Semantics and the Organization of Knowledge in Design

R. D. Coyne and J. S. Gero
Computer Applications Research Unit, Department of Architectural Science, The University of Sydney, NSW 2006 Australia

Accepted for publication January 1, 1986.

The linguistic paradigm of design is considered. The idea of meta-languages is developed as a means of generating designs that are imbued with semantic content. The formulation of meta-languages therefore becomes a way of organizing knowledge about design. The usefulness of these ideas is demonstrated by means of programs developed in Prolog.

INTRODUCTION

Computer-aided design systems often contain models of the artifacts they assist in producing. We may also expect computer-aided design systems to contain other models, including models of designers, or at least of the knowledge of designers. This raises the question of how knowledge about design can be conveniently and effectively organized within computer systems. We pursue the *linguistic paradigm* as a useful basis for structuring this knowledge.

Design can be discussed conveniently as operations within a language. A system can be characterized as a language if it consists of a set of indivisible elements (an alphabet), a set of operations (such as union or difference), a vocabulary, and a grammar.[1,2] The grammar defines a legal syntax. In natural language systems the output is primarily an ordered list of vocabulary elements called a sentence. In design the output is generally an *artifact,* or the representation of an artifact.

We discuss this paradigm by considering *design as a linguistic model* taking place at various operational levels. There are therefore such things as meta-languages in design. We explore this view by considering the critical nature of semantics in the context of design and by discussing how meta-languages can facilitate mappings between meaning and artifact. Production systems are discussed as appropriate ways of implementing meta-languages. A simple system is described that demonstrates these concepts.

SEMANTICS AND DESIGN

Considering artifacts as "sentences" within a language enables us to formalize certain concepts about design that have had long-standing intuitive appeal. (Edward's *Architectural Style*[3] and Zevi's *The Modern Language of Architecture*[4] exemplify the rich body of writings that explore the intuitive link between language and architecture.) One such concept is that it is possible to talk about an artifact having meaning. It soon becomes evident that there is little to say about an artifact, such as a building, if it is considered purely in terms of the vocabulary elements of which it is composed. It is also essential to talk about attributes not immediately evident from an inventory of its component parts. Implicit in the linguistic model is the notion of *semantics*.

The Interpretation of Designs

An artifact possesses attributes other than those explicitly represented as syntax. These can be described as derived, or implicit, attributes, and a set of such attributes constitutes its semantic content. The semantic content can be deduced from looking at the vocabulary elements individually, in combination, and within a context. Each vocabulary element carries with it various descriptions and, in natural language, dictionaries are used to formalize the mappings between vocabulary elements and their descriptions. The mapping of compositions of vocabulary elements within a context onto semantic descriptions is less direct, however.

How is semantic content derived from artifacts that are represented syntactically? The issue is particularly interesting when we consider the representation and interpretation of objects in computer-aided design systems. Here we are dealing with the representation of designs within a medium that is conceptually very distant from the built artifact. Conventional representations of buildings, such as line drawings, possess geometrical properties not dissimilar to those of built objects, and designers have little difficulty in providing the necessary mapping from one to the other. In order to derive basic meaning from a digital representation within a computer, however, it is necessary to make the mappings between syntactic representations and their semantic content explicit. This mapping is generally achieved by means of programs.

It becomes apparent when talking about syntax and semantic content within a computer system that it is possible to describe these mappings hierarchically.[5] The hierarchical levels are generally separable at the lower levels of description, although they may involve complex connections at the higher levels. For example, the syntactic representation of numbers within a data structure may be interpreted as points and lines, points and lines can be interpreted as primitive shapes, primitive shapes as building elements, and collections of elements as objects in the sense of belonging to aggregation or part hierarchies.[6] Certain properties of these objects and collections of objects can also be derived. Elaborate programs have been devised for calculating the complex properties of building designs described in this way. The derived properties of designs are also often called "performances" or "evaluations."

It is also apparent that the distinction between the syntax and the semantic content depends upon one's point of view. Data items, such as arrays of numbers, may be interpreted as points and lines and be regarded as syntactic representations, the semantic descriptions of which are building components. The building components may, in turn, be regarded as syntactic elements and their meanings described in terms of compositions and performances. At any level in this hierarchy of representations we look "downward" to primitive descriptions (syntax) and "upward" to less explicit attributes (semantics). This makes it possible to talk about doors and windows as vocabulary elements within a computer-aided drafting system even though, at a lower level, the system operates on the syntax of points and lines, and the meanings of these element as doors and windows must be derived from programs.

The semantic interpretation of designs represented syntactically can be achieved by means of a type of knowledge. When described as a cognitive activity the process of interpretation is often considered as *inference*. This knowledge can therefore be regarded as inferential knowledge. It can be embedded within algorithms or represented in a form that makes the mappings between syntax and semantic content explicit. A useful and intuitively appealing method of representing inferential knowledge is as rules of the form

$$\text{IF } a_1, \ldots, a_n \text{ THEN } b_1, \ldots, b_n$$

This states that if a design possesses attributes $a_1, \ldots, a_n$, then infer that attributes $b_1, \ldots, b_n$ are also possessed. We may suppose that the knowledge by which a design is interpreted consists of many such rules. This method of representing knowledge has direct parallels with formal logic systems. It is assumed that for a static design within an unchanging context, anything that is inferred to be true remains true. Hence, as we make more and more inferences, we are building up a more complex and complete understanding of the design. (It has been argued that this assumption of monotonicity is not necessarily a good model of cognitive processes, although it appears to be a useful one.[7])

For a given set of inference rules, the number of attributes that can be derived is likely to be very large. Therefore, there will be more work involved in asking of a design, "What attributes can be inferred from its syntactic description?" than asking "Does the design have this particular set of characteristics?" The former suggests a data-driven approach where we begin with a syntactic description and try to infer as much as the rules will allow. The latter is a goal-directed approach where we commence with various attributes and try to discover if the artifact possesses those attributes. The value of different approaches to inferring meaning have been discussed in the context of expert systems, which are a class of automatic inference system.[8,9]

The tasks of representing design descriptions[10] and that of deriving meaning from syntactic descriptions of artifacts raise complex issues. (An example of a knowledge-based approach using rules of inference is that described by Akiner.[11]) The generation of syntactic representations, that is, artifacts, from semantic descriptions poses even greater practical and theoretical difficulties. A generative system takes a set of semantic descriptions as a set of goals and addresses the problem: "What configuration of syntactic elements will constitute an artifact exhibiting these characteristics?" Whereas a goal-driven inference system seeks to discover whether or not a certain set of attributes is true of a design, a generative system seeks to *produce* a design that exhibits a set of goal characteristics. It is unlikely that the rules by which performances can be derived will, on their own, be of much assistance in enabling designs to be generated.[12] In order to generate designs it is necessary to make use of a *grammar* of design.

Design Grammars

The principles by which vocabulary elements can be put together at the syntactic level to form artifacts constitute a grammar. Inherent in a grammar is a set of mappings between vocabulary elements such that certain groupings of elements can be transformed into other groupings. The formulation of phrase structure grammars by Chomsky[13] is based on transformation rules that make explicit the mappings between vocabulary elements. Like the rules of inference described above, each grammar rule has an antecedent part and a consequent part (or a "left-hand side" and a "right-hand side"). If the vocabulary elements in the antecedent part of the rule are present, then they are replaced by the vocabulary elements in the consequent part of the rule. Rules of grammar in natural language can be used to parse sentences, that is, decompose them into specialized vocabulary elements (called "syntactic categories" or "nonterminal elements"), to see whether the sentences conform to the rules of the grammar. A syntactic grammar can also be used to generate legal sentences in the language.

In natural language, strings can be generated by applying rules successively to the nonterminal vocabulary element, *sentence,* to produce a list of words that constitutes a sentence in the language. The linguistic paradigm is an attempt to describe design processes in this way. Whereas rules of inference build up a more complete description of the design, under a grammar a design undergoes developmental or evolutionary changes. From some initial representation of a design (the "initial state"), therefore, the grammar rules are applied to bring about changes, and intermediate stages in the process may exhibit characteristics quite different from the end state. It is possible to generate designs that conform to a syntactic design grammar.

Design Generation

That grammars of composition can be derived for generating architectural designs has been demonstrated both informally by design theorists such as Alexander[14] and formally by Stiny and Gips,[15] Mitchell,[16] and others. But are purely syntactic grammars sufficient on their own to model meaningful design activity? How can compositions of syntactic elements be generated given some semantic content? In natural language, it appears, competent speakers know which grammar rules to select, depending on context, to satisfy certain semantic requirements. Theories of meaning, however, are less well developed than theories of syntax, and this constitutes a major hurdle in our understanding of how *utterances* in natural language are produced.[17] An understanding of how we achieve mappings between intended meaning and suitable strings of words is crucial if we are interested in the automated generation of meaningful sentences. Presumably this mapping is achieved by a kind of knowledge. In design, too, it seems as though we bring knowledge to bear on the creation of artifacts other than the syntactic grammar.

A simple paradigm of design (which is perhaps less immediately applicable to natural language) is that of "generate and test". It is possible to formulate the required semantic content as a set of goals. A grammar is used to generate designs, and inferential knowledge is used to determine which of the designs possesses the attributes meeting the requirements of the goal set. The knowledge that makes possible the mapping between goals and designs is therefore embedded in the control of such a system. In general, the computational effort required for the generation of all possible states in order to carry out this evaluation is prohibitive.

One way to ensure that goals are satisfied without exhaustively generating all the possibilities of the grammar is to develop a system that first finds out which rule, or rules, immediately produces the goal condition. When these rules have been identified the task is to find out which rules need to be applied in order to satisfy their preconditions. The process is therefore to chain backwards through the rules from the goals until the initial state is reached. Once a rule sequence has been determined it can be used to generate the artifact that is "guaranteed" to contain the required attributes. Unfortunately, this is not a simple process since design goals are generally nonserializable; that is, they cannot be satisfied independently of one another, and mechanisms have to be devised for efficiently handling goal interaction.

This view of design (as language) therefore exhibits the problems inherent in general production systems, that of the satisfaction of nonserializable goals. Various techniques have been proposed that attempt to address this problem in production systems[18] and these are applicable to language systems. The applicability of some of these techniques to design has been discussed by Coyne and Gero.[19] However, not all of the techniques assist in preserving the intuitive appeal of the linguistic paradigm, nor do they provide organizational structures that make it possible to explicitly represent knowledge of the type readily available to design experts.

The view explored here is that, for certain design domains, the knowledge by which grammar rules are selected can be made explicit. This is because the grammar rules tend to form a semantic structure on their own. The rules of grammar in design languages are not always anonymous but have names. These names label the rules as *actions,* and these actions form a semantic structure. It is also possible to identify a grammar of actions, or a meta-grammar, and there is often a link between the semantics of actions and the semantics exhibited by the final artifact.

Semantics of Actions

In architectural design the product is generally an artifact occupying three-dimensional space. But design in general can be manifest in other dimensions and media, such as music and art. In certain artistic endeavours the process by which the artifact is accomplished is also considered an important part of the artifact; for example, in such "art events" as wrapping up a mile of Australian coastline in fabric by Christo Javacheff or action painting by Jackson Pollock. Pursuing the idea of art and design as language, actions should therefore form a language of their own with their own syntax and semantics.

In the process of constructing a building (as opposed to the process of design) it is possible to identify certain syntactic elements (a vocabulary of actions). These might be activities such as pouring concrete, laying bricks, etc. If one asks what is being done on a particular part of a building site, the answer "Bricks are being laid" is an appeal to the syntactic content of the process, whereas the answer "A wall is being built" is an appeal to the semantic content of the process. If the answer is "A building is being built" or "They are constructing a single family house" then higher levels of meaning are being explored.

In the process of design similar activities may be identified, such as creating and moving spaces, partitioning spaces, labelling shapes, etc. These are actions that transform design states and constitute the grammar of a design language. Unlike the grammar rules of natural language, as outlined by Chomsky, this design grammar appears to be rich with meaning. It is possible to talk about mappings between actions in the same way that we consider mappings between physical attributes. For example, the action *move the kitchen closer to the dining room* could be part of the syntactic expression of the semantic goal *keep travel distances short*. Whereas it is very difficult to map the natural language grammar rule, *replace a noun phrase with an article and a noun,* onto higher level actions, designers are able to readily articulate such mappings within their languages.

These design actions therefore constitute the vocabulary and semantics of a meta-language. Does this meta-language have a grammar? In order to satisfy the requirements of a language there should be rules for selecting and ordering design actions. An example of such a rule appropriate to any vocabulary of actions is:

> *If* action *a* and action *b* are to be performed, and action *a* requires a consequent of action *b* before it can be implemented,
> *then* order the actions such that *b* is before *a*.

Rules about ordering appropriate actions therefore constitute a type of grammar.

How does the recognition of a meta-language assist in the generation of designs that have been given some semantic description? In some cases there appears to be an obvious mapping between semantic descriptions of artifacts and the semantic representations of actions. By undertaking a particular type of process it is known that designs with particular attributes will be produced. The links between process and form have often been made in the context of architectural design. At a simple level, such a mapping is exemplified by the rule: in order to produce a compact building layout, start with a square perimeter and attempt to arrange the functions within that; or the rule: the actions of laying out grids and axes tends to produce formal and symmetrical building plans. By making such mappings explicit it may be possible to employ this knowledge in the generation of designs. Further mappings will be considered below.

