Environment and Planning B: Planning and Design, 1985, volume 12, pages 419-442

Design knowledge and context

R D Coyne, J S Gero
Computer Applications Research Unit, Department of Architectural Science, University of Sydney, NSW 2006, Australia
Received 2 January 1985; in revised form 25 March 1985

Abstract. Design is regarded here as the generation of a language of designs from a grammar. We consider the application of context sensitive grammars to design. Meaning in design is equated with purposeful design activity where mechanisms are devised for finding states which match goals. Regression is discussed as a tool for goal satisfaction using context sensitive grammars. Some of the complexities of implementation introduced by contextual considerations are handled by hierarchical planning. The ideas are demonstrated by means of an example where the artifact is a simple building form which is to fit into a site context. The example has been implemented in PROLOG.

Introduction

Many of the terms we use to describe design and design processes are related to the way we talk about language. The notion of semantics or meaning is of particular interest. In natural language, meaning elevates sentences from mere strings of words to mechanisms for communicating ideas. In design, semantics elevate a configuration of building elements to something with purpose. Another linguistic concept, that of 'context', is closely bound up with 'meaning'. An understanding of surrounding events and accompanying dialogue is vital if our utterances are to mean what we want them to mean. So too in design it is necessary to take account of the context within which an artifact and its various components and subcomponents are to function if the design is to achieve its intended purpose.

We attempt here to formalise some ideas about context in design, taking the linguistic idea of context. This will involve us in a discussion of mechanisms for handling the computational complexity resulting from the consideration of complex contextual constraints.

State-space search

One important view of problem solving is that the act of discovering a solution constitutes a search through a space of states (Newell and Simon, 1972). A state is a representation of the world we are dealing with at any moment during the problem-solving process. In design this world may be seen as the artifact undergoing processes of change during its creation, such as a block of marble undergoing transformation at the hands of a sculptor, or a sketch being manipulated on an architect's drawing board. One state is transformed into another by means of operators. A powerful type of operator is the transformation rule, and we can view certain design knowledge as a corpus of such rules. A transformation rule can be represented as a pair of patterns: if the first pattern is recognised as forming part of the current state description, it can be replaced by the second pattern. The design space is therefore that collection of states which would result from the implementation of all applicable transformation rules acting on all possible states.

In the case of conventional architectural sketches we are used to dealing with an abstraction of the real world. Abstractions provide obvious economic advantages

over the manipulation of states in the real world. Design states are represented in terms of line segments, shapes, colours, labels, etc and with these we build up complex descriptions of objects. Most of what the designer understands about states is implicit rather than explicit. The lines assume properties other than those of graphite on paper, but become walls with three-dimensional properties and these bound spaces which are rooms performing particular functions. So the description of states is much richer than is immediately evident from an inspection of the representation. Transformation rules act on an abstract description of the designer's world.

Design rules as grammar

There are direct parallels between the sort of knowledge by which we operate on states and the linguistic notion of a 'grammar' as described by Chomsky (1963). A phrase structure grammar is described as consisting of a vocabulary and a set of productions. The vocabulary is composed of two types of elements: terminal symbols and nonterminal symbols. Nonterminals are symbols in the language which can be operated on by production rules to create other vocabulary elements, and terminal symbols are those which cannot be transformed. The productions cause transformations to be made. Beginning with a start symbol (which is a nonterminal), the successive application of productions produces strings of terminal and nonterminal symbols. When a string produced in this way consists entirely of terminal symbols we have a sentence. In natural languages, nonterminal symbols are the syntactic categories: sentence (S), noun phrase (NP), verb phrase (VP), verb (V), noun (N), determinant (DET), etc, and the terminals are words in the language. Examples of production rules are:

1 $S \rightarrow NP\ VP$
2 $VP \rightarrow V\ NP$
3 $NP \rightarrow N$
4 $NP \rightarrow DET\ N$
5 $V \rightarrow$ *eats*
6 $N \rightarrow$ *John*
7 $N \rightarrow$ *apple*
8 $DET \rightarrow$ *the*

Beginning with the start symbol, S, it is possible to generate simple sentences by applying different rules. For example, the rule sequence, 1, 3, 2, 6, 5, 4, 8, 7, produces the sentence: John eats the apple.

Conversely, with such a grammar, we can parse a sentence to see if it conforms to the rules of the grammar. The generative aspect of grammars is of most interest to us here, however. As grammars deal in symbols it is possible to construct a language from a vocabulary of any type of symbols from any domain. In the state-space model of design we have a grammar of transformation rules. The vocabulary consists of collections of literals or attributes which describe the world. Terminal symbols are those attributes which cannot be transformed, and we reach a termination point in the design process when the only attributes which describe the world are terminals. The state-space model also requires that we begin with an initial symbol or set of symbols. The solution space of the design problem therefore becomes a 'universe of discourse' as defined by the grammar. This analogy between language and design has been discussed extensively by Stiny (1975), Gips (1975), and others interested in language and semiotics in architecture (Broadbent et al, 1980).

Context

Chomsky (1963) defines a hierarchy of language classes according to the form of the grammar rules. The four types of grammars are numbered from 0 to 3, proceeding from the most general class to the most restrictive. The ordering also indicates an increase in computational difficulty.

A type 3 grammar (regular grammar) is the most restrictive form of grammar. Rules must take the form:

$$X \rightarrow aY \quad \text{or} \quad X \rightarrow a,$$

where X and Y are strings of nonterminal symbols and a is a single terminal. Nonterminals are taken one at a time and transformed into another nonterminal plus a terminal or into a terminal on its own. No account is taken of the context of the symbol on the left-hand side of the rule.

A type 2 grammar is also one in which context is ignored. The only restriction on the rules is that the left-hand side must be a single nonterminal:

$$X \rightarrow aY, \quad X \rightarrow Y, \quad \text{or} \quad X \rightarrow a,$$

where Y and a are arbitrary strings of nonterminal and terminal symbols respectively.

A type 1 grammar can take the context of a symbol into consideration. It is of the general form:

$$uXv \rightarrow uYv,$$

where X is a single nonterminal symbol, u and v are arbitrary strings of symbols, and Y is a string of symbols (comprised either of terminals or of nonterminals). This can be interpreted as: X may be rewritten as Y in the context of u and v. The context of course is unchanged. These grammars are also called 'context sensitive grammars'.

A type 0 grammar is the most general form. Any string of symbols can be transformed into any other string:

$$X \rightarrow Y.$$

These categories are of interest when classifying different types of languages. Computer programming languages, for example, are generally of type 2. The type of grammar has a bearing on the ease with which we are able to conduct parsing procedures. It is generally recognised that in natural language we are dealing with context sensitive grammars, that is, type 1 grammars, although we often assume they are of type 2 for ease of parsing. In the linguistic example above, the rules were those of a context free grammar. In design, a context sensitive grammar appears to afford the greatest scope for capturing design knowledge of the type that interests us.

Context in design

We can describe a transformation rule belonging to a context sensitive design grammar as one which operates on a particular pattern in the state description by transforming it into another pattern, provided other patterns are present. Those other patterns constitute the context for that rule and are not changed by the rule. They must be present if the rule is to be fired. A context sensitive grammar rule therefore takes the general form:

$$uXv \rightarrow uYv,$$

where X and Y are lists of attributes or literals contained in the current state description, and u and v are lists of attributes which remain unchanged by the rule. It is apparent that although certain literals may constitute the context for one rule they may just as readily be transformed by other rules. There is nothing in this definition which states that contextual attributes must remain so for all rules.

Context and meaning

Meaning is instilled in a design when we intend it to achieve some purpose. An understanding of context is important in the realisation of intentions. The same design may achieve apparently different intentions in different contexts, just as the same sentence may mean different things in different contexts. The sentence 'pass the jam' spoken at the dinner table has a different meaning than when spoken to a motorist approaching a traffic jam. Stating the context not only solves problems of ambiguity, but also enables the interpretation of nuances in conversation. For example, after a scathing argument about the merits of the marmalade, 'pass the jam' may be interpreted as an expression of contempt. Something similar applies to design. The intention to build a thermally efficient building may be translated as 'massive one metre thick walls' in one context (for example, an arid desert) and as 'thin galvanised iron cladding' in another (for example, the tropics). Context determines how our intentions are translated into actuality. 'Meaningful' designs are achieved, therefore, when certain goals are achieved within a particular context. The context of a design becomes the conditions under which goals are to be satisfied.

Automated design generation

With an automated system of design generation the satisfaction of goals within a context must be taken into account. Goals are essentially attributes which the final design is to exhibit. A design state constitutes a design if it possesses those attributes contained in a goal set. This involves the evaluation of design states using the goals as evaluators. The goal facts can be matched directly against the literals which make up the state description or they can be inferred from the state description. If no such state is reached and no more transformation rules can be applied (that is, the state is described only in terms of terminal symbols) then one procedure is to backtrack to an earlier state which afforded a choice of rules. Another transformation rule is selected to produce a different sequence of states. This process of generating states and backtracking when 'dead ends' are reached continues until a state which satisfies the goals is found.

