NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

A Generative Expert System for the Design of Building Layouts

Version 2

by

U.Flemming, R.Coyne, T.Glavin and M.Rychener

EDRC-48-08-88

UNIVERSITY LIBRARIES
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA

Table of Contents

1 Introduction 2
2 Representation of Layouts 2
3 Overview of System 4
4 Generator 6
5 Control Strategy 10
6 Domain 1: Residential Kitchens 11
7 Domain 2: Service Cores of High-Rise Office Buildings 16
8 Implementation Details 17

UNIVERSITY LIBRARIES
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA 15213

Abstract.

The present paper describes an attempt to increase the intelligence of a CAD system by adding capabilities (i) to *systematically* enumerate alternative solutions to a design problem, and (ii) to take, at the same time, a *broad spectrum* of criteria or concerns into account. These capabilities are intended to complement the designer's abilities and performance. In connection with such attempts, fundamental problems arise when the objects to be designed have shape and are located in space. These problems are identified, and an approach to solve them is outlined. This approach is currently being tested over a range of domains all of which deal with the design of layouts of rectangles subject to constraints and criteria. The search for alternatives takes place in a *state space* with properties that make it possible to systematically explore and evaluate the power of various *search strategies* or *planning paradigms.* The state space is established through a domain-independent *generator,* while the evaluation of points in that space is carried out by a domain-dependent *tester* built up through a process of knowledge acquisition familiar from work with expert systems.[1]

[1]The project is funded by a grant from NSF and currently in its second and final year.

1 Introduction

A particularly promising avenue for expanding the intelligence of a CAD system is to add capabilities that *complement* the human designer's abilities and performance. The present paper deals with two such capabilities:

- the ability to *systematically* enumerate alternative solutions to a design problem, and
- the ability to take, at the same time, a *broad spectrum* of criteria or concerns into account

The underlying assumption is that the human cognitive apparatus does not perform particularly well in either of these capabilities.

The paper concentrates on fundamental problems that often arise in connection with these tasks, especially when the object to be designed consists of parts that have shape and are placed in space (that is, when the *geometry* of the object is of prime importance):

1. The design has to deal with both *discrete* and *continuous* variables (e. g. the relations between parts and the dimensions or location of a part). This gives rise, at the outset, to an infinite solution space, and the meaning of 'systematic enumeration' has to be clarified. Furthermore, some variables are dependent on each other (e. g. constraints on the dimensions of a part might depend on and vary with the topological relations between that part and other parts).
2. Even if continuous variables are neglected when differences between solutions are defined, the solution space tends to explode combinatorially as the number of parts increases, which introduces the need for efficient *search* strategies.
3. The search for promising alternatives is complicated by the fact that the relations between *design variables* (describing the physical properties of a design) *and performance variables* (describing the quality of a design) are many-to-many. This makes it impossible to consider any (design or performance) variable in isolation.
4. Designers normally develop efficient implicit search strategies and a broad spectrum of relevant criteria over years of practice, thus accumulating expertise that is not easily accessible, but important for the efficient generation of good solutions.

We introduce an approach to solve these problems using a concrete class of design problems as an environment for experimentation: the design of layouts of rectangles in 2D space. This class covers a broad range of specific domains which cut across disciplines. Although we are primarily dealing with domains taken from architectural design (configurations of rooms on a floor, layouts of equipment or work stations in a room etc.), our approach easily extends to other disciplines confronted with layout or floor plan problems.

We outlined the approach taken by us in [4]. Since then, we have developed a first version that deals with layouts in a specific domain (described in [5]), and based on the experience gained with this version, we have started to implement a second version and to expand our approach into a second domain. The present paper summarizes our experience with these systems.

2 Representation of Layouts

We accept, at the outset, the restriction that the layouts produced contain only rectangles with sides parallel to the axes of an orthogonal system of coordinates. A rectangle, z, is then completely described by the coordinates of its lower left corner, (x_z,y_2), and by the coordinates of its upper right comer, (X_Z,Y_Z). This restriction makes the formalizations described in this and the following sections possible,

while allowing for a broad spectrum of realistic design problems to be handled. In particular, layouts are not restricted to 'densely-packed' arrangements; they can have 'holes' or irregular boundaries.

We use a *formal* (graph theory based) *representation* for layouts of rectangles that suppresses continuous variables. It thus defines not a single solution but a solution class. The set of such classes is finite for a given number of rectangles and can be enumerated, at least in principle, if not efficiently. The fundamental, discrete design variables in terms of which a class of layouts is defined are the *spatial relations above!below* and *right!left.* Formally, if *c* and z are two rectangles, then

c t z (read *c is above* z) <=> $y_c \geq Y_z$

zic (read z *is below c)* <=> c t z

c **-»z (read *c is to the right ofz)* <=>** $x_c \geq X_z$

z <- *c* (read z *is to the left ofc)* <=> *c* -> *z.*

We represent the set of relations that define a class of rectangles by a directed, arc-colored graph whose vertices represent rectangles and whose arcs represent spatial relations between selected pairs of rectangles. An example of such a graph is shown in Figure 1 (the vertices labelled 'E' represent the outside or the 'external rectangle').

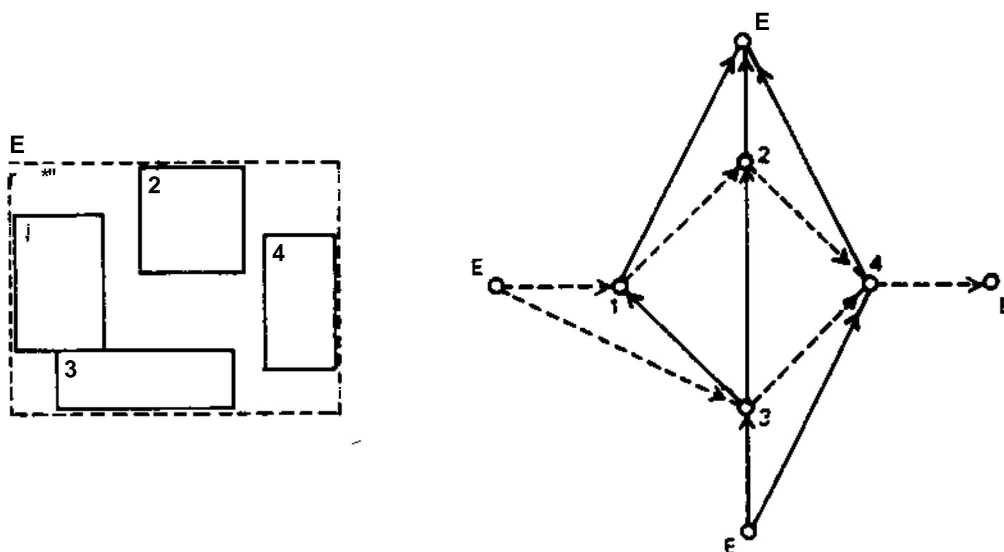


Figure 1: An orthogonal structure and a layout represented by it

Conditions of well-formedness have been established that guarantee that the relations in a graph can **be *simultaneously realized* so that *no two rectangles, in the layouts represented, overlap;* we call a well-**formed graph an *orthogonal structure.* The conditions of well-formedness are formulated in a parsimonious way so that an orthogonal structure defines (either directly or by transitivity) exactly one spatial relation between any (ordered) pair of rectangles (this is sufficient to guarantee non-overlap). It should be noted that the relations used by us are not 'topological* in the strict sense of the word (since they are not based on connectivity).

