roof-material-type    ?f-rel-BB    ?material )
  )
)
)
```

Table 5. CLIPS form of a Conflict Resolver rule

A brief explanation of the rule in Table 5 follows:

1. Identifiers that begin with a '?' are pattern matching variables. So, the first three patterns in the rule, all of which begin with 'RELATION', will be matched if some instance of a 'floor' frame, denoted by the value bound to ?floor, is the same frame related to 'floor-ceiling/roof-structural', 'floor-ceiling/roof-thermal', and 'floor-ceiling/roof-BB' frames.
2. The next three forms remember in ?A, ?B, and ?C the location of the facts that match the patterns beginning with 'VALUE'. These patterns are matched only when the values for the 'suggested-roof-material-type' slots suggested by the structural and thermal IDTs are the same, and when those values are different from the current value for the slot on the Blackboard.
3. In other words, the left hand side of the rule, the part preceding the ' $=>$ ' symbol, is matched when the structural and thermal IDTs suggest the same roof material type, which is different from the current value on the Blackboard. This causes the rule to 'fire'.
4. The facts that record the suggested values and the current Blackboard value are removed from the fact list, or memory, of the Blackboard by the 'retract' operation.
5. The 'bb_assert' function and the 'bb_end_message' fact are used to send the information that establishes the new current value for the 'suggested-roof-material-type' slot to all IDTs.
6. The Conflict Resolver remembers this new value itself through the 'assert' action.

ISSUES OF CONCERN

The model of the ICADS Expert Design Advisor described above presents several issues of concern. Some of these are of an operational nature and others are related to fundamental concepts embodied in the model. The issues addressed here are:

- (a) how to reduce the time required for knowledge acquisition;
- (b) how to settle conflicts between domain experts;
- (c) how to know when the time is right for making decisions;
- (d) how to focus attention to achieve convergence of original problems;
- (e) how to enable the system to work with unforeseen situations.

The first concern is basic to all knowledge based systems. Some efficiency has been achieved at the domain expert development level through the use of an off-line program, DBRESOLVE, that resolves frame information. This program can be used on a file of CLIPS rules to mark any facts that use the keywords that identify parts of frames. It checks all such references against the database that is used to establish the design object information. As a result, improper references are caught early in the expert system development cycle. However, there is a much more serious problem in developing the rules for the Conflict Resolver.

Whenever an IDT is expanded or a new IDT is added to the system, rules must be added to the Conflict Resolver to handle new advice. Unlike IDT development, these rules require the developer to know about the new advice from a total system point of view. In other

words, the modification to the Conflict Resolver must reflect an understanding of how this new advice relates to that provided from all other IDTs. The developer must also consider whether communication with the designer is required and provide for the appropriate interface as necessary. Generally, the developer of rules for the Conflict Resolver must have a significant amount of knowledge about the ICADS system itself, as well as a considerable degree of experience in architectural design.

The ICADS team is developing an intelligent interface to help with knowledge acquisition. Initially this program will incorporate the off-line frame resolution function of DBRESOLVE and perform other syntactic checking in an interactive environment. Later it will provide advice for developers by comparing the patterns found in new rules with those already at use in the Conflict Resolver.

The second concern is to prevent the Conflict Resolver from entering into an 'endless argument' state that may arise when two domain experts always return with conflicting values for the same solution parameter. For instance, in reference to the previous example, the Structural expert may insist for good reasons that the roof construction system should be 'timber'. For reasons that are different but just as persuasive, the Thermal expert may be unwilling to deviate from its original proposal of a 'concrete' roof system. The two domain experts are now locked into an ad absurdum argument. In the current ICADS working model the Conflict Resolver monitors this type of situation by checking for repetitive cycles in current values. Future ICADS models will present such disagreements to the user and provide convenient access to helpful presentations of the problem. It should even be possible for the user to experiment with potential ways of resolving the problem, by having the system predict consequences of specific actions, and return to the original problem state to resolve the conflict.

Another concern is related to the timeliness of the conflict resolution process. In the implemented model of the Expert Design Advisor, the Conflict Resolver will not post a current value to the Blackboard until it has seen all of the 'solution design object' frames which are involved in a given conflict. In a full ICADS implementation it may be desirable for the Conflict Resolver to post a current value based on partial information (ie., based on only some of the required 'solution design object' frames). An improvement to the current model could be made to permit evidence to accumulate that would identify the appropriateness of making decisions based on partial input. This requires enhancement of the coordination expert to give that system more knowledge of the progress being made in the design process and greater awareness of the significance of design factors in affecting that progress.

It is possible that a problem with convergence to a design solution will arise as the prototype is extended into dozens of expert systems. The potential for a 'cascading' condition may exist when a minor change by a domain expert causes a major re-evaluation of the current solution by several domain experts. In the ICADS implementation of the Expert Design Advisor, the possibility of such a condition occurring is exacerbated because domain experts will fire on any change to a current value, and the Conflict Resolver will respond to any proposed changes posted by the domain experts. An attempt has been made to foresee events that could conceivably lead to this undesirable condition. Where appropriate, tests have been included in the rules of domain experts to determine whether the current divergence between the most recent suggestion and the corresponding current value

(proposed by the Conflict Resolver) is sufficiently large to warrant further action by the domain expert. Again, as the model is extended more attention will need to be given as to what constitutes a difference in values suggested for a design factor. It may even be necessary to have the constraints on acceptable values change with respect to time or phase.

An overriding concern of the Expert Design Advisor model described in this paper is that it is based on the assumption that the conflict resolution set required for the coordination of a representative number of domain experts can be largely predefined. This assumption is obviously not valid. The very nature of design as a process that involves creativity and intuition, operating in a design space that expands dynamically, defies the establishment of the type of detailed symbolic description that is required in support of a comprehensive design reasoning system. The reasoning capabilities of the current ICADS working model, while perhaps satisfactory within the limited scope of the Expert Design Advisor, cannot be extrapolated to a real world model involving two or three dozen domain experts and a comparable number of knowledge bases. Any attempt to predefine the necessary conflict resolution set, even if this were possible, would serve to constrain the design space and restrict rather than enhance the creative ambience of the design environment.

An Alternative to A Predefined Conflict Resolver

A better approach to conflict resolution may be found in considering design as a largely unstructured environment in which coordination is accomplished through agents that are endowed with simple tools. These agents gain knowledge of the environment through the application of their tools to the conflicts encountered. An analogy is found in the behavior of ants. The ant interacts with a complicated world and its seemingly sophisticated path is not the result of an intelligent brain, but the application of primitive behavior patterns to the complexity of its environment.

Work in this area has been conducted by others in respect to robots (Brooks 1990). Existing industrial robots that function perfectly in the carefully structured environment of automated factories, tend to perform poorly in the unstructured environment of ordinary factories. Recognition of the fact that the unstructured environment simply contains too many potentially significant details to be structured into a symbolic model, is suggesting the need for an alternative approach: the progressive acquisition of knowledge through the interaction of simple mechanisms with each other and a complicated world.

The application of this principle to the computer-aided design environment would suggest a conflict resolution structure that differs in several ways from the current ICADS Expert Design Advisor model. First, it is clear that the conflict resolution structure must be generated in a more dynamic manner and that it must be provide the capability for making decisions that are customized by the user, or user group. However, this does not mean that all of the functions of the Conflict Resolver need to be developed specifically for each user. Indeed, there are many requirements and constraints that are quite static.

It should be noted that in order to provide for this customizing, it is necessary to expand the scope of the conflict resolution structure beyond that of handling only conflicts. It is essential to provide new knowledge to the system that it can use to reason with respect to its own function, and of course, the facilities for using that knowledge must be added.

Several new features have been considered for enhancement of the Conflict Resolver, including:

- identification of predefined universals
- design project constraints
- preferences learned during previous use
- implications deduced from changes in the structure

The new conflict resolution system for ICADS should identify predefined universals, values that are independent of project and designer. This would make it possible to provide a core of knowledge that would be constant to all users and design projects. It would also make it easier to make certain that such values are not changed in a learning process.

Project constraints are handled in the current model through incorporation into the conflict resolution system by the Attribute Loader, when the initial design session is begun. The user interface under current development will make it possible to modify these constraints during the design session and to dynamically update the database from which these values originally came. By modifying the software that will provide for the dynamic modification of project constraints, it would also be possible to develop a mechanism for dealing with user preferences.