This generate and test model has the drawback that, in practice, design spaces tend to be extremely large, and it is impractical to evaluate each state in terms of the goals before arriving at a solution. A second approach is to consider knowledge about the potential of states to become goal states, and so limit the amount of searching.

Another approach is to see design as a process of working backwards from goals, such that we ask: 'What action is necessary to achieve this goal? This action has certain preconditions. What actions are necessary before these preconditions can be achieved?' And so the process is one of chaining backwards from goals to an initial state. In the process we also build up a description of the whole design. The selection and ordering of operators in this way prior to actually implementing them is termed 'planning'. The implementational advantages of this approach within design have been discussed elsewhere (Coyne and Gero, 1985).

Knowing the context assists in determining the applicability of a rule for meeting a particular goal or subgoal. A context sensitive rule takes the general form:

$$a_1, \ldots a_i \ldots, a_n, c_1, \ldots c_i \ldots, c_n \rightarrow b_1, \ldots b_i \ldots, b_n, c_1, \ldots c_i \ldots, c_n,$$

where a_i, $i = 1, \ldots, n$, are literals or attributes describing the current state of the world; c_i, $i = 1, \ldots, n$, are literals describing the context; and b_i, $i = 1, \ldots, n$, are the new state descriptors which are to replace the a_i. b_i can alternatively be viewed as a goal. If the conditions on the left-hand side of the rule can be met then the goal can be satisfied. If the context changes such that the c_i are not true in the current state description then the rule will be inapplicable as a means of achieving the goal.

The satisfaction of contextual constraints need not be different implementationally from the manipulation of any other state descriptors by transformation rules. Contextual attributes are simply those that appear on both sides of a rule. When planning by backward chaining contextual constraints are treated in the same way as other literals. The consideration of complicated contextual requirements within a grammar can, however, impose a heavy computational toll on a planning system. Various strategies have been discussed for imposing a higher level of control above an automated planning system. These generally involve augmenting the knowledge contained in the production rules with further knowledge about the domain under consideration. One such strategy is to note that certain attributes in the state description are going to be more influential or critical than others in the formulation of a plan. The more critical facts should be considered first. This is basically the approach taken in the ABSTRIPS planning system (Sacerdoti, 1974). This amounts to discriminating between important information and details in a problem space. It is handled by dealing with the more important factors first, then introducing successive levels of detail. The analogy is often made of planning a car journey between two cities. It is not necessary to 'search' a map of every street and laneway; instead we consider the problem by planning the journey at a coarse level first, that is, we consider the major highways and stopping points, and then resolve the problem at successively finer levels of detail. We consider this hierarchical approach here as a means of handling the complexities imposed by a contextual grammar.

Figure 1 illustrates the idea of formulating a sequence of design actions (a plan) in a three-level planning system. The three levels constitute subsystems. At level 3 the most important or influential attributes are considered and a sequence of rules to satisfy an input goal set is produced. The plan is passed down to the next level and evaluated taking into account the attributes considered at level 3 plus those considered to be of secondary importance. If it passes the approval of this subsystem it is channeled down to level 1 and evaluated to see if it is an appropriate plan when all attributes are taken into account. If at any level the plan fails then the subsystem at the level above must generate another plan. This is one way in which the idea of planning at different levels can be exploited. A more sophisticated model is that in which a partial or imperfect plan is passed down from a higher level and is filled in or 'corrected' at the lower levels. It can be shown that even the approach outlined in figure 1 can introduce efficiencies in the planning process, and this approach will be adopted here.

A set of design rules is shown diagrammatically in figure 2. The patterns in the rules are squares each with a label (*a*, *b*, *c*, *d*, *e*) and an orientation. An edge indicated with a dot is the top of a square. This can be regarded as a context sensitive grammar when we consider that there are certain conditions which must be present in the domain before certain rules are applicable, but which are not changed by the rules.

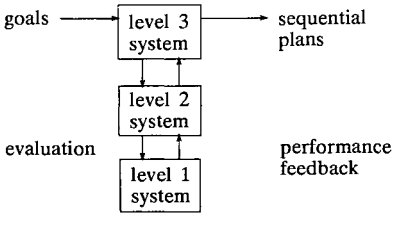

Figure 1. A three-level hierarchical planning system.

This can be seen more clearly if we represent the rules by means of a high-level abstraction using the following form for a rule:

rule_name # preconditions > > consequents,

for example,

rule(2) # [*located*(*a*), *clear*(*a, right*)]
> >
[*located*(*a*), *located*(*c*), *clear*(*c, top*), *clear*(*c, right*), *clear*(*c, left*)].

This states that the fact that *a* is clear on the right can be replaced by the fact that *c* is located and that it is clear on its top, right-hand, and left-hand sides. 'located(*a*)' is a contextual literal as it is a necessary precondition of the rule yet remains unchanged by it.

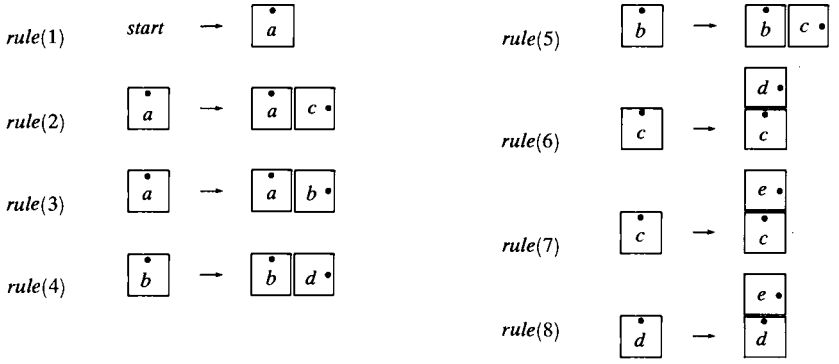

Figure 2. A set of design rules.

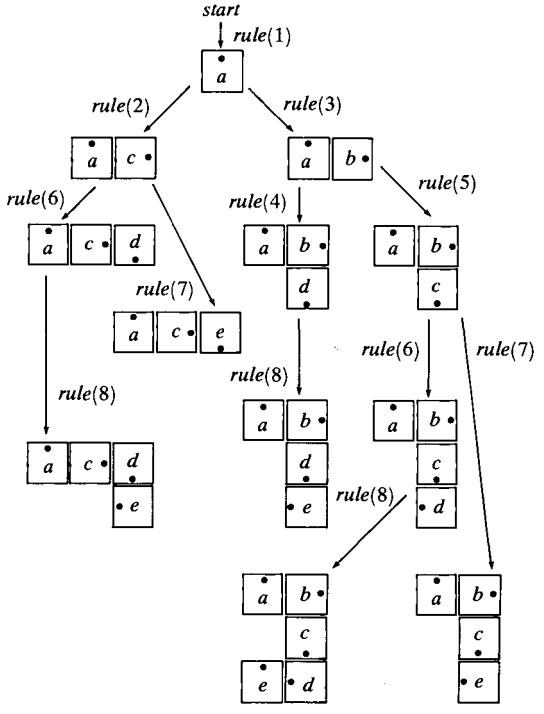

Figure 3. Design space generated by the rules of figure 2.

The design space generated by the rules is shown diagrammatically in figure 3. This network or tree diagram, indicating the various states and the rules that are required to generate them, is also referred to as a 'search space'.

An abstraction of all the rules depicted in figure 2 is as follows:

rule(1) # [*start*]
>>
[*located*(*a*), *clear*(*a*, *top*), *clear*(*a*, *bottom*), *clear*(*a*, *right*), *clear*(*a*, *left*)].

rule(2) # [*located*(*a*), *clear*(*a*, *right*)]
>>
[*located*(*a*), *located*(*c*), *clear*(*c*, *top*), *clear*(*c*, *right*), *clear*(*c*, *left*)].

rule(3) # [*located*(*a*), *clear*(*a*, *right*)]
>>
[*located*(*a*), *located*(*b*), *clear*(*b*, *top*), *clear*(*b*, *right*), *clear*(*b*, *left*)].

rule(4) # [*located*(*b*), *clear*(*b*, *right*)]
>>
[*located*(*b*), *located*(*d*), *clear*(*d*, *top*), *clear*(*d*, *right*), *clear*(*d*, *left*)].

rule(5) # [*located*(*b*), *clear*(*b*, *right*)]
>>
[*located*(*b*), *located*(*c*), *clear*(*c*, *top*), *clear*(*c*, *right*), *clear*(*c*, *left*)].

rule(6) # [*located*(*c*), *clear*(*c*, *top*)]
>>
[*located*(*c*), *located*(*d*), *clear*(*d*, *top*), *clear*(*d*, *bottom*), *clear*(*d*, *left*)].

rule(7) # [*located*(*c*), *clear*(*c*, *top*)]
>>
[*located*(*c*), *located*(*e*), *clear*(*e*, *top*), *clear*(*e*, *bottom*), *clear*(*e*, *left*)].

rule(8) # [*located*(*d*), *clear*(*d*, *top*)]
>>
[*located*(*d*), *located*(*e*), *clear*(*e*, *top*), *clear*(*e*, *bottom*), *clear*(*e*, *left*)].