The spatial relations imply *constraints* on the dimensions and location of the rectangles contained in a structure. Specifically, they imply lower and upper bounds for the comer dimensions of a rectangle. We maintain these bounds explicitly through attributes attached to the vertices in an orthogonal structure; we call the resulting representation a *configuration.* The layouts represented by a configuration can be evaluated over a broad spectrum of constraints and criteria. Changes in the values of dimensional

attributes of any vertex can influence those of other vertices and are easily propagated along paths. Thereby the representation allows us to model explicitly and with ease the distinction between discrete and continuous variables and their interactions.

Furthermore, we generate well-formed structures from well-fbimed structures by application of *operators* (in the form of recursive re-write rules that add one vertex at a time), and we were able to show that any structure can be generated by a unique sequence of rule applications. We thus have available a complete and well-structured *state space* that can, in principle, be systematically searched. The representation that we selected and the operators that work on this representation show how problem 1 can be solved.

3 Overview of System

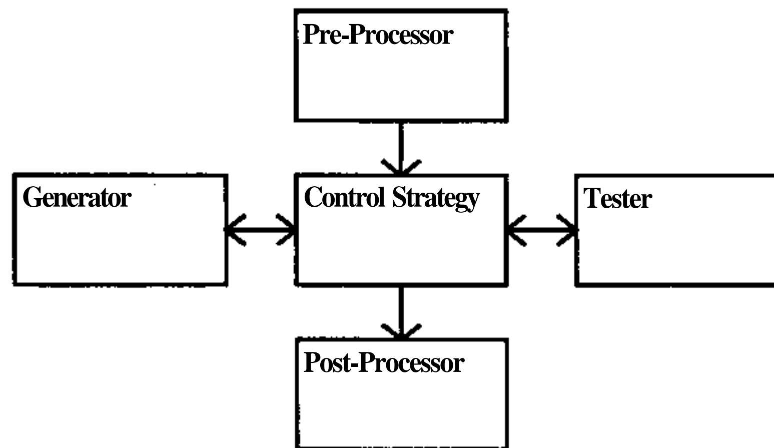


Figure 2: System Components

The approach that we take to solve the remaining problems is reflected in the overall architecture of the system as shown in Figure 2. The system has five major components, each of which can be modified (in fact, replaced) independent of the other components:

1. a *generator* that accepts a configuration (that is, a state in the state space) and expands it in all possible ways; that is, it generates all possibilities for adding a rectangle to the configuration (or all states that can be directly reached from the given state)
2. a *tester* that accepts a configuration and evaluates it with respect to all criteria that can be evaluted, given the information contained in the configuration
3. a *control strategy* that inpects the results obtained from the tester and selects a next state for expansion or terminates generation
4. a *preprocessor* that accepts a problem statement from the user and does preliminary work needed to start generation; specifically, it generates an initial state which is the first state passed to the generator
5. a *post-processor* that fine-tunes the alternatives selected by the control strategy and displays them to the user

How these components work together can be seen from the very simple example used by us to set up the system (see Figure 3). The layout context (configuration A1) consists of four walls that enclose a

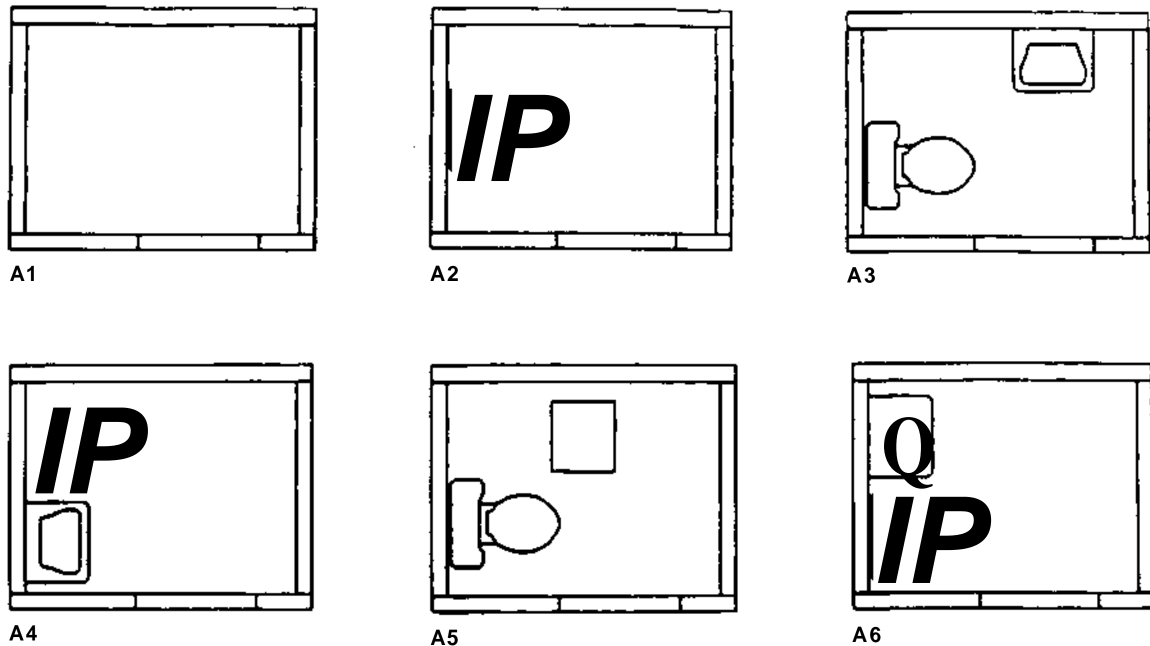


Figure 3: Sample problem

bathroom and a door in the lower wall that swings inward. The problem is to allocate, in that room, a water closet (WC) and a sink.

The pre-processor accepts from the user a description of that context (as a list of rectangles already allocated with each rectangle described by its comer coordinates and by its type, e.g. 'wall') and of the objects to be allocated. It generates an initial state in the form of a configuration representing the context.

The control strategy passes the initial state to the generator, which finds only one possibility for allocating the first object (the WC) 'somewhere' in the room. When evaluating this configuration, the tester discovers that in order to guarantee proper clearance for the door, the WC must be pushed to the left, and the dimensional attributes for that object are updated accordingly. The tester also finds that the WC can now be *oriented,* that is, attached to a wall, which determines the direction of its front. Oriented objects can be displayed through icons that make their function and orientation obvious (configuration A2); all other objects are displayed as general rectangles. Objects are displayed in the center of the range given by the upper and lower limits of the comer coordinates of the dimensional regions attributed to the vertices which represent them.

Three possibilities are found for the placement of the sink: to the right of the WC (configuration A3); below the WC (configuration A4); and above the WC (configuration A5). In the first two, the sink can immediately be oriented. In configuration A5, on the other hand, it can be attached to either of two walls. The post-processor resolves this ambiguity by using a rule that selects a wall because an object has already been attached to it (configuration A6).