Particularly in architectural design, it is necessary to allow the users to have different opinions about what constitutes a conflict, in at least some of the design variables. It is also necessary to permit a single user to change an opinion expressed in a previous design session. The current working model permits the designer to ignore an unresolved conflict, or to assign values to current values in the solution design object frames to eliminate the conflict; but it does not remember this action so as to resolve the conflict in the chosen manner for a future design session. By enabling the designer to enter a 'learning' mode, the assignments that resolve the conflict could be entered into the Conflict Resolver software. The form of most of the rules in the current working model is such that it would be easy to automatically generate such rules by asking the designer to answer a few questions.

The major difficulty in permitting the user to specify rules that resolve conflicts is basic to all of knowledge engineering. It is quite possible that by making a series of decisions, the designer will create an environment that requires more global, or causal, knowledge to be resolved than the particular user can easily handle. However, it may also be the case that the only way to eventually develop robust sophisticated design systems is to permit a great deal of experimentation. Perhaps this difficulty can be moderated by incorporating knowledge based programs to help formulate the rules to be added, because some success has been achieved in generating good rules via acquisition programs (Gruber 1989).

Once the users begin to develop the Conflict Resolver, it may also be possible to improve the rules that are initially generated. After the designer has worked with the system in a learning mode, off-line programs could be employed to examine the rulesets. Both consistency and efficiency have been improved by programs that examine such rulesets (Muggleton 1990). However, it is widely known that as rule-based systems grow, generally they become increasingly difficult to understand and maintain (Metrey 1986). One possible way to ameliorate this problem is to reduce the size of the original ruleset, perhaps by 'chunking' (Laird 1986). It would be even nicer if improvements in the effectiveness of rulesets could be made by programs. Unfortunately, very little progress has been made in improving the quality of a set of rules. However, the nature of the rules in the Conflict

Resolver may afford an opportunity for better results. Unlike other areas, the Expert Design Advisor has practically no planning nor scheduling activities. Instead, the actions of the system are in reaction to the guiding activities of the designer.

In fact, the ICADS working model provides a convenient environment for experiments with the conflict resolution structure. In the current model, the Conflict Resolver is depicted as a single component. However, it could be implemented as a set of processes that communicate with each other through the existing Blackboard structure. In addition, it could be implemented as a miniature Blackboard structure itself, or as a combination of Blackboard and non-Blackboard structures. The underlying implementation framework of the working model does not limit the agents to have any particular purpose, it merely provides the mechanism for communication. As a result, parts of the conceptual conflict resolution could be realized through different types of software. They could execute on different kinds of hardware, and they could have their own unique control mechanisms. Furthermore, the automatic means by which the system as a whole is configured makes it possible to quickly create a new combination of Blackboard connections.

CONCLUSIONS

The current working model of the ICADS system has the ability to advise a designer in a real-time design activity. It also exhibits a considerable amount of knowledge about the non-geometric attributes of the objects drawn and the real environment for the design project.

There are two major problems in extending the current ICADS model to become worthwhile in performing commercial design work. The first is the classic difficulty with knowledge acquisition. By separating domain specific knowledge from control knowledge through the use of a blackboard system, some moderation of that problem has been achieved. With the development of interface tools that will enable both designers and knowledge engineers to create and test rules more quickly, some additional reduction in effort should be achieved.

The second problem is that many of the design attributes are evaluated in different ways by different designers, and in different ways at different points in time by the same designer. Therefore, in order for a computer system to assist the designer in close agreement with the designer's preferences, it is necessary that the system be able to learn those preferences. The designer differences pertain not only to the evaluation of specific design attributes, but also to the manner in which incompatible assignments to attributes should be resolved.

The software framework of the ICADS working model makes it possible to begin experiments with mechanisms to enable the system to enhance and customize its knowledge by learning from its users. It is possible to quickly change both the hardware and software configurations of the system. In particular it is easy to provide blackboard systems as components within the major blackboard system itself. It is known that similar applications have been overwhelmed with information to a single coordinator (Durfee 1988), so it may become necessary to use this reorganization facility as the model is extended.

No engineering design methodology or performance tools have yet been developed for blackboard systems, so that in spite of the fact that significant performance and operation

facilities can be provided by such systems, it is necessary to provide evaluation exclusively through implementation experiments.

The success of the current ICADS prototype and the clarity with which the implementation of the next enhancement can be made are very encouraging. Hopefully, Expert Design Advisors that provide valuable assistance in the design process without imposing undesirable constraints on the user can be developed in this century.

REFERENCES

- Akin, O. (1978). How do architects design, in Latombe (ed), *Artificial Intelligence and Pattern Recognition in Computer-Aided Design*, IFIPS, North Holland.
- Assal, H. and Myers, L. (1990). An implementation of a frame-based representation in CLIPS, *First CLIPS Conference Proceedings*, Houston, pp.570-580.
- Au, T. and Stelson, T. E. (1969). *Introduction to Systems Engineering: Deterministic Models*, Addison-Wesley, Reading.
- Bridges, A. H. (1976). The interrelationship of design and performance variables, in D. R. Smith and C. W. Jones (eds), *CAD76*, IPC Science and Technology Press, Guildford, UK, pp.22-30.
- Brooks, R. A. (1990). A robot that walks: emergent behaviours from a carefully evolved network, in P. H. Winston and S. H. Shellard (eds), *Artificial Intelligence at MIT - Expanding Frontiers*, MIT Press, Boston.
- Coyne, R. C., M. A. Rosenman, A. D. Radford, M. Balachandran and J. S. Gero (1989). *Knowledge-Based Design Systems*, Addison-Wesley, Reading.
- Cross, N. (ed) (1984). *Developments in Design Methods*, Wiley, New York.
- Dudnik, E. E. (1977). Uncertainty and the design of building subsystems: a linear programming approach, *Building and Environment*, 12: 111-16.
- Durfee E. (1988). *Coordination of Distributed Problem Solvers*, Kluwer Academic, Boston.
- Durrant-Whyte, H. F. (1988). *Integration, Coordination and Control of Multi-Sensor Robot Systems*, Kluwer Academic, Boston.
- Dym, C. L. (ed) (1985). *Applications of Knowledge-Based Systems to Engineering Analysis and Design*, American Society of Mechanical Engineers, New York.
- Fenves, S., U. Flemming, C. Hendrickson, M. L. Maher and G. Schmitt (1988). An integrated software environment for building design and construction, *Proceedings of Fifth Conference on Computing in Civil Engineering*, ASCE.
- Flemming, U., R. Coyne, T. Glavin and M. Rychener (1986). A generative expert system for the design of building layouts, *Applications of Artificial Intelligence in Engineering Problems*, Vol.II, Springer-Verlag, Berlin, pp.811-822.
- Gero, J. S. (1976). Dynamic programming in the CAD of buildings, in D. R. Smith and C. W. Jones (eds), *CAD76*, IPC Science and Technology Press, Guildford, UK, pp.31-70.
- Gero, J. S. (ed) (1985a). *Design Optimization*, Academic Press, Reading.
- Gero, J. S. (ed) (1985b). *Knowledge Engineering in Computer-Aided Design*, North-Holland, Amsterdam.

- Gero, J. S., M. L. Maher and W. Zhang (1988). Chunking structural design knowledge as prototypes, in J.S. Gero (ed), *Artificial Intelligence in Engineering: Design*, Elsevier/CMP, New York.
- Gruber, T. R. (1989). *The Acquisition of Strategic Knowledge*, Academic Press, Reading.
- Harfmann, A. C. (1987). The rationalizing of design, in Y. Kalay (ed), *Computability of Design*, Wiley-Interscience, New York, pp.1-8.
- Laird J., P. Rosenbloom and A. Newell (1986). *Universal Subgoal and Chunking*, Kluwer Academic, Boston.
- Mackinder, M. and Marvin, H. (1982). *Design Decision-Making in Architectural Practice*, Building Research Establishment, Dept.of Environment, UK, July.
- Maher, M. L. (ed.) (1987). *Expert Systems for Civil Engineers: Technology and Application*, American Society of Civil Engineers, New York.
- Maher, M. L. (1988). HI-RISE: an expert system for preliminary structural design, in M. D. Rychener (ed), *Expert Systems for Engineering Design*, Academic Press, pp.37-52.
- Maher, M. L. and Zhao, F. (1987). Using experience to plan the synthesis of new designs, in J. S. Gero (ed), *Expert Systems in Computer-Aided Design*, North Holland, Amsterdam, pp.349-369.