These rules build up a partial description of states. They do not include geometrical descriptions. The advantage of this type of abstraction is that states can be generated relatively economically. Once the desired sequence of actions for achieving a particular goal state is discovered, it is a relatively simple procedure to run through the same sequence of rules, this time described in more complete terms, to generate geometrical designs.

Regression

The mechanism for satisfying goals by backward chaining, or regression, is more efficient than that of forward chaining (which is a type of generate and test approach) as a method of planning, provided that goals can be stated in the same terms as the descriptions of states. We adopt here some of the ideas behind the STRIPS planning system (Fikes and Nilsson, 1971), which relates to planning sequences of robot actions. The different approaches to planning are also discussed extensively by Nilsson (1982).

To illustrate how contextual requirements can be satisfied by backward chaining we begin by looking at the mechanism for satisfying a single goal by means of the rules in figure 2. We decide that one of the attributes of the final state is 'located(*d*)'. The search tree for satisfying the single goal, located(*d*), is shown in figure 4. The goal is

shown at the top of the tree. Rule(4) will satisfy that goal if there is a rule (or rules) that can satisfy located(b) and clear(b, right). Located(b) can be satisfied by rule(3), the preconditions of which are satisfied by rule(1). The precondition for rule(1) is 'start', which is given as true (indicated by 'T') as it constitutes the initial condition. Now that located(a) can be satisfied, the subgoal, clear(a, right), can be tested against the state description produced by rule(1). As this is true the goal, located(b), is true and clear(b, right) can be tested. This matches the state description generated by rule(1) and rule(3). All the preconditions necessary to achieve the goal, located(d), are satisfied by the rule sequence: rule(1), rule(3), rule(4). So this is one of the plans for achieving the goal: located(d). We explore this approach to the same problem elsewhere (Coyne and Gero, 1985).

A goal or subgoal is satisfied if it can be produced by a rule or if it matches the current state description. In figure 4 it can be seen how the state description is built up with the implementation of each rule. For clarity, states are here shown geometrically, remembering that states are actually described only in terms of the facts in the rules in figure 2. In the case of compound goals, that is, where more than one top-level goal is to be satisfied, we may find it necessary to search unsuccessful solution paths. Where a sequence of rules satisfies one goal but not another, it is necessary to backtrack to investigate alternative sequences of rules. Backtracking also enables us to generate alternative rule sequences to satisfy the same goal. Backtracking is illustrated later in this paper.

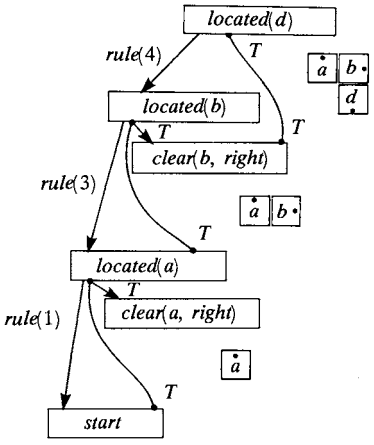

Figure 4. Search graph for the satisfaction of the goal: located(d).

Extended context

Contextual attributes which remain constant throughout a particular design process but which can change from one 'project' to the next are of particular interest. This is illustrated by means of an example. We represent the context in this case as an envelope within which the configuration of squares produced by the rule set must fit. The 'artifact' can assume any spatial characteristics, provided it does not extend beyond the boundaries of this envelope. We further abstract this context by describing it as a set of candidate squares on a gridded world. This envelope is shown diagrammatically in figure 5. We represent this information as a fact in predicate calculus.

envelope([[4, 6], [3, 5], [4, 5], [2, 4], [3, 4], [4, 4], [5, 4], [2, 3], [3, 3], [4, 3], [5, 3], [6, 3], [3, 2], [4, 2], [4, 1]]).

The predicate has one argument, which is simply a list of cells contained within the envelope, each cell identified by an address giving its x and y grid coordinates. Coordinate pairs are grouped together in square brackets. The centre of each square in the rules is to coincide with the centre point of a grid cell in the context.

We reformulate the design rules so that they are sensitive to this context: for example, rule(2) can be fired if the object a is located and if it is clear on its right-hand side. But to fit the context certain conditions about the object's location and orientation must also be satisfied. The cell to the right of a must be within the envelope, that is, if the cell location of a is $[X, Y]$ and the cell to the right of this is $[X1, Y1]$ then $[X1, Y1]$ must be a member of the list L of cells contained within the envelope.

Finding the address of the cell to the right of a introduces a slight complication, as the design space in figure 3 shows that an object can be at almost any orientation. The 'address' of the cell to the right of a can be deduced from the address and orientation of a. It is therefore desirable to maintain facts about the orientation of objects in the current state description and to provide knowledge about how to determine the location of the cell to the right of an object from its position and its orientation. The knowledge for finding the cell (and its orientation) to the right of a oriented to the north, is given by the PROLOG rule:

```
right_cell(north, east, [X, Y], [X1, Y]) :— X1 is X +1.
```

where the first argument of the predicate, right_cell, is the orientation of a, the second argument is the orientation of the object in the neighbouring cell, the third argument is the address of the cell containing a, and $[X1, Y]$ is the address of the neighbouring cell. The x coordinate of the neighbouring cell is found by adding 1 to the value of the x coordinate of the a cell. There are similar inference rules for finding neighbouring cells under different conditions.

The preconditions for the firing of rule(2), taking account of this context, are therefore given formally by the following list of literals:

[*located*(*a*),
clear(*a*, *right*),
cell(*a*, [*X*, *Y*]),
orientation(*a*, *D*1),
right_cell(*D*1, *D*2, [*X*, *Y*], [*X*1, *Y*1]),
envelope(*L*),
member([*X*1, *Y*1], *L*)].

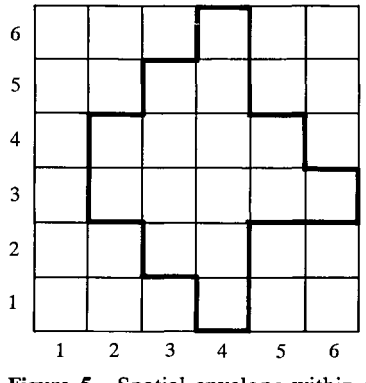

Figure 5. Spatial envelope within a 'gridded world'.

The first three literals match directly the current state description. The last four literals are contextual. In this case the last three are 'evaluatable functions' or facts about the world which must be inferred from other facts. In this implementation such attributes are unchanged by the transformation rule and they are not stored as part of the current state description. The consequent of this rule is given by the following list of facts; these are asserted as part of the current state description:

[*located*(*a*),
located(*c*),
clear(*c*, *top*),
clear(*c*, *right*),
clear(*c*, *left*),
cell(*c*, [*X*1, *Y*1]),
orientation(*c*, *D*2)].

The values of the variables in the consequent list are the same as those instantiated in the preconditions list. The following shows the rules of figure 2, such that the antecedents and the consequents of the rules refer to facts about the context:

rule(1) # [*start*, *envelope*(*L*), *member*([*X*, *Y*], *L*)]
>>
[*located*(*a*), *clear*(*a*, *top*), *clear*(*a*, *bottom*), *clear*(*a*, *right*), *cell*(*a*, [*X*, *Y*]), *orientation*(*a*, *north*)].

rule(2) # [*located*(*a*), *clear*(*a*, *right*), *cell*(*a*, [*X*, *Y*]), *orientation*(*a*, *D*1), *right_cell*(*D*1, *D*2, [*X*, *Y*], [*X*1, *Y*1]), *envelope*(*L*), *member*([*X*1, *Y*1], *L*)]
>>
[*located*(*a*), *located*(*c*), *clear*(*c*, *top*), *clear*(*c*, *right*), *clear*(*c*, *left*), *cell*(*c*, [*X*1, *Y*1]), *orientation*(*c*, *D*2)].

rule(3) # [*located*(*a*), *clear*(*a*, *right*), *cell*(*a*, [*X*, *Y*]), *orientation*(*a*, *D*1), *right_cell*(*D*1, *D*2, [*X*, *Y*], [*X*1, *Y*1]), *envelope*(*L*), *member*([*X*1, *Y*1], *L*)]
>>
[*located*(*a*), *located*(*b*), *clear*(*b*, *top*), *clear*(*b*, *right*), *clear*(*b*, *left*), *cell*(*b*, [*X*1, *Y*1]), *orientation*(*b*, *D*2)].

rule(4) # [*located*(*b*), *clear*(*b*, *right*), *cell*(*b*, [*X*, *Y*]), *orientation*(*b*, *D*1), *right_cell*(*D*1, *D*2, [*X*, *Y*], [*X*1, *Y*1]), *envelope*(*L*), *member*([*X*1, *Y*1], *L*)]
>>
[*located*(*b*), *located*(*d*), *clear*(*d*, *top*), *clear*(*d*, *right*), *clear*(*d*, *left*), *cell*(*d*, [*X*1, *Y*1]), *orientation*(*d*, *D*2)].

