Environment and Planning B: Planning and Design, 1985, volume 12, pages 401 - 418

Design knowledge and sequential plans

R D Coyne, J S Gero
Computer Applications Research Unit, Department of Architectural Science, University of Sydney, NSW 2006, Australia
Received 31 August 1984; in revised form 25 March 1985

Abstract. The application of sequential planning to the design process is discussed, considering design as a search through a space of states which are acted upon by transformation rules. Various approaches to goal satisfaction are considered, including forward inference, regression, the satisfaction of implicit goals, and metaplanning. These issues are illustrated with an example from a simple design domain. The example has been implemented in PROLOG.

Introduction

Meaning can be extracted from an artifact when we analyse it, determine why its parts are arranged in a particular way, and discover something about its intended or assumed purpose. When we *create* an artifact we instill meaning into it by organising its parts to express some sort of purpose. The parts which make up the whole are selected and configured to achieve some end. We here equate 'meaning' with 'intentionality'. The idea of capturing design knowledge in such a way that the generation of designs can be achieved mechanistically has fascinated artists and artisans for some time, and various methods have been devised for producing complex designs from the combinatorial possibilities presented by a set of simple rules. The utility of rules as a basis of human problem solving, and as a way of modeling human behaviour, has been widely recognised. Attention must be directed towards the selection and ordering of rules if a mechanistic model of the design process is to take account of the meaning of artifacts (Gero and Coyne, 1985; Gero et al, 1984).

Design as state space search

Newell and Simon (1972) discuss human problem solving as information processing, whereby a set of operators act on patterns to produce solution states. We can regard a pattern as a set of characteristics belonging to some object which enable us to recognise similarities with other objects. Patterns also enable us to copy objects. They need describe not only dimensional attributes (as with a dressmaker's pattern), but also topological relationships between elements: colour, texture, orientation, temperature, etc. It is generally understood that humans are extraordinarily adept at recognising patterns. The term 'cognitive map' is often used to describe a network of patterns and associations. This has been discussed in the context of human perception of the environment (Downs and Stea, 1973; Lynch, 1960), but it can be argued that it is generally applicable to all types of mental activity (Kaplan and Kaplan, 1982). We have difficulty, however, in externalising patterns, that is, describing patterns that enable us to class similar objects together; and even greater difficulty in representing the knowledge which enables us to recognise patterns. The implementation of a mechanistic view of the design process as transformation rules acting on patterns will undoubtedly be simplistic, representing but a pale imitation of human cognitive activity, yet, we find that at a pragmatic level this model provides a basis for demonstrably workable systems. Empirical studies of designers at work have been discussed in the context of this model, notably by Eastman (1970) and Akin (1978).

Design states, therefore, represent conglomerations of patterns. The designer is able to recognise patterns contained within the design state which match those contained within some mental repository of patterns. A strong association between patterns prompts the designer to replace one pattern with another. These associations can be regarded as transformation rules, the general form of which is

IF pattern(X) THEN REPLACE WITH pattern(Y).

Transformation rules are important as a basis for describing language understanding and other cognitive activities (Chomsky, 1975), and provide a powerful way of representing design knowledge. In this context we consider transformation rules as 'design rules'.

Design theorists, such as Alexander, have given formal expression to design rules in a practical context (Alexander et al, 1968), although with little thought as to how such rules may be used entirely mechanistically. A 'shape grammar' is the name given to a collection of formal rules for manipulating patterns of lines and labeled points to produce designs resembling architectural plans after the style of a certain body of architectural work (Stiny and Mitchell, 1978). This latter approach is of interest as it demonstrates the degree of precision with which rules have to be described to facilitate mechanistic design generation.

The model of design as transformation rules acting on patterns within design states requires an understanding of how goal states are achieved. Problem-solving activity is goal-directed. We are trying to find a state which contains a particular pattern, and the process of generation can cease when that pattern is achieved. There may also be 'end states' that are nongoal states, that is, there are no more rules that can be applied and yet the goal state is not achieved. The general mechanism for satisfying goals under such circumstances is backtracking. This involves a return to an earlier state, where a different rule can be tried. This results in the exploration of alternative solution paths. Design activity is therefore described as a goal-directed search through a space of states or potential design solutions.

Planning

The above model is complete in that it describes how, with a set of rules and a goal set, we are certain to generate designs. In practice, however, it is insufficient for producing solutions to real design problems. There are high costs, in terms of computational time, associated with forward search and backtracking. The process of transforming one state into another may be costly; the cost of 'undoing' partial designs (that is, backtracking to earlier states) may be expensive; and there may just be too many intermediate states requiring evaluation to be practicable (that is, the space of design states may be subject to a combinatorial explosion).

One mechanism for reducing the number of 'branches' in a design space (and therefore the amount of generating and backtracking) is to evaluate intermediate states for their potential to develop into the goal state and to investigate only those branches that are likely to lead to a goal state. This task is not as easy as it may seem at first. The criterion used to evaluate when a goal state has been reached is generally inappropriate as an evaluator of intermediate states. The design problem illustrated in figure 1 demonstrates this. The goal consists of two subgoals: create a structure which spans the canal and which connects *A* to *B*. The initial state is S: a canal and five structural elements—two pylons and three planks. If we adopt a naive approach, using the goal as an evaluator for intermediate states, it may seem that the intermediate state M will lead to the goal state (as it satisfies the subgoal 'span canal'), whereas the intermediate state N is considered unsuitable. (After all, why build upwards when you want to go across?) The end state G shows that, in fact, state N

lies on the path leading to the achievement of the goal. The knowledge required for evaluating intermediate states therefore has to be stated explicitly as it can rarely be inferred from the end goals. No doubt an experienced designer possesses a strong faculty for effectively evaluating intermediate states, but the representation of this knowledge places a burden on any mechanised design system. We therefore turn to other devices for reducing the cost of goal achievement.

Already, in the above example, we have utilised one of the major components of a planning system, that of abstraction. The design states were explored by considering only certain selected attributes of planks and pylons. For artifacts that are created in the 'ground' level of abstraction the cost of backtracking tends to be high (although it is less for sand castles than for marble sculptures). It is obviously cheaper to design buildings by considering only certain attributes of walls and spaces, on paper, before committing the design to bricks and concrete, but it may also be appropriate to consider higher levels of abstraction again. We discuss this below. The second mechanism is that of working backwards from goals. This is similar to spanning the canal with the planks and then working out how to support them. It will be noted that this type of exploration is one that can be carried out only in an abstraction of the real world. For most domains, working backwards from goals involves less searching and in some instances can reduce a complex problem to a trivial one.

This is the process of planning: determining sequences of actions that enable goals to be satisfied. The actions, of course, are the design rules. The theory of planning, in artificial intelligence, stems mainly from an interest in determining robot actions, but also proves applicable to design theory. A comprehensive discussion of the general theory of planning is given by Nilsson (1982). We develop this idea of the role of sequential planning in design by means of an example.