Design is therefore described as a language operating on multiple levels. Just as there is a grammar (or set of actions) that governs how physical vocabulary elements fit together, there is a meta-grammar that determines how those actions are to be combined. There may also be levels of grammar above that.

How does this formal view accord with our understanding of the process of design as attempting to deal with ill-defined problems? In design a complete and definitive formulation (that is, a grammar, vocabulary, set of goals, etc.) is not obvious in any sense from the design problem, but must be extracted from an information-rich and often poorly understood context. Once the formulation is achieved, it is not rigorous and can be expected to change as the design progresses.[20] The view of design, as a multilayered language system, accommodates the view put forward by Simon[21] that design problems may appear to be ill-structured at the large scale of the design process but can be formulated such that they are well-structured at the detailed level. The major task of design therefore appears to be that of structuring the problem. This is a problem-solving task in itself that involves partitioning the domain into well-structured subproblems that are constantly undergoing reformulation. The problem formulation can therefore be seen as a discourse in a meta-language that takes as its syntax the semantics and grammar of the lower level language.

Design languages are therefore highly dynamic. The rules of grammar appear to undergo change, not only as the designer's style develops in the long term, but also as the particular design is being developed. It appears that the reason it is necessary for designers to articulate their processes as sketches is that, in doing so, attributes of the design come to light that could not have been readily anticipated. Not only does a partial design on paper exhibit attributes that were intended, but it also exhibits unintended or "extensional" semantic attributes. Goals are also undergoing evaluation and change. Whereas such processes appear to be less obvious in natural language, design languages are characterized by this visible and obvious exploratory component.

The view adopted here is that the dynamic characteristics of design processes can be accommodated by meta-languages and that the grammars of such languages can be made explicit. However, only the static view will be explored in detail here. The design process will be constrained to that of selecting and ordering rules from a statically defined grammar within a static context. In order to demonstrate how such grammars can be made explicit it is necessary to see how design knowledge can be represented and manipulated.

KNOWLEDGE STRUCTURES

Language systems can be formalized as production systems. The production system formalism is a way of organizing a computational system into particular functional components. David and King[22] and Gips and Stiny[23] provide summaries of the underlying structure common to most production system formalisms. These can be described in various ways, but generally consist of three basic components:

1. the domain on which the system is to work;
2. production rules for transforming the domain from one state into another;
3. a control strategy.

The *domain* of operation is that which undergoes transition as a series of states. This domain can also be described as a "global database." The global database is generally represented using some uniform structure, such as an ordered list, an array, a network, or a set. The elements that make up the global database are literals of some kind. In a language system these are the vocabulary elements. The task of a production system is generally to transform the global database from some initial state to an end state, and there may be any number of intermediate steps along the way.

The application of the *production rules* transforms one state into another. In a language system these rules constitute the grammar. The set of all possible states generated by a set of production rules is termed a space of states. This can be represented as a connected graph where nodes are states and directed arcs are the rules that produce them.

Control is concerned with three issues: the first two have to do with implementing rules, the third concerns selecting rules. The implementation of rules concerns matching and execution. A matching mechanism may search for a one-to-one mapping between literals in the conditions part of a production rule and the global database. However, more complicated mechanisms may be involved, for example, where the conditions in the production rule must be inferred from the global database. In this case the attributes to which the production rule is to respond are not represented explicitly in the form of literals in the global database. This suggests the need for a body of inferential knowledge by which these mappings can be made.

Control is also concerned with the mechanism by which production rules are executed. In a production system it is expected that productions are represented in a uniform way and that there is a mechanism appropriate to the execution of all rules. If there is a match between the conditions part of the production rule and the current state, then those matching elements are deleted from the global database, and those in the consequent part of the rule are substituted. The consequent part of the rule may also consist of a set of actions, performing complex operations on the global database. This requires a set of procedures for carrying out actions, in the same way that the conditions may require a special body of knowledge for matching. There may, of course, be more than one type of production. That is, there may be classes of productions in the system with each class requiring different matching and execution mechanisms.

The third control issue concerns the selection of rules. Given any state of the global database there will be several rules applicable for transforming that state. Typically, there will be many choices, and hence many paths through the space of states, and it will be necessary to adopt some strategy for choosing which rules to execute in order to achieve an end state. There are different ways in which the control can operate in order to do this. It can be designed to produce an arbitrary end state, exhaustively produce all possible end states, or produce an end state (or states) with particular attributes (that is, it can be goal directed). An example of the first type of control mechanism is one that always selects the first rule applicable from an ordered list of rules. This is equivalent to tracing down the left-hand path of a state-space graph in a "depth-first" manner until an end state is reached, at which point the process terminates. A controller for generating all end states may adopt an exhaustive search and backtrack mechanism. This is where paths are explored until an end state is achieved. The controller then backtracks to the last state where there was a choice of rules, selects another rule, and then proceeds to find another end state. This continues until all possible states have been encountered. Goal-directed control generally involves the evaluation of states in some way. It, too, may involve exhaustive search and backtracking, but states are evaluated against some set of criteria, and this directs the search.

This is the basic representation of a production system, and it is the basis for the formulation of many automated problem-solving methods.[18,24] The difficult part of this formulation is in devising methods for efficiently selecting and ordering production rules. In a language system this issue of control concerns the mechanism that achieves a mapping between the intended semantic content of the artifact and the final syntactic expression. A meta-grammar is essentially a method of controlling a production system. Control is achieved by formulating a production system that has as its global database an ordered set of production rules and operates on these by means of meta-rules. The output of such a control system provides an *a priori* rule list for the system it is controlling.

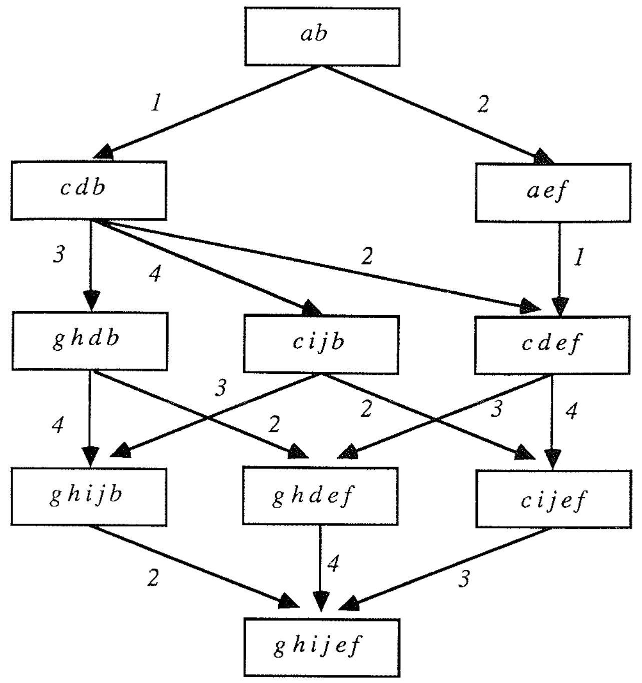


Figure 1. Search space for a commutative system.

Characteristics of Grammars

A production system begins with a global database in some initial state. This is then operated upon by the production rules. Terminal states are those in which no rules are applicable.

Search spaces display different characteristics depending on the characteristics of the production rules and the initial conditions. Certain rule sets operate *commutatively,* that is, at any state where there are several rules applicable any one of these can be selected and executed. The system will always reach the same terminal state even though rules are selected arbitrarily. This does not mean that all states will necessarily be encountered by arbitrary rule selection. Nor does it imply that the terminal state will be reached with equal efficiency along paths. Neither is it necessary that each path encounters all production rules. If a rule set can be identified as being commutative, then that affords certain advantages. There is only one terminal state, and if the goal of the system is to reach this state, then it means an irrevocable control regime can be employed. Backtracking is unnecessary. An example of a set of rules that operates on character strings in a commutative fashion is given as follows:

$$a \rightarrow c, d \qquad (1)$$

$$b \rightarrow e, f \qquad (2)$$

$$c \rightarrow g, h \qquad (3)$$

$$d \rightarrow i, j \qquad (4)$$

A rule can be executed if the symbols on its left-hand side can be matched against the global database. When the rule is executed it results in the symbol on the left being deleted from the database and the symbols on the right substituted. Each rule is numbered. If the initial state consists of the symbols

$$a, b$$

then the system is commutative. The states and the rules that produce them are depicted in Figure 1. The terminal state consists of the symbol string:

$$g, h, i, j, e, f$$

Certain rule sets have the advantage of being *decomposable*. This is where states can be divided into substates that are operated on independently. The terminal states so produced need then only be combined. Certain rule sets display both characteristics, and some interesting symmetries between commutative and decomposable systems are discussed by Nilsson.[18] The rule system described above is also decomposable; this is illustrated in Figure 2. The vertical bars indicate how states can be decomposed and operated on independently. The terminal state consists of the concatenation of the three states at the branch ends indicated (*). (This system has the advantage that not only is rule order unimportant in the achievement of the terminal state, but the rules can also be processed using a control regime that makes use of parallel processing.)

Examples of rule sets exhibiting these characteristics are evident in design. At certain states in the design process the rules that operate on partial design states can be applied independently of one another. In the design of a single story house, for example, it can generally be assumed that rules operating on the living room that locate a fireplace have little or no effect on the rules operating on bedrooms that locate wardrobes. The rule set is, to some extent, decomposable at this stage. The order in which certain rules are implemented may not matter either. So, such a system is also commutative. Attempts to formalize architectural design principles often result in the explication of rule sets of this type, as exemplified in Alexander's *A Pattern Language*.[14] Because rule sets of this type are

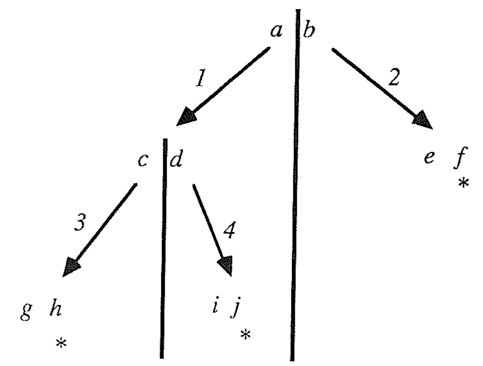


Figure 2. A decomposable system.

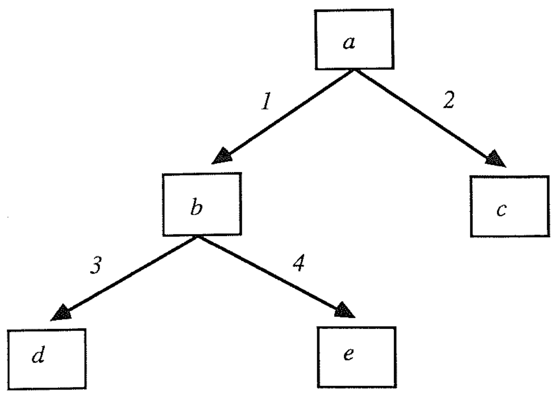


Figure 3. Search space for a poorly behaved rule set.

relatively easy to control, they can be described as "well behaved" rule sets.

A simple rule set that is "poorly behaved", that is, neither decomposable nor commutative, is illustrated below:

$$a \rightarrow b \tag{1}$$

$$a \rightarrow c \tag{2}$$

$$b \rightarrow d \tag{3}$$

$$b \rightarrow e \tag{4}$$

where the initial state is the single symbol:

$$a$$

The search space for this system is shown in Figure 3. The application of any rule commits the system to a path that does not converge with any other path, and there is more than one terminal state. The rules exhibit a high degree of interdependence.

In design we are often concerned with rule sets that have a high degree of interdependence. Rules that govern the placement of spaces in buildings are typically like this. Rules about objects competing for placement tend to produce different results depending on order because a commitment to the location of one object obviates that location as a possible site for another object. The problem of laying out a building could therefore be partitioned into several subproblems: those concerned with interacting rule sets that locate important spaces, and those concerned with decomposable and commutative rule sets about details such as fireplaces and wardrobes. This is, of course, a simplification, but it is the sort of assumption that designers are prepared to make during the development of a design. The most difficult types of problems in design tend to require a grammar that is highly interactive. The shape grammar rules of Stiny and Mitchell[25] are mostly of this type.

Knowledge About Control

It happens that any production system can be reformulated as a commutative system. There is normally no advantage in such a reformulation except in providing a structure that will accommodate explicit control knowledge if it is available. One way of accomplishing this for the search space of Figure 3 is to devise the following control regime:

1. Match the left side of the first rule with the global database.
2. Retain the element that has been matched.
3. Add the right side of the rule to the global database.
4. Join the element added to the element just matched with an arrow symbol.
5. Label the arrow with the name of the rule.
6. Cycle through this procedure for all the rules until none are applicable.