rule(5) # [*located*(*b*), *clear*(*b*, *right*), *cell*(*b*, [*X*, *Y*]), *orientation*(*b*, *D*1), *right_cell*(*D*1, *D*2, [*X*, *Y*], [*X*1, *Y*1]), *envelope*(*L*), *member*([*X*1, *Y*1], *L*)]
>>
[*located*(*b*), *located*(*c*), *clear*(*c*, *top*), *clear*(*c*, *right*), *clear*(*c*, *left*), *cell*(*c*, [*X*1, *Y*1]), *orientation*(*c*, *D*2)].

rule(6) # [*located*(*c*), *clear*(*c*, *top*), *cell*(*c*, [*X*, *Y*]), *orientation*(*c*, *D*1), *top_cell*(*D*1, *D*2, [*X*, *Y*], [*X*1, *Y*1]), *envelope*(*L*), *member*([*X*1, *Y*1], *L*)]
>>
[*located*(*c*), *located*(*d*), *clear*(*d*, *top*), *clear*(*d*, *bottom*), *clear*(*d*, *left*), *cell*(*d*, [*X*1, *Y*1]), *orientation*(*d*, *D*2)].

rule(7) # [*located*(*c*), *clear*(*c*, *top*), *cell*(*c*, [*X*, *Y*]), *orientation*(*c*, *D*1), *top_cell*(*D*1, *D*2, [*X*, *Y*], [*X*1, *Y*1]), *envelope*(*L*), *member*([*X*1, *Y*1], *L*)] >> [*located*(*c*), *located*(*e*), *clear*(*e*, *top*), *clear*(*e*, *bottom*), *clear*(*e*, *left*), *cell*(*e*, [*X*1, *Y*1]), *orientation*(*e*, *D*2)].

rule(8) # [*located*(*d*), *clear*(*d*, *top*), *cell*(*d*, [*X*, *Y*]), *orientation*(*d*, *D*1), *top_cell*(*D*1, *D*2, [*X*, *Y*], [*X*1, *Y*1]), *envelope*(*L*), *member*([*X*1, *Y*1], *L*)] >> [*located*(*d*), *located*(*e*), *clear*(*e*, *top*), *clear*(*e*, *bottom*), *clear*(*e*, *left*), *cell*(*e*, [*X*1, *Y*1]), *orientation*(*e*, *D*2)].

Figure 6 shows the appropriate search graph for satisfying the goal: located(*d*). It is similar to figure 4, except that there are now several more subgoals to be satisfied at each level in the tree, and these subgoals contain variables. The goal, located(*d*), can be satisfied by chaining backwards through rule(4), rule(3), and rule(1). Rule(1) requires as its precondition the fact 'start' and the existence of a cell with the address [*X*, *Y*] which is a member of the envelope list *L*. The first cell to satisfy this condition is

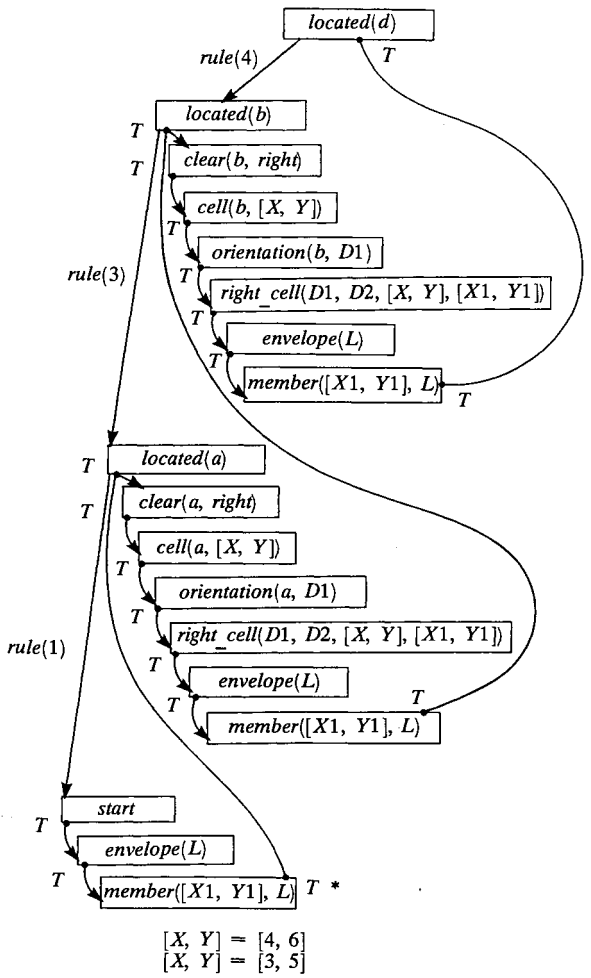

Figure 6. Search graph for satisfying the goal: located(*d*), using the rules of figure 2, taking account of the spatial context.

the first member of the list, that is, [4, 6]. So all the preconditions of rule(1) are satisfied and the values of *X and Y* are instantiated to 4 and 6, respectively. These values form part of the current state description along with the consequents produced by rule(1). The other subgoals forming the preconditions to rule(3) are evaluated against the current state descriptions as follows:

clear(*a, right*)	*TRUE*
cell(*a*, [4, 6])	*TRUE*
orientation(*a, north*)	*TRUE*
right_cell(*north, east*, [4, 6], [5, 6])	*TRUE*
envelope([[4, 6], [3, 5], [4, 5], [2, 4], [3, 4], [4, 4], [5, 4], [2, 3], [3, 3], [4, 3], [5, 3], [6, 3], [3, 2], [4, 2], [4, 1]])	*TRUE*
member([5, 6], [[4, 6], [3, 5], [4, 5], [2, 4], [3, 4], [4, 4], [5, 4], [2, 3], [3, 3], [4, 3], [5, 3], [6, 3], [3, 2], [4, 2], [4, 1]])	*FAIL*

The fail causes backtracking to the last goal that could be resatisfied in some other way. This happens to be the subgoal indicated with a * in figure 6. The next member of the list *L* is the element [3, 5], and so we chain forwards again, this time with *X* and *Y* instantiated to 3 and 5, respectively. This time all the preconditions of rule(3) are satisfied:

clear(*a, right*)	*TRUE*
cell(*a*, [3, 5])	*TRUE*
orientation(*a, north*)	*TRUE*
right_cell(*north, east*, [3, 5], [4, 5])	*TRUE*
envelope([[4, 6], [3, 5], [4, 5], [2, 4], [3, 4], [4, 4], [5, 4], [2, 3], [3, 3], [4, 3], [5, 3], [6, 3], [3, 2], [4, 2], [4, 1]])	*TRUE*
member([4, 5], [[4, 6], [3, 5], [4, 5], [2, 4], [3, 4], [4, 4], [5, 4], [2, 3], [3, 3], [4, 3], [5, 3], [6, 3], [3, 2], [4, 2], [4, 1]])	*TRUE*

All the preconditions of rule(4) are also satisfied in a similar manner. The final state is represented diagrammatically in figure 7. Further solutions can be generated by subsequent backtracking.

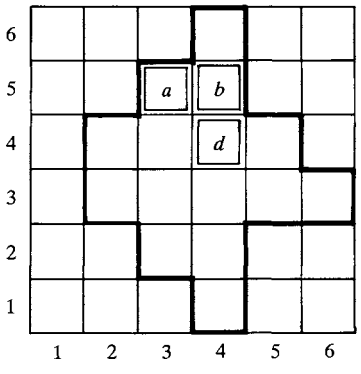

Figure 7. Configuration of objects which satisfies the goal: located(*d*). This is the first of several solutions. The rest are discovered by backtracking.

Controlling regression

In this case the approach to planning described above results in a procedure similar to that a designer might adopt. Conceptually, this amounts to generating a configuration of objects, then moving the configuration about until it fits within the envelope. If that configuration will not fit, then an attempt is made to generate another configuration.

Although this mechanism has some intuitive appeal, the search procedure can become extremely inefficient when there is more than one goal to be satisfied at the top of the search tree. This is illustrated in figures 8, 9, and 10.

The goals are: located(d) and located(c). Rule(4) satisfies the first goal. The first precondition of rule(4) is: located(b). This can be satisfied by rule(3), which has as its first precondition: located(a). This can be satisfied by rule(1), which has as its first precondition: start. As 'start' is the initial condition it is satisfied. The other preconditions of rule(1) are also satisfied, with X and Y instantiated to 4 and 6, respectively. The other preconditions of rule(3) are now also tested. The process is the same as that described for figure 6, and involves backtracking until the configuration and positions shown in figure 7 are generated. A problem arises when we attempt to satisfy the second goal: located(c). This can be satisfied by rule(2), provided its preconditions can be met. The second precondition of rule(2) does not match the current state description and so fails. Backtracking proceeds to the last goal or subgoal that could be satisfied in an alternative way. This is the goal indicated with a *. Located(c) could also be achieved by rule(5), except that it also fails when an attempt is made to match its second precondition against the current state description. Backtracking now proceeds back to the subgoal indicated (*) in figure 9. This can be resatisfied by the selection of an alternative cell from the envelope list L. The procedure will continue as in figure 8, again failing inevitably on the subgoal: clear(a, right).