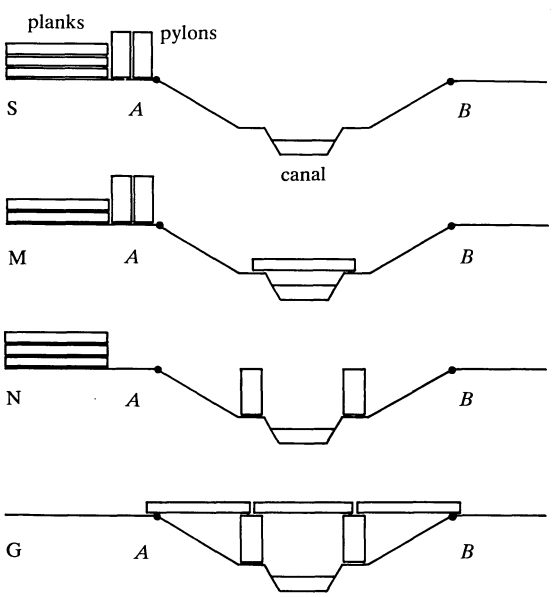

Figure 1. States in a bridge-building domain.

Planning in design

Figure 2 shows a set of design rules for generating simple building forms from wall elements and columns. The elements form rooms, each of which has a name and an orientation. A dot indicates the top of a room. The design space is depicted in

figure 3 as a tree structure, with the root node as the initial state (the fact 'start') and the ends of the branches as various end states, where no more rules can be applied. It will be noted that the application of certain rules precludes the achievement of certain states. So the end states are all different. The design space of this set of rules is atypical of most design domains in that it is small enough to be represented on one page.

One of the rules, rule(2), is depicted in program form as follows:

rule(2) # [*match*: *room_a*(1), *match*: *short_wall*(2)]
>>
[*copy*: *room_a*(1), *copy*: *room_c*(2), *copy*: *short_wall*(4), *copy*: *short_wall*(5), *copy*: *long_wall*(5), *copy*: *open_wall*(2)].

The rule name is given first, followed by a list of preconditions. These preconditions must be satisfied before the rule can be applied or 'fired'. The second list is a set of actions or consequents. When the rule is fired, the objects matched in the precondition part of the rule are deleted from the current description of the world and replaced with the objects on the consequent side of the rule. The rule is interpreted by a general-purpose design system written in the logic programming language, PROLOG (Clocksin and Mellish, 1981). The rule is also shown diagrammatically in figure 4. Here a room is defined by four columns. For this rule to be fired, it is necessary to find patterns in the current design state which match the descriptions of room_*a*(1) and short_wall(2). These objects are deleted from the description of the design state and replaced with the objects room_*a*(1), again, room_*c*(2), short_wall(4), short_wall(5), long_wall(5), and open_wall(2). The arguments denote different instances of the same object. The elements used in the rules have real coordinates in a two-dimensional world and it is possible to construct different rules from a vocabulary of such objects.

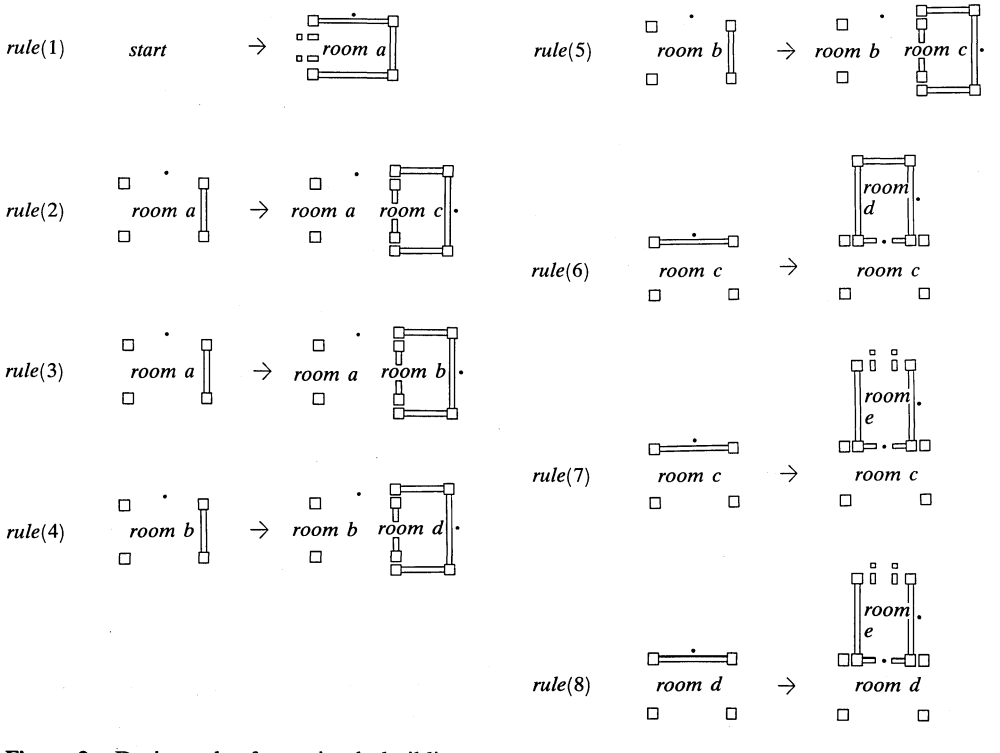

Figure 2. Design rules for a simple building.