This procedure produces the search space illustrated in Figure 4. This formulation introduces a complexity of representation in the global database, but the process of selecting between rules is simplified since any ordering of rules produces the one terminal state that is simply the search graph of Figure 3. By introducing the following rule it is possible to make explicit the knowledge by which a goal state is matched:

$$goal(X),\ X \rightarrow found \tag{5}$$

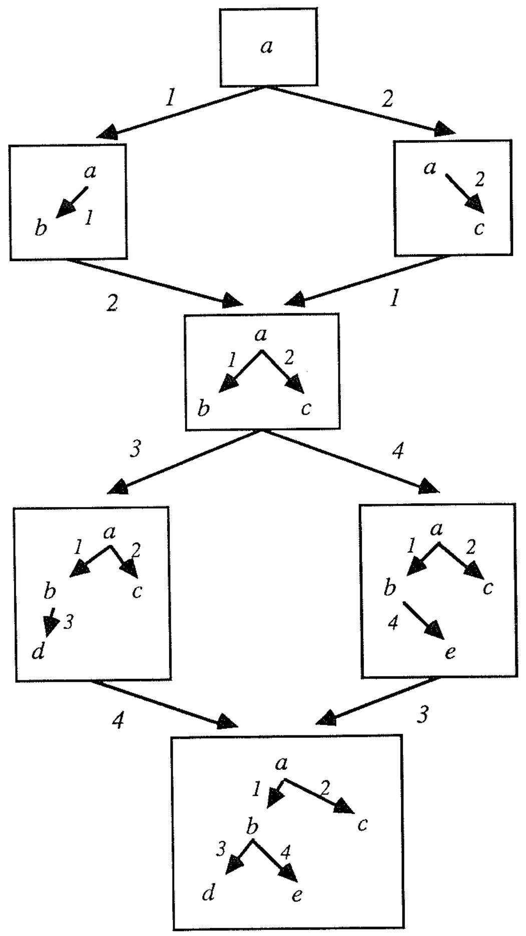


Figure 4. Partial search space of Figure 3 as a commutative system.

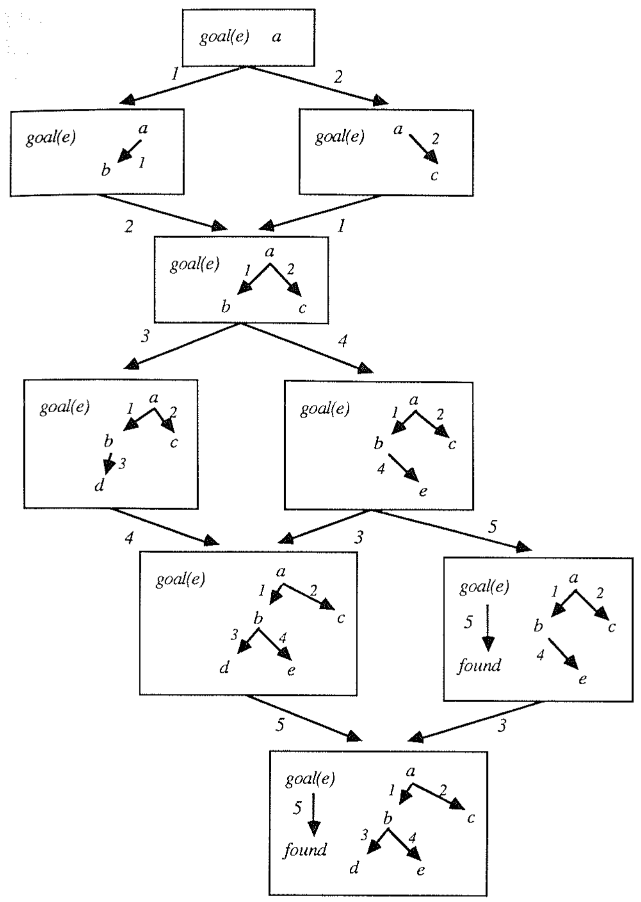


Figure 5. Partial search space of Figure 4 incorporating rule 5.

The rule states that if there is a literal of the form $goal(X)$ in the global database, where X is a variable, and X also exists as a symbol in the database, then add the word *found* to the database. This word is a flag that signals that the goal has been satisfied. The initial state of this system therefore contains a statement about how to find a goal state. The search space for this system, using the same control regime outlined above, is illustrated in Figure 5. There are therefore two goal states: these are states containing the word *found*. What is achieved is simply a way of structuring the problem such that knowledge about control can be made explicit. It follows that if knowledge about efficient search is available, then this can be made explicit as rules in the same way.

Basically, this is the technique employed in the NOAH planning system.[26] Instead of representing states as partially formed search trees, however, the approach is to represent the global database as a network of actions called a *procedural network*. The nodes of the network are therefore the actions or production rules that transform the global database. Nodes are connected in such a way that the system begins with little commitment to order, and the final state is one in which the ordering of the actions is resolved. The operations that transform the procedural network are the actions themselves and a type of rule called the "critic."

This approach has been discussed in the context of learning systems[27] and, more recently, in the context of natural language generation.[28] An explanation of procedural networks in the context of design has also been discussed by Gero and Coyne.[29] This will be developed further below.

Scheduling

If a system is poorly behaved, then the ordering of rules is determined by their interactions. If a system is well behaved, then the control issue shifts to that of finding a path that is, in some sense, optimal. The scheduling system for the HEARSAY speech understanding system appears to work on this assumption.[30] Production system control is provided by types of meta-rules called *scheduling rules* that select between competing rules. (The arrangement of applicable rules is therefore the schedule.) The information available to the system is that found by *triggering* each of the competing rules to see what changes these make to the global database. Scheduling rules check which rule seems to create the most desirable changes, and that rule is then executed. This process is repeated for each state. This has been demonstrated to be a useful basis for simulating cognitive activities, such as errand planning.[31,32]

The use of a scheduler can be demonstrated simply by considering a production system that contains the following three rules:

$$a \rightarrow a,\ b \tag{1}$$

$$a \rightarrow a,\ b,\ c \tag{2}$$

$$b \rightarrow a \tag{3}$$

The first few branches of the (infinite) search graph are shown in Figure 6. Scheduling rules are meta-rules that decide between possible rules at each state. Two such meta-rules might be

s.1 Select the rule that produces the smallest state.
s.2 Select the rule that produces the largest state.

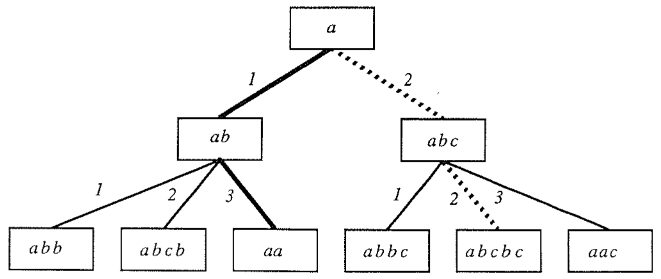


Figure 6. Search graph showing alternative paths generated by scheduling rules *s.1* (bold) and *s.2* (dotted).

If *s.1* is operating, then the search procedure will follow the path indicated with bold arcs in Figure 6. If *s.2* is operating, then the dotted path will be followed. These scheduling rules are entirely arbitrary in this context, but similar strategies are often followed in general problem solving when there appear to be no better guides. However, *s.1* bears some intuitive similarity to the strategy of *consolidation,* whereas *s.2* is similar to *diversifying*. Scheduling generally takes advantage of such heuristics.

One way of tackling the problem of determining rule order in a poorly behaved system is therefore to formulate it as a commutative system that is controlled by a scheduler. This is the same as controlling a language of simple rules but complex interactions by means of a meta-language that contains complex rules driven by a simple controller. This will be developed further.

STRUCTURING DESIGN KNOWLEDGE

Knowledge-based systems are computer programs that are intended to make the knowledge about a particular domain explicit and operable. In general, this can be achieved if the knowledge is represented and organized to meet certain objectives. These can be summarized as:

1. uniform methods of knowledge representation;
2. uniform control mechanisms;
3. the knowledge is explicit and has some meaning in terms of the domain;
4. the processes resulting from the application of the knowledge are visible and comprehensible in terms of the domain.

It appears that it is possible to meet these objectives, at least in part, with formulations based on production systems, although other methods are actively used in knowledge-based systems.[33,34] It is desirable that the knowledge contained in the production rules bears some resemblance to the way that human experts understand their knowledge. So the way in which the rules are formulated is also important. The fourth goal suggests that it should be possible to "see" what is being done with the knowledge, perhaps graphically, and that the intermediate steps during the development of the artifact resemble, in some way, the processes carried out by designers. The formulation of a knowledge-based system that is appropriate to design and that makes use of production systems is considered here as a means of addressing these objectives.

In the linguistic formulation proposed, knowledge is made explicit primarily in the form of grammars and rules of inference. Control knowledge is made explicit in the form of meta-grammars. This can be demonstrated by structuring a domain, that concerned with layout planning in buildings, as three layers of language. The layering takes advantage of the properties of commutative production systems as discussed above. So, as well as providing a structure for making knowledge explicit, there are some implementational advantages as well. The particular layering proposed (and the naming of levels) is pragmatic and is not intended to capture a universally applicable view of design, but is intended merely to illustrate that such layering is possible and advantageous. (As has already been suggested, an even richer formulation would be the provision of meta-languages that permit the grammars of the languages they control to undergo dynamic change.)

Language of Form

There is a language that operates at the object level and is concerned with transformations involving spatial forms. The grammars demonstrated by Stiny and Mitchell are of this type. They operate on shapes that consist of points and maximal lines. These grammars do not require any representation of the semantic content implicit in those shapes in order for transformations to be effective. (A separate grammar is proposed for building up descriptions of designs by Stiny[35] in a way different from that discussed here.) A simple grammar for generating designs has been demonstrated and implemented as a computer program by Coyne and Gero.[19] A different approach has been employed; the vocabulary is a set of primitive building elements such as walls and columns, and the grammar rules consist of explicit representations of those elements that are to be matched and those that are to be substituted. This has the limitation that building elements are treated as if they were items of furniture: entirely discrete and of fixed size.

There may be any number of ways of formulating object level grammars, but a useful type of rule is that which takes dimensionless spaces as vocabulary elements and operates on these. This is an attempt to capture something of the "fluidity" of spatial representation during the design process.

The grammar proposed is similar to the set of rules described by Mitchell et al.[36] for generating "rectangular dissections." Other rules for the same purpose are proposed by Earl[37] and Flemming,[38] as discussed by Steadman.[39] The study of rectangular dissections is ostensibly concerned with enumerating the different ways in which a rectangle can be divided into subrectangles, but it can also be applied to the generation of building layouts. Dimensioning is of only secondary importance in this context. Dimensioning can be considered separately, and there are numerical techniques for handling this problem.[40] Examples of rules by which dissections can be generated are considered here as a means of generating layouts. These are shown schematically in Figure 7. Rule *a* adds a rectangle by appending it to the right side of an existing composition. Rule *b* appends a rectangle to the top of

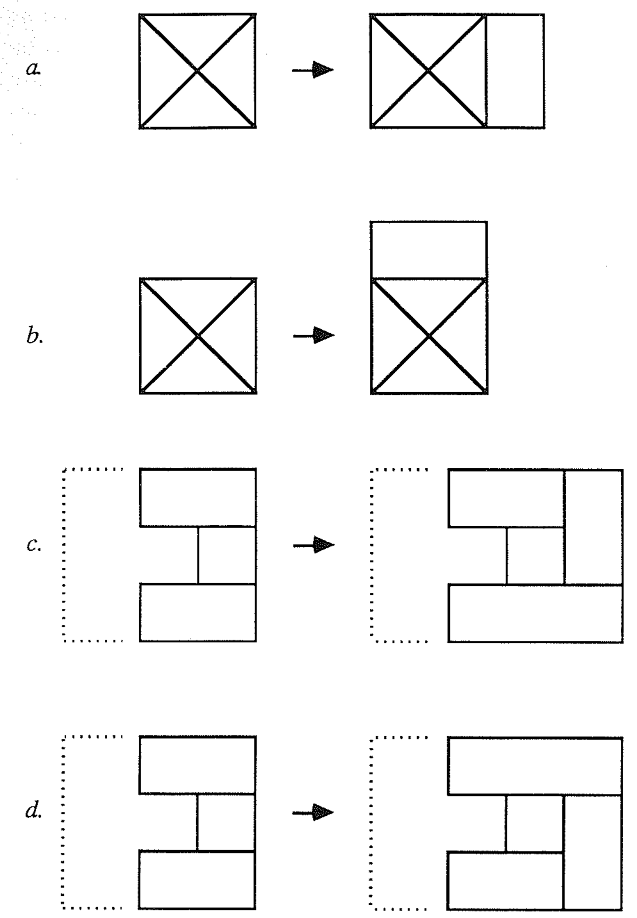


Figure 7. Examples of rules for generating rectangular dissections.

the configuration. Rules *c* and *d* operate by extending certain of the rectangles at the edge of the configuration and adding a new rectangle in the space thus formed. Other rules can be devised. The idea is to begin with a single rectangle (a space) that then undergoes successive transformations resulting in a complex composition of rectangles. It should be noted that after the operation of each rule the composition retains its overall rectangular shape (as well as being composed of rectangles). States in the development of a simple rectangular configuration are illustrated in Figure 8. The success of the grammar for generating building layouts relies on the following set of assumptions:

1. There is a small set of rules with which it is possible to generate every possible arrangement of rectangular dissections.
2. Any geometrically orthogonal arrangement of rectangular spaces can be derived by first creating a rectangular dissection (allowing for the deletion of spaces).
3. The generation of rectangular dissections is best accomplished if every intermediate state in the generation process is also a rectangular dissection.