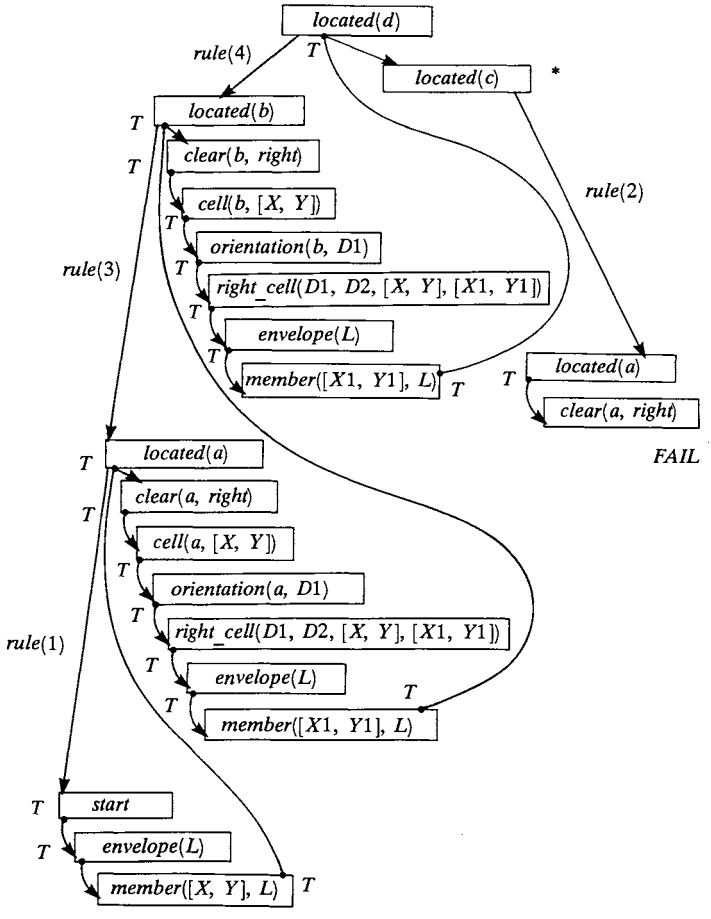

Figure 8. Search graph for satisfying the goals: located(d) and located(c). The search results in failure at the subgoal: clear(a, right).

When all the cells have been 'searched', backtracking returns us to the goal, located(d), and a new rule sequence is generated, as shown in figure 9. Again, each cell is considered in turn until the rule sequence is validated or the list exhausted. This involves an unnecessary amount of searching. It should not be necessary to investigate every cell in the envelope to try and satisfy the goals. It is not the inappropriate selection of cells which is causing failure, but the rule sequence itself. It would fail irrespective of the context. This would become an even greater problem with a larger envelope. It would be more efficient, therefore, first to find an appropriate sequence of rules which will generate a configuration of objects, without taking the context into account, and then to adjust the position of the configuration to fit the context.

This requires the addition of some knowledge about the importance of different attributes. In this case we may decide that facts relating to the configuration of objects are more important than those which relate to context. Furthermore, we may decide that the fact that an object is located is more important than whether or not it has a clear side. We therefore formulate a plan of actions, taking into account the most important considerations first. This is the highest level of the hierarchy. We then reconsider the plan also taking into account factors which exhibit the next level of importance. If the plan is satisfactory when those factors are considered, we reconsider the plan at the next level. There are two main ways that a plan can prove unsatisfactory, at any level. There may be gaps in the plan that require 'patching'.

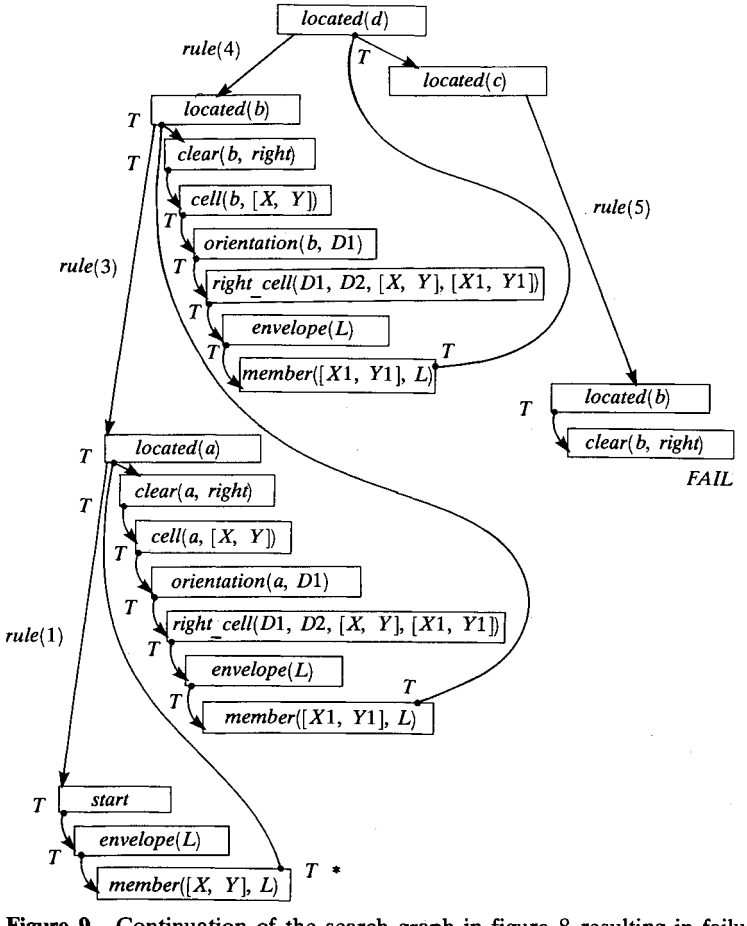

Figure 9. Continuation of the search graph in figure 8 resulting in failure at the subgoal: clear(b, right).

In this case the plan passed down from a higher level to a lower level is a skeletal plan which requires the insertion of certain actions to ensure that factors at the next level of importance receive consideration. We require a mechanism for patching plans that takes account of successive levels of detail. The second way that a plan proves unsatisfactory is if the plan proves totally impossible, with or without patching, at the next level down. In this case the plan could be amended or we could take advantage of backtracking to produce new plans. The second approach is perhaps the simplest and only this one will be considered here. It is less powerful than patching or amending plans, but still produces significant increases in efficiency compared with a non-hierarchical approach.

We adopt a simplified version of the ideas contained in the ABSTRIPS problem solver (Sacerdoti, 1974). The preconditions of each rule are assigned to criticality levels. The higher the level the more critical the precondition. Preconditions and levels are as follows (with variables labeled anonymously):

level 3 *start*, *located*(—)
level 2 *clear*(—, —)
level 1 *cell*(—, —), *orientation*(—,—), *right_cell*(—, —, —, —), *top_cell*(—, —, —, —), *envelope*(—), *member*(—, —)

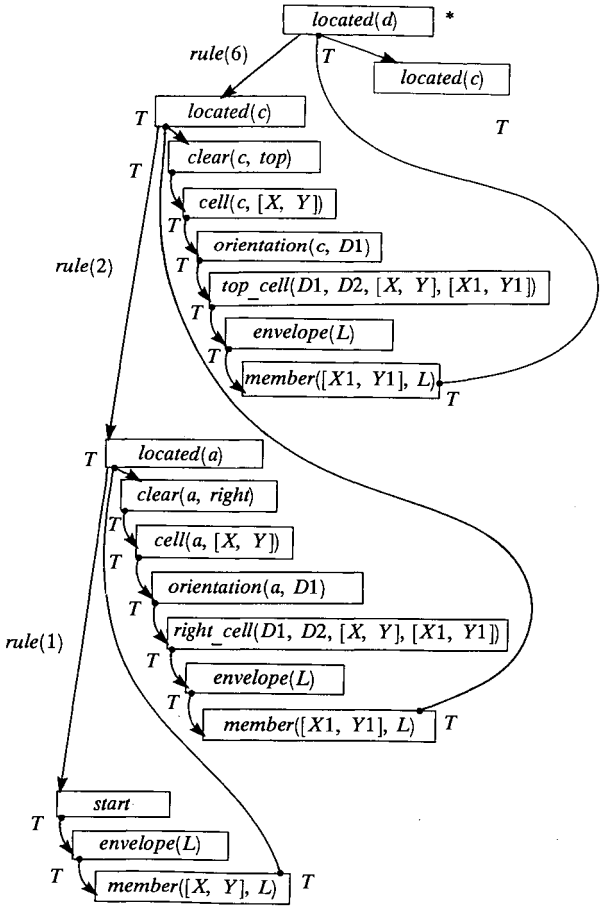

Figure 10. Continuation of the search graph in figure 9 resulting in the satisfaction of both goals: located(d) and located(c).