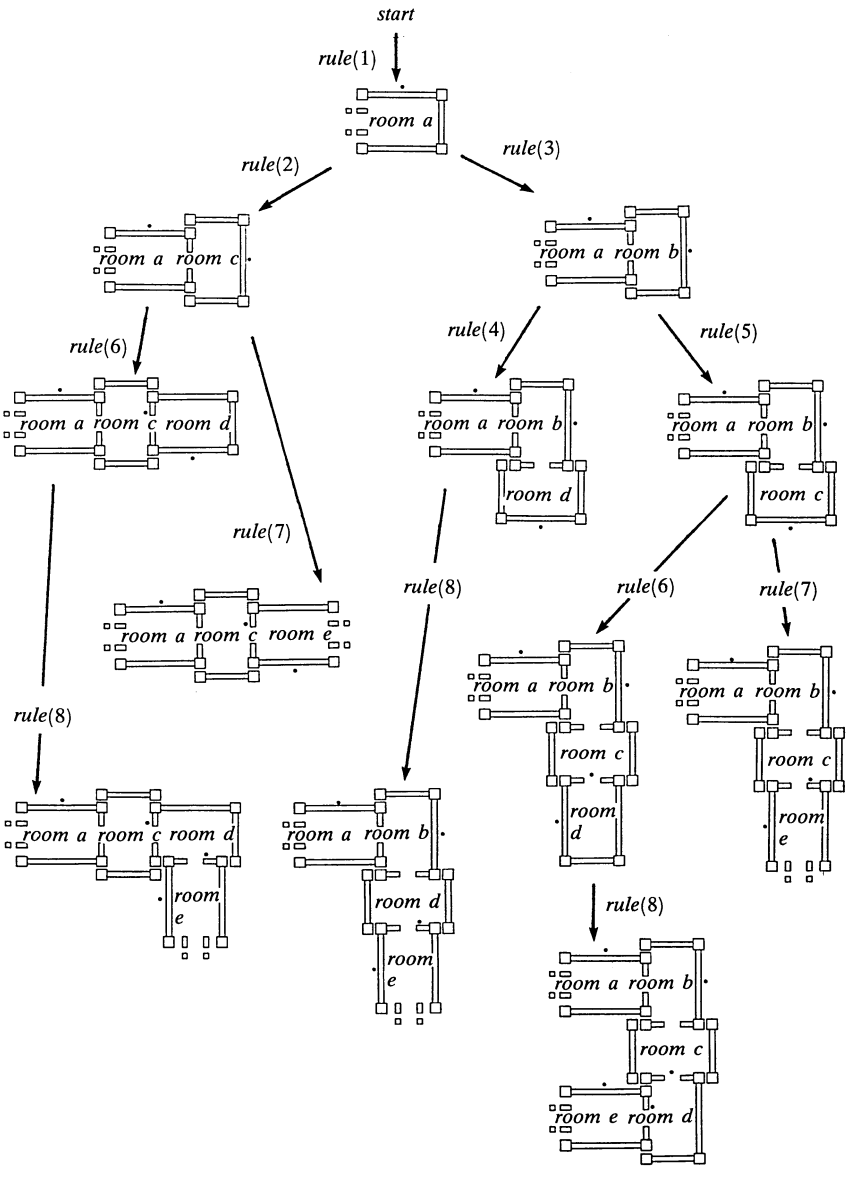

Figure 3. State space for the rules in figure 2.

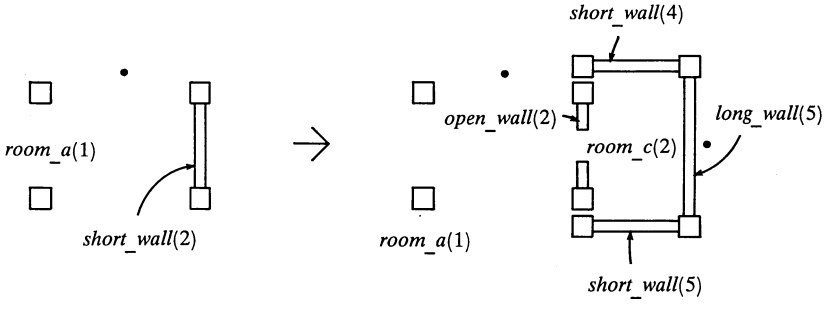

Figure 4. A graphical representation of rule(2).

The current state is depicted in terms of facts, in predicate calculus notation. Facts are therefore deleted and asserted with the firing of each rule. The representation of objects is essentially hierarchical: for example, room_*a*(1) is a composite object made up of the objects column(1), column(2), column(3), and column(4); these objects consist of line segments, which are defined by points expressed in homogeneous coordinate notation:

composite(*room_a*(1), [*column*(1), *column*(2), *column*(3), *column*(4)]).
object(*column*(1), [*s*(1), *s*(2), *s*(3), *s*(4)]).
.
.
.
line(*s*(1), [*p*(1), *p*(2)]).
.
.
.
point(*p*(1), [260, 200, 1]).
point(*p*(2), [260, 280, 1]).

A composite object in a rule matches the current state if there exists a composite object described in the facts base which has the same predicate name, but different instance number, and both are made up of similar objects. The geometrical transformation which enables the rule object to be mapped onto the object in the state description is calculated from the points defining the ends of the line segments. This task is made simpler by restricting transformations to translations and rotations of 0°, 90°, 180°, and 270°. So here we do not consider mappings between objects that involve reflection or changes of scale. A representation of the knowledge required for this type of pattern matching is given in appendix 1. The process of matching nongeometrical descriptions is relatively simple as it is necessary to find only a one-to-one match between a fact within a rule and the corresponding fact in the state description.

The costs involved in determining whether rules are to be fired, and the cost of backtracking, are fairly high in terms of computational time. It is therefore not entirely practical to use this abstraction space when searching for goal states. To reduce the need for costly backtracking it is necessary to derive the sequence of rules which will achieve particular goal states, by searching within a higher level of abstraction before implementing the rules to produce geometrical designs.

Suppose we have the goal that both room *b* and room *d* are to be located. This is a simple goal, but one that cannot be achieved without some backtracking, as figure 5 illustrates. To see if any state satisfies this goal it is not necessary actually to generate geometric descriptions as long as we can find a suitable rule abstraction. For example, rule(2) can be represented by using simple facts in the following way:

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

This produces the search space shown in figure 6. With rules in this form it is possible to generate abstracted design states and evaluate them for compliance with the goal, and to backtrack if necessary without actually generating geometrical descriptions.

Even when operating in an abstracted search space, however, forward reasoning, as described here, can be very inefficient, particularly when we consider that in a real

design problem it may be necessary to search many long branches before arriving at a state that matches the goals. The branches marked * in figure 5 could have been very long, for example. We therefore turn to reasoning backwards from goals within an abstraction space as a more effective method of planning.

Backward reasoning, or regression (the backward application of rules or operators), involves knowing what rules must be applied to satisfy goals. This knowledge is contained in the rules themselves. If the goal is 'located(room_*d*)', we find that there are two rules that satisfy this condition, rule(4) and rule(6). Selecting the first rule, we look to see what conditions must be satisfied before that rule can be implemented. These are that room *b* is located and that the right-hand side of room *b* is clear. These conditions can be satisfied by rule(3); and so we form a backward chain through these conditions until we come to the initial state. Regression is a trivial process when there is only one goal to be achieved, but becomes more difficult when there are more, as satisfaction of one goal can produce a state which precludes the satisfaction of another.