- Mallen, G. L. and Goumain, P. G. R. (1973). *The Analysis of Architectural Design Activity in the Working Environment*, Report 108/4 DDR, Royal College of Art, London, UK.
- March, L. and Stiny, G. (1985). Spatial systems in architecture and design: some history and logic, *Environment and Planning B*, 12: 31-53.
- McLaughlin, J. L. and Gero, J. S. (1987). Acquiring expert knowledge from characterized designs, *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 1(2): 73-87.
- Mettrey, W. (1987). An assessment of tools for building large KB systems, *A I Magazine*, 8(4): 81-93.
- Mitchell, W.J., J. P. Steadman and R. S. Liggett (1976). Synthesis and optimization of small rectangular floor plans, *Environment and Planning B*, 3: 37-70.
- Muggleton, S. (1990). *Inductive Acquisition of Expert Knowledge*, Addison-Wesley, Reading.
- Myers, L. and Pohl, J. (1989). ICADS DEMO1: a prototype working model, *Fourth Eurographics Workshop on Intelligent CAD Systems*, Paris, France, pp.33-71.
- NASA (1989). *CLIPS Architecture Manual (version 4.3)*, Artificial Intelligence Center, Lyndon B. Johnson Space Center, Houston.
- Nijkamp, P. and Spronk, J. (eds) (1981). *Multicriteria Analysis in Practice*, Gower, London.
- Page, J. K. (1974). The optimization of building shape to conserve energy, *Journal of Architectural Research*, 3(3): 20-8.
- Pohl, J., A. Chapman, L. Chirica, R. Howell and L. Myers (1988). *Implementation Strategies for a Prototype ICADS Working Model*, Technical Report: CADRU-02-88, CAD Research Unit, Design Institute, California Polytechnic State University, San Luis Obispo, California.
- Pohl, J., L. Myers, A. Chapman and J. Cotton (1989). *ICADS: Working Model Version 1*, Technical Report: CADRU-03-89, CAD Research Unit, Design Institute, School of

- Architecture and Environmental Design, California Polytechnic State University, San Luis Obispo, California.
- Radford, A. D. and Gero, J. S. (1985). *Design by Optimization in Architecture, Building and Construction*, Van Nostrand Reinhold, New York.
- Rychener, M. D., R. Banares-Alcantara and E. Subrahmanian (1984). *A Rule-Based Blackboard Kernel System: Some Principles in Design*, Design Research Center, Carnegie-Mellon University, November.
- Rychener, M. (ed.) (1988). *Expert Systems for Engineering Design*, Academic Press, Reading.
- Schoen, D. A. (1983); *The Reflective Practitioner: How Professionals Think in Action*, Basic Books, New York.
- Sriram, D. and Adey, R. (eds) (1986). *Applications of Artificial Intelligence in Engineering Problems*, Vols. I and II, Springer-Verlag, Berlin.
- Stark, R. M. and Nicholls, R. L. (1972). *Mathematical Foundations for Design*, McGraw-Hill, New York.
- Taylor, J. and Myers, L. (1990). Executing CLIPS expert systems in a distributed environment, *First CLIPS Conference Proceedings*, Houston, pp.686-697.
- Taylor, J. and Pohl, J. (1990). A geometry interpreter for extracting architectural objects from the point/line schema of a CAD database, *Proceedings of Intersymp-90: Knowledge-Based Systems in Building Design*, Baden-Baden, West Germany.

A knowledge-level analysis of several design tools*

A. Balkany, W. P. Birmingham and I. D. Tommelein

Department of Electrical Engineering and Computer Science

Department of Civil Engineering

The University of Michigan

Ann Arbor MI 48109 USA

Abstract. Design has been extensively studied by artificial intelligence researchers for many years. These studies have resulted in a large number of design tools that perform interesting tasks. Understanding the capabilities of these tools is, however, very difficult, which seriously impedes progress in the field. In this paper, we suggest that a knowledge-level analysis of design tools will allow us to more fully understand different tools, and to constructively compare them. The initial analysis of five tools is presented.

1. INTRODUCTION

Design is an important aspect of an engineer's job and is of great economic importance. Increasing designer productivity is essential to remaining competitive. Design is, however, as complex as it is important. Thus, it represents an appealing area of study. The artificial intelligence (AI) community, in addition to others, has been active in design research for some time (Mostow, 1985). Much of this activity is manifested as tool development that attempts to automate some portion of an engineering design task. The most immediate result of this research has been the development of a large number of tools for a number of different domains.

These tools have demonstrated the feasibility of automating particular tasks, but have not necessarily lead to an increased understanding of the design process itself. Two observations

* This work was funded, in part, by a gift from Digital Equipment Corporation, and by the National Science Foundation Grant Mips-9057981. The opinions expressed here are our own, and not necessarily those of Digital Equipment Corporation or NSF.

support this statement. First, it is difficult to objectively compare the operation of various design tools. Most comparisons are based on particular functions, without much emphasis on the problem-solving approach taken. Stated differently, it is difficult to unambiguously identify the problem-solving process and knowledge representation employed by various tools. Second, there has yet to emerge a generally accepted theory of the design process.

The current state of understanding is not surprising. Research into design, at least at the level taken by AI researchers, is young and, in large part, remains enigmatic. With the large number of developed systems available, however, it is appropriate to begin to study and understand their operation. Through this study, we hope to more thoroughly understand the design process. We suggest that a good place to begin such a study is to consider the following points:

- Problem-Solving Method: what is the method used to perform the design task? This is where the *action* of the tool comes from.
- Task Type: what is the fundamental nature of the design task? The type of task can both indicate complexity and provide insight into the types of problem solving required.

In order to identify these points about design systems, a careful analysis must be undertaken. At present, this is a difficult process since there is a lack of a common vocabulary for describing systems, there appears to be no strong consensus on what aspects of a system should be reported, and the system descriptions are limited. We propose that a knowledge-level description (Newell, 1981) of these tools, as defined in Section 4, is a good starting point. Depicting systems at this level emphasizes precisely the points that are important for understanding behavior, which, in turn, leads to understanding the nature of various design tasks.

In order to focus our efforts, we have restricted our scope initially to a specific type of design, called *configuration*. Roughly speaking, configuration systems construct a solution from a fixed library of parts. These parts are well-characterized with respect to their function and relationships among each other. During configuration, parts are selected, and perhaps interconnected, such that a set of specifications provided by the user are met. (See Mittal and Frayman (1989) for a more formal definition of configuration problem solving.) This type of problem solving is an ideal initial starting position for the following reasons: the definition of the task is cogent; there are number of working systems that perform the task; and, there is a great deal of variety inherent to the task, *i.e.* not all configuration tasks are the same.

This paper reports the initial results of a knowledge-level analysis of a number of interesting and diverse configuration design tools. Following a discussion of nomenclature,

the analysis technique and task type characterization are provided in Section 3. Next, the results of the analysis of five tools from a variety of domains are given in Section 4. In Section 5, a comparison of our approach to that of other researchers is presented. A summary and future research directions are given in Section 6.

2. NOMENCLATURE

Definitions in the design area are often inconsistent and confusing. In order to precisely describe our work, we first define the terms used throughout this paper. It is possible that these definitions will change as researchers begin to converge on common terms. Nonetheless, the following discussion provides a consistent starting point.

A design process is characterized by the **problem-solving method** (PSM) that performs a certain task in the domain of the application. We shall use the terms PSM and problem solver interchangeably. The term **domain** refers to an area of expertise, for example that of *computer engineering*, or that of *civil engineering*. A domain introduces terms and concepts needed to communicate effectively and may also imply specialized knowledge needed to perform a task. The term **task** refers to an activity performed by someone, such as designing a single-board computer, designing a single-family home, or laying out the site of a coal-fired power plant. We consider a task to be a specification of a PSM.

A design process consists of the successive application of **mechanisms** (See Section 4.3), each with its own operators. A mechanism is a procedure with well-defined inputs and outputs that performs an elementary step in a task. A mechanism may require domain-specific knowledge for its operation.

