Heuristic Generation of Layouts (HeGel)
Based on a Paradigm for Problem Structuring

by

O. Akin, B. Dave and S. Pithavadian

EDRC-48-10-88

Heuristic Generation of Layouts (HeGeL) Based on a Paradigm for Problem Structuring

Omer Akin, Bharat Dave
and Shakunthala Pithavadian
Department of Architecture
Carnegie Mellon University
Pittsburgh, PA 15213

5 November 1987

Abstract

This report describes a computer based system called HeGeL which is developed to calibrate and verify a model of problem structuring in the design process. Implemented as a production system, HeGeL simulates the design behaviors of humans observed during protocol experiments. This report discusses the representations used, flow of actions and the computing environment of HeGeL as well as the validity of the proposed design paradigm and the adequacy of methods used.

Heuristic Generation of Layouts (HeGeL): Based on a Paradigm for Problem Structuring

Omer Akin, Bharat Dave,
and Shakunthala Pithavadian

Department of Architecture
Carnegie-Mellon University
Pittsburgh PA 15213

5 November 1987

Abstract

Based on protocol analysis of the designers solving spatial problems in architecture, a paradigm for the designers' behavior was proposed [Akin87] in terms of: *Problem (Re)Stnjcturing,* when problem parameters are established or transformed, and *Problem Solving,* when these parameters are satisfied in a design solution.

In order to calibrate and verify the paradigm, we implemented a computer based system called HeGeL (Heuristic Generation of Layouts). In this report, we describe our approach to developing HeGeL, its architecture and performance. Implemented as a production system, the computer program simulates behavior of the designers observed in the protocol experiments conducted earlier. This report intends to serve multiple purposes. First, we describe architecture of HeGeL in terms of representations used, flow oTactions and computing environment in which it was implemented. Based on performance of HeGeL, we discuss validity of the proposed paradigm for the designers' behavior, adequacy of our methodology, and some other issues faced during the development of HeGeL. Some research issues that need further exploration ^tre also highlighted.

This research is funded by NSF Grant No: CEE-8411632

Table of Contents

List of Figures

1. Background

The purpose of our research is to understand the designers' behavior when solving spatial problems in architecture and to develop an operational model that accounts for this behavior. Based on empirical observations underlying patterns of phenomena have been elucidated in various disciplines. Once a coherent paradigm is developed in this fashion, it is then validated by showing how it accounts for, explains or predicts similar phenomena [Baylor71, Moran70, Eastman70, Foz73, Akin78]. In a similar vein, we approached our research task in two stages: (a) developing a paradigm for the designers[1] behavior and (b) simulating and verifying the paradigm as a computer program.

As part of the first stage in our research work, empirical data were collected through protocol experiments of designers solving spatial problems. Based on the analysis of protocols, a paradigm for the designers' behavior was presented by Akin *et al.* [Akin86a, Akin86b, Akin87] in terms of two functionalities: *Problem (Re)Structuring,* when problem parameters are established or transformed, and *Problem Solving,* when these parameters are satisfied in a design solution. In the second stage, we implemented a computer program to test and validate the paradigm. This report is mainly concerned with the implementation of this system.

Previous studies have addressed some of the issues that we pursued in our work. One of the central concerns in our work has been understanding the process of structuring a design problem. This issue has drawn attention of many researchers from the area of artificial intelligence [Reitman64, Freeman71, Simon73, Akin78]. These studies suggest that most design problems acquire structure during the very process of design development and even then, at best, a designer is likely to settle for a *satisficing* solution. Another salient observation to emerge from some studies [Foz73, Simon73, Baykan84, Akin86c] is that the designers employ heuristic techniques in searching for a solution to design problems.

Many such studies have concentrated on modeling a specific component of the design process. In this regard, some of these studies are directed towards developing techniques and tools that are as good, if not better, than the performance of a human designer. Our work has slightly different objectives. Primarily our interest in this research lies in understanding and modeling the design process as a cognitive skill. We did not aim to develop any prescriptive methods for design; rather we have attempted to demonstrate what the designers do. To this end, we describe a system that simulates the behavior of designers as recorded in our protocol experiments.

2. Objectives

A paradigm for the designers' behavior was presented earlier [Akin87]. In order to validate this paradigm we implemented a computer program which we call HeGeL for Heuristic Generation of Layouts. By comparing the performance of HeGeL against the behavior of designers, we gradually developed computer program that enabled us to look closely at our paradigm. This effort also pointed out certain areas which need further work in order to test the validity of the suggested paradigm.