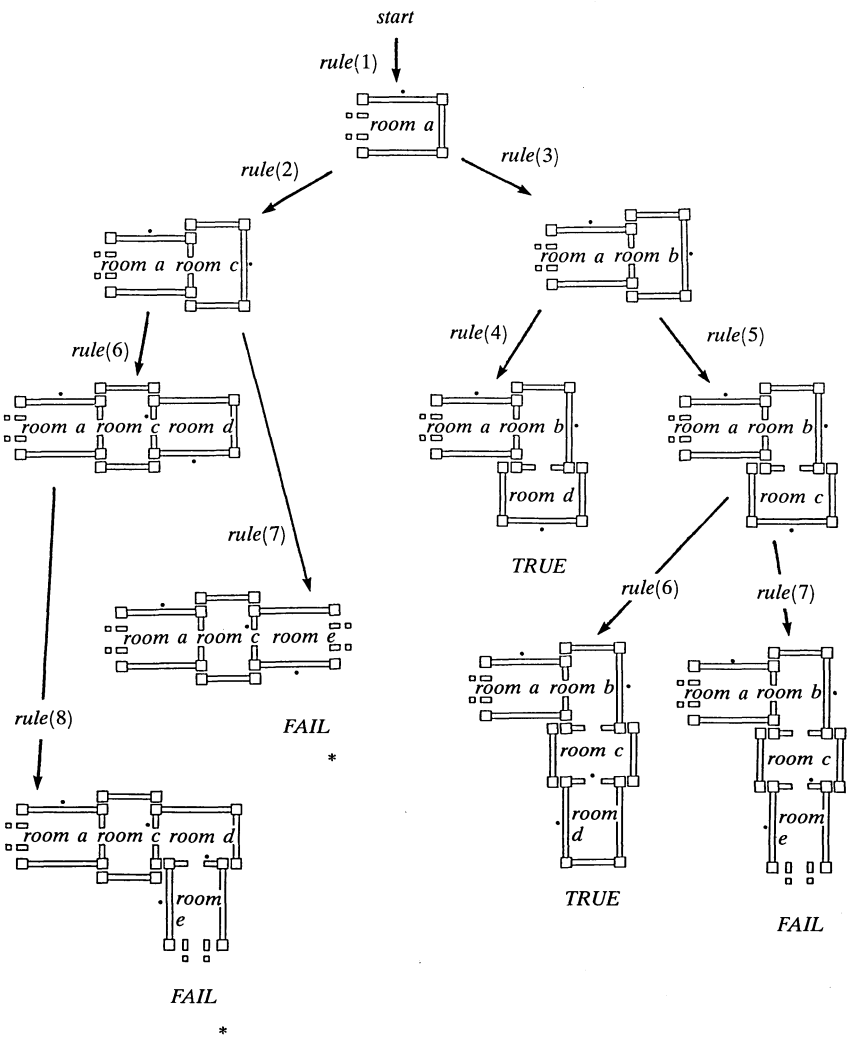

Figure 5. The search graph for satisfying the goal: located(room_*b*) and located(room_*d*).

The planning system STRIPS represents a general paradigm for backward chaining to satisfy multiple goals (Fikes and Nilsson, 1971). Kowalski (1979) has formulated the planning problem in such a way that it is able to be solved by a PROLOG-type theorem prover, and a planning system based on a different approach has been developed using PROLOG by Warren (1974). We present here a simple STRIPS-like planner which maintains a current description of the design state as well as a list of appropriate actions by which states are achieved. The planning knowledge is shown in appendix 2.

In figure 7 we illustrate the mechanism by which the multiple goals: located(room_*d*) and located(room_*b*) are satisfied. The boxes contain goals and subgoals. A downward arrow points to the subgoals which must be satisfied before a particular goal can be met. Goals are considered one at a time in a recursive fashion. A goal is satisfied when it matches the current state description or can be achieved by means of some rule. The condition 'start' is the first to be satisfied, as this matches the initial condition. Once it has been found that rule(1) achieves the subgoal, located(room_*a*), this rule can be used to change the current state. The subgoal, clear(room_*a*, right), matches the current state of the world and is therefore true (indicated by '*T*').

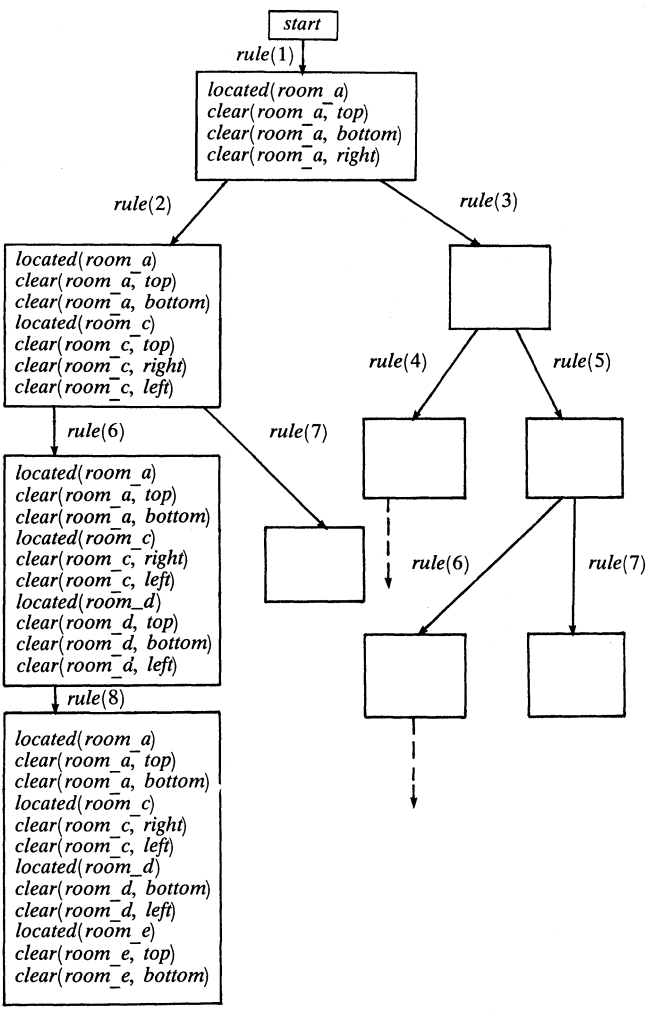

Figure 6. A further abstraction of the search space in figure 5.

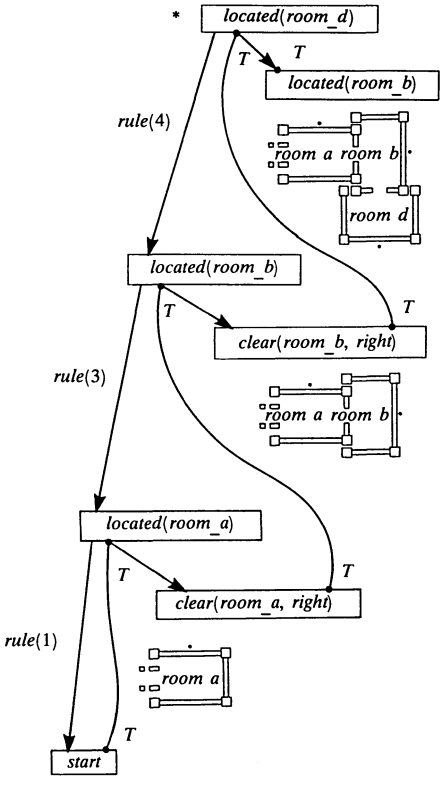

Figure 7. Backward search graph to satisfy the goal: located(room_d) and located(room_b).

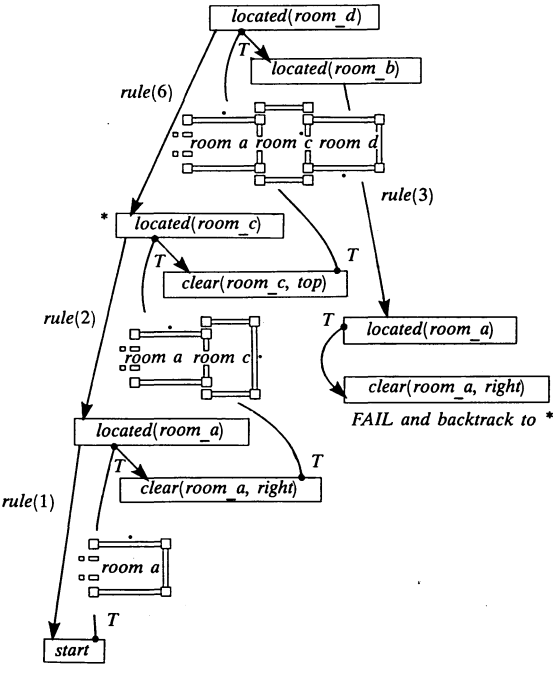

Figure 8. Continuation of the search graph in figure 7 to produce a second solution, resulting in failure.