While it is possible that only a single mechanism will be needed for some tasks, most tasks will use a combination of mechanisms. This requires additional knowledge to control their sequencing. A PSM is a high-level procedure that uses control knowledge to invoke mechanisms in the proper order to perform the task. For example, the Emycin (van Melle, 1980) shell embodies the heuristic classification method that successively applies mechanisms for data abstraction, heuristic match, and solution refinement. Note, however, that not all control knowledge resides in the PSM; some domain-specific knowledge also may be required to properly sequence the mechanisms. So, **control knowledge**, a more general term, includes all knowledge required.

Domain and PSM are distinct and independent. On the one hand, it is possible for a PSM to be applied to several different domains. For example, Emycin could be applied to domains other than blood disease, its original domain. In fact, it appears as though there is significant

overlap between PSMs that exist in many domains, as noted by several researchers who are either attempting to codify general-purpose PSMs (Chandrasekaran, 1986, Klinker *et al.*, 1990, Mittal and Frayman, 1989, Clancey, 1985), or have demonstrated generality by porting tools from one domain to another (Birmingham *et al.*, 1989a, Langrana *et al.*, 1986, Johnson and Hayes-Roth, 1988). On the other hand, there are no examples yet of problem solvers that are task independent.

A conceptual model of the relationships between the terms described here is a cube, as shown in Figure 1. Domain, control knowledge, and mechanisms are shown on orthogonal planes, as we contend that they represent sets of design knowledge that may be shared among different tasks. Thus, there may be "sharing of faces" between similar tasks, or domains. Mechanisms may be shared by exchanging "slices", where each slice represents a mechanism. Each design tool can be represented by a cube.

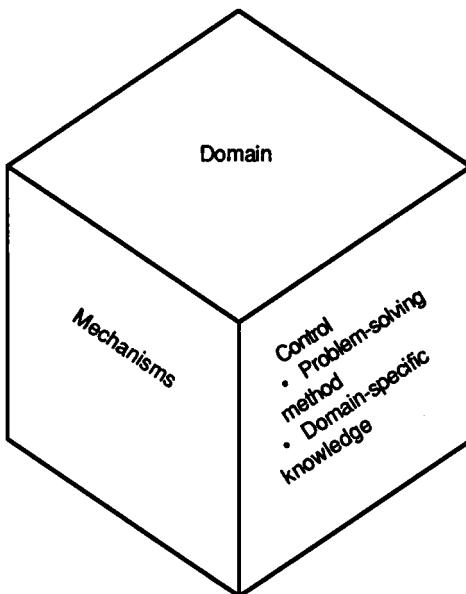


Figure 1. The *cube* representation of a task.

3. KNOWLEDGE-LEVEL ANALYSIS AND PROBLEM CHARACTERIZATION

The knowledge-level analysis method, described next, allows us to distill the essence of a design tool's problem-solving approach and the mechanisms used to effect that behavior. From this distillation, the behavior of the system becomes apparent. We also present the task characterization based on subtask interactions. We begin with an overview of the knowledge level.

3.1 The Knowledge Level

It is common to describe computing systems at various levels of abstraction. The knowledge level, proposed by Newell, is an abstraction that attempts to isolate the behavior of a system from the symbolic notation used to implement that behavior. The elements of this level are goals, and means of achieving those goals via behaviors. We refer to these behaviors as PSMs; the actual means of effecting those behaviors are not considered. The implementation of a behavior, via a program, occurs at the symbol level.

Creating knowledge-level descriptions of design systems is valuable. Since these descriptions will, by definition, describe behavior, not implementation issues, a more general understanding of the design process can be obtained from them. Essentially, these descriptions both highlight the important problem-solving processes required, and imply the types of knowledge needed at the symbol level. As shown in Section 4, knowledge-level descriptions facilitate making meaningful comparisons between systems.

3.2 The Analysis Method

Describing systems at the knowledge level is more art than science, especially if the descriptions are created by those who did not build the system. It is not possible yet to describe an exact procedure for creating knowledge-level descriptions. We hope, however, to demonstrate how these descriptions can be formulated by providing examples of knowledge-level descriptions of systems whose behaviors can be obtained from the literature, and by outlining the method used to create those descriptions.

The analysis method consists of identifying a system's PSM and its associated mechanisms, attempting to capture the overall behavior of a system. While system descriptions usually provide this information, it is often intermingled with symbol-level details. Discerning the PSM is an iterative process, where initial conjectures about the system's operation are refined by comparison to execution traces, if provided, or by simulating system operation by hand. Mechanisms are found by isolating atomic parts of the system behavior. For example, most of the systems studied thus far use dependency-directed backtracking to recover from a design failure. This backtracking is encapsulated as a mechanism, that is used to achieve a higher-level behavior such as "fix-constraint-violation".

The following are a set of rough guidelines that were used in preparing the knowledge-level descriptions; the set will grow over time:

- Identify the outermost or main loop in the system. This loop will also appear in the knowledge-level description of the system.

- Encapsulate the major functions of the system into mechanisms, identifying the inputs and outputs. There are a number of criteria for doing this. One is to choose a partitioning that will maximize "cohesion" (Constantine and Yourdon, 1979). Two of the strongest indicators suggesting the presence of a mechanism are units that exhibit "functional" or "sequential" cohesion. With functional cohesion, each operation is necessary for the execution of a single function. With sequential cohesion, data flow through consecutive steps produces one logical action.
- Outline the flow of control between these mechanisms that characterizes the PSM.
- Contain all loops either in the PSM or within a mechanism, *i.e.* loops cannot transcend mechanisms.
- Apply the preceding steps iteratively to any mechanism, treating it as if it were a PSM to be broken down and described. This can be repeated until the "granularity" of a mechanism is found to be adequate.

3.3 The Task Characterization

Understanding the type of task a tool operates on is as important as knowing how the tool works. Even the comparatively well-defined task of configuration design has a wide variety of task types. Typically, characterization has been done relative to abstract measures, such as the degree of novelty the problem exhibits, or relative to task types, such as design *versus* classification. It is also meaningful, to characterize tasks by the interaction between their subtasks. These interactions differentiate various types of configuration design tasks from one another, and point to differences between the design problem solvers.

Task characterization is based on three measures: the type of information dependencies between subtasks, when those dependencies can be determined, and when an ordering on subtasks can be established. Information dependencies arise when the solution to a particular subtask requires information generated from finding the solution to another subtask. Since all tools studied thus far exploit some form of subtask decomposition, all have some type of dependency. It is possible, however, that some class of tasks will have no dependencies. Dependencies can be mutual, *i.e.* two or more subtasks may depend on each other for information, or linear, when they are not mutual. In some types of tasks, dependencies can be determined when the system is being built, which is termed *a priori*. In other cases, the dependencies can only be uncovered during the design process, which is termed *runtime*.

Based on these measures, there are four task types. Type 0, a trivial case, has no subtask interactions; Type 0 will not be considered further. Problems may be classified as Type 1 when linear interactions exist, and it is possible to provide an invariant ordering on solving those subtasks. With Type 2 tasks, all potential interactions are known *a priori*, but the

ordering of the tasks is not known. In this case, it may be known that two subtasks depend on each other, but which gets solved first may change from design to design. Furthermore, which parts of the dependency network become actual dependencies is determined at runtime. Finally, with Type 3, both the potential dependencies and the ordering of subtask solutions are determined at run time. Examples of Type 2 and Type 3 tasks are given in Section 4.4.

4. PRELIMINARY RESULTS

This section presents the preliminary results of our analysis, the PSMs and mechanisms used by five tools. From the *design task* viewpoint, we notice that the tools fall under the task types described in the previous section. This section begins with an overview of the design tools studied.

4.1 The Systems Studied

The following five systems have been studied: Air-cyl (Brown and Chandrasekaran, 1989), VT (Marcus *et al.*, 1987), Pride (Mittal *et al.*, 1986), M1 (Birmingham *et al.*, 1989b), and SightPlan (Tommelein *et al.*, 1991). These tools were selected according to the following criteria:

- They solve configuration tasks.
- Air-cyl, VT, and Pride are design systems that are well documented in the literature. M1 and SightPlan perform tasks not covered by the other tools.
- There are various degrees of similarities between these systems. Thus, we can determine the knowledge-level description's resolution, that is, the ability to distinguish similar and dissimilar PSMs and mechanisms.

We assume that these systems represent interesting points in the space of configuration tools, thereby allowing us to get reasonable coverage in testing the validity and expressiveness of the descriptions. As time permits, we will study more systems, articulate their methods and mechanisms, and compare them to the systems already understood. We encourage others to report the results of their systems using a scheme similar to ours.