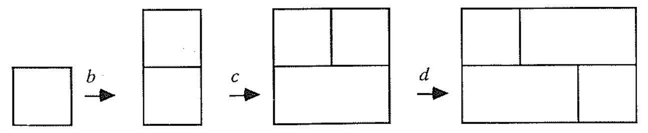


Figure 8. States in the development of a rectangular configuration achieved by executing the rule sequence *b, c, d* from Figure 7.

The validity of these assumptions is not of concern here. It is also assumed that any deficiencies in the rules proposed can be overcome by the provision of more rules, or the refinement of the rules. The general idea is therefore not dependent on the sophistication of the proposed grammar.

This language is fairly rich and can be employed to generate simple building layouts. It appears to exhibit the characteristics of the poorly behaved rule set described above, since the order in which rules are executed clearly determines the outcome, and there can be many end states. (That there is an infinite number of possible layouts can be proven by considering the repeated application of rule *a*.) There are other rules relating to this domain that can be added and that display well behaved properties. These are illustrated in Figure 9 and are concerned with the locations of windows, the locations of openings between rooms, and the deletion of spaces. These can be considered as "enhancements" since they do not really affect the location of spaces as determined by the other rule set, nor do they impinge appreciably on one another. It can be assumed that the implementation of one of these rules is not affected by the implementation of other rules of the same type. The rules of Figures 7 and 9 could operate as a useful set of "commands" for a

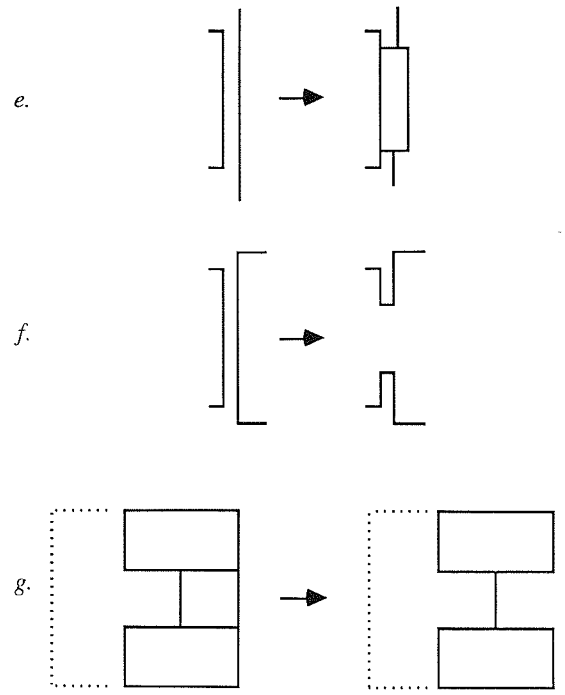


Figure 9. Examples of rules for positioning windows (*e*), positioning doors (*f*), and deleting spaces (*g*).

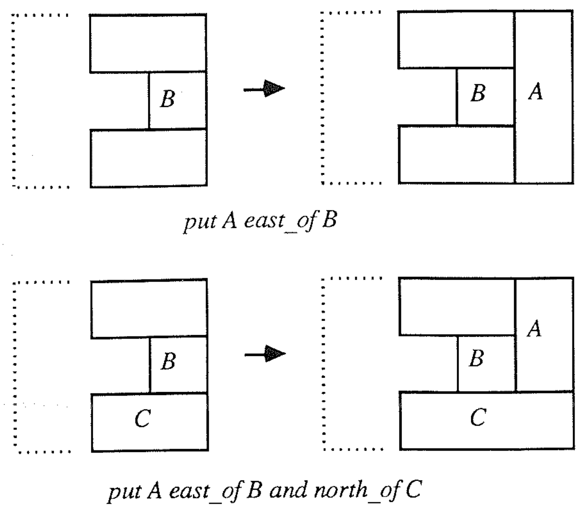


Figure 10. Descriptive names associated with rules *a* and *c* in Figure 6. *A*, *B*, and *C* are variables naming spaces.

design system. The rules only contain knowledge about what preconditions are necessary before the consequent can be carried out. They contain no knowledge about whether or not it is desirable that a rule be executed in order to meet certain semantic requirements. This is the role of the language of actions that controls the language of form. In this discussion the rules of Figure 7 only are considered in detail.

Language of Actions

The rules of Figure 7 can be given names that describe what they achieve. They can be represented in the general form

put A Dir B,
put A Dir_1 B and Dir_2 C

where *A*, *B*, and *C* represent the names of spaces, *Dir*, *Dir_1*, and *Dir_2* are directions (north_of, south_of, east_of, and west_of), *put* is a descriptive predicate indicating that the spaces should be *put* in place, and *and* carries its normal meaning as conjunction. As well as containing a set of conditions that must be matched against a global database and a consequent that describes the changes to be made to the database, the rules therefore also carry a descriptive name. Rules *a* and *c* in Figure 7 can therefore bear the names

put A east_of B
put A east_of B and north_of C

where *A*, *B*, and *C* are variables representing spaces (Fig. 10). The actions

put bathroom east_of bedroom
put dining_room east_of kitchen and north_of living_room

are therefore specific instances of these rules.

Rules that transform states can also be regarded as actions. The grammar of Figure 7 can therefore be described as a set of actions. These actions constitute the *vocabulary* of a *language of actions*. In this language the way in which the actions operate (as rules) can be ignored. So we only consider their names. The relationship between these actions, however, is important. The "artifact" (final state) of the language consists of a sequence of actions that provides the control for a language of form. There are two useful relationships between actions that we wish to consider: *serial* and *parallel* relationships. In serial relationships actions are configured in the order in which they are to be executed. In parallel relationships actions are arranged in groups as conjunctions and disjunctions. It is therefore possible to represent actions with varying degrees of commitment to order. These relationships can be represented in a procedural network.

The example given by Sacerdoti[26] to demonstrate a procedural network in the NOAH planning system is that of painting a ceiling and painting a ladder. The two actions *paint the ceiling* and *paint the ladder* are depicted in parallel as *state.1* in Figure 11. In effect, there is no commitment to order, that is, no suggestion that one action should be performed before the other. In order to work out the correct sequence, as well as to provide a more specific set of actions to perform, it is necessary to consider how the actions in *state.1* can be broken down into more specific actions. This can be achieved with the mappings represented as transformation rules in Figure 12. *State.2* is achieved by simply implementing these transformations, that is, expanding the tasks *paint the ceiling* and *paint the ladder* into their component actions. There is only a partial commitment to order as yet. Provided we have some knowledge about the domain, it becomes clear that certain actions are going to conflict if an attempt is made to arrange the network serially. (For example, it is not possible to climb something if it has just been painted.) Specialized rules (called critics) are introduced to adjust the network in the light of such knowledge. *State.3* shows how the linkages in the network have been adjusted such that *apply paint to ladder* occurs after *apply paint to ceiling*. The application of further critics would recognize that the same paint used for the ceiling can also be used for the ladder and would cause one of the actions, *get paint,* to be eliminated.

This approach ensures that partial sequences of actions are refined by critics before they become too large. The system therefore proceeds from a less detailed to a more detailed sequence, the critics making sure that the actions make some type of global sense before the network is expanded further. It can also be seen as an hierarchical approach in which attempts are made to create simple, correct, high-level sequences before they are expanded into greater detail where the

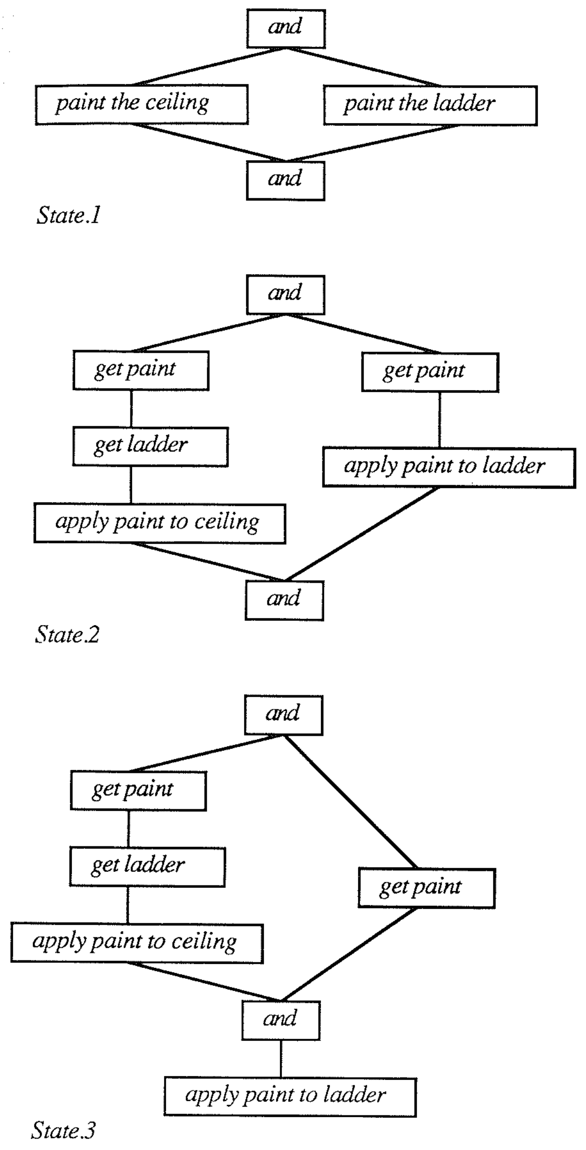


Figure 11. States in the transformation of a procedural network (after Sacerdoti[26]).

costs of rectifying the networks are greater. Although the use of critics is intended primarily to resolve conflicts between actions, the device can be extended to incorporate heuristics that also help to *avoid* conflicts.

Translating this method into the linguistic paradigm, it becomes apparent that, in the same way that we required a richer understanding of a building than just a knowledge of its basic components, we require a richer understanding of actions. Actions bear a semantic relationship to other actions as exemplified in Figure 12. In the context of design the action *put dining_room* ***east_of*** *kitchen* is therefore also a specific instance of the action *put dining_room* ***next_to*** *kitchen,* which is a specific instance of *put dining_room* ***near*** *kitchen*. The actions eventually map onto high level actions such as *design_house*. These are semantic mappings.

We therefore have a language system that takes as its initial condition some high level action (or actions) and produces a predominantly serial configuration of executable actions as an end state. Intermediate states consist of actions on different semantic levels arranged with various degrees of commitment to order. The transition is therefore from a semantic set of actions to an executable set of actions, and from a set of actions in which there is relatively little commitment to order to a set of actions with a more complete commitment to order.

What is the grammar of this language? The semantic mappings between actions can be employed to transform states. These mappings can be utilized by a grammar rule called an "expansion rule," which replaces higher level actions with lower level actions. (An alternative would be to formulate the mappings as rules in themselves, as in Figure 12. This approach has not been adopted in order to preserve generality.)

As was illustrated with the example about painting a ceiling, just substituting executable actions for higher level actions is insufficient for producing appropriate sequences since actions interact with one another. Other grammar rules are required, concerned with interactions between actions. In the NOAH system, critics tend to resolve conflicts within a network, but the general idea can be extended to other operations on networks. Such rules are intended to encapsulate more complex knowledge than can be represented by simple mapped pairs. An example of a rule appropriate to the domain of layout planning is one that states

If the procedural network contains a set of actions concerned with locating spaces and these actions are arranged in parallel (that is, there is no commitment to order)
then order the actions according to the importance of the spaces on which they operate.

This can be interpreted as: if there are several spaces to be located and the desirable interconnections be-

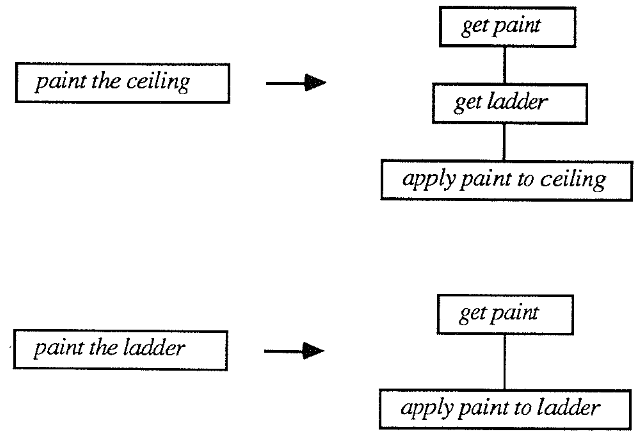


Figure 12. Examples of rules applicable to the language of Figure 11.

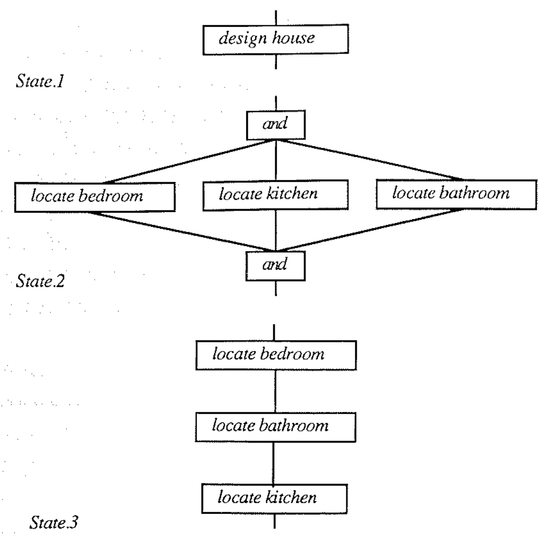


Figure 13. States in the transformation of a procedural network in the domain of spatial layout.

tween spaces are known, then locate the space with the most interconnections first. Although not entirely general, this is a simple heuristic of the type that a designer might use in beginning a layout exercise.