The sequence of rules, rule(1), rule(3), rule(4), therefore results in a state that satisfies the goal: located(room_*d*). The second goal, located(room_*b*), also matches the current state. So both goals are satisfied.

It will be noted that the goal, located(room_*d*), could have been achieved by another rule: rule(6). It should be possible to generate other plans by backtracking to the last goal that can be achieved by another rule. In all of these examples that goal or subgoal is indicated by the symbol *. Figure 8 shows an attempt to achieve a second plan. The rule sequence, rule(1), rule(2), rule(6), satisfies the goal: located(room_*d*). The only rule that can be invoked to satisfy the second goal, located(room_*b*), is rule(3), which requires as its precondition that room *a* is located and that the right-hand side of room *a* is clear. As the second of these subgoals cannot be matched against the current state description, and as there is no rule which enables this goal to be satisfied, the plan as formulated so far fails. In a logic programming framework, a fail causes backtracking to the last goal which presents an alternative proof, that is, the last goal that is able to be achieved by means of another rule. This is the goal indicated *. Figure 9 shows how a new plan is constructed from that goal. In this case both goals: located(room_*d*) and located(room_*b*) are satisfied. Figures 10, 11, and 12 show the procedures to satisfy the same goals but in reverse

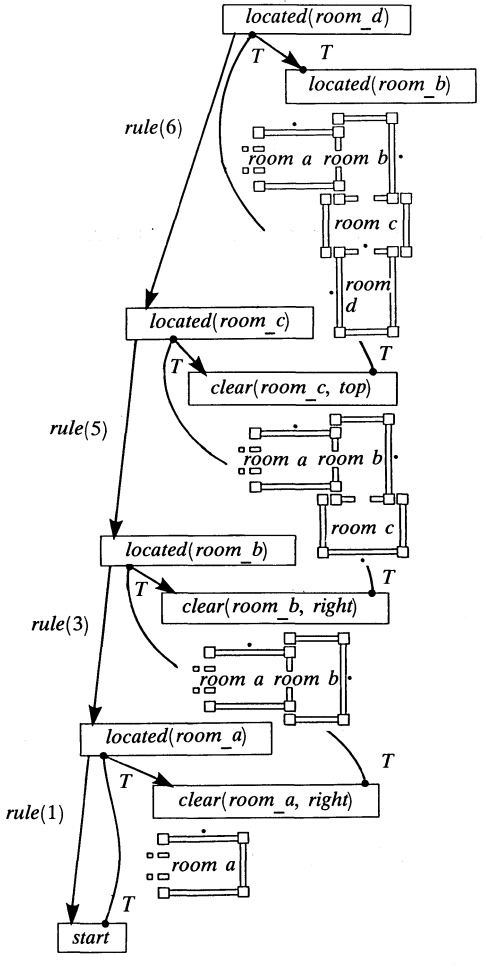

Figure 9. Continuation of the search graph in figure 8.

order, that is, 'located(room_b)' is before 'located(room_d)'. It should be noted that the representations of states in these diagrams are depicted as full geometrical entities, whereas the state descriptions maintained by the planner are actually similar to those abstractions shown in figure 6. The diagrammatic representation of states has been illustrated to make the figures easier to follow.

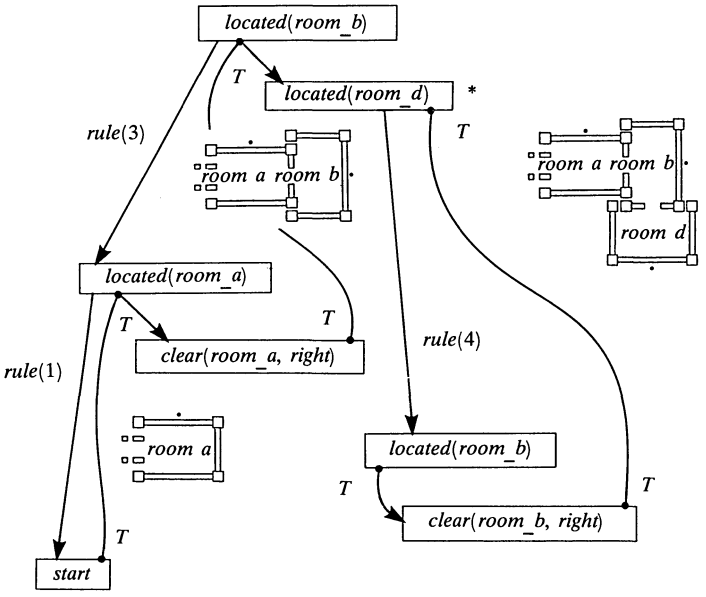

Figure 10. Backward search graph to satisfy the same goals as figure 7 but where the subgoals are expressed in reverse order.

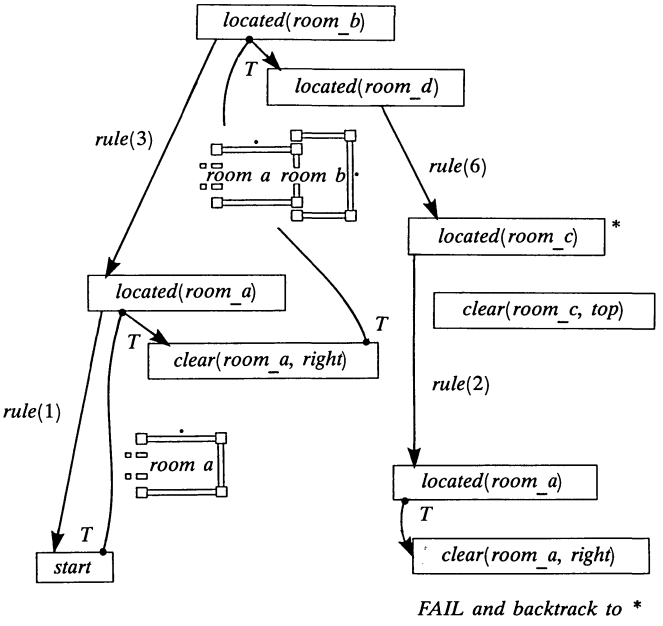

Figure 11. Continuation of graph in figure 10 to produce a second solution, resulting in failure.