Clearly, the system is based on a generate-and-test approach. It tries to take advantage of the generality and modularity that are possible within this approach, while avoiding its pitfalls. We said in the introduction that neither design nor performance variables can be considered in isolation in building design. Strictly speaking, only *complete* solutions should be evaluated over the/a// spectrum of relevant criteria. This makes generate-and-test immediately attractive because it separates the generation of a

solution from its evaluation ([8] pages 71-72). In particular, it handles many-to-many relations between design and performance variables with ease (because every criterion can be evaluated if the design variables on which it depends have been given values). It furthermore allows us to develop a generator that is domain-independent: it generates well-formed structures from well-formed structures independent of what the represented configurations 'mean'; that is, it is defined in purely syntactic terms and thus applicable across domains.

Domain-specific knowledge is concentrated in the tester in the form of test rules, each of which evaluates a specific criterion and thus contributes to an interpretaion of the configurations under consideration. The tester does not have to be specified at the outset, but can be built up through the process of 'knowledge acquisition' as known from current woric on expert systems. With this approach we try to extract relevant, domain-specific knowledge about the quality of configurations in that domain from experts in the broadest sense of the term.

The control strategy directs the search for alternatives into promising directions and prunes the search tree. It is the major device that we use to avoid the inefficiencies usually connected with a blind generator producing a vast number of non-sensical solutions, thus making generate-and-test infeasible. We can do this without violating our principles because in our state space, every state represents complete configurations (in the sense that they represent realizable arrangements of rectangles; they may be incomplete only in the sense that not all rectangles have been allocated). These configurations can be evaluated with respect to those criteria that do not depend on additional information. However, these evaluations can be used for pruning only if the state space has certain monotonicity properties; this poses specific requirements on the generator (see Section 4).

We are sure that for more complex domains, the kind of 'tactical planning' performed by the control strategy must turn into a more 'strategic' form. The modular architecture of the system makes it possible to test a succession of planning strategies of increasing sophistication in terms of the complexity of the domains for which they are efficient. (Our general confidence in a modified generate-and-test strengthened by a planning capability is based not only on the characteristics of building design mentioned in Section 1, but also on the success of the DENDRAL system which exhibits strong parallels to our approach [2]).

The pre- and post-processors are, like the tester, highly domain-dependent. All of these components will be described in greater detail in subsequent sections, starting with the domain-independent ones.

4 Generator

The general approach outlined in the previous section strongly suggests that the generator satisfy the following requirements:

1. *Completeness: Every* conceivable solution, in our case, every well-formed structure, can be generated, in principle, by a sequence of rule applications. Given our objectives, this is the most basic requirement the generator must satisfy because it guarantees a complete state space, a prerequisite for a systematic search.
2. *Closure:* Every object produced by a sequence of rule applications is a well-formed structure.
3. *Uniqueness* (or *non-redundancy):* Any well-formed structure is generated by a unique sequence of rule applications.
4. *Fixed, but arbitrary order for adding rectangles:* This property assures, on the one hand, that in every sequence of rule applications, rectangles are added in a predetermined order. This is desirable because it facilitates control of the generation process; it also makes it

easier for human observers to understand that process and to compare alternative sequences of rule applications. On the other hand, the property assures that the order itself is arbitrary; that is, the set of terminal states does not change with the order in which rectangles are added. The order consequently can be chosen by the user or by a search strategy based on criteria such as efficiency.

5. *Monotonicity:* The spatial relations that exist between allocated rectangles do not change when new rectangles are added. Based on this property, the power of various search strategies can be explored based on performance criteria that are functions of the spatial relations and of the dimensional constraints implied by them (we have not found a single criterion that does not belong to this class); see Section 5 below.

Properties 1 to 3 are standard requirements for a generator to be used in the intended sense ([8], pages 71-72 and 99-102), while properties 4 and 5 are important for our particular application (see the following sections). In order to be able to prove properties for the generator, we found it essential that the operations performed by it are formulated as recursive re-write rules that, specifically, support inductive proofs (see [6] for details). Technical difficulties arise nevertheless when the last two properties are to be satisfied simultaneously.

The generator used in version 1 consisted of the single rule specified in Figure 4. This rule can be applied to an orthogonal structure containing the left-hand side as a sub-structure; the application consists in substituting the right-hand side for the left-hand side in that structure. Intuitively, one can view the rule as 'pushing' rectangles $v_j, \ldots, v_p$ in the configurations represented by the structure 'to the side', thus creating space for the insertion of a new rectangle, *n*. In Figure 4, the rule is specified in a particular orientation. But it can be applied also in rotated versions. Figure 5 shows geometric interpretations of the effects resulting from applications in various rotations and contexts.

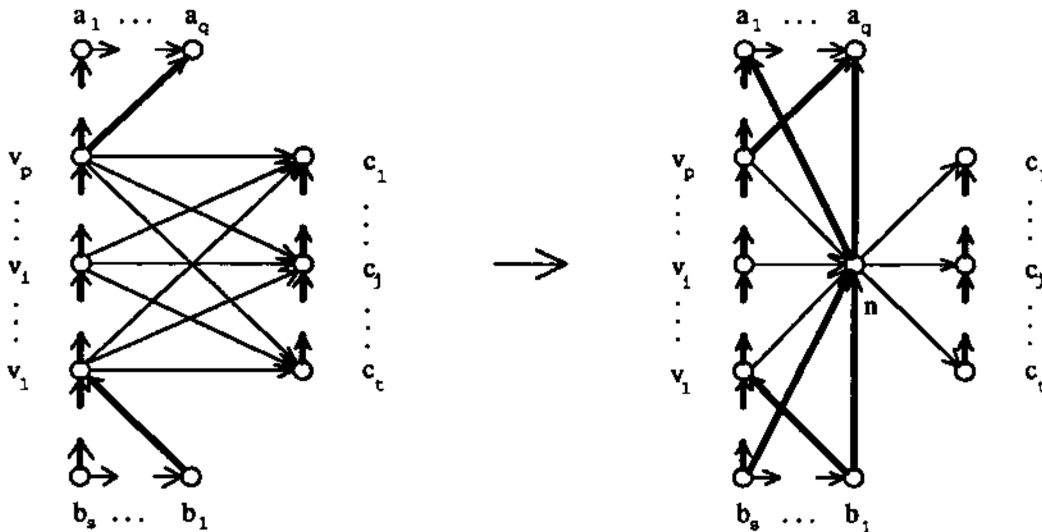


Figure 4: Rule 1 (shown in a particular orientation)

The rule proved sufficient to generate alternative spatial relations for a small number of rectangles (for example, the configurations shown in Figure 3). It even generates certain pin-wheel configurations. An example is configuration (a) shown in Figure 6, in which rectangle 4 is the center of a pin-wheel (we call such rectangles *hubs) thai* is created when rectangle 5 is added to configuration (b) by application of rule 1. But if we want rectangle 5 to be a hub (configuration (c) in Figure 6), rule 1 fails to achieve this effect; that is, rectangles cannot be added in an arbitrary order using rule 1 alone.