A simple example of states in a domain about actions in the design of a building is illustrated in Figure 13. *State.1* contains the single action *design house*. Given that there is a simple mapping between this action and the actions of locating the components of *house, state.2* can be produced by an expansion rule. *State.3* is produced by applying the rule about locating the most important element first (described above). The application of further rules may expand the network into more specific details and take account of more complex mappings between actions. Further information about building components is required in order to bring about the transformations of Figure 13 than is evident from the semantic mappings described. This will be discussed below in connection with *context*.

The grammar of the language of actions therefore captures knowledge about the ordering of actions. As illustrated by the simple example of Figure 13 we might expect that designers make use of this sort of knowledge all the time. A further point that can be made about this formulation is that it can be constructed as a well-behaved system. In the same way that the system demonstrated in Figure 4 has only a single end state, it is possible that the continuation of the sequence of states in Figure 13 would produce a single end state, and that the order in which the rules are applied only affects the efficiency with which the end state is reached. This is not to say that only a single building layout is produced. The end state may consist of many sequences of actions in disjunction. When executed, the sequences of actions produced by this system generate artifacts in the language of form. The sequences may result in the generation of many artifacts. The end state of the language of actions is a *plan*. We now consider what mechanism can be employed to control the grammar in order to produce plans.

Language of Plans

The type of knowledge that we wish to encapsulate is that which comes under the category of "meta-planning" knowledge, as suggested by Wilensky[41] and Stefik[42] or control knowledge, as discussed by Davis.[43] The objective is to capture universal principles employed in decision making.

Knowledge about the selection and ordering of the rules constitute the grammar of actions (described above). This knowledge can be represented as a grammar within a language that is concerned with formulating plans. In the same way that the rules in the language of form can be given names that enable them to be treated as vocabulary elements in the language of actions, it is possible to give the rules of the language of actions names and regard them as vocabulary elements in the meta language called the *language of plans*. The vocabulary of this language therefore consists of *planning tasks* such as those that were effective in bringing about the state changes in Figure 13:

expand actions,
order actions, etc.

Tasks such as these can be organized according to their relationships with one another and they can also be organized as groups that are related to other tasks. Rather than the procedural network schema discussed above, it seems convenient to relate these tasks within a simple hierarchical graph. The levels within this graph make explicit the semantic relationships between the tasks. They can be considered as similar to macrooperators in a structured computer program or, in this context, it is perhaps more meaningful to talk about a *schedule* of tasks. A schedule (unconnected with any domain) is depicted graphically in Figure 14. Task *1* can be broken up into the subtasks *1.1* and *1.2*. Task *1.1* can be broken up into the subtasks *1.1.1* and *1.1.2*, etc. There is also an implicit order in that task

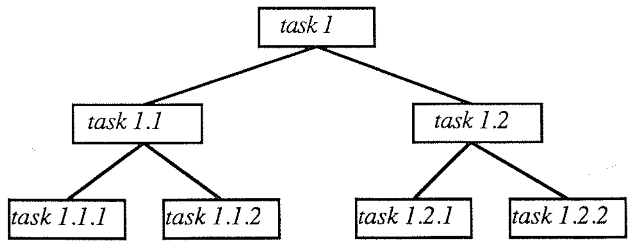


Figure 14. A hierarchical schedule of tasks.

1.1 is to be performed before task *1.2*, and task *1.1.1* is to be performed before task *1.1.2*, etc.

The advantage of arranging tasks hierarchically in this fashion is that those which naturally group together can be treated as a block if it becomes necessary to rearrange the schedule. So, if the order of tasks *1.1* and *1.2* is to be changed, then the tasks below these are exchanged by implication as well.

The syntactic output of this language system is therefore a graph of hierarchically arranged tasks called a schedule. Such a schedule could be built up from some simple initial state, but it is proposed that the initial state of this system is a complete schedule (as in Figure 14), modified in some way to make it more responsive to the domain in which the language is operating. The grammar for this system should consist of rules that manipulate the schedule. There are two ways in which we could envisage this happening. The first, that of modifying the structure of the schedule by means of rules, will not be considered here.

The other way in which a grammar can operate on the schedule is where there is a choice of tasks to be undertaken and the ordering has to be decided upon, such as at the branch ends of the graph. One way of handling this problem is to partition the global database of the language of actions in such a way that the effects of each task can be stored temporarily before they are actually executed. The grammar then operates on this information. The procedure therefore is to discover the tasks that are applicable to the current state of the global database of the lower-level language. This is achieved by *triggering* the tasks, that is, checking that their preconditions exist in the global database and recording what changes each task will make to the database. The language of plans therefore requires "cooperation" from the language of actions.

The language that this meta-language is controlling (the language of actions) tends toward well-behaved characteristics, and the grammar of the language of plans consists of scheduling rules. An advantage of this formulation is that the importance of domain-independent principles becomes apparent. Rules for manipulating tasks could be based on general strategies for problem solving, such as

1. Consolidation
2. Diversification
3. Convergence
4. Divergence

A strategy to consolidate might be expressed as a rule that takes as its precondition the relative sizes of the states produced by each of the planning tasks competing to be executed, and as a consequence selects the task that tends to reduce the size of the database. Alternatively, it may select the task that produces a state with the smallest "conflict set" (set of competing tasks). A rule about convergence may select the task that operates on the most recently affected database item. This will cause the system to concentrate on particular elements. Rules about convergence may also operate by selecting planning tasks that operate on the most important database items first. It could be maintained that important elements are those which occur earliest on a procedural network.

The control strategy for this system could be one in which only a single scheduling rule is adopted, or there might be an overriding strategy that states, for example,

Attempt to consolidate, and converge attention (whichever is applicable at any given state); if, after a particular period of time, no satisfactory end state is reached, then attempt to diversify.

It is possible to conceive of a meta-language operating above the language of plans to facilitate the representation of this sort of knowledge. This will not be developed here, however. In the following section we concentrate on the implementation of a system concerned with representing and manipulating knowledge that develops plans of actions.

REPRESENTING PLANNING KNOWLEDGE

Three layers of language appropriate to any domain have been discussed, but here they will be considered specifically in relation to layout planning. The first, the language of form, takes as its vocabulary elements dimensionless spaces. The grammar is a set of rules for manipulating spaces such that the layout "evolves" as a configuration of rectangles. The grammar can also be seen as a set of actions that form the vocabulary elements of a meta-language (the language of actions). In this language, actions are configured on a structure called a "procedural network." The outcome of this language system is a statement about actions and their ordering. This provides the control for the language of form, that is, it provides an *a priori* rule list that, when executed, produces artifacts.

The grammar of the language of actions consists of rules that transform procedural networks. These rules can be identified by descriptive names and are a type of planning task. The language of plans takes these tasks as its vocabulary elements arranged within a hierarchical schedule and selects tasks according to its grammar. This language operates by deciding between tasks dynamically as choices are demanded by the language of actions. So this language operates not by providing an *a priori* list of operations for the language below it but by operating in consort. It can be seen therefore that the three languages operate differently and they each bear different relationships with one another. These relationships are illustrated schematically in Figure 15. The arrows link the syntactic prod-

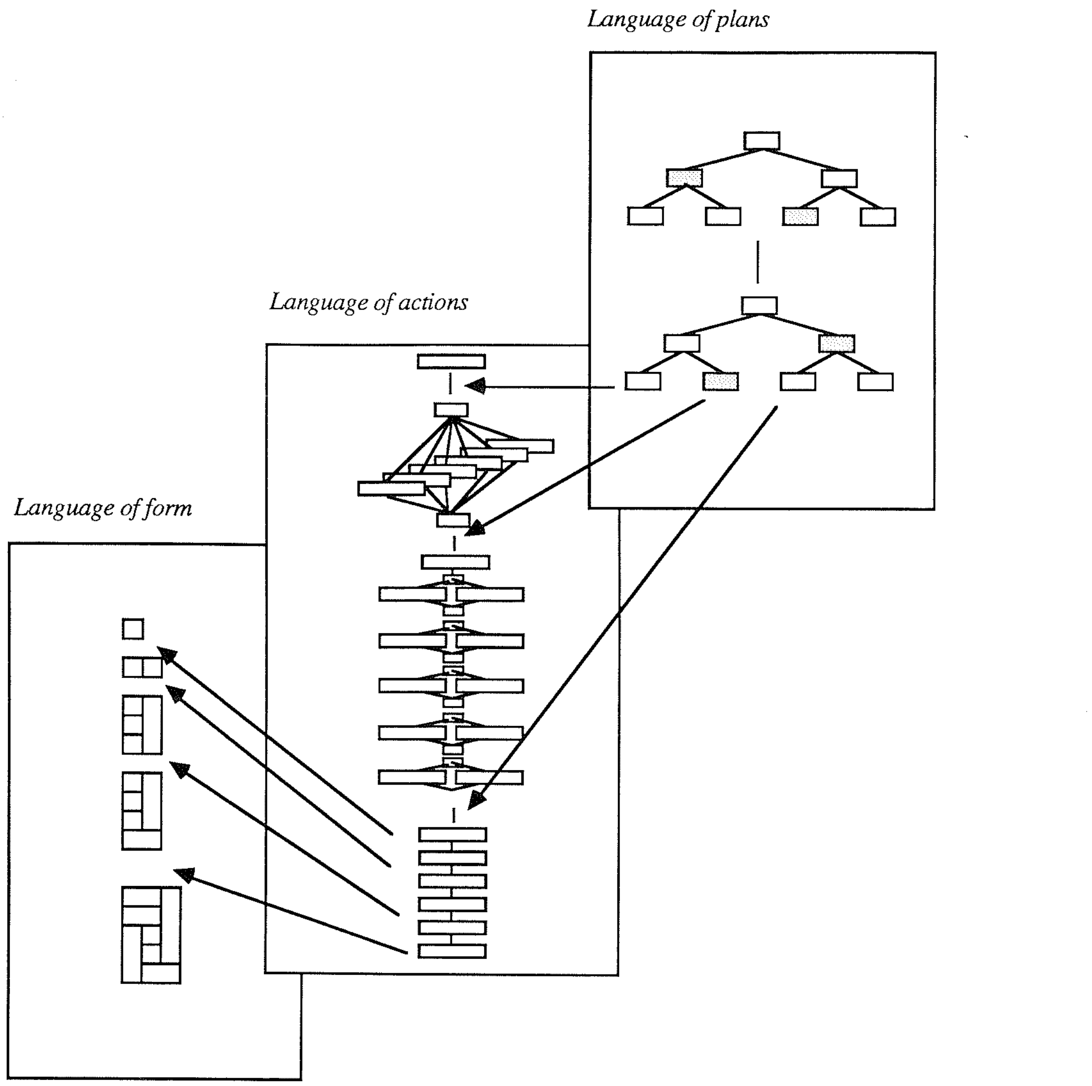


Figure 15. Schematic representation of the relationship between a language and meta-languages in the spatial layout problem.

uct of each language with the transformational grammar of the language below it.

Context

Transformational grammar rules can respond, therefore, to control exercised from above (a meta-language) and to conditions within the global database and also to *context*. We may regard as context those parts of a problem domain that remain constant throughout the process but vary from one problem to the next and therefore contribute to the unique nature of any design. In this formulation it seems appropriate to make this information explicit. Contextual information in the layout problem may therefore be concerned with desirable spatial relationships, the hierarchical relationship between parts, relative room sizes, etc. The contents of the context could also be seen as a design brief or a set of performance specifications. (The flexible nature of a design brief suggests that a more complete formulation should, of course, allow for languages that actually operate on the context.)

A set of facts constituting a design context is represented as terms in predicate calculus notation able to be manipulated in Prolog[44] and is depicted graphically in Figure 16. The first fact is that a particular building (called *building(x)*) is composed of seven spaces, six of which are rooms, the last being a *void* (that is, a courtyard or other open space). The desired relationships between the spaces are depicted in an adjacency network. The context constitutes a type of input to the system, but it may itself be a product of a similarly formulated design system. It is not entirely realistic to assume that such relationships will remain static throughout the design process, but they are assumed static for this exercise. In this example we also consider the relative sizes of some of the rooms to be important. This will serve to constrain or, in this example, *guide* the selection of plan actions. The spatial configuration should be such that these relative sizes

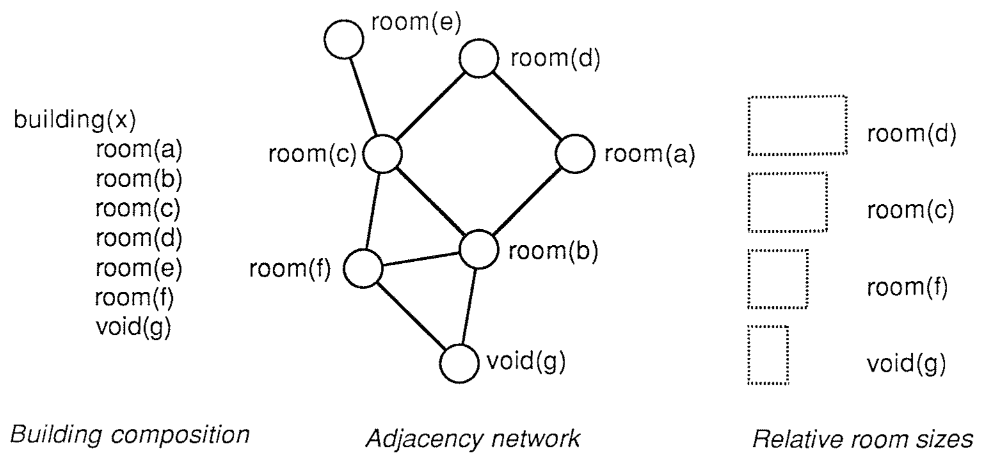


Figure 16. Design context for a spatial layout problem.

can be achieved when the rooms are dimensioned, even though the relative size differences are not actually evident from the dimensionless configuration.