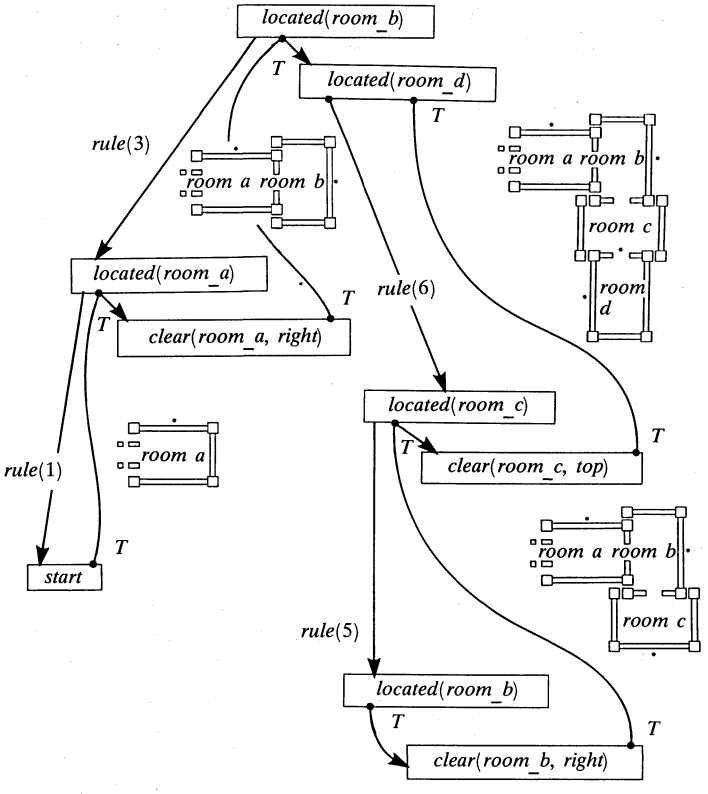

Figure 12. Continuation of the graph in figure 11.

Implicit goals

In most design domains it is unlikely that goals will be able to be matched against facts contained within a state description, that is, the goals are not expressed in the same terms as the facts describing states, so a one-to-one match is not possible. Under such circumstances it is necessary to use inference to determine whether a goal state has been achieved. We consider two approaches to this problem here, the use of knowledge about the design rules and the use of knowledge about design states. Both approaches involve augmenting the design knowledge (the design rules themselves) with some form of metaknowledge, that is, knowledge about that knowledge.

A simple example of this is where we wish to achieve the goal that two rooms are next to each other, for example, next_to(room_*c*, room_*d*). By looking at the rules we can determine something about them which will prove useful in achieving this goal. This can be expressed as the metarule:

> IF the goal is that two rooms are to be next to each other THEN the room in the precondition part of the rule is NEXT TO the room in the resultant part of the rule.

When our planning system is searching for a match with the goal: next_to(*A*, *B*), this metarule ensures that a 'next_to' fact appears on the resultant side of a design rule, by implication. The search tree which produces the first sequence of rules for satisfying this goal is illustrated in figure 13.

The second approach is to employ knowledge about design states in determining whether goals have been achieved. For example, we can infer from the abstracted state description that a particular room is located on a corner of the building by

noting if two walls from that room are clear and form a corner. This can be expressed in the same form as a design rule:

inference_rule(1) # [*clear*(*X*, *A*), *corner_wall*(*A*, *B*), *clear*(*X*, *B*)]
>>
[*corner_room*(*X*)].

Corner_wall(*A*, *B*) is derived by using the following knowledge, which states that *A* and *B* are corner walls if they are both found within a fact headed by the predicate 'corner':

corner_wall(*A*, *B*):— *corner*(*A*, *B*); *corner*(*B*, *A*).
corner(*top*, *right*).
corner(*top*, *left*).
corner(*bottom*, *right*).
corner(*bottom*, *left*).

Consider the goal: corner_room(room_*d*) and located(room_*e*). Figure 14 shows how the first rule that enables the goal, corner_room(room_*d*), to be satisfied is inference_rule(1). The preconditions that must be satisfied are the subgoals: clear(room_*d*, *A*), corner_wall(*A*, *B*), and clear(room_*d*, *B*). Note that these subgoals contain variables. These variables cannot be instantiated until the subgoals are matched against the current state description. So their values will not be known until a partial sequential plan for achieving these subgoals has been discovered. The rule sequence, rule(1), rule(3), rule(4), produces a design state that satisfies these conditions and instantiates the variable *A* to 'top' and *B* to 'left'. The goal, corner_room(room_*d*), is satisfied by this rule sequence, but the second part of the goal, located(room_*e*), fails. Figure 15 shows the rule sequence that results from backtracking. The sequence is: rule(1), rule(2), rule(6), rule(8).

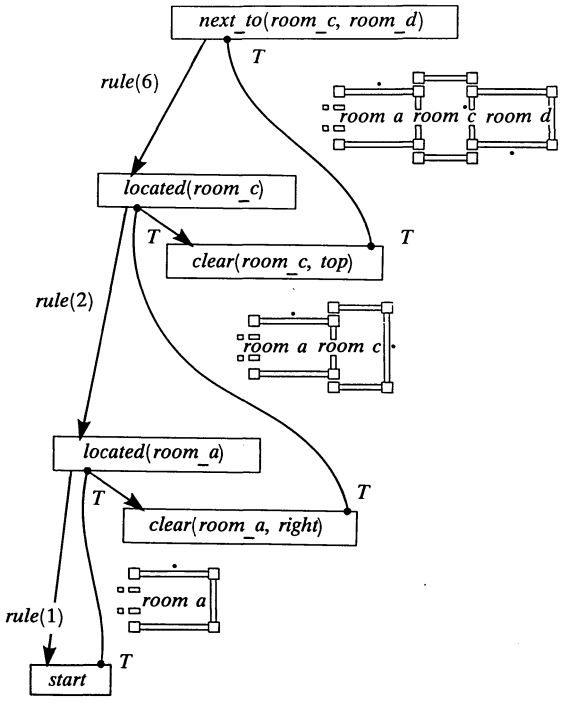

Figure 13. Search graph showing the satisfaction of the implicit goal: next_to(room_*c*, room_*d*).

Design goals are rarely expressed in these terms however. The aspirations of a designer for the object being created are generally articulated at a higher level. It becomes necessary to explicate the knowledge which enables high-level goals to be interpreted in terms that can be understood by a mechanistic planning system. Figure 16 illustrates this. Suppose that the building under consideration is some kind of public building. One of the ways in which the goal 'high public utility' can be achieved is by providing an attractive restaurant facility. This is achieved if the restaurant has good views and is accessible to the public. For the restaurant to have good views it should be on the corner of the building. This last goal is a 'system' goal, that is, one that can be understood by the planning system. The selection of appropriate planning goals may be regarded as an expert task and one that could be handled by an 'expert system'. The utility of expert systems in architectural design has been discussed elsewhere (Gero and Coyne, 1984). It should be possible to derive an appropriate goal set where goal conflicts are minimised and where, perhaps, goals are ordered in importance in some way, for use by a planning system.