In the following sections, a knowledge-level description of each tool is provided. There is no standard notation for knowledge-level descriptions; Newell proposed logic, while other researchers have suggested less formal techniques (Clancey, 1985).

We have adopted a three-part model. One part describes the PSM in a high-level programming language-like manner. The second part lists the mechanisms exploited by the method. The third part details control knowledge implicit in the systems, but not given by

the PSM. Section 4.2 describes the PSMs and their mechanisms. Section 4.3 details the mechanisms by giving the knowledge-level description, input parameters, and output parameters for each mechanism. Please note that the following represents our best effort at creating knowledge-level descriptions from published material. Due to the substantial semantic difference between that material and our descriptions, it is possible that errors may have been inadvertently introduced.

4.1.1 Air-cyl

Air-cyl is a system for designing air cylinders that uses a hierarchical community of design agents, called specialists, each with a repertoire of design plans to accomplish a design task. The specialist at the root of the hierarchy contains the most abstract task (design an air cylinder), while the lower-level specialists contain less abstract design tasks, such as designing subsystems or components. The plans of abstract specialists call less abstract specialists to accomplish their tasks. When a specialist's plan fails to accomplish its task and satisfy the given constraints, the specialist's next plan is tried. Failure of a specialist to accomplish its task is handled by chronological backtracking. If all a specialist's plans fail, it returns a failure message to its parent, which considers *its* plan to have failed. The design fails if the root-level specialist returns failure, and is successful otherwise.

Air-cyl uses the following implicit control knowledge:

- (a) Which tasks to perform and which specialists to invoke.
- (b) How to order the plans to be tried. Each specialist has this knowledge in addition to the list of plans. Plans are ordered to try to minimize backtracking.

4.1.2 VT

VT is a system for designing elevators. VT has a flexible forward chaining and backtracking scheme. Steps to extend the design are taken whenever the information required by them is available. When a design extension is made, VT makes an entry in a dependency network that records for each value, the other values used to obtain it. When a constraint violation is detected, VT chooses a fix for that violation from a list ranked by cost. When a fix is selected, VT makes it, then uses the dependency network to correct any values that might be inconsistent with the changed value. The design is complete if the last step is completed with no constraint violations.

VT uses the following implicit control knowledge:

- (a) How to rank each design extension.
- (b) How to rank constraint violation fixes by cost.

4.1.3 Pride

Pride is a system for designing paper-handling systems in photocopiers. **Pride** uses a hierarchy of design goals to decompose the process into simpler steps. Each design goal has a corresponding design method, called a plan, for accomplishing it. Constraint violations are handled by dependency-directed backtracking with an extension: redesign advice is given to the method doing the redesign. The design is complete when the top-level goal is accomplished with no constraint violations.

Pride uses the following implicit control knowledge:

- (a) How to decompose the task and represent a hierarchy of components.
- (b) How to select a fix for a constraint violation when one occurs.

4.1.4 M1

M1 is a system for designing small computer systems. **M1** uses a hierarchy to represent electrical and electronic parts. These parts are organized functionally, and are abstracted to yield a graph with abstract parts at the top (representing major functions such as memory and processor) and with physical components as the leaves. In addition to parts, **M1** also represents pieces of computer designs in *templates*, that show the structural relationships between various parts. The fundamental mechanism for producing a design is called the *design cycle*, which is conceptually divided into two phases: the search for function (SFF) and the search for structure (SFS). During SFF, the functional hierarchy is traversed to find components that meet specifications either given by the user or generated by **M1**. In SFS, appropriate templates are chosen from a library based on the components used in the design. **M1** can fail during the design process. Failure is recognized by the inability to select a part that matches the specifications. Failure is handled by redesigning some part, then using dependency-directed backtracking to propagate the change by redesigning all affected parts. The design is complete if all abstract parts have been realized by physical parts.

M1 uses the following implicit control knowledge:

- (a) How to decompose the task and represent a hierarchy of components.
- (b) How to select a fix for a constraint violation when one occurs.

4.1.5 SightPlan

SightPlan is a system for laying out temporary facilities on construction sites. **SightPlan** possesses hierarchical descriptions of the type of desired design steps, represented in several

skeletal plans (also called *methods* in this paper). One such plan describes how SightPlan can construct a layout for the overall site. Another plan describes how SightPlan can create a layout for a selected set of objects. The system pursues each plan in a depth-first manner, and posts the leaf nodes as an active control strategy. Each step can involve none, one, or multiple problem-solving actions. Which action should be taken at a given point in the problem-solving process is determined opportunistically at run time. The program stops when the goal of a plan is met.

SightPlan uses the following implicit control knowledge:

- (a) How to identify steps to take.
- (b) How to rate the goodness of steps based on hierarchical skeletal control plans.

4.2 The Problem-Solving Methods

The following are the PSMs for each system, with their mechanisms printed in boldface. When there are multiple mechanisms in a PSM step, they are invoked sequentially unless otherwise specified.

4.2.1 Air-cyl

1. Extend the design by sending a design request to a specialist who has plans for making that extension.
invoke-specialist
2. The specialist invokes the next untried plan. If there are no more untried plans, return failure to the parent specialist and repeat this step with the parent. If Air-cyl is already at the top-level specialist, quit with failure.
select-a-plan
3. Execute the plan. Plans consist of one or more tasks, which consist of one or more steps. Steps can invoke other specialists.
 - (a) **execute-plan-AC**
 - (b) **perform-task**
 - (c) **do-step**
4. Identify constraint violations; if none, go to step 5, else go to step 2.
check-for-constraint-violations
5. If the design is complete, quit with success, else go to step 1.
test-if-done-top-level-plan-completed

4.2.2 VT (Adapted from McDermott, 1988)

1. Extend the design and identify constraints on the extension just formed.
 - (a) **extend-design**
 - (b) **find-constraints**
2. Identify constraint violations; if none, go to step 8.

check-for-constraint-violations

3. Suggest potential fixes for a constraint violation.

suggest-fixes

4. Select the least costly fix not yet attempted. If there are no more fixes, quit with failure.

rank-select-fixes

5. Tentatively modify the design and identify constraints on the modification just formed.

- (a) **make-modification**
- (b) **find-constraints**

6. Identify constraint violations due to the revision; if any, go to 3.

check-for-constraint-violations

7. Remove relationships incompatible with the revision.

propagate-changes

8. If the design is incomplete, go to step 1, else quit with success.

test-if-done-variable-values-assigned

4.2.3 Pride

1. Extend the design by applying a design method to accomplish a design goal.

SELECT ONE OF:

- (a) **select-design-method**
- (b) **apply-design-sequence-method**
- (c) **apply-design-rule-group-method**
- (d) **apply-design-generator-method**

2. Identify constraint violations; if none, go to step 5.

detect-constraint-violations

3. Suggest or construct a fix for a constraint violation. If no fix is possible, quit with failure.

SELECT ONE OF:

- (a) **suggest-fixes**
- (b) **rank-select-fixes**

OR:

construct-modification-to-satisfy-constraint-violation

4. Modify the design and go to step 2.

make-modification

5. If the design is complete, quit with success, else go to step 1.

test-if-done-top-level-goal-accomplished

4.2.4 M1

1. Select an abstract part to expand.

choose-abstract-part-to-expand

2. Begin the design cycle by generating specifications for the abstract parts.

generate-specifications

3. Select a candidate part. If no satisfactory candidate part can be found, go to step 6.

select-from-candidates

4. Determine structure of part.

- (a) **expand-part**
- (b) **select-template**
- (c) **do-calculations**

5. If the design is complete, quit with success, else go to step 1.

test-if-done-no-more-abstract-parts

6. Invoke the failure handling module. If a fix is impossible, quit with failure, else go to step 1.

construct-modification-to-satisfy-constraint-violation
propagate-changes

4.2.5 SightPlan

1. Extend a design by identifying applicable plans

invoke-all-plans

2. Execute the plan.

- (a) **execute-plan-SP**
- (b) **identify-domain-action**
- (c) **select-action**
- (d) **perform-domain-action**
- (e) **propagate-change-on-domain-action**
- (f) **update-plan**

3. Identify unsatisfactory constraint satisfaction.

detect-constraint-violation

4. If the goal of a plan has been satisfied, quit with success for that plan.

test-if-done-top-level-plan-completed

4.3 The Mechanism Library

This section describes the mechanisms at the knowledge level and groups them by function. Although the terminology of the original systems has been preserved, many of the mechanisms in each group appear to be similar, and in some cases identical. We are working on a reformulation of the PSM for each system with subsuming generic mechanisms.