In this report, we first describe architecture of HeGeL in terms of the representations used, flow of actions and computing environment in which it was implemented. This section, entitled "System Architecture," describes in detail the data structures, operations, input and output of data as developed in HeGeL Although we describe computational aspects of heGeL, we have attempted to correlate the discussion with major components of the paradigm presented earlier. Here, a detailed example run of HeGeL is also included.

In the next section entitled "Evaluation," HeGeL i£ assessed in terms of its performance compared to the behavior of designers as seen in the protocol experiments. We discuss the validity of the proposed paradigm in light of the results demonstrated through the implementation of HeGeL just described.- Here, some open issues concerning methodology, problem structuring in design and architectural layout problems are also delineated in general terms. Based on our implementation efforts, we also discuss the advantages and disadvantages of adopting a production system architecture. General issues pertaining to implementation of a computer system are described. The study ends with a "Conclusion" section.

3. System Architecture

3.1. The Task

HeGeL is designed to develop solutions to a space planning problem identical to the one given to the subjects in our protocol experiments. Specifically, the task given to HeGeL is to design an office layout given a list of personnel and furniture to be accommodated. Based on the analysis of 18 different subjects in our experiments, we developed a paradigm that accommodates all different solutions produced by the subjects. HeGeL is implemented to reflect this paradigm and broadly comprises of the following operations [Akin87].

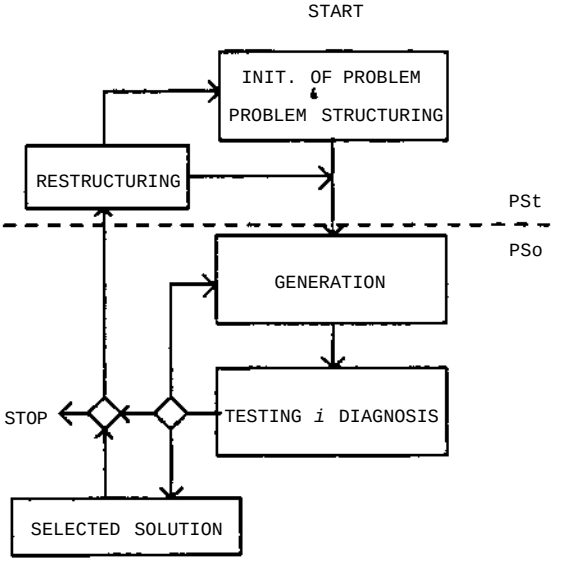


Figure 3-1: System Outline

First, structure the problem by establishing pertinent requirements to be satisfied. Next, select one of these requirements and generate alternative solutions that satisfy this requirement. Test the generated alternatives to find those that satify the remaining pertinent requirements. If more than one alternative are feasible, select one according to additional requirements. If no solution is feasible, either restructure the problem by modifying the set of requirements or search for alternate ways of satisfying the original set of requirements based on a diagnosis of the situation. This process continues until all the requirements are satisfied. At this point, one may either have a final solution and the process stops or one may want to look for another solution that may be better than the one found so far. In its broadest outline, the problem solving portion of the paradigm follows a *generate-and-test* sequence of operations as shown below the dotted line in Fig.3-1.

A successful run of HeGeL is capable of finding solutions that are remarkably similar to the ones produced by the subjects. Three final solutions produced by the subjects and the corresponding layouts generated by HeGeL are shown in Figures 3-2 and 3-3. As explained in the next few sections, HeGeL generates layouts at a particular level of detail, specifically in terms of functional areas. It does not yet work at the next detailed level of furniture arrangement for individual functional areas. This simply reflects a decision on our part to set HeGeL up hierarchically. In future iterations, we intend to implement individual furniture placement once functional zones are established using the same paradigm underlying HeGeL Incidentally, in our protocol studies, human subjects treat furniture placement in similar, hierarchic fashion.