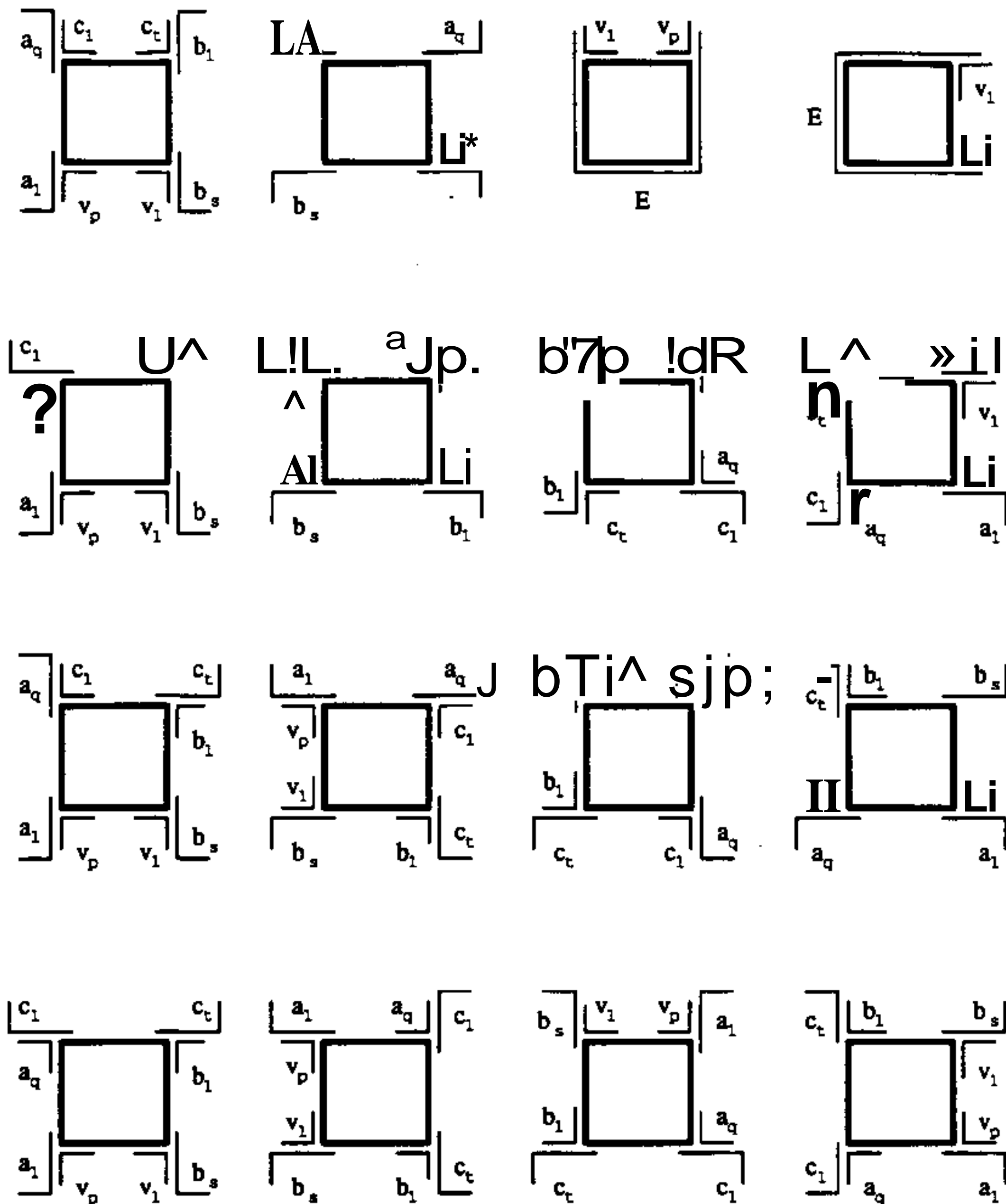


Figure 5: Application of rule 1 - geometric interpretations

Nor is it possible to generate configuration (d) containing four rectangles surrounding what we call a *non-trivial hole* (labelled *h* in the figure). A non-trivial hole is a hole that cannot be generated in a configuration simply by reducing the sides of certain rectangles; it might be neccessary to also *enlarge* certain sides, that is, *change the spatial relations* between certain rectangles. The problem is how to assure that all hubs and non-trivial holes can be generated without violating our requirements for monotonicity and arbitrary order of insertion. The present paper only allows us to sketch the approach adopted by us; an exhaustive description is given in [6].

We maintain rule 1 as the main work horse of the generator because of its simplicity and general usefuleness. In addition, we specify a rule that can be used to insert special vertices representing non-trivial holes (as rectangles) at a time when this can be done without violating monotonicity. These

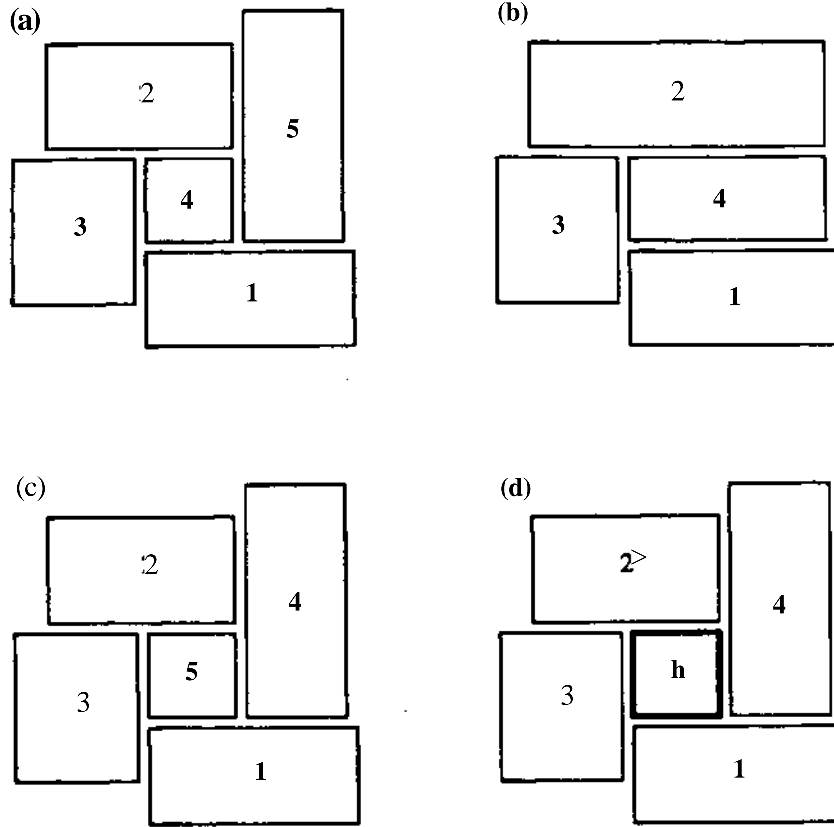


Figure 6: Examples of configurations

vertices serve two purposes: (i) they can become permanent components of a configuration when non-trivial holes arc allowed; or (ii) they can serve as *place holders* for hubs [such as rectangle 5 in configuration (c)] that cannot be inserted at the time they are scheduled for insertion. For example, we generate configuration (c) from configuration (d) simply by replacing the non-trivial hole by rectangle 5. The rule used is rule 2, which simply replaces one identifier (A) by another one *(N)* (Figure 7).