4.3.1 Select Design Extension Mechanisms

choose-abstract-part-to-expand

“Select an abstract part to expand. Part selected determines subtask ordering.”

input:	functional hierarchy, specifications, design state
output:	an abstract part
identify-domain-action	"Identify all possible domain actions."
input:	design state, design state change
output:	possible domain action
select-action	"Rate all possible actions by matching possible domain actions against current control and select the one with the highest rating."
input:	set of actions
output:	selected action
select-a-plan	"Select the next plan to try. Plans are heuristically ordered to minimize backtracking."
input:	plan list, design state, specifications
output:	a plan
select-design-method	"Select the appropriate method (algorithm) for the given design goal."
input:	design goal, design state
output:	design method
select-from-candidates	"Select a less abstract part from the candidates below in the functional hierarchy which best fits the specifications and minimizes resource use. The inability to select a part indicates failure."
input:	abstract part, specifications, functional hierarchy
output:	a part
select-template	"Select a template that specifies how the part is to be used, and possibly what subparts are required."
input:	templates, specifications, design state, abstract part
output:	a template

4.3.2 Make Design Extension Mechanisms

apply-design-generator-method	"Heuristically generate a guess at initial values for design parameters."
input:	design state, specifications
output:	changed design state
apply-design-rule-group-method	"Apply a set of rules to make a design decision."
input:	design state, specifications, rules
output:	changed design state
apply-design-sequence-method	"Decompose a design goal into subgoals."
input:	design goal, design state, specifications
output:	subgoals
do-calculations	"Use the known values to compute new values."
input:	a template, specifications
output:	design state change
do-step	"Do one step of a task. A step is the lowest level operation in Air-cyl."
input:	step to be done
output:	design state change, success (Boolean)
execute-plan-AC	"Execute the given plan."
input:	plan to be executed, design state
output:	design state change, success (Boolean)
execute-plan-SP	"Execute a plan, i.e., enlist the prescribed types of actions as active control."
input:	plan, current control
output:	updated control
expand-part	"Modify the part to match other characteristics of the design."
input:	a part, constraints, design state
output:	modified part
extend-design	"Select an extension to the design and make it."
input:	design state, list of extensions, extension selection function
output:	design state change, dependency network entry.
generate-specifications	"Determine desired parameter values for the part to be expanded."

input: design state or input from user	
output: specifications for the selected part	
invoke-all-plans	"Activate all plans with true preconditions. Active plans can run in parallel or in sequence."
input: plan, design state	
output: applicable plans	
invoke-specialist	"Send a design request to a specialist"
input: specifications, design state, specialist list	
output: design state change, success (Boolean)	
make-modification	"Make the given modification."
input: modification, design state, dependency network	
output: modified design state, modified dependency network	
perform-domain-action	"Meet constraint(s) on design parts."
input: design state, design state change, constraint to be met	
output: design state change	
perform-task	"Perform a task of the plan consisting of one or more steps."
input: task to be performed, design state	
output: design state change, success (Boolean)	
propagate-change-on-domain-action	"Propagate state change on domain action."
input: design state, design state change	
output: proposed new action(s)	
propagate-changes	"Recalculate values dependent on variables which were changed."
input: fix, design state, dependency network	
output: changed design state	
update-plan	"Propagate state change on control action."
input: design state, design state change	
output: move to next step in the plan	

4.3.3 Handle Constraint Violation Mechanisms

detect-constraint-violations	"Detect constraint violations."
input: design state, constraint list.	
output: violated constraints	
construct-modification-to-satisfy-constraint-violation	"Construct a modification based on the dependency network and design knowledge."
input: constraint, design state, dependency network	
output: a modification	
find-constraints	"Identify constraints that apply to the last design state change."
input: design state change, list of constraints	
output: list of constraints	
rank-select-fixes	"Use domain knowledge to rank the fixes and choose the best."
input: design state, list of fixes	
output: a fix	
suggest-fixes	"Find design alterations to resolve the constraint violation."
input: design state, violated constraint, dependency network	
output: list of fixes	

4.3.4 Test If Done Mechanisms

test-if-done-no-more-abstract-parts	"Answer TRUE if there are no more abstract parts, indicating that the design is complete."
input: design state, constraints	
output: done (Boolean)	
test-if-done-top-level-goal-accomplished	"Answer TRUE if the top-level goal is accomplished, indicating that the design is complete."
input: design state, constraints	
output: done (Boolean)	

test-if-done-top-level-plan-completed

input: design state, constraints
output: done (Boolean)

test-if-done-variable-values-assigned

input: design state
output: done (Boolean)

“Answer TRUE if the top-level plan has been completed, indicating that the design is complete”

“Answer TRUE if all design variables have had values assigned, and the design is therefore complete.”

4.4 Task Characterization

The characterization of the type of task solved by each system is given in Table 1. By comparing the PSMs with the task characterization, it appears that systems that can tackle higher numbered task types can also tackle lower numbered ones, although with some overhead. This conjecture, however, is still subject to experimental confirmation.

<u>System</u>	<u>Problem Type</u>
Air-cyl	2
VT	2
Pride	2
M1	3
SightPlan	2

Table 1. Task characterization of each system.

The specialists invoked by Air-cyl’s plans (and therefore the subtask dependencies) are all known *a priori*. When one of Air-cyl’s specialists fails, control returns to the parent specialist, which tries its next plan. Since this introduces a new actual dependency, it implies that Air-cyl is of Type 2. Likewise, all of VT’s design extensions and potential dependencies are known *a priori*, but its opportunistic forward chaining scheme means that the actual dependencies are only known at runtime. Therefore, VT is also of Type 2. Pride’s hierarchy of design goals also define all the potential subtask dependencies *a priori*, but its backtracking behavior in response to constraint violations means that the actual dependencies, here too, are only known at runtime, making it Type 2. M1, as described here, is also Type 2 because the potential dependencies are defined by its functional hierarchy and templates, but the subtask ordering and actual dependencies are only known at runtime because of its backtracking. At the time of this writing, however, an addition is being made to M1 that will cause Type 3 behavior. This addition will permit non-template-specified design extensions, implying that even the potential subtask dependencies will not be known *a priori*. SightPlan’s plan hierarchy defines the potential subtask dependencies *a priori*, but its opportunistic forward chaining means that the actual dependencies are only known at runtime. Therefore, SightPlan is of Type 2.

5. DISCUSSION

Our work builds on Newell's notion that knowledge is an abstraction that can be separated from the symbols representing it. Clancey clearly illustrated the value of knowledge-level analysis in characterizing Mycin's PSM as that of domain-independent heuristic classification (Clancey, 1985). While heuristic classification is an appropriate method for solving diagnostic tasks, it does not apply to solving configuration tasks. It is clear, though, that the generality of systems can be assessed and benefits gained, by abstracting specific aspects of implemented systems. For example, Emycin lent itself to implementation of diagnostic problem solvers in a multitude of domains. Similarly, Accord, the domain-independent language to express constructive assembly actions in Protean's biochemistry domain, served SightPlan's construction site layout application equally well (Hayes-Roth *et al.*, 1988).

By developing knowledge-level descriptions of various design tools, we are attempting to find the commonalities and differences between these tools, which will lead to a set of domain-independent problem solvers, such as Clancey has done with heuristic classification. One significant difference between our work and his is the variety that exists in configuration tasks, as shown in Section 3.3. Because of this variety, we believe one or several PSMs will be needed for each task type.

An excellent knowledge-level analysis of configuration tasks was performed by Mittal and Frayman (Mittal and Frayman, 1989). While our work is similar in nature, our interpretation of what constitutes configuration systems is slightly broader; Mittal and Frayman may not consider SightPlan to be a configuration system. Our analysis differentiates between systems based on lower-level descriptions using mechanisms, whereas Mittal and Frayman analyze higher-level PSMs. In addition, their work seems to point to only a single PSM for configuration tasks.