This set of facts therefore constitutes an explicit, high level description of the final artifact. Also part of the context is the initial state of the planning process. Here it is the action, *configure(building(x));* that is, the task is to arrange the components of *building(x)*. This is the semantic action that must be expanded and shaped into a sequence of low level syntactic elements able to be executed. Therefore, the grammar for achieving this "reads" the other facts in the context as well as the current state of the plan.

Plan Grammar

Some rules of a planning grammar are represented schematically in Figure 17. (These rules are simplified. They have been represented more precisely than illustrated here in predicate logic.) The expressions in the boxes are Prolog terms, where upper case characters are variables. As a rule is executed, its left side is matched against the current state of the global database (that is, the facts representing the state of the plan). The variables are instantiated to corresponding values in the matched facts, and the terms on the right side of the rule are substituted in the current state.

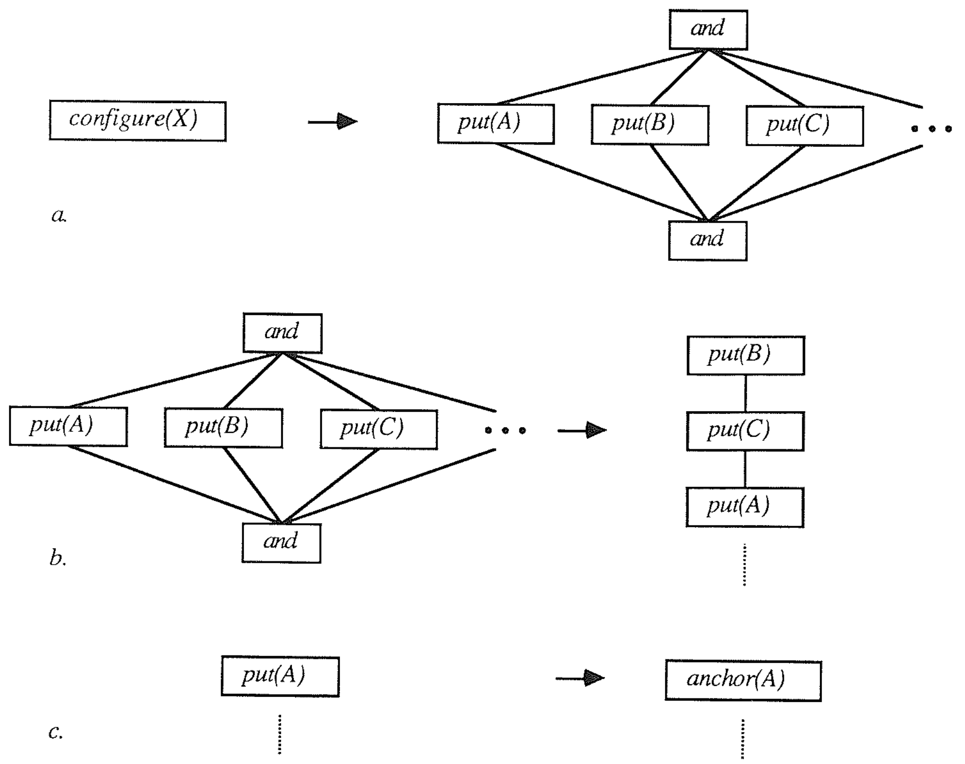


Figure 17. Planning grammar.

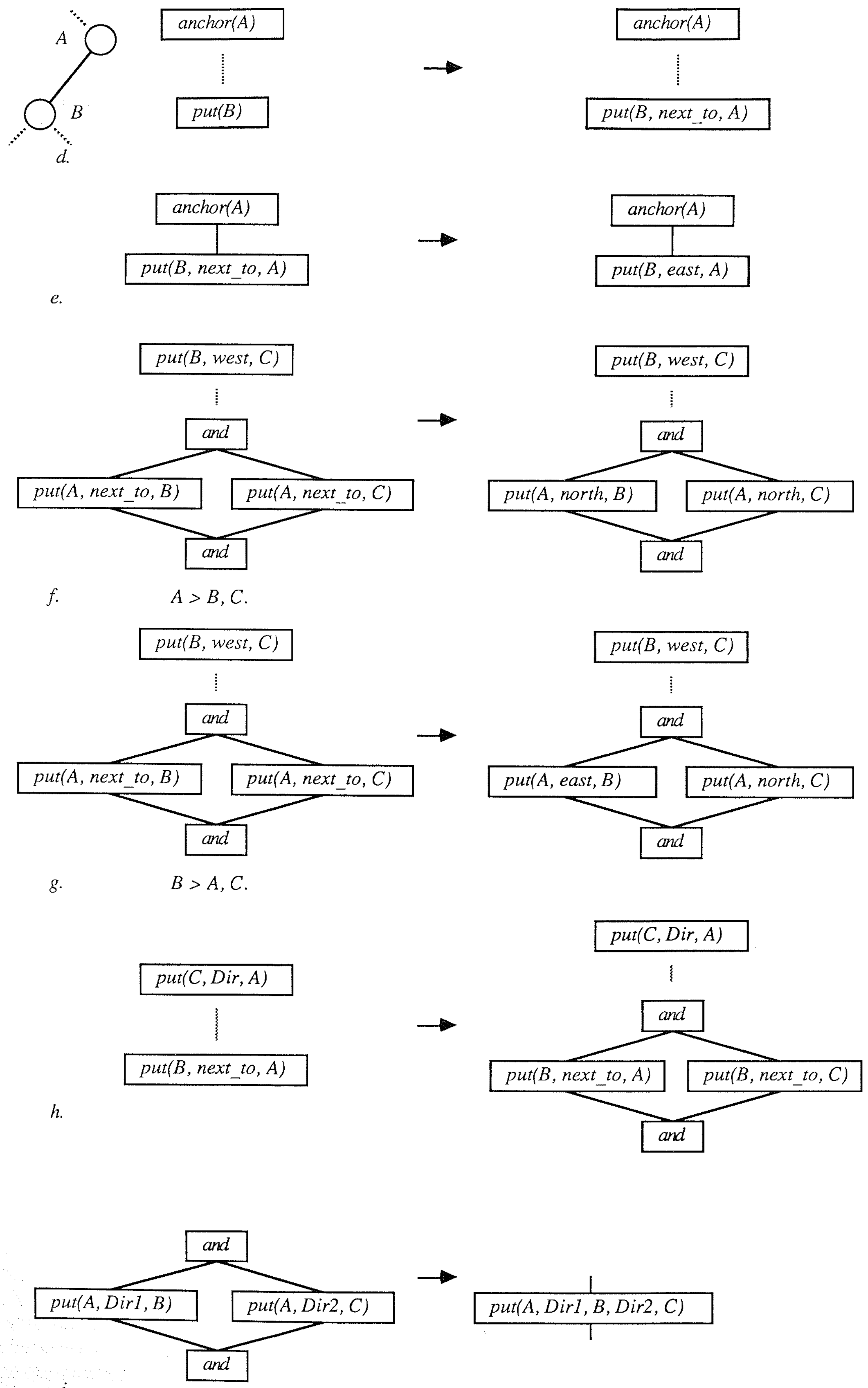


Figure 17. *(cont.)*

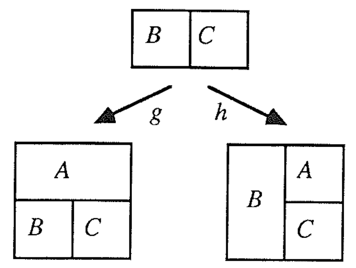


Figure 18. Alternative configurations of spaces that result from sequential plans generated by planning rules *g* and *h* in Figure 17.

Rule *a* is an expansion rule. It means that the action to *configure* an object should be expanded into the conjunction of several actions placing the components of that object. Rule *b* orders a set of such actions according to the valency of the rooms on the adjacency network. This assumes that the placement of rooms with the most interconnections is somehow critical and that the placement of these rooms should be accomplished first. Other rules could be devised that order on some other basis, such as size. Rule *c* simply says to anchor the first room on the "work plane."

Rule *d* states: if there is a room *B* and, according to the set of adjacency relationships it should be linked to a room *A*, and *A* has already been anchored, then substitute *put(B, next_to, A)* for *put(B)*. This is equivalent to placing objects adjacent to objects in place and with which they should be linked. Rule *e* arbitrarily locates the second object to the *east* of the first object. This is a simple expedient. It is considered that other configurations can be handled by the rotation and reflection of the entire configuration. Rules *f* and *g* are examples of rules for orientating the rest of the rooms. They take

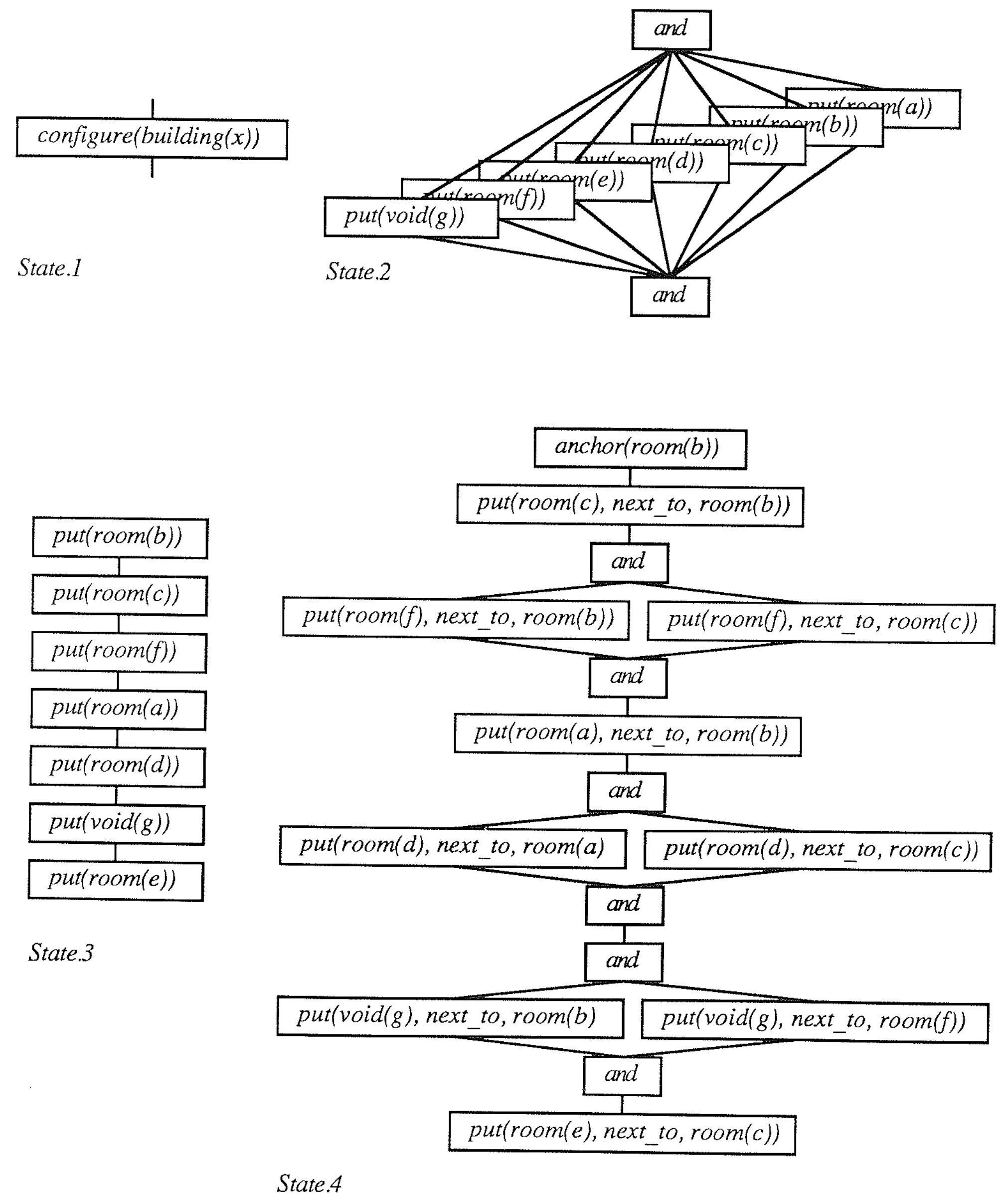

configure(building(x))
State.1
and
put(room(a))
put(room(b))
put(room(c))
put(room(d))
put(room(e))
put(room(f))
put(void(g))
and
State.2
put(room(b))
put(room(c))
put(room(f))
put(room(a))
put(room(d))
put(void(g))
put(room(e))
State.3
anchor(room(b))
put(room(c), next_to, room(b))
and
put(room(f), next_to, room(b))
put(room(f), next_to, room(c))
and
put(room(a), next_to, room(b))
and
put(room(d), next_to, room(a)
put(room(d), next_to, room(c))
and
and
put(void(g), next_to, room(b)
put(void(g), next_to, room(f))
and
put(room(e), next_to, room(c))
State.4

Figure 19. States in the development of a plan for generating a floor layout.