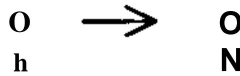


Figure 7: Rule 2

Non-trivial holes are inserted by rule 3, which is shown in Figure 8 together with a geometric interpretation; the rule is shown in a specific orientation, but can again be applied in rotated and reflected versions. Intuitively, it splits two sequences of rectangles that face each other, v_x,..., v_p and c_x,..., c_t, and inserts a non-trivial hole at the rupture. This rule applies recursively after a rectangle has been inserted by an application of rule 1; it generates alternatives *at the same ply* and changes spatial relations only between rectangles *inserted at this ply,* provided that rectangles v_x,..., y- and b_x,..., b_s have been added at the same ply.

The interactions between rule 1 and 3 are carefully monitored by a calling routine described in [6]. The rules can also be called for individual application characterized by parameters that describe the

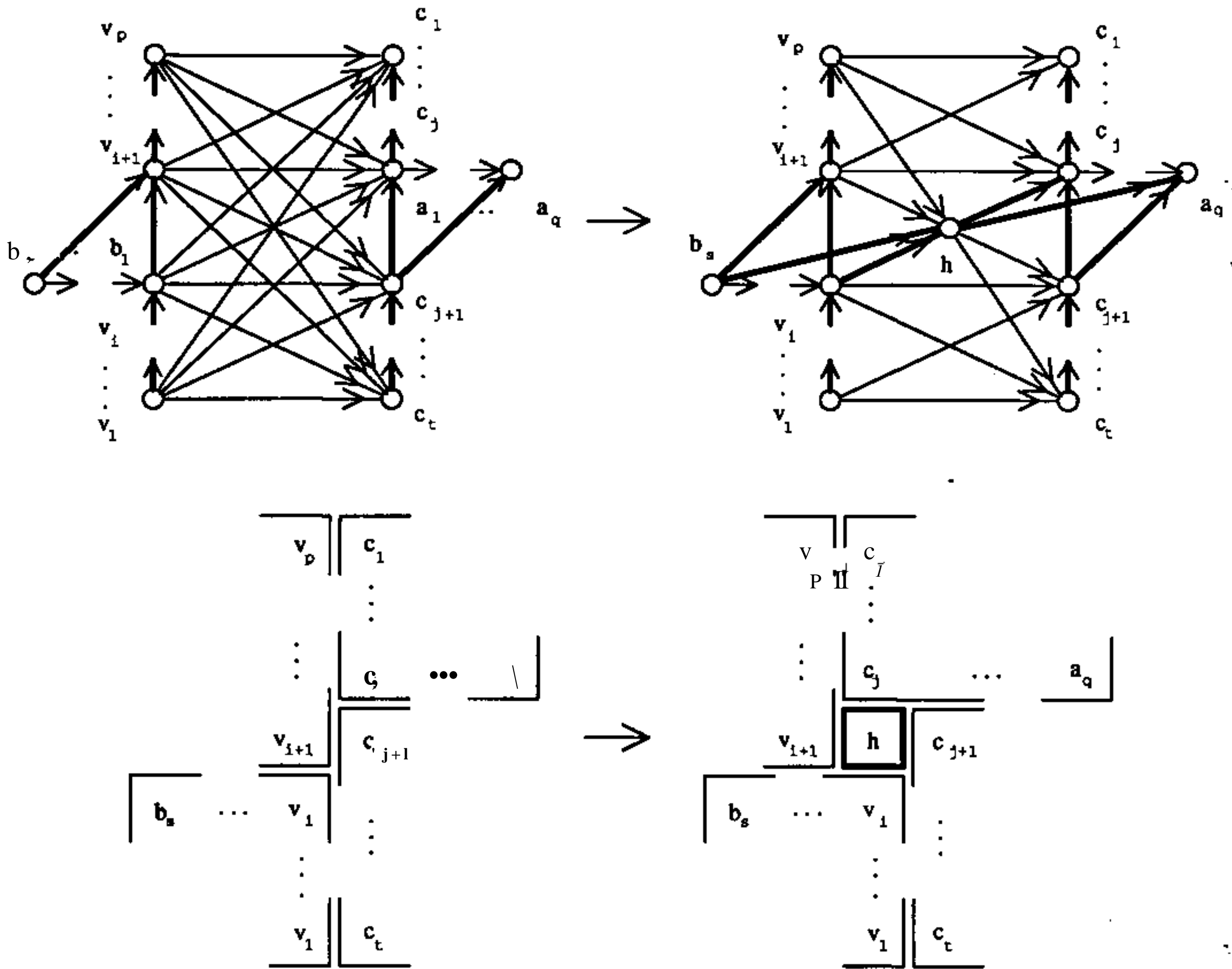


Figure 8: Rule 3 (shown for a specific orientation)

application uniquely (for rule 1, three parameters are needed: vertices v_x and v_p and orientation). This possibility is exploited by the pre- and post-processors (see Section 6).

5 Control Strategy

For the first two versions of the system, we selected a *branch-and-bound* strategy expanding those and only those intermediate states which are at least as good as any other state generated before (independent of its depth in the search tree). To evaluate a state, *s,* we use the triple

$$\phi(s) = \langle c(s), d(s), e(s) \rangle,$$

where $c(s)$ is the number of constraints, $d(s)$ the number of 'strong' criteria and $e(s)$ the number of 'weak' criteria violated by s.

Based on these triples, the control strategy ranks states 'lexicographically'. The triples are computed by the control strategy from an *evaluation record* containing the result of the tester's actions in the form of an unordered list of constraints and criteria violated by the state under consideration. (These records

are used not only by the control strategy, but also as the base for an explanation capability that informs the user in plain English about the reasons for the system's actions.)

The evaluation functions captures, albeit in a crude way, the notion of trade-offs between alternatives. We selected it because of its simplicity; it is easy to implement and proved very effective for the simple domains that we have been working with.

The constraints and criteria evaluated by the function depend on the spatial relations that characterize a configuration. Control strategies like branch-and-bound are applicable because the generator guarantees monotonicity for these relations.

6 Domain 1: Residential Kitchens

Background

The domain that we used to set up and test the system in its initial versions is the layout of fixtures or appliances in residential spaces such as bathrooms or kitchens. We concentrated specifically on the remodelling of kitchens, a design problem that occurs frequently in older houses with kitchens designed for free-standing objects (stove, sink, table, cupboard etc.). Attempts to remodel these kitchens in accordance with current habits that prefer built-in appliances and uninterrupted counter space frequently meet obstacles because window sills are too low, chimneys or radiators are in the way, or doors are misplaced. To find a layout that is pleasing and functionally appropriate is not really difficult, but also not entirely trivial and often left to experts.