Figure 11 shows the search graph generated by the two goals, located(d) and located(c), taking into account preconditions at criticality level 3. The plan: rule(1), rule(3), rule(4), rule(2), is produced. This plan is then reconsidered taking into account preconditions at criticality level 2 (figure 12). The plan proves satisfactory until we attempt to satisfy the subgoal: clear(a, right). The plan fails at this level and so we backtrack to the previous level of abstraction. The last goal at criticality level 3 that could be satisfied in a different way is indicated with a * in figure 11. An attempt to resatisfy this goal results in the rule sequence: rule(1), rule(3), rule(4), rule(5), as shown in the search graph in figure 13. This rule sequence is then reconsidered at criticality level 2, and also fails (figure 14). Backtracking to the goal indicated (*) in figure 13 produces the rule sequence: rule(1), rule(2), rule(6) (figure 15—see over).

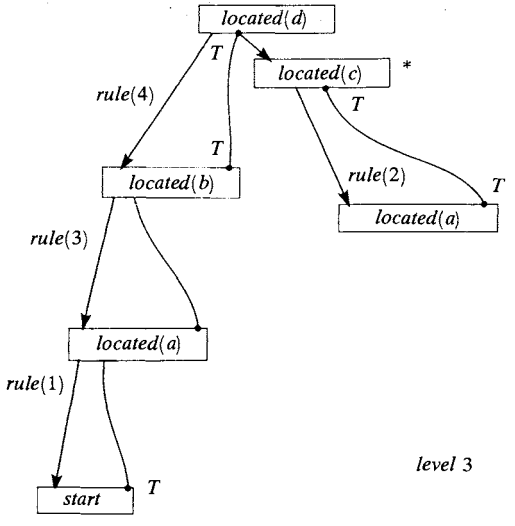

Figure 11. Search graph for the goals, located(d) and located(c), considering preconditions at criticality level 3.

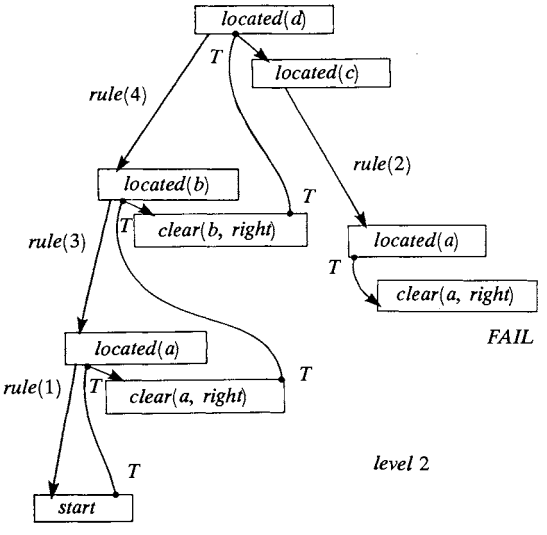

Figure 12. Reconsideration of plan generated in figure 11 by considering preconditions at criticality level 2 as well as level 3.

This plan also proves satisfactory at criticality level 2, as shown in figure 16. The plan is next reconsidered at criticality level 1. The search graph for this level is the same as that shown in figure 10, and it involves a small amount of backtracking to find grid cells which suit the plan and which lie within the envelope.

The efficiency of the planning process in this demonstration could have been enhanced had we considered only two levels of abstraction: that concerned with configuration and that which relates to the context, as this would have involved less backtracking. We selected three levels to demonstrate the mechanism more clearly. The advantage of the hierarchical approach is that it is not necessary to repeatedly search the entire set of grid cells before a plan is proved unsatisfactory. Unsatisfactory plans are detected much earlier, and the amount of searching is reduced. As well as

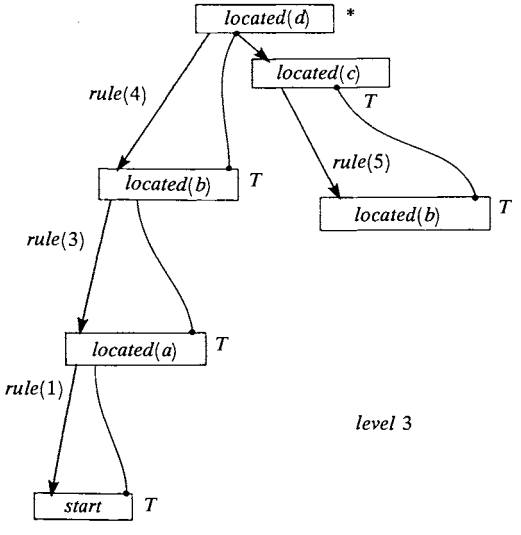

Figure 13. Generation of a new plan by backtracking to criticality level 3.

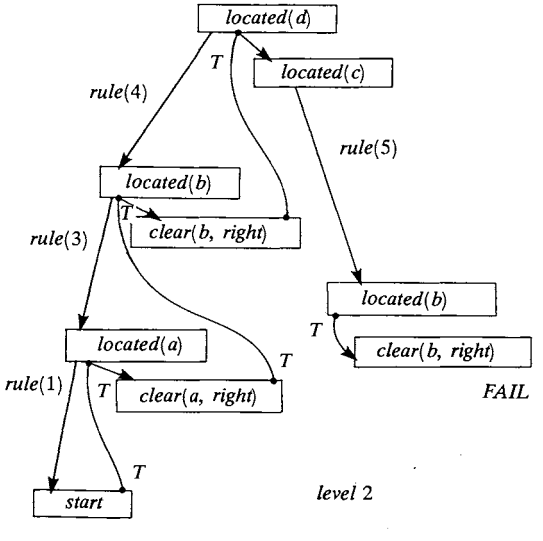

Figure 14. Reconsideration of plan generated in figure 13 by considering preconditions at criticality level 2. This results in failure at the subgoal: clear(*b*, right).

the hierarchical approach to planning suiting our purposes here it has also been shown to be a more efficient mechanism for planning in a general sense (Sacerdoti, 1974).

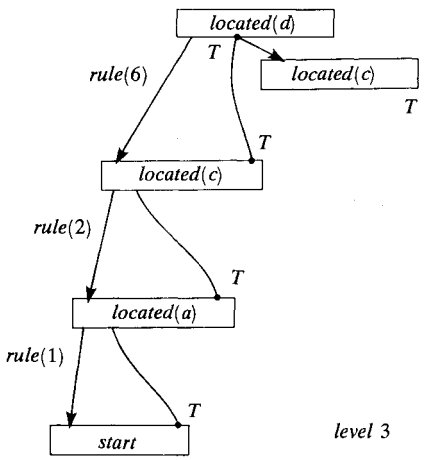

Figure 15. Generation of a new plan by backtracking to criticality level 3.

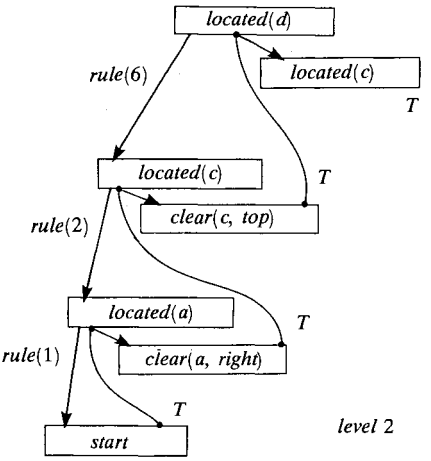

Figure 16. Reconsideration of plan generated in figure 15 by considering preconditions at criticality level 2, resulting in the satisfaction of both goals, located(d) and located(c), at that level.

Reasoning about context

We consider a set of design rules for generating simple building forms from wall elements, windows, and columns. These elements form rooms, each of which has a name and an orientation. A dot indicates the 'top' of a room. The rules are represented diagrammatically in figure 17. It will be noted that the first eight rules bear a certain similarity to those in figure 2. It could be said that the rules in figure 2 are an abstraction of those in figure 17. Rules (9), (10), and (11) concern the addition of an external terrace area to the building, and rules (12), (13), and (14) the addition of external rooms to the terraces.

Part of the context is represented in figure 18 as a gridded building site. The site is partitioned into areas each of which is homogeneous with respect to some particular attributes. These are labeled: attribute 1, attribute 2, attribute 3, attribute 4, attribute 5, and attribute 6, and could refer to any spatially distributed site attributes such as slope categories, soil types, drainage easements, etc. The distribution of each attribute is shown in figure 19. If we assume that this context will have some bearing on the form of the artifact, it is necessary to interpret this information about the context so that it can be used in determining the building form and location. This is achieved by means of reasoning, given some knowledge about suitability. We assume that different parts of the site are suitable for different uses.

We include in the precondition list for each rule the condition that the neighbouring cell on which the neighbouring object is to be located is suitable for its proposed use. This can be tested by the literal:

suitable([X, Y], *USE*),

where X and Y are the coordinates of the address of the cell. *USE* is the name of

the object to be located on the cell. An example of a transformation rule is therefore:

rule(2) # [*located*(*room_a*), *clear*(*room_a*, *right*), *cell*(*room_a*, [*X*, *Y*], *orientation*(*room_a*, *D*1), *right_cell*(*D*1, *D*2, [*X*, *Y*], [*X*1, *Y*1]), *suitable*([*X*1, *Y*1], *room_c*)] >> [*located*(*room_a*), *located*(*room_c*), *clear*(*room_c*, *top*), *clear*(*room_c*, *right*), *clear*(*room_c*, *left*), *cell*(*room_c*, [*X*1, *Y*1]), *orientation*(*room_c*, *D*2)].