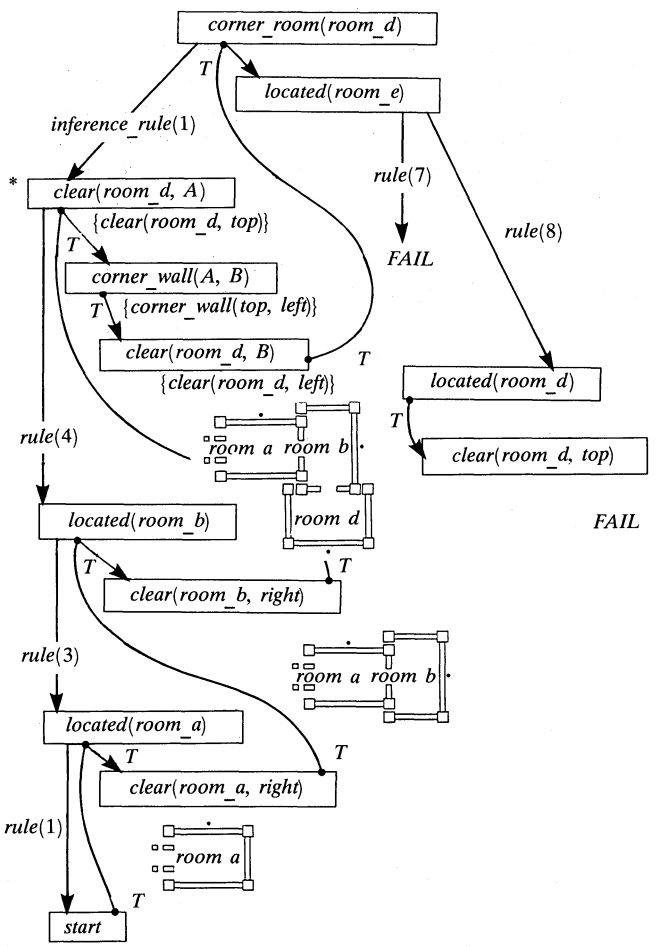

Figure 14. Search graph for the satisfaction of the implicit goal: corner_room(room_*d*) and located(room_*e*), resulting in failure.

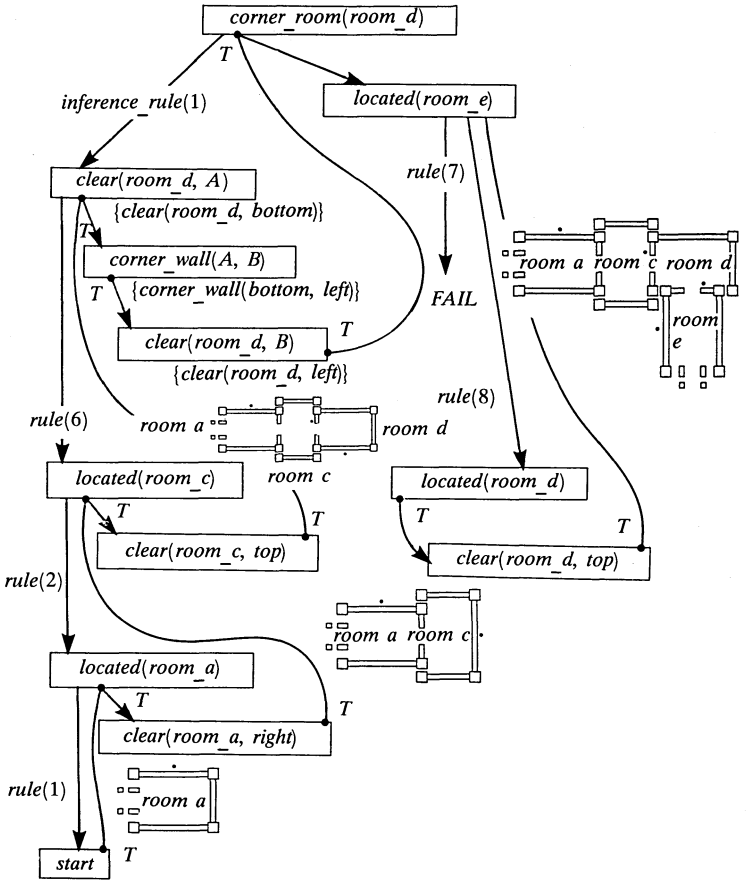

Figure 15. Continuation of the graph in figure 14.

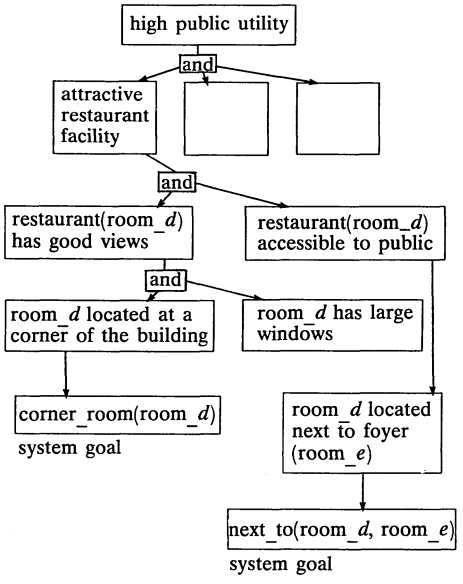

Figure 16. Part of the inference net for translating high-level goals to system goals.

Conclusion

We have considered the model of the design process as a search through a space of design states. The operators which transform one state into another are transformation rules, which in this context we have called 'design rules'. Design rules capture design knowledge that facilitates the mechanistic generation of designs. If such designs are to be imbued with any meaning then they must express some kind of purpose. Purposeful design is synonymous with goal achievement in this context. In realistic design domains the process of generating and backtracking to achieve goals is expensive. We have discussed the importance of selecting and ordering design rules within an appropriate level of abstraction as a means of reducing this cost. Regression is the process of chaining backwards from goal states to the initial state and proves effective in reducing the amount of search necessary. This becomes increasingly difficult when more than one goal has to be satisfied. A STRIPS-like planner is demonstrated as a mechanism for satisfying goals which produce conflicts. Implicit goals can be satisfied if we supplement the design rules with a type of metaknowledge with which we explicate some of our knowledge about the design rules themselves. We can also use our knowledge about design states to infer whether or not they achieve goals. The mechanism of inference also proves important in interpreting the aspirations of designers in terms that a planner can understand.

Within realistic design domains the mechanisms we have discussed so far for achieving goals are insufficient. The domain-independent planning strategies of GPS (Newell and Simon, 1972), STRIPS, and WARPLAN (Warren, 1974), etc still present us with a search strategy that can involve the consideration of an unwieldy set of states, before reaching a solution. It is necessary to provide a mechanism for representing and using our knowledge about formulating plans. Systems which do this generally consider planning within a hierarchy of abstractions. ABSTRIPS (Sacerdoti, 1974) and the blackboard model of HEARSAY-II (Erman et al, 1980) are two such paradigms. The analogy is made of planning a car journey between two cities. We do not 'search' a map of every street and laneway, but rather we consider the problem by planning the journey at a coarse scale first, that is, we consider the major highways and stopping points, and then resolve the problem at successively finer levels of detail. This type of planning strategy requires some prior knowledge about the domain under consideration, and we may consider the procedures for selecting appropriate planning strategies as metaplanning. This will have to be considered if we are to satisfy goals within realistic domains.