Pre-Processor

The pre-processor accepts from the user a problem statement consisting of a *context description* and a *list of objects* to be allocated. For the present domain, the context typically consists of walls, windows and doors forming the boundary of a space; to these can be added existing fixtures such as radiators. Typical contexts are shown in Figure 9.

All of the objects in a context are considered rectangles, and their shape and position are specified by the user through their corner coordinates. From these specifications, the pre-processor generates an orthogonal structure that represents the configuration of context elements; it functions as the starting configuration for the search (an example is shown in Figure 10). In constructing this configuration, we use specific applications of the generation rules to successively allocate the rectangles in the context. This is possible because their comer coordinates are given, and based on these parameters, a sequence of unique applications can be computed. This approach assures that the starting state is well-formed without recourse to special tests or procedures. More generally, it demonstrates how the generation rules can be used in a more constraint-directed mode.

Tester

When we start working on a new domain, the tester is empty. We present a first case to the system and observe its behavior. Even when the first object is allocated in a context that is as complicated as the examples shown above, most of the alternatives produced are bad, if not non-sensical. Moreover, it is usually easy to give explicit reasons for the deficiencies of a configuration (e. g. the object is inserted in a way that obstructs the door). To transform any such reason into a test rule is straight-forward: one simply has to describe explicitly the conditions that lead to the criticism. These conditions give the left-hand side of a rule whose right-hand side writes an entry in the explanation record (see Section 5).

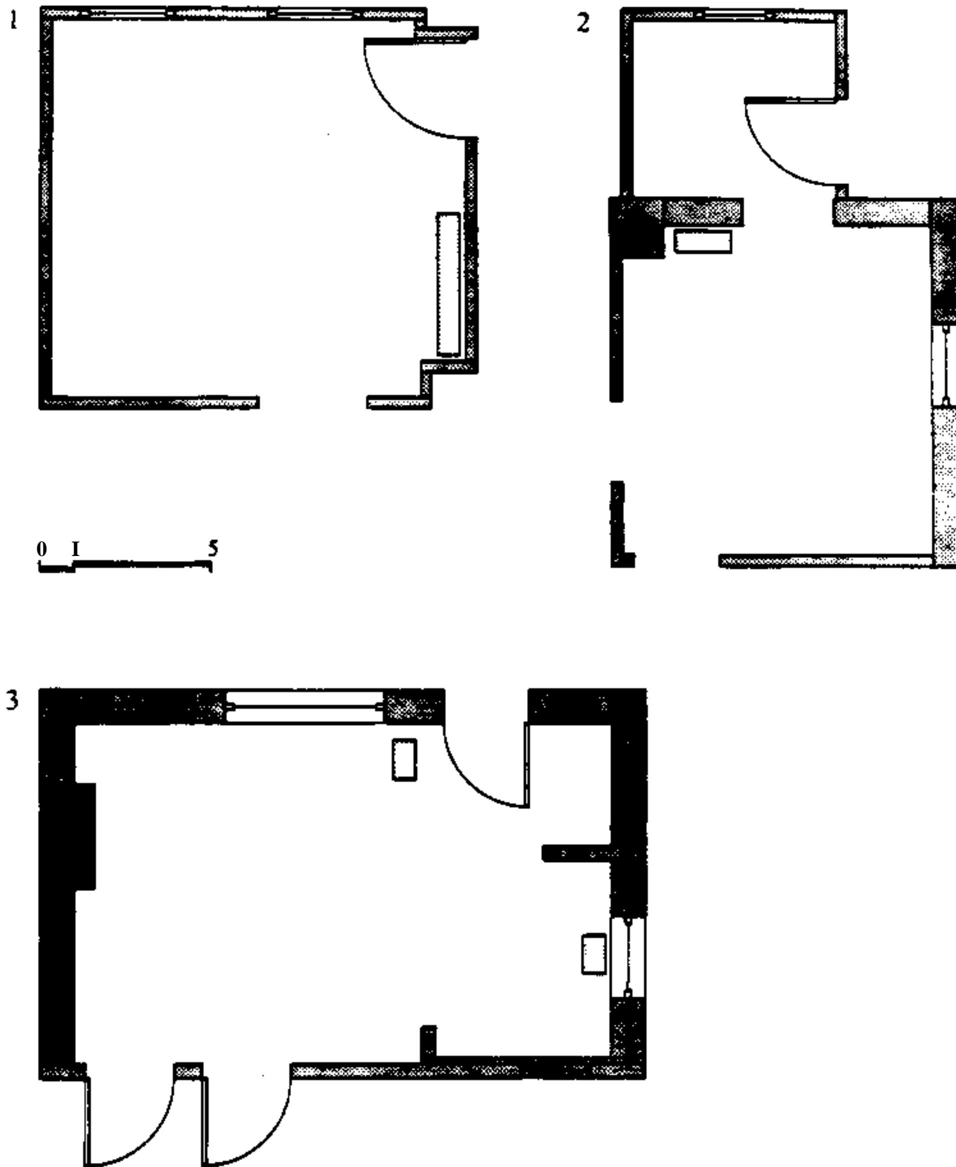


Figure 9: Examples of contexts

We had the experience that whenever an object is allocated for which no test rule applies, the number of alternative allocations is too large to even allow us to look at each alternative. But we found that only two or three rules are nonnally needed to detect the most glaring mistakes and to eliminate most alternatives from further consideration. A more careful analysis of the remaining states is then possible and might lead not only to a fine-tuning of existing rules, but also to the discovery of more subtle points and the formulation of additional rules. Less than half a dozen rules for each object are sufficient to comfortably solve the problems posed by the contexts shown in Figure 9.

At the time of this writing, we insert objects in the following order:

1. Sink. This is a plausible first object because the context nonnally contains more clues for

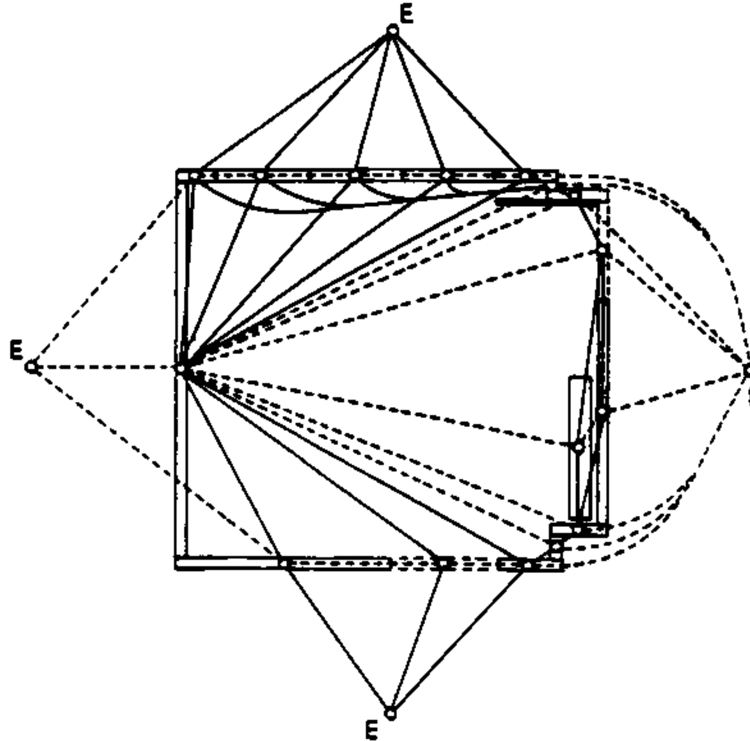


Figure 10: A context and its internal representation

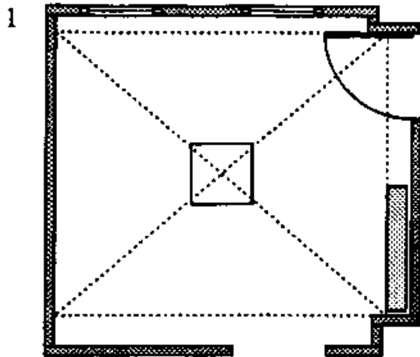
evaluating its position than for other objects.