In an attempt to facilitate construction of knowledge bases, Chandrasekaran engages in *information processing analysis* by decomposing design into sub-processes and analyzing these processes based on several criteria (Chandrasekaran, 1988). So far, the following six elementary generic tasks were distinguished (Chandrasekaran, 1986): hierarchical classification, hypothesis matching, knowledge-directed information passing, object synthesis by plan selection and refinement, state abstraction, and abductive assembly of hypotheses. When combined, these tasks implement problem solvers. Chandrasekaran's work concentrates on identifying generic tasks, which we believe are similar to mechanisms, without considering how these tasks would be combined to form PSMs. We find the issue of combining tasks very important, but agree with him, however, that the construction of a library of mechanisms facilitates the construction of systems. In fact, Chandrasekaran's (1990) notations of problem-solving method and subtask are much in the same spirit as our

model.

Brown and Chandrasekaran's work has strongly relied on the distinction between Class 1, Class 2, and Class 3 design, roughly speaking, ranging from innovative to routine design (Brown and Chandrasekaran, 1985). We steer away from these class distinctions because we feel that mechanisms are independent of these distinctions. That is, during routine and creative design the same mechanisms may be used. Thus, mechanisms may not necessarily belong to one specific design class. We feel that the task characterization scheme described in this paper provides insight into the difficulty of building a system to perform a task, as opposed to metering its intellectual complexity. In particular, the more complex tasks require more sophisticated mechanisms for ordering subtasks (*e.g.* M1's choose-abstract-part-to-expand).

The characterization based on subtask dependencies has until now prevented us from differentiating the examined systems from one another. This finding suggests that either subproblem interaction does not cast a worthwhile perspective on the design systems we studied, or our examination is not sufficiently refined to highlight distinctions. As we suspect the latter to be the case, a closer examination of the delineation and the scope of the subproblems might yet reveal interesting differences between the systems we studied.

We tried to compare mechanisms with each other in order to understand better what the similarities or differences between systems might be in this regard. One difficulty in doing so lies in achieving a consistent formulation of the mechanisms and their behavior across application systems. Another difficulty lies in understanding the exact input and output of mechanisms. We recognize that such input and output may depend on the model representing domain concepts and relations between them, that is used by each system. If this is the case, a class of mechanisms might apply to a functional decomposition hierarchy of domain concepts, while another class might apply to an abstraction hierarchy of parts. Thus far we have ignored characterizing domain models, although other researchers have stressed the importance of doing so (Clancey, 1985, Steels, 1990). Our present efforts focus on this facet of design system characterization.

Our work is closely related to that of Klinker *et al.* (1990), who are working on simplifying programming by collecting mechanisms that are both *usable* and *reusable*. A mechanism is *usable* if it can be used to perform a task by someone who understands the task, but not necessarily how to program. A mechanism is *reusable* if it can be employed for several domains and tasks. While we are using a similar paradigm, our focus is different. We are attempting understand the particulars of sophisticated problem solvers within the context of a particular task type (configuration); they are concentrating on building a broad range of mechanisms for a wide range of tasks and domains, while limiting the scope of the problem solvers.

Our knowledge-level analysis spans both the knowledge level and the function level in Tong's framework (1987). His knowledge level is composed of a problem characterization, a domain theory (containing problem-solving knowledge), and a characterization of acceptable solutions. We consider the characterization of acceptable solutions ("constraints") to be included in the problem domain. His function level specifies the "problem-solving process", which is equivalent to our PSMs and mechanisms. Tong's third level, the "program level" specifies the primitive structures used in the implementation of the problem-solving system. This is identical to our definition of the symbol level, which directly follows Newell. Tong presents a new approach to knowledge-based design (*goal-directed planning*), while we are interested in the differences between problem solving systems.

Treur introduces a rigorous model of the design process, which includes theories of objects and requirements (Treur, 1989). While creating a powerful model, it does not address the same issues as discussed in this paper. In particular, we attempt to identify the knowledge and PSMs (*strategies* in Treur's terminology) used by various systems. These are specifically the issues that Treur does not address.

Our work is aimed at understanding how design systems operate, namely the way in which they solve their particular problems. What makes these systems different is the design knowledge they use and the domains in which they operate. Thus, our work is very different from those developing languages for constructing design systems, such as DSPL (Brown and Chandrasekaran, 1989) and Edesyn (Maher, 1988). These languages provide programming constructs, at a lower level of sophistication than mechanisms, to capture design knowledge for specific tasks in specific domains. In fact, once a developer has used one of these languages to construct a system, we would attempt to analyze it within the context of our framework.

6. SUMMARY AND CONCLUSIONS

Research on design tools has generated a large number of systems. Through analysis of these systems, we can begin to more thoroughly understand the design process. In this paper, we have outlined a knowledge-level analysis method. This method identifies the key elements of design tools: the PSMs and other related control knowledge, and the mechanisms underpinning those methods. In addition, we have presented a task characterization scheme based on subtask interaction that helps to classify the generality of various design tools. Thus, the difficulty of the task solved by a particular tool can be determined on qualitative features, not abstract measures of skill or ingenuity.

The analysis method and task characterization scheme provide a framework for describing design tools at the knowledge level, thereby free of implementation details. Such

descriptions facilitate both the understanding of individual systems and comparisons between systems. Knowledge-level descriptions have been used effectively for this purpose in other disciplines, and would, we believe, benefit the design research community.

The results of the analysis point to interesting differences and similarities between the tools studied. The PSMs used appear to be relatively similar for a particular task type, but vary between different types. Perhaps a small set of methods could be developed to cover configuration design for a particular task type. In addition, the mechanisms employed have remarkable similarity among tools, and do not seem to vary with respect to task type. Thus, given a sufficiently large set of mechanisms, a wide set of PSMs could be produced.

We are continuing this research. Our intention is to develop both a set of methods for each task type, and a library of design mechanisms. From these, we hope to be able to rapidly configure design tools for various domains. We are also extending this work to include knowledge-acquisition tools.

Acknowledgements. We would like to thank the members of the Knowledge-based Design Group at the University of Michigan, whose members include Dan Haworth, Pierrette Zouein, Viju Menon, Jay Runkel, and Manjot Sandhu, for their very helpful contributions to our discussions. In addition, we thank Georg Klinker for getting us to think about our systems along the lines described here. We would also like to thank John McDermott, Mark Musen, and the reviewers of an earlier draft of this paper, who made helpful comments.

REFERENCES

- Birmingham, W. P., Gupta, A. P. and Siewiorek, D. P. (1989a). *A General Synthesis Engine: Making MICON Domain-Independent*, Technical Report, EDRC, Carnegie Mellon University, EDRC-18-07-89.
- Birmingham, W. P., Gupta, A. P., and Siewiorek, D. P. (1989b). The Micon System for Computer Design, *IEEE Micro*, 9(5): 61-67.
- Brown, D.C., Chandrasekaran, B. (1985). *Expert Systems for a Class of Mechanical Design Activity*, IFIP, Elsevier Publishers.
- Brown, D.C., Chandrasekaran, B. (1986). Knowledge and Control for a Mechanical Design Expert System, *IEEE Computer*, 19(7): 92-100.
- Brown, D.C., Chandrasekaran, B. (1989). *Design Problem Solving*, Pitman Publishing, London.
- Chandrasekaran, B. (1986). Generic Tasks in Knowledge-based Reasoning: High Level Building Blocks for Expert System Design, *IEEE Expert*, Fall, 23-30.
- Chandrasekaran, B. (1988). *Design, An Information Processing-level Analysis*, Technical Report, The Ohio State University, Department of Computer and Information Science, Laboratory for Artificial Intelligence Research, Revised January 1988.
- Chandrasekaran, B. (1990). Design Problem Solving: A Task Analysis, *AI Magazine*, 11(4): 59-71.

- Clancey, W.J. (1985). Heuristic Classification, *Artificial Intelligence*, 27 (3) 289-350.
- Constantine, L., Yourdon, E. (1979). *Structured Design*, Prentice-Hall, Englewood Cliffs, NJ.
- Hayes-Roth, B., Hewett, M., Johnson, M.V.Jr., Garvey, A. (1988). *Accord: A Framework for a Class of Design Tasks*, Stanford University, Department of Computer Science, Knowledge Systems Laboratory, Report No. KSL 88-19, March.
- Johnson, M.V. Jr., Hayes-Roth, B. (1988). *Learning to Solve Problems by Analogy*, Stanford University, Department of Computer Science, Knowledge Systems Laboratory, Report No. KSL-88-01.