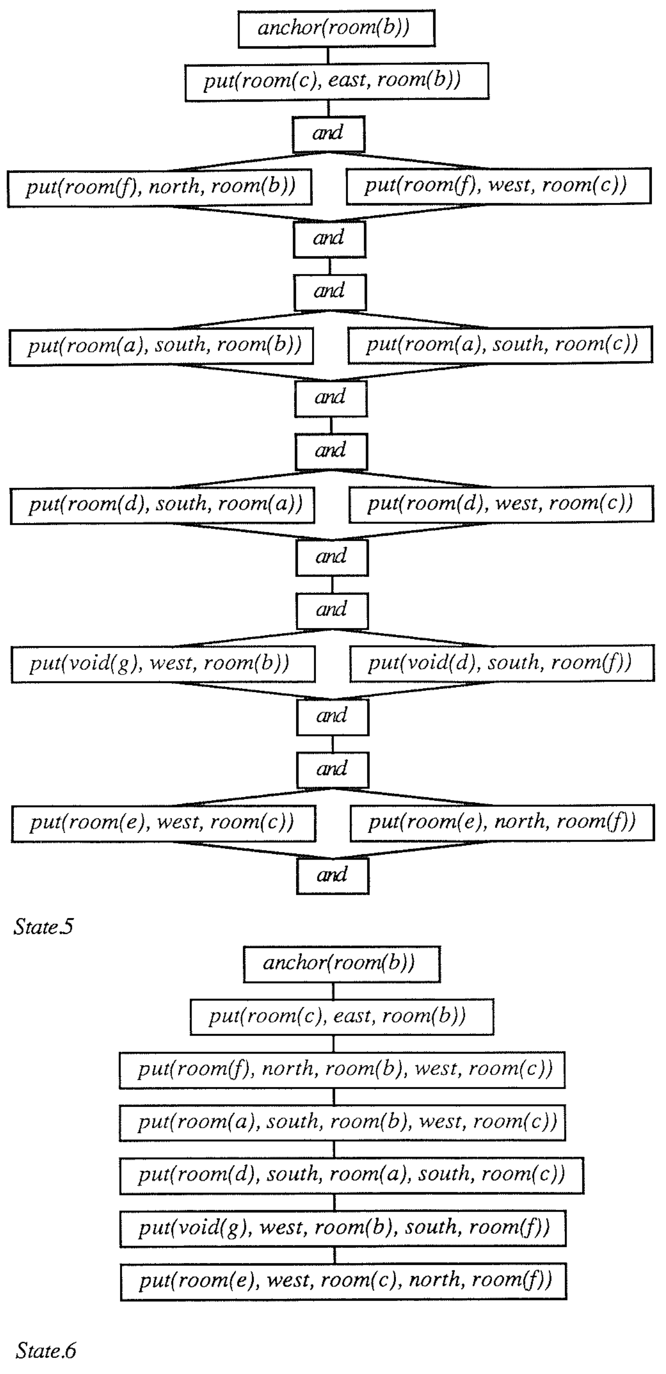

anchor(room(b))
put(room(c), east, room(b))
and
put(room(f), north, room(b))
put(room(f), west, room(c))
and
and
put(room(a), south, room(b))
put(room(a), south, room(c))
and
and
put(room(d), south, room(a))
put(room(d), west, room(c))
and
and
put(void(g), west, room(b))
put(void(d), south, room(f))
and
and
put(room(e), west, room(c))
put(room(e), north, room(f))
and
State.5
anchor(room(b))
put(room(c), east, room(b))
put(room(f), north, room(b), west, room(c))
put(room(a), south, room(b), west, room(c))
put(room(d), south, room(a), south, room(c))
put(void(g), west, room(b), south, room(f))
put(room(e), west, room(c), north, room(f))
State.6

Figure 19. (*cont.*)

the *next_to* relationships and refine them into more specific relationships, depending on the relative sizes of the rooms. Rule *f* therefore states that if room *A* is to be *next_to* room *B* and also room *C* and the action for putting *B* west of *C* has already been established, and *A* is to be the larger of the three rooms, then place *A* north of *B* and north of *C* (provided nothing has already been placed there). Rule *g* applies if *B* is to be larger than *A* and *C*. The effects of these alternative rules on the eventual spatial configuration is shown graphically in Figure 18. The rules for achieving this and other orientations can be represented in a single rule, but they have been separated here for clarity.

Rule *h* states that if there are no other rules to determine the placement of room *B* the action *put(B, next_to, A)* can be expanded to the conjunction of two actions as shown. *B* is simply located adjacent to *C* and the room next to which *C* is already positioned. Rule *i* replaces the conjunction of two actions with a single action. Implicit in these rules is the general strategy to proceed from less specific to more specific actions (that is, *next_to* relationships to *north, south, east* and *west* directional relationships), and the strategy of using triangulation for fixing objects in place. That is, objects are placed in relation to at least two objects already in place.

These are examples of the types of rules that can be employed in the manipulation of plans of simple design actions. The rules are intended to reduce the likelihood of conflicts between actions by means of simple heuristics. Where conflicts occur these can be handled by means of rules after the fashion of the critics of the NOAH system[26] that might adjust the network by reordering actions. Rules that specifically handle conflict resolution have not been illustrated here, but these operate in the same way as the other rules of the grammar.

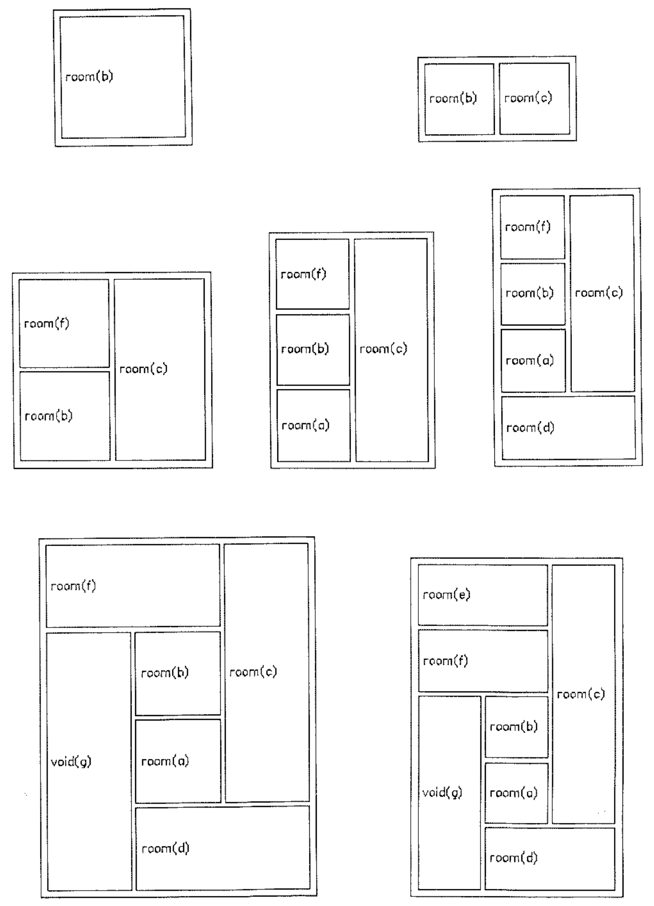
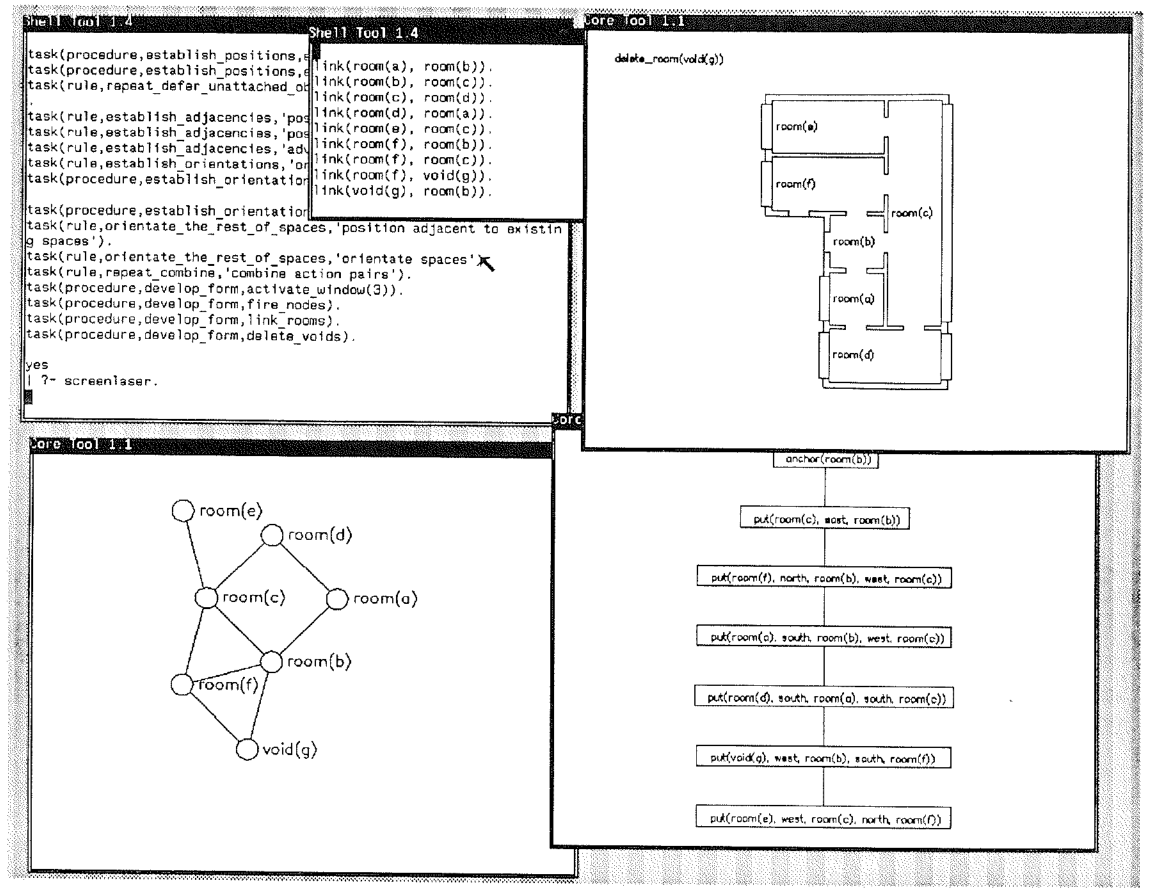


Figure 20. States in the generation of a floor layout.

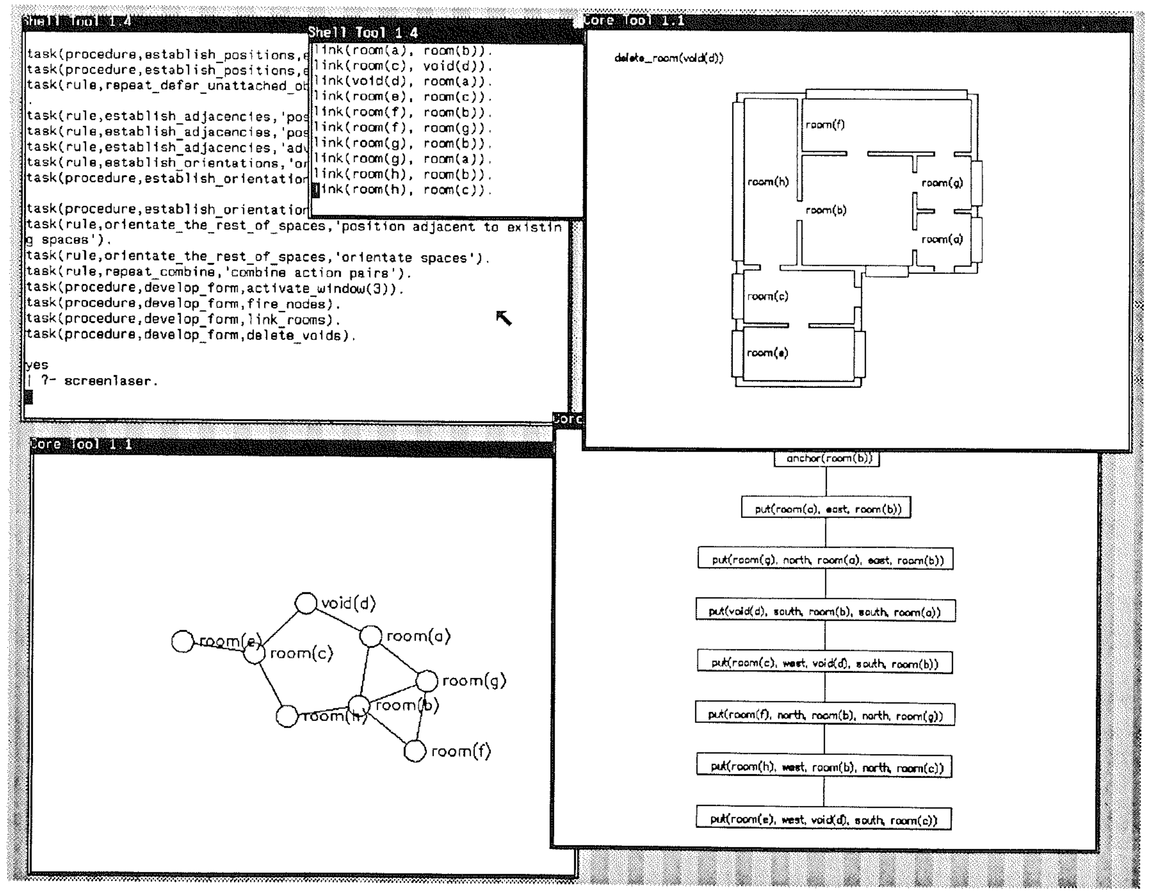


```
task(procedure,establish_positions,e
task(procedure,establish_positions,e
task(rule,repeat_defer_unattached_ob
.
task(rule,establish_adjacencies,'pos
task(rule,establish_adjacencies,'pos
task(rule,establish_adjacencies,'adv
task(rule,establish_orientations,'or
task(procedure,establish_orientation

task(procedure,establish_orientation
task(rule,orientate_the_rest_of_spaces,'position adjacent to existin
g spaces').
task(rule,orientate_the_rest_of_spaces,'orientate spaces')
task(rule,repeat_combine,'combine action pairs').
task(procedure,develop_form,activate_window(3)).
task(procedure,develop_form,fire_nodes).
task(procedure,develop_form,link_rooms).
task(procedure,develop_form,delete_voids).

yes
| ?- screenlaser.
```

```
link(room(a), room(b)).
link(room(b), room(c)).
link(room(c), room(d)).
link(room(d), room(a)).
link(room(e), room(c)).
link(room(f), room(b)).
link(room(f), room(c)).
link(room(f), void(g)).
link(void(g), room(b)).
```

```
anchor(room(b))
put(room(c), east, room(b))
put(room(f), north, room(b), west, room(c))
put(room(a), south, room(b), west, room(c))
put(room(d), south, room(a), south, room(c))
put(void(g), west, room(b), south, room(f))
put(room(e), west, room(c), north, room(f))
```