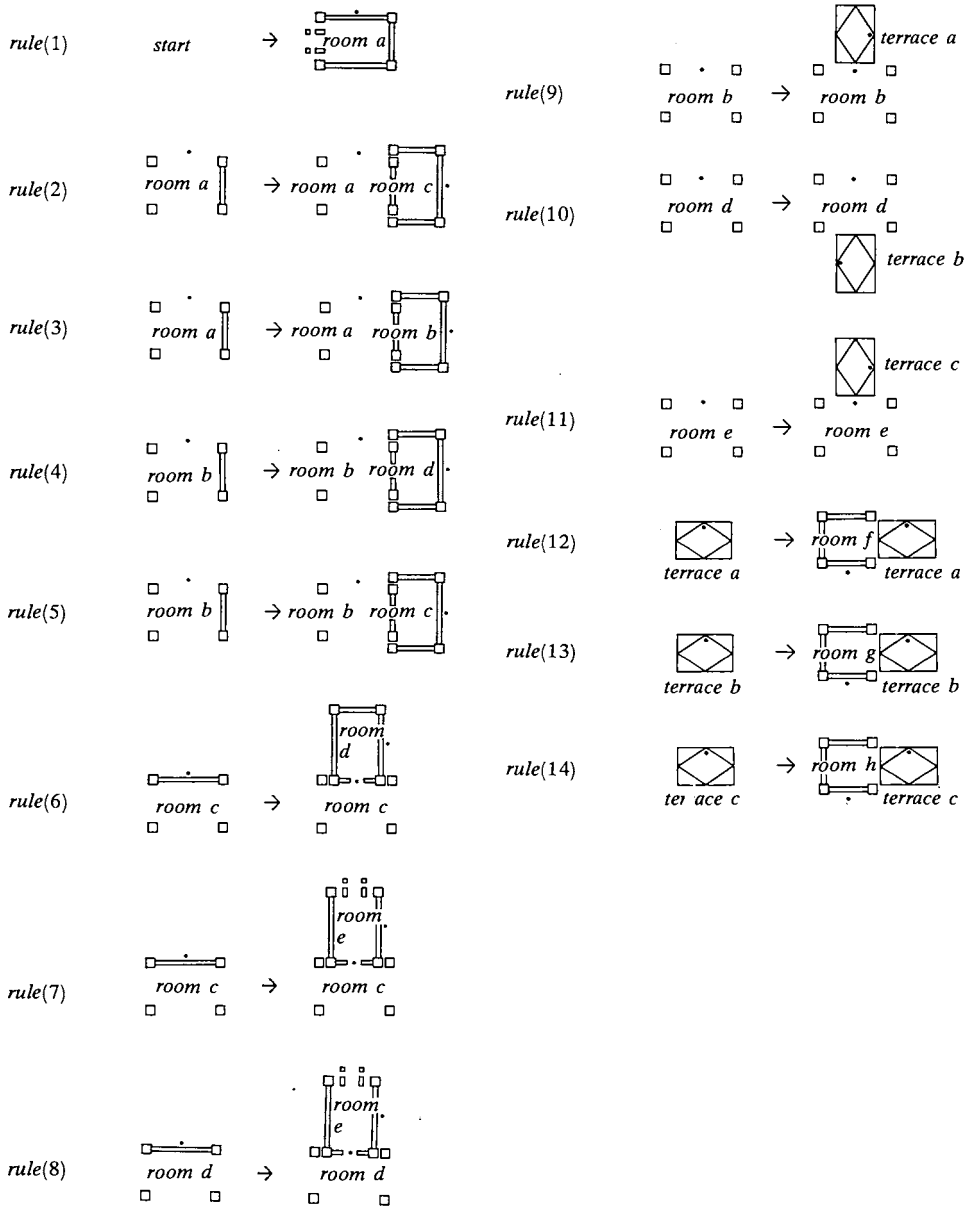

Figure 17. Design rules for a simple building.

The knowledge by which we infer that a particular cell is suitable for particular uses can be represented as the PROLOG inference rule:

```
suitable([X, Y], USE) :—
                    member(USE, [room_a, room_b, room_c, room_d, room_e,
                                 room_f, room_g, room_h, terrace_a,
                                 terrace_b, terrace_c]),
                    site(attribute_1, LIST_1),
                    member([X, Y], LIST_1),
                    site(attribute_3, LIST_3),
                    member([X, Y], LIST_3).
```

This can be read as: a cell with the address $[X, Y]$ is suitable as a location for a room or a terrace if it has attribute_1 (that is, it is a member of the list of cells with attribute_1) and it has attribute_3. This rule requires that facts about cells and suitabilities are stored as:

```
site(A, B),
```

where A is an attribute and B is the list of addresses of cells with that attribute.

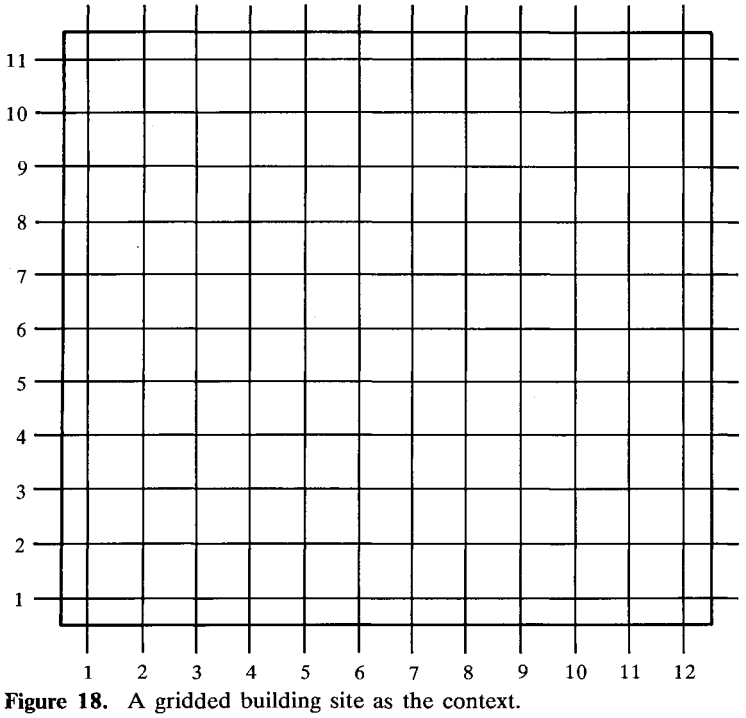

Figure 18. A gridded building site as the context.

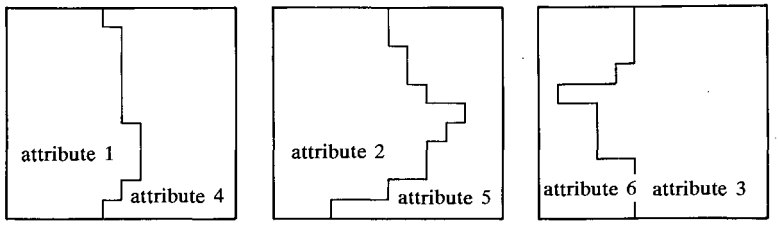

Figure 19. Distribution of different attributes across the site.

It is, of course, possible to represent quite complex knowledge about site suitability using rules of this form. For simplicity, we limit ourselves to two rules represented in tabular form as follows:

Use	Site attributes
room_a, room_b, room_c, room_d, room_e, room_f, room _g, room_h, terrace_a, terrace_b, terrace_c	*attribute_1, attribute_3*
room_f, room_g, room_h, terrace_a, terrace_b, terrace_c	*attribute_2, attribute_3*

The task is, therefore, to generate a building form (or forms) from the design rules, such that the building exhibits a close fit with the site context in terms of suitability. We consider the design problem at two levels by assigning literals to the following criticality levels:

level 2 *start, located*(—), *clear*(—, —)
level 1 *cell*(—, —), *orientation*(—, —), *right_cell*(—,-—, —, —), *top_cell*(—, —, —, —), *suitable*(—, —)

The planning mechanism is the same as that shown in the simpler example above. We show only some of the graphic output from such a system (figures 20, 21, and 22). The 'planner' produces a sequence of design actions, in the form of a list of rules, which achieve simple goals. This sequence is then used by a design generator which implements the rules to produce more complete geometrical descriptions of the artifact.