- Klinker, G., Bhola, C., Dallemane, G., Marques, D., McDermott, J. (1990). Usable and Reusable Programming Constructs, *Proceedings of the 5th Knowledge Acquisition Workshop*, AAAI.
- Langrana, N. A., Mitchell, T. M., Ramachandran, N. (1986). *Progress Towards a Knowledge-based Aid for Mechanical Design*, Symposium on Integrated and Intelligent Manufacturing, The American Society of Manufacturing Engineers.
- Maher, M. L. (1988). Engineering Design Synthesis: A Domain-Independent Representation, *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 1(3): 207-213.
- Marcus, S., Stout, J., McDermott, J. (1988). VT: An Expert Elevator Designer That Uses Knowledge-based Backtracking, *AI Magazine*, 9(1): 95-112.
- McDermott, J. (1988). Preliminary Steps Toward a Taxonomy of Problem-solving Methods, in *Automating Knowledge Acquisition for Expert Systems*, ed. S. Marcus, Kluwer Academic Publishers, Boston, MA.
- Mittal, S., Dym, C.L., Morjaria, M. (1986). Pride: An Expert System for the Design of Paper Handling Systems, *IEEE Computer*, 19(7): 102-114.
- Mittal, S., Frayman, F. (1989). Towards a Generic Model of Configuration Tasks, *Proceedings of the 11th IJCAI*, 1395-1401.
- Mostow, J. (1985). Toward Better Models of the Design Process, *AI Magazine*, Spring, 44-63.
- Musen, M. (1989). *Automated Generation of Model-based Knowledge-Acquisition Tools*, Morgan Kaufmann Publishers, San Mateo, CA.
- Newell, A. (1981). The Knowledge Level, *AI Magazine*, 2(2): 1-20 and 33.
- Steels, L. (1990). Components of Expertise, *AI Magazine*, 11(2): 28-49.
- Tommelein, I.D., Levitt, R.E., Hayes-Roth, B., Confrey, T. (1991). SightPlan Experiments: Alternate Strategies for Site Layout Design, *ASCE Journal of Computing in Civil Engineering*, 5(1): 42-63.
- Tong, C. (1987). Toward an Engineering Science of Knowledge-based Design, *Artificial Intelligence in Engineering*, 2(3): 133-166.
- Treur, J. (1989). A Logical Analysis of Design Tasks for Expert Systems, *International Journal of Expert Systems*, 2(2): 233-253.
- van Melle, W. (1980). *A Domain-independent Production-Rule System for Consultation Programs*, Ph. D. Dissertation, Stanford University, Department of Computer Science, Report No. STAN-CS-820, June.

Author index

- Agogino, A. M. 815
Arakawa, M. 839
Babin, B. A. 249
Balachandran, M. 757
Balkany, A. 921
Bañares-Alcántara, R. 1
Birmingham, W. P. 921
Bond, A. H. 339
Bradley, S. R. 815
Brown, D. C. 323
Buck, P. 583
Cagan, J. 665
Carter, I. M. 859
Chase, S. C. 339
Chapman, A. 897
Clarke, B. 583
Colajinni, B. 23
Colgan, L. 211, 543
Coupal, C. M. 95
Coyne, R. D. 49
Coyne, R. F. 303
Cugini, U. 407
Dasgupta, S. 447
de Grassi, M. 23
di Manzo, M. 23
Eastman, C. M. 339
Falcidieno, B. 407
Faltings, B. 467, 645
Finger, S. 737
Fischer, G. 191
Floyd, C. 583
Gan, M. 785
Ganeshan, R. 737
Garcia, A. C. B. 723
Garrett, J. 737
Gero, J. S. 525, 757
Giannini, F. 407
Greef, A. 79
Gupta, A. 623
Haroud, D. 467
Held, H.-J. 883
Hodgkin, E. 583
Hornberger, L. 563
Horváth, I. 703
Howard, H. C. 151, 723
Ishii, K. 563
Iwai, S. 281
Jäger, K.-W. 883
Kannapan, S. M. 683
Katai, O. 281
Kawakami, H. 281
Kemp, G. J. L. 387
Kratz, N. 883
Krige, G. J. 801
Lawson, B. R. 231
Leckie, C. 603
Liu, H. 603
Liu, X. 785
Lloyd, G. 583
Logan, B. 423
Loganantharaj, R. 249
MacCallum, K. J. 859
MacKellar, B. K. 115
Maher, M. L. 137
Marshek, K. M. 683
Matwin, S. 269
Millington, K. 423
Modi, A. 303
Mussio, P. 407
Myers, L. 897
Nakakoji, K. 191
Naticchia, B. 23
Newton, S. 49
Nguyen, G. T. 367
O'Grady, P. 79
Oppacher, F. 269
O'Reilly, U.-M. 269
Ozel, F. 115
Patel, S. 447
Pelletier, B. 269
Protti, M. 407
Pohl, J. 897
Poulter, K. 583
Pu, P. 171
Purcell, A. T. 525
Rankin, P. 211, 623
Reich, Y. 303
Reschberger, M. 171
Rieu, D. 367
Rosenman, M. A. 757
Rowles, C. 603
Ryu, J. 231
Sawaragi, T. 281
Schneider, M. 883
Scott, P. J. 231
Smith, I. F. C. 467
Smithers, T. 423, 583
Sorenson, P. G. 95
Spence, R. 211, 543
Spillane, M. B. 323
Steier, D. 303
Subrahmanian, E. 303
Tang, M. X. 583
Tomes, N. 583
Tommelein, I. D. 921
Tremblay, J.-P. 95
Tsang, J. P. 485
Visser, W. 505
Wang, J. 151
Wen, W. 603
Yamakawa, H. 839
Young, R. E. 79

Author electronic addresses

Agogino, A. M. aagogino@tycho.berkeley.edu	Liu, H. h.liu@trl.oz.au
Arakawa, M. +81 3 209 9176	Liu, X. +86 1 256 2768
Bañares-Alcántara, R. rene@chem-eng.edinburgh.ac.uk	Logan, B. brian@aifh.edinburgh.ac.uk
Bradley, S. R. sbradley@tycho.berkeley.edu	Logananthraj, R. logan@cacs.usl.edu
Brown, D. C. dcb@cs.wpi.edu	MacCallum, K. J. ken@cad-centre.strathclyde.ac.uk
Cagan, J. jcagan@globe.edrc.cmu.edu	MacKellar, B. K. bonnie@vienna.njit.edu
Clarke, B. logcam!brc@relay.eu.net	Maher, M. L. mary@archsci.arch.su.oz.au
Colgan, L. lynne@titan.ee.ic.ac.uk	Matwin, S. stan@csi.uottawa.ca
Coupal, C. M. coupa@skorpio.usask.ca	Myers, L. lmyers@polyslo.calpoly.edu
Dasgupta, S. das@cacs.usl.edu	Newton, S. sid@archsci.arch.su.oz.au
Eastman, C. M. cs.ucla.edu!aalto!chuck	Nguyen, G. T. nguyen@imag.fr
Faltings, B. faltings@elma.epfl.ch	O'Grady, P. o.grady@ie.ncsu.edu
Finger, S. sfinger@isl1.ri.cmu.edu	Protti, M. +39 10 51 7801
Fischer, G. gerhard@boulder.colorado.edu	Pu, P. pu@cse1.cse.uconn.edu
Gero, J. S. john@archsci.arch.su.oz.au	Purcell, A. T. terry@archsci.arch.su.oz.au
Gupta, A. gupta@prl.philips.co.uk	Reich, Y. yoram@edrc.cmu.edu
Horváth, I. +36 1 166 6808	Rosenman, M. A. mike@archsci.arch.su.oz.au
Howard, H. C. hch@cive.stanford.edu	Scott, P. J. pc1pjs@primea.sheffield.ac.uk
Ishii, K. ishii+@osu.edu	Smith, I. F. C. smith@elgc.epfl.ch
Kannapan, S. M. kannap@emx.utexas.edu	Tommelein, I. D. irist@caen.engin.umich.edu
Katai, O. d52754@jpnkudpc.bitnet	Tsang, J. P. tsang@aar.alcatel-alsthom.fr
Kemp, G. J. L. gilk@cs.abdn.ac.uk	Visser, W. vissei@psycho.inria.fr
Krige, G. J. +27 11 403 1926	Young, R. E. young@ie.ncsu.edu