2. Work area. This is not a 'solid' rectangle, but the area from which all objects must be accessible; it can be viewed as incarnation of the 'work triangle' elaborated in the usual literature on kitchen design. We found it convenient to treat this area explicitly because once it is inserted, objects can immediately be oriented, which allows for the execution of several tests that depend on this information.
3. Refrigerator
4. Range
5. Work counter

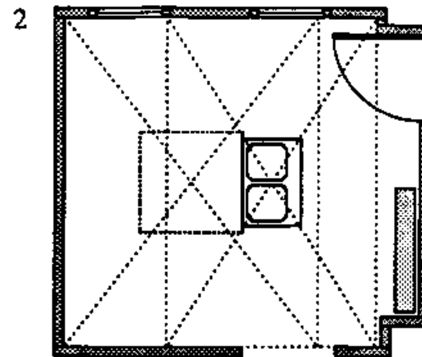
Taken together, these objects give a minimally functional kitchen. Figure 11 shows selected intermediate states reached when dealing with context 1 and parts of the evaluations that caused their rejection for expansion under branch-and-bound.[2] The rule used in case 4 shows that some form of look-ahead is possible for purposes of directing the branch-and-bound into preferred directions.

When testing for a certain condition, a rule might discover that the condition can be met only if the dimensional bounds are tightened. The tester is allowed to make the appropriate updates, provided that no choice exists. For example, when testing if the refrigerator can be placed against a wall, it might find

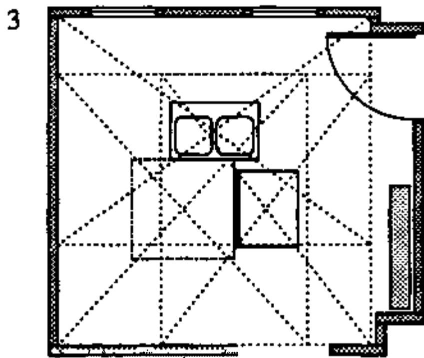
[2]In this figure, an object is shown in the center of the region available for its location; the boundaries of the region itself are indicated through dotted lines. We use this kind of diagram to display spatial relations in an evolving configuration because a display of the underlying structure would be virtually unreadable.



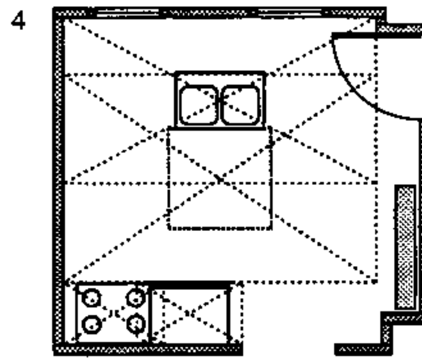
The sink cannot reach the existing plumbing stubs (scored as a weak criterion).



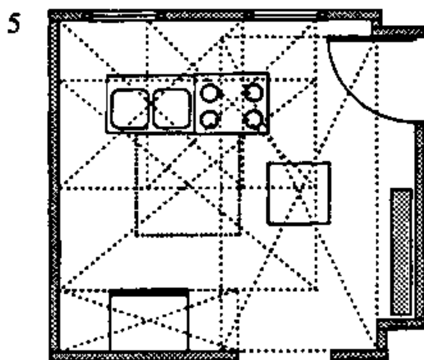
The sink cannot have a window at its back (scored as a strong criterion).



The refrigerator cannot have a wall to its back (scored as a strong criterion).



The range does not have space at its sides for a work counter (scored as a strong criterion).



The work counter is not on the preferred side of the stove (scored as a weak criterion).

Figure 11: Examples of evaluation rules used by the tester

that only one wall is available for that puipose and might consequently update the dimensional attributes of the vertex representing the refrigerator so that its position is restricted to the front of the walL In making these decisions, the tester follows a *least-commitment principle.* In no case is it allowed to change the underlying structure because that would mean a shift of state and a duplication of effort

Post-Processor

The post-processor is invoked by the control strategy when no further expansions are possible; that is, when all intermediate states that are not yet expanded are worse than at least one terminal state. For context 1, the eight states shown in Figure 12 maintain a perfect score in the present version and are passed to the post-processor. These solutions might need some refinements, which are carried out by the post-processor. In the end, they are displayed to the user.

Figure 12: Solutions for context 1

In refining a solution, the post-processor fixes the position of objects whose dimensional range still leaves some choice; they might become oriented in the process. These actions are again based on rules that detect conditions and take the appropriate actions. In determining the final placement for the range, for example, the post-processor has to decide whether it should be placed against a wall or should remain accessible from two opposing sides; the latter possibility leads to the creation of 'islands' or 'peninsulas'. It is interesting to note that the post-processor has no pre-programmed knowledge about these concepts; they emerge naturally from a combination of tests which, at this stage, also deal with issues of 'good' design.

In addition, the post-processor adds optional objects like supplementary woik counters needed to create uninterrupted work areas along walls. Again, it uses generation rules to insert these objects. Figure 13 shows refined versions for solutions 2 and 3 as produced by the post-processor.

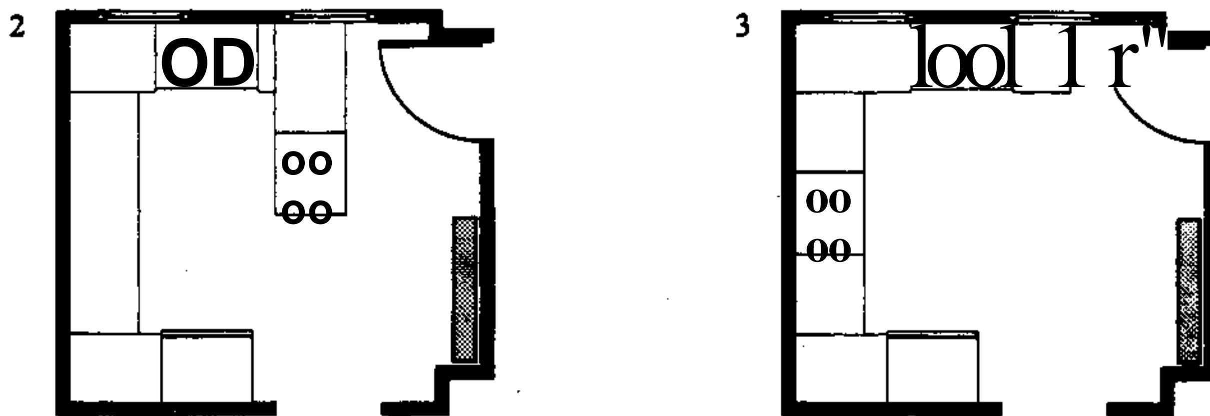


Figure 13: Refined solutions

7 Domain 2: Service Cores of High-Rise Office Buildings

When developing version 2, we repeated some of the experiences that we had with version 1. We spent the most time on programming or re-programming the generator and general suppoit capabilities (such as the display of configurations). Once these components were in place, die domain-specific components could be built rapidly.