Figure 21. Screen display of a planning system.



```
task(procedure,establish_positions,e
task(procedure,establish_positions,e
task(rule,repeat_defer_unattached_ob
.
task(rule,establish_adjacencies,'pos
task(rule,establish_adjacencies,'pos
task(rule,establish_adjacencies,'adv
task(rule,establish_orientations,'or
task(procedure,establish_orientation

task(procedure,establish_orientation
task(rule,orientate_the_rest_of_spaces,'position adjacent to existin
g spaces').
task(rule,orientate_the_rest_of_spaces,'orientate spaces').
task(rule,repeat_combine,'combine action pairs').
task(procedure,develop_form,activate_window(3)).
task(procedure,develop_form,fire_nodes).
task(procedure,develop_form,link_rooms).
task(procedure,develop_form,delete_voids).

yes
| ?- screenlaser.
```

```
link(room(a), room(b)).
link(room(c), void(d)).
link(void(d), room(a)).
link(room(e), room(c)).
link(room(f), room(b)).
link(room(f), room(g)).
link(room(g), room(b)).
link(room(g), room(a)).
link(room(h), room(b)).
link(room(h), room(c)).
```

```
anchor(room(b))
put(room(a), east, room(b))
put(room(g), north, room(a), east, room(b))
put(void(d), south, room(b), south, room(a))
put(room(c), west, void(d), south, room(b))
put(room(f), north, room(b), north, room(g))
put(room(h), west, room(b), north, room(c))
put(room(e), west, void(d), south, room(c))
```

Figure 22. Output from a set of contextual requirements involving eight spaces.

Some of the rules are independent of any domain and would appear to be universally applicable; others might be regarded as idiosyncratic and of limited applicability. The programming environment is such that the system is not entirely dependent on the appropriateness of the rules. The knowledge base (rules) can be enhanced and developed without detriment to the overall framework. An example of the development of a plan of actions employing the rules of Figure 17 is shown in Figure 19.

Six states in the development of the plan are shown. *State.1* is the initial state; this is expanded to the conjunction of actions in *state.2* derived from rule *a* of Figure 17. *State.3* depicts the ordering produced by rule *b;* it arbitrarily selects one of the two spaces with equivalent highest valencies to be positioned first. *State.4* results from the application of rule *c*, and the repeated application of rules *d* and *e*. *State.4* therefore depicts a sequence of actions starting with the placement of the most important space. Subsequent actions locate rooms next to other rooms already in place and with which they are to be linked. *State.5* is a plan produced by the repeated application of rules *f, g,* and *h*. The repeated application of rule *i* produces the end *state.6*.

When this sequence of actions is interpreted by the language of form it results in the states depicted in Figure 20. The final state is a floor layout. Further rules can be applied to delete voids, to draw openings between spaces that are linked, and to position windows.

Figures 21 and 22 show a computer graphics environment devised primarily for experimenting with planning grammars. The computer screen is divided into "windows" displaying information about context, schedules, plans, and the development of the form of the artifact. Figure 21 depicts the layout developed in Figure 20, and Figure 22 shows a different layout generated from another body of contextual information. This environment can be extended to cover each of the meta-languages discussed above. The intention is to create a flexible environment in which contextual information, knowledge, and system processes are visible and amenable to modification and development.

CONCLUSION

Meta-languages have been discussed as a way of formalizing the complex mappings between meaning and artifact in design, such that designs can be generated that exhibit desired attributes. The rules of a grammar that operate on a vocabulary of form are considered as actions. We have considered how other grammars might operate on those actions. The assumption is made that designers are readily able to articulate such grammars and that they constitute a type of knowledge about design. Some of the semantic relationships between these actions and between actions and design attributes can also be known. Some attempt has been made to demonstrate how this view can form the foundation of a knowledge-based computer system. In the example, three levels of language and meta-language were discussed as a way of structuring knowledge: a language of form, a language of actions, and a language of plans, although the languages could be defined otherwise.

A basic principle being exercised here is that of exploiting the redundancies inherent in the way designers (or any problem solvers) view the world. Design systems can therefore operate on multiple abstractions of the world, such as adjacency graphs, procedural representations (at various levels), and formal and geometrical arrangements of spaces.

The implementational advantage of this multilevel view of design is that languages can be made to resemble what we have termed well-behaved systems, that is, systems that require only simple control mechanisms. The ideas put forward here are not dependent on the sophistication of the grammars described, although more complex grammars may be required to demonstrate conclusively the utility of this view.

In this paradigm design processes require grammars that operate upon other grammars not only by the selection of rules but in the creation and modification of rules. Further investigation into the dynamics of the design process is required in order to model effectively the exploratory nature of design.

This work has been supported by the Australian Research Grants Scheme and a Sydney University Postgraduate Research Studentship. Editorial assistance was given by Fay Sudweeks.

References

1. Harrison, M. A., *Introduction to Formal Language Theory*, Addison-Wesley, Reading, 1978.
2. Winograd, T., *Language as a Cognitive Process*, Addison-Wesley, Reading, 1983.
3. Edwards, A. T., *Architectural Style*, Faber and Gwyer, London, 1926.
4. Zevi, B., *The Modern Language of Architecture*, University of Washington Press, Seattle, 1978.
5. Dahl, O.-J., E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, London, 1972.
6. Johnson, J. H., "Hierarchical Structure in Design," *Design Policy Conference Proceedings*, Vol. 3, Design Council, London, 1984, pp. 51–59.
7. Doyle, J., "A Truth Maintenance System," in B. L. Webber and N. J. Nilsson (eds.), *Readings in Artificial Intelligence*, Tioga, Palo Alto, 1981, pp. 496–516.
8. Buchanan, B. G., "New Research on Expert Systems," in J. E. Hayes and D. Michie (eds), *Machine Intelligence 10*, Wiley, London, 1982, pp. 269–299.
9. Hayes-Roth, F., D. A. Waterman, and D. B. Lenat, (eds.), *Building Expert Systems*, Addison-Wesley, Reading, 1983.
10. Bijl, A., "An Approach to Design Theory," in H. Yoshikawa (ed.), *Preprints of Design Theory of CAD*, Tokyo University, Tokyo, 1985, pp. 1–23.
11. Akiner, T., *TOPOLOGY1: A System That Reasons About Objects and Spaces in Buildings*, Unpublished Ph.D. Thesis, Sydney University, 1985.
12. O'Cathain, C. S., "Why is Design Logically Impossible?" *Design Policy Conference Proceedings*, Vol. 3, Design Council, London, 1984, pp. 33–36.
13. Chomsky, N., *The Logical Structure of Linguistic Theory*, Plenum Press, New York, 1975.
14. Alexander, C., *A Pattern Language*, Oxford University Press, London, 1977.
15. G. Stiny and J. Gips, *Algorithmic Aesthetics*, University of California Press, 1978.
16. Mitchell, W. J., "Synthesis with Style," *PArC79*, AMK, Berlin, 1979, pp. 119–134.
17. Hornstein, N., *Logic as Grammar*, MIT Press, Cambridge, 1984.
18. Nilsson, N. J., *Principles of Artificial Intelligence*, Springer-Verlag, Berlin, 1982.
19. Coyne, R. D., and J. S. Gero, "Design Knowledge and Sequential Plans," *Environment and Planning B*, 1985, Volume 12, pp. 401–418.
20. Mitchell, W. J., "The Theoretical Foundation of Computer-Aided Architectural Design," *Environment and Planning B*, Vol. 2 (1975), pp. 127–150.
21. Simon, H. A., "The Structure of Ill-Structured Problems," *Artificial Intelligence*, Vol. 4 (1973), pp. 181–201.
22. Davis, R., and J. King, "An Overview of Production Systems," in E. W. Elcock and D. Michie (eds.) *Machine Intelligence 8*, Ellis Horwood, Chichester, 1977, pp. 300–332.
23. Gips, J., and G. Stiny, "Production Systems and Grammars: A Uniform Characterisation," *Environment and Planning B*, Vol. 7 (1980), pp. 399–408.
24. Newell, A., and H. A. Simon, *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, 1972.
25. Stiny, G., and W. J. Mitchell, "The Palladian Grammar," *Environment and Planning B*, Vol. 5 (1978), pp. 5–18.
26. Sacerdoti, E. D., *A Structure for Plans and Behavior*, Elsevier, New York, 1977.
27. Sussman, G. J., *A Computer Model of Skill Acquisition*, Elsevier, New York, 1975.
28. Appelt, D. E., "Planning English Referring Expressions," *Artificial Intelligence*, Vol. 26 (1985), pp. 1–33.
29. Gero, J. S., and R. D. Coyne, "Knowledge-Based Planning as a Design Paradigm," in H. Yoshikawa (ed.), *Preprints of Design Theory in CAD*, Tokyo University, Tokyo, 1985, pp. 261–295.
30. Hayes-Roth, F., and V. R. Lesser, "Focus of Attention in the Hearsay-II System," *Proc. IJCAI5* 1977, pp. 27–35.
31. Hayes-Roth, B., and B. Hayes-Roth, "A Cognitive Model of Planning," *Cognitive Science*, Vol. 3, No. 4 (1979), pp. 275–309.
32. Hayes-Roth, B., "A Blackboard Architecture for Control," *Artificial Intelligence*, Vol. 26 (1985), pp. 251–321.
33. Quinlan, J. R., "An Introduction to Knowledge-Based Expert Systems," *The Australian Computer Journal*, Vol. 12, No. 2 (1980), pp. 56–62.
34. Barr, A., and E. A. Feigenbaum, *The Handbook of Artificial Intelligence*, Pitman Books, London, 1981, Vols. I–III.
35. Stiny, G., "A Note on the Description of Designs," *Environment and Planning B*, Vol. 8 (1981), pp. 257–267.
36. Mitchell, W. J., J. P. Steadman, and R. S. Liggett, "Synthesis and Optimization of Small Rectangular Floor Plans," *Environment and Planning B*, Vol. 3 (1976), pp. 37–70.
37. Earl, C. F., "A Note on the Generation of Rectangular Dissections," *Environment and Planning B*, Vol. 4 (1977), pp. 241–246.
38. Flemming, U., "Wall Representations of Rectangular Dissections and Their Use in Automated Space Allocations," *Environment and Planning B*, Vol. 5 (1978), pp. 215–232.
39. Steadman, J. P., *Architectural Morphology*, Pion, London, 1983.
40. Balachandran, M., and J. S. Gero, "Dimensioning of Architectural Floor Plans under Conflicting Objectives," *Working Paper*, Department of Architectural Science, University of Sydney, 1985.
41. Wilensky, R., "Meta-Planning: Representing and Using Knowledge about Planning in Problem Solving and Natural Language Understanding," *Cognitive Sciences*, Vol. 5 (1981), pp. 197–233.
42. Stefik, M., "Planning and Meta-Planning," *Artificial Intelligence*, Vol. 16 (1981), pp. 141–169.
43. Davis, R., "Meta-rules: Reasoning about Control," *Artificial Intelligence*, Vol. 15 (1980), pp. 179–222.
44. Clocksin, W. F., and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, Berlin, 1981.