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

References

Akin O, 1978, "How do architects design?" in *Artificial Intelligence and Pattern Recognition in Computer Aided Design* Ed. J Latomb (North-Holland, Amsterdam) pp 65-98

Alexander C, Ishikawa S, Silverstein M, 1968 *A Pattern Language Which Generates Multi-service Centres* Center for Environmental Structure, 2701 Shasta Road, Berkeley, CA

Chomsky N, 1975 *The Logical Structure of Linguistic Theory* (Plenum, New York)

Clocksin W F, Mellish C S, 1981 *Programming in PROLOG* (Springer, Berlin)

Downs R M, Stea D, 1973, "Cognitive maps and spatial behavior: process and products" in *Image and Environment* Eds R M Downs, D Stea (Aldine, Chicago, IL) pp 8-26

Eastman C, 1970, "On the analysis of intuitive design processes" in *Emerging Methods in Environmental Design and Planning* Ed. G T Moore (MIT Press, Cambridge, MA) pp 21-37

Erman L D, Hayes-Roth F, Lesser V R, Reddy D R, 1980, "The HEARSAY-II speech understanding system: integrating knowledge to resolve uncertainty" *Computing Surveys* **12** 213-253

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

Gero J S, Coyne R D, 1984, "The place of expert systems in architecture" *CAD84* Ed. J Wexler (Butterworth, Sevenoaks, Kent) pp 529-546
Gero J S, Coyne R D, 1985, "Logic programming as a means of representing semantics in design languages" *Environment and Planning B: Planning and Design* **12** 351-369
Gero J S, Radford A D, Coyne R D, Akiner V T, 1985, "Knowledge-based computer-aided architectural design" *Knowledge Engineering in Computer-aided Design* Ed. J S Gero (North-Holland, Amsterdam) pp 57-81
Kaplan S, Kaplan R, 1982 *Cognition and Environment* (Praeger, New York)
Kowalski R, 1979 *Logic for Problem Solving* (North-Holland, New York)
Lynch K, 1960 *The Image of the City* (MIT Press, Cambridge, MA)
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
Sammut R A, Sammut C A, 1982, "PROLOG: a tutorial introduction" *The Australian Computer Journal* **15**(2) 42-51
Stiny G, Mitchell W J, 1978, "The Palladian grammar" *Environment and Planning B* **5** 5-18
Warren D H D, 1974, "WARPLAN: A system for generating plans" memo 76, Department of Computational Logic, University of Edinburgh, Edinburgh

APPENDIX 1

We present here the knowledge required for matching geometrical descriptions of objects, as described above. The formalism used is the logic programming language PROLOG (Clocksin and Mellish, 1981), using the UNSWPROLOG syntax (Sammut and Sammut, 1982). The goal is to match a composite object *C1* with the facts, describing objects, in the current state description. *C2* is the composite object against which *C1* has been matched and T is the transformation by which the mapping has been made. T is expressed as a combination of rotation and translation with the predicate name 'rot_translation'. The three arguments are: the rotation in degrees and the *x* and *y* cartesian translation factors. This transformation is converted into matrix form for copying objects (not shown).

```
match_composite(C1, C2, T):— composite(C1, L1),
                             composite(C2, L2),
                             match_objects(L1, L2, T).
match_objects([], [], T).
match_objects([O1, ..R1], [O2, ..R2], T):— object(O1, L1),
                                            object(O2, L2),
                                            match_lines(L1, L2, T),
                                            match_objects(R1, R2, T).
match_lines([], [], T).
match_lines([L1, ..R1], [L2, ..R2], T):— line(L1, P1), line(L2, P2),
                                         match_points(P1, P2, T),
                                         match_lines(R1, R2, T).
match_points([], [], T).
match_points([P, ..R1], [Q, ..R2], rot_translation(R, Tx, Ty)):— point(P, [P1, P2, P3]),
                                                                 point(Q, [Q1, Q2, Q3]),
                                                                 rotx(R, P1, Q1, Q2, P4),
                                                                 roty(R, P2, Q1, Q2, Q4),
                                                                 Tx is P4/P3,
                                                                 Ty is Q4/Q3.
```

```
rotx(0, P1, Q1, —, P4):— P4 is P1 − Q1.
rotx(90, P1, —, Q2, P4):— P4 is P1 − Q2.
rotx(180, P1, Q1, —, P4):— P4 is P1 + Q1.
rotx(270, P1, —, Q2, P4):— P4 is P1 + Q2.
roty(0, P2, —, Q2, Q4):— Q4 is P2 − Q2.
roty(90, P2, Q1, —, Q4):— Q4 is P2 + Q1.
roty(180, P2, —, Q2, Q4):—Q4 is P2 + Q2.
roty(270, P2, Q1, —, Q4):— Q4 is P2 − Q1.
```

APPENDIX 2

The basic knowledge required for constructing plans to satisfy simple goals is given below. Design rules take the general form: ACTION # PRECONDITIONS >> RESULTANT, as indicated in the text. Certain rules are required to ensure that plans are always in the right order, but to aid clarity these have not been shown here. The program goal is the predicate, plan, which has five arguments: the list of planning goals, the final state list (to be discovered), the current state (the empty list), the final plan list (to be discovered), and the initial plan (the empty list).

```
plan([], [], −, CURRENT_PLAN, CURRENT_PLAN).
plan([SUBGOAL, ..OTHER_GOALS], CURRENT_STATE, PREVIOUS_STATE,
  CURRENT_PLAN, PREVIOUS_PLAN):—
    solve(SUBGOAL, PARTIAL_STATE, PREVIOUS_STATE,
      PARTIAL_PLAN, PREVIOUS_PLAN),
    plan(OTHER_GOALS, STATE, PARTIAL_STATE, CURRENT_PLAN,
      PARTIAL_PLAN),
    union(PARTIAL_STATE, STATE, CURRENT_STATE).

solve(SUBGOAL, PARTIAL_STATE, PARTIAL_STATE, PARTIAL_PLAN,
  PARTIAL_PLAN):—
    member(SUBGOAL, PARTIAL_STATE).
solve(SUBGOAL, PARTIAL_STATE, INITIAL_STATE, PARTIAL_PLAN,
  PARTIAL_PLAN):—
    given(PARTIAL_STATE, INITIAL_STATE),
    member(SUBGOAL, PARTIAL_STATE).
solve(SUBGOAL, PARTIAL_STATE, PREVIOUS_STATE, PARTIAL_PLAN,
  PREVIOUS_PLAN):—
    ACTION # PRECONDITIONS >> RESULTANT,
    member(SUBGOAL, RESULTANT),
    plan(PRECONDITIONS, CURRENT_STATE, PREVIOUS_STATE,
      PARTIAL_PLAN, [ACTION, ..PREVIOUS_PLAN]),
    delete_list(PRECONDITIONS, CURRENT_STATE, NEW_STATE),
    union(NEW_STATE, RESULTANT, PARTIAL_STATE).
```

The following utilities are required:

member(X, Y)	*returns TRUE if atom X is a member of the list Y*
union(X, Y, Z)	*returns a list Z, which is the union of two lists X and Y*
delete_list(X, Y, Z)	*returns a list Z, which is the inverse of the intersection of lists X and Y.*

The initial state is represented with the fact: given(start, []).