This is true in particular for the tester. Its development appears easy both conceptually and technically. There is probably no easier way for extracting implicit expertise from experts than to confront them with bad solutions. Observations of mistakes normally trigger an immediate response that is often quite explicit in terms of the conditions observed and their consequences. This information, in turn, nonnally needs little transformation if it is to be put into a test rule of the form that we have been using: the important part, the left-hand side that causes a critique, is not more than a re-statement of the condition causing the response in a particular syntax (and possibly in a cleaned-up version). To be sure, an initial formulation is often too general or too specific, but this can be discovered through further cases. We found this process so effective that we do not even attempt any more to define test rules in the abstract Why should one worry about a problem before it has occurred?

This suggests that once the generator and support parts have stabilized, the system is easy to adapt to different domains. In order to test this idea, we have started woik on a second domain. This application arose in connection with with a larger project dealing with the creation of an integrated software environment for building design and constructioa It consists of different processes, each adressing a particular task, that are vertically and horizontally integrated (see [3] for a general description). The

particular design problem that is used as context for developing the system is the design of high-rise, speculative office buildings. In this context, layout tasks arise mainly in two forms: (1) the design of the common service core that contains the main means of vertical circulation together with service rooms; and (2) the layout of the rental areas. The present domain deals with the first task. We sketch below how the architecture of the system can be adapted to this domain.

The pre-processor has to accomplish tasks that are quite different from domain 1. It starts with only rudimentary information about the overall geometry of the building, its division into floors and their anticipated occupancy, and the rough position of the core (these data are made available by a module developed independently of our project). The pre-processor first computes a good *banking arrangement* for the elevators; that is, it determines how many banks are needed, how many elevators are in each bank, and which floors are served by each bank. It also specifies the auxiliary spaces needed (emergency stairs, bathrooms, utility rooms etc.) These elements, taken together, are the objects to be allocated; they establish an *architectural program* in the traditional sense.

The pre-processor then generates layouts for the elevator banks. This step is analoguous to the transformation of a context description into an initial state in domain 1; only in the present case, we expect that there may exist alternative starting configurations. Each starting configuration will then be developed into a complete core by adding the auxiliary rooms, a process that will be analoguous to, but possibly even simpler, than the design of a kitchen.

The post-processor might have to finalize a solution. More importantly, it will have to insert structural elements between certain objects in the form of load-bearing walls, a process that is facilitated by certain properties of our representation.

8 Implementation Details

In the first and pilot version, each component was programmed as a collection of production rules, using the OPS83 language [7]. We started in this way because we intend to study, as a side issue, the appropriateness of various shells and programming paradigms for our kind of problem. Given that both generator and tester are conceived as a collections of rules, it seemed natural to start with a production system.

We abandoned this approach for version 2 for various reasons. Prominent among these were the following:

- It was difficult to find an internal representation for orthogonal structures within the restrictions of the language that made the matches needed to apply a rule easy and efficient The problem was compounded by the fact that we had to store many configurations simultaneously in working memory.
- One of the advantages of production systems is that they support the unanticipated firing of rules. In our case, however, this was welcome only in a few circumstances. We were mostly occupied with assuring desired sequences of rule applications by elaborate goal hierarchies and found ourselves continuously programming *against* the grain of the system.

With these difficulties in mind, we wanted an environment for version 2 which allowed us to meet distinct and sometimes conflicting representation requirements for various parts of the system. We wanted the highly structured, algorithmic parts of our system, such as the generator, to be easily and concisely expressed and manipulated. We also wanted the knowledge acquisition and utilization parts to meet the requirements normally associated with a knowledge-based system: adequate representation of domain objects; "chunking" or modularity of knowledge for incremental acquisition and modification; and the ability to express and control relationships and interactions between chunks of knowledge.

We briefly considered using different representations and languages for the components of version 2. But we rejected this approach due to the overhead that would result from the need to translate between representations as the system cycled through its components. We did implement a hybrid system; but it is built in an environment that facilitates a powerful and smooth connection between programming paradigms because all required representations are compatible without translation. We are using Common Lisp for well-understood components (such as the generator) and Portable Common Loops (PCL), an object-oriented extension of Common Lisp [1]v when modularity and incremental development are essential.

We take advantage of inheritance throughout the implementatioa During generation, the child of a parent configuration shares its parent's structure in all but small local changes. We also make significant use of the inheritance of properties and procedures provided by PCL in the creation of class hierarchies and the application of tests.

References

[1] Bobrow, D. et al.
CommonLoops: Merging Common Lisp and Object-Oriented Programming.
Technical Report ISL-85-8, Xerox Palo Alto Research Center (PARC), Palo Alto, CA, August, 1985.

[2] Buchanan, B.; Sutherland, Georgia and Feigenbaum, E.A.
HEURISTIC DENDRAL: a program for generating explanatory hypotheses in organic chemistry.
In Meltzer, B. and Michie, D. (editor), *Machine Intelligence 4,* pages 209-254. Edinburgh University Press, Edinburgh, 1969.

[3] Fenves, S.; Flemming, U.; Hendrickson, G; Maher, M.L.; Schmitt, G.
An Intergrated Software Environment for Building Design and Construction.
In *Proceedings of the Fifth Conference on Computing in Civil Engineering.* American Society of Civil Engineers, 1988.
forthcoming.

[4] Flemming, U.; Coyne, R.; Glavin; T. and Rychener, M.
A generative expert system for the design of building layouts.
In R. Adey, and Sriram, D. (editors), *Applications of Artificial Intelligence in Engineering Problems,* pages 811-821. Springer, New York, 1986.

[5] Flemming, U.; Coyne, R.; Glavin, T.; Rychener, M.
ROOS1 - Version 1 of a generative expert system for the design of building layouts.
In *Proceedings of the International Joint Conference on CAD and Robotics in Architecture and Construction,* pages 157-166. Hermes, Paris (France), 1986.

[6] Flemming, U.; Coyne, R.; Glavin, T.; Rychener, M.
A Generative Expert System for the Design of Building Layouts.
Technical Report (in preparation), Engineering Design Research Center, Carnegie-Mellon University, Pittsburgh, PA, 1988.

[7] Forgy, C.L.
OPS83 User's Manual and Report.
Production Systems Technologies, Pittsburgh, PA, 1985.

[8] Hayes-Roth, F. et al. (ed.).
Building Expert Systems.
Addison-Wesley, Reading, MA, 1983*