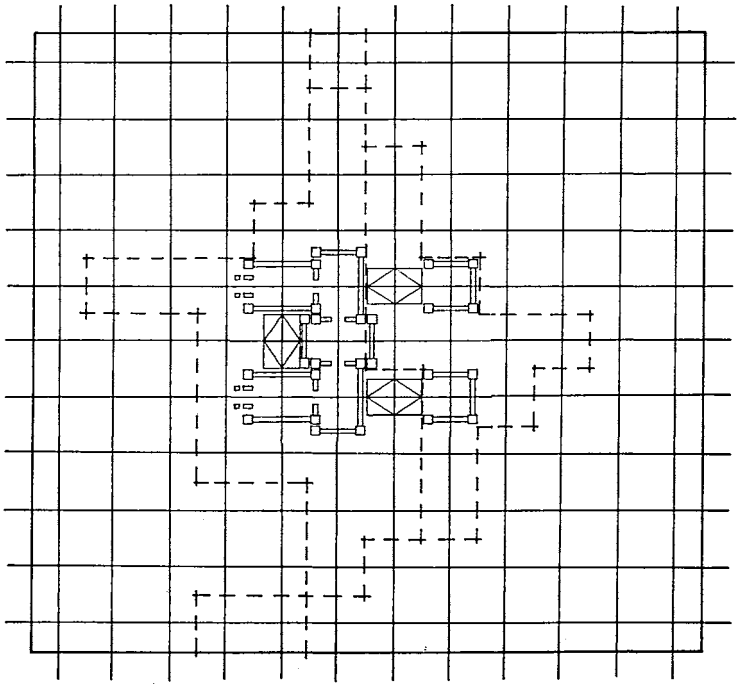

Figure 20. One of the layouts achieved by the rule sequence: rule(1), rule(3), rule(5), rule(6), rule(8), rule(9), rule(12), rule(10), rule(13), rule(11).

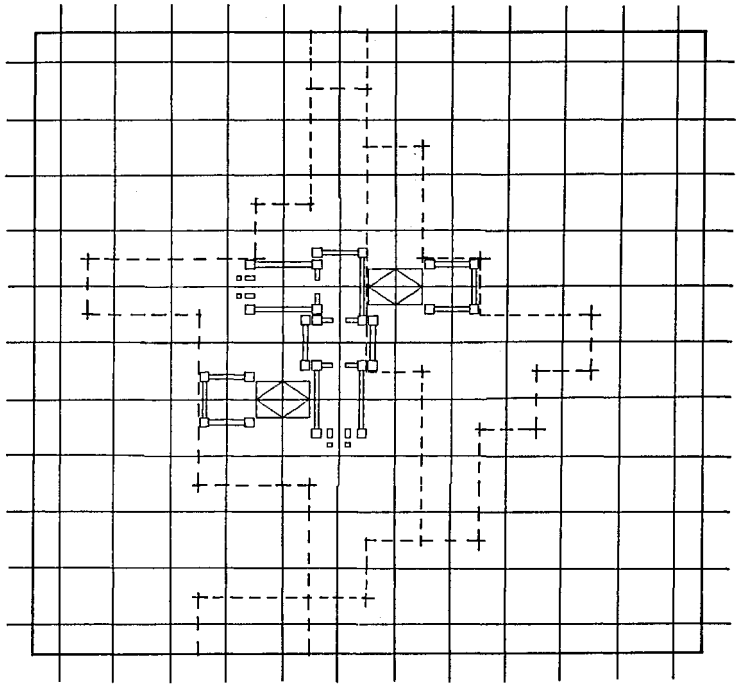

Figure 21. One of the layouts achieved by the rule sequence: rule(1), rule(3), rule(5), rule(7), rule(9), rule(12), rule(11), rule(14).

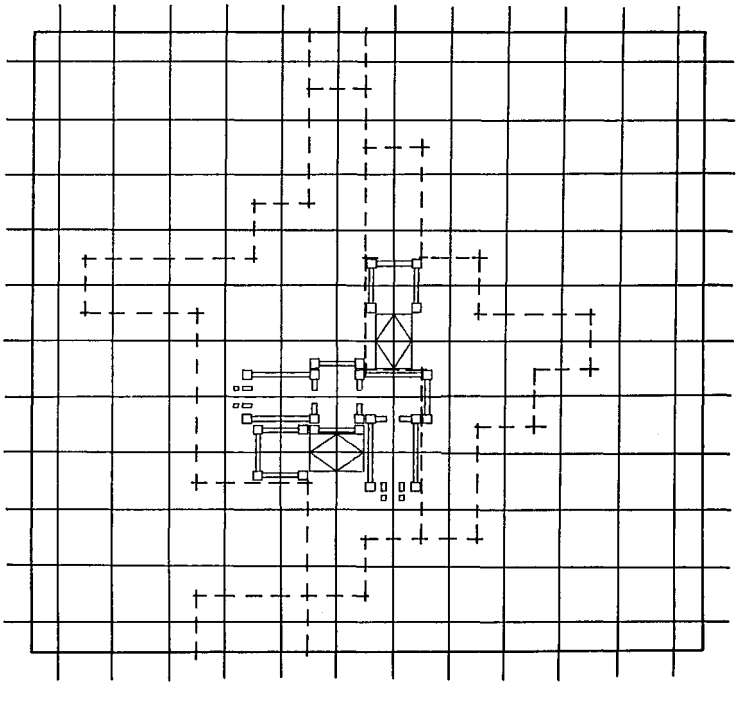

Figure 22. One of the layouts achieved by the rule sequence: rule(1), rule(2), rule(6), rule(8), rule(10), rule(13), rule(11), rule(14).

Conclusion

Planning in an abstracted view of the world by backward chaining was considered as an efficient means of generating designs. In domains where it is necessary to search through a large number of possible design states before reaching a goal it becomes necessary to impose some higher level of control over the planning process. In the examples above we considered the method of assigning criticality values to different literals. This requires some prior knowledge about the importance certain types of facts are going to have in determining the sequence of design actions.

In problem solving the key task seems to be the 'correct' formulation of the problem. In design we can see this as assembling appropriate design rules and goal sets, and also as selecting appropriate abstractions and representations. A major gap in the process as described so far is that formed by the difference between the rule set concerned with the manipulation of geometrical entities (shown diagrammatically in figure 2) and an abstracted version of the rules suitable for the efficient formulation of sequential plans (shown at the end of the section on automated design generation). This gap is bridged by the knowledge required to formulate the appropriate abstraction suitable for the search for a design. Such knowledge is required in an automated system which deals with realistic design problems.

In the implementation discussed here knowledge about context is bound up with the operators which produce transformations. There are advantages in being able to separate out from the operators those literals which relate to the wider context. This would allow greater freedom in the formulation of design rules. Design rules could be formulated and processed as context independent operators. All the conditions and exclusions which relate to the wider context could be invoked by a metarule:

> IF the preconditions of rule X are satisfied by the current state description and IF all the contextual constraints applicable to rule X are satisfied THEN rule X will achieve the goals contained in the consequent of the rule.

It should be possible to infer the applicability of the contextual constraints on a given rule from a body of specific contextual knowledge which links context with state descriptors.

We have demonstrated the applicability of various planning concepts to simple well-defined design problems. Real design problems require more complex control mechanisms than those described here. Other approaches to problem solving in realistic domains include more constructivist views of planning, where skeletal or abstract plans are refined successively under the control of high-level programs called 'critics'. Conflicts within plans are not handled by backtracking (which is a destructive process), but are addressed by critics whose job it is to detect conflicts and to revise plans and partial plans to resolve them. The representation of control knowledge which operates over the planning process (of which the critics idea is one example) has been explored by Sussman (1975), Sacerdoti (1977), Hayes-Roth and Hayes-Roth (1979), and Wilensky (1981) (to name a few), and the applicability of these approaches to design problems must be considered. Human problem solvers resort to a variety of methods in tackling difficult domains. They also know when to change strategies when one approach seems to hold little promise. The representation of this sort of knowledge would be an important part of a design system.

Acknowledgements. This work has been supported by the Australian Research Grants scheme and a Sydney University Postgraduate Research Studentship.

References

Broadbent G, Bunt R, Jencks C, 1980 *Signs, Symbols and Architecture* (John Wiley, Chichester, Sussex)

Chomsky N, 1963, "Formal properties of grammars" in *Handbook of Mathematical Psychology. Volume 2* Eds R D Luce, R Bush, E Galanter (John Wiley, New York) pp 323-418

Coyne R D, Gero J S, 1985, "Design knowledge and sequential plans" *Environment and Planning B: Planning and Design* **12** 401-418

Fikes R E, Nilsson R E, 1971, "STRIPS: A new approach to the application of theorem proving to problem solving" *Artificial Intelligence* **2** 189-208

Gips J, 1975 *Shape Grammars and their Uses* (Birkhäuser, Basel)

Hayes-Roth B, Hayes-Roth F, 1979, "A cognitive model of planning" *Cognitive Science* **3** 275-310

Newell A, Simon H A, 1972 *Human Problem Solving* (Prentice-Hall, Englewood Cliffs, NJ)

Nilsson N J, 1982 *Principles of Artificial Intelligence* (Springer, Berlin)

Sacerdoti E D, 1974, "Planning in a hierarchy of abstraction spaces" *Artificial Intelligence* **5** 115-135

Sacerdoti E D, 1977 *A Structure for Plans and Behaviour* (Elsevier, New York)

Stiny G, 1975 *Pictorial and Formal Aspects of Shape and Shape Grammars* (Birkhäuser, Basel)

Sussman G J, 1975 *A Computer Model of Skill Acquisition* (Elsevier, New York)

Wilensky R, 1981, "Meta-planning: representing and using knowledge about planning in problem solving and natural language understanding" *Cognitive Science* **5** 197-233