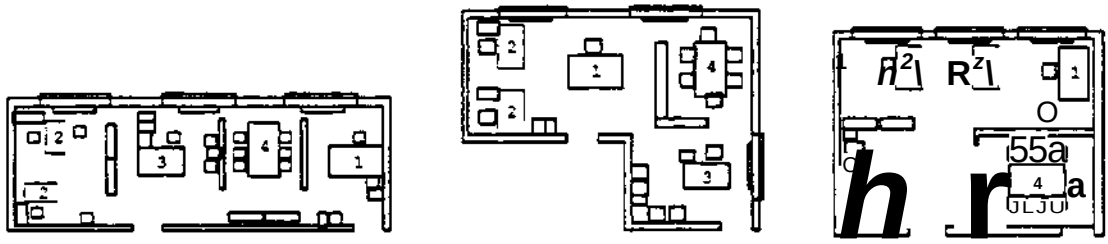


Figure 3-2: Solutions generated by the subjects

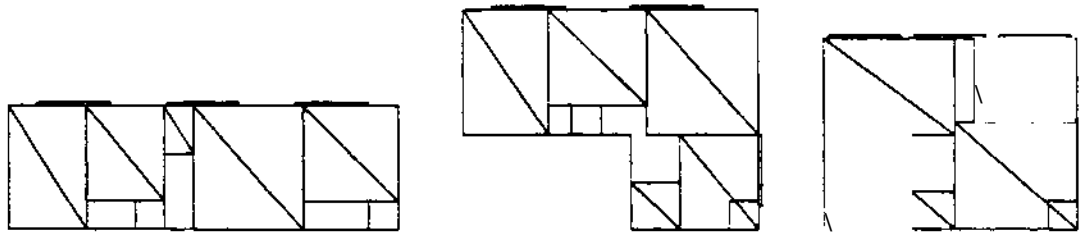


Figure 3-3: Solutions generated by HeGeL

Next we shall describe in depth **each component operation of HeGeL and** the underlying representations.

3.2. Computational Model

When we embarked on Implementation of HeGeL, a couple of choices in terms of programming tools were available to us. We were looking for an implementation environment that would facilitate representation of the following:

1. To perform generate and test operations in a graphic domain
2. To manipulate spatial constraints easily
3. To modify the constraints list with ease
4. To use relations for generation and testing interchangeably

We had developed the paradigm in a descriptive form and we anticipated a gradual and incremental development of its implementation as a computer program. Clearly, the imperative programming languages were not suitable for our purposes.

Since our research subject involves representation and manipulation of substantial domain knowledge, we were naturally faced with choosing one from among the major architectures for knowledge-based systems: frame-based, rule-based and logic-based represenation. Specific advantages provided by each of these architectures in certain domains are nicely summed up by Friedland [Friedland85]:

> ...*logic* where the domain can be readily axiomatized and where complete causal models are available, *rules* where most of the knowledge can be conveniently expressed as experiential heuristics, and *frames* where complex structural descriptions are necessary to adequately describe the domain.

Initially, we settled on a Lisp-based frame representation environment. A particular frame-based lanaguage, Schema Representation Language(SRL) [Wright83] was available to us at the time. SRL is a Usp-based environment with some special features like maintaining incremental frame-based databases and object oriented computational facilities. Additionally, SRL has an interpreter that seemed ideal for our application. Subsequently, due to circumstances beyond our control, SRL was no longer supported on the computing machines available to us. And we had to select another implementation environment which ended up being OPS83- a production system language [Forgy85]. In retrospect, we realized both advantages and disadvantages existed in this choice.

3.2.1. OPS83- A Production System

HeGeL is programmed as a production system in OPS83 [Forgy85][1]. Since in the following sections w- will frequently refer to the data structures and rules as encoded in this programming language, we fir; present a brief overview of OPS83.

OPS83 is a production system language and owes its lineage to the earlier classic production systc languages like OPS4 and OPS5. Basic components of a production system are: (a) a global databc called *working memory(WM)* represented as WM *elements* (WME), (b) a collection of *if-then* rules, and » a *conflict resolution strategy(CRS).*

[1] **While other versions of OPS83 running under different operating systems and machines are also available, we used C -83 running under 4.2 Unix on a VAX 11-780 in the Computer Science Department, CMU.**

The WM is represented as a collection of *working memory eiements(WME).* A WME consists of an identifier name and a list of field tags each of which can store values of a defined type. A formal declaration of WME in OPS83 looks very similar to a record structure found in procedural languages like Pascal. In the following example (Fig.3-4), a WM element of type vertex is declared with two fields or attributes named x and y, each of which can store real values.

```
type vertex - element
    {
      x : real;
      y : real;
    };
```

Figure 3-4: A sample WME declaration in OPS83

Each rule contains a left-hand-side(LHS) that specifies a condition part under which the rule can fire and a right-hand-side(RHS) that specifies actions to be executed if the rule fires. The LHS of the rule consists of one or more *patterns* that are to be compared to the elements in working memory. If a match is found, then the rule becomes a potential candidate for firing and the matching elements from working memory are available to the RHS of the rule for whatever action is specified. The RHS of the rule may contain system defined primitive actions (e.g. make, modify, remove) on the elements in WM. The RHS may also be specified in a manner that is very similar to procedural syntax found in other languages like Pascal[2]. In the following example (Fig.3-5), a rule called example consists of a LHS pattern specifying a WME of type vertex whose field x is equal to zero and y is greater than zero. If there is such a WME and the rule fires, the RHS action will modify the WME associated with &V (on the LHS) by changing the value of field x to be equal to y and then setting the field y equal to zero.

```
rule example
  {
   AV {vertex x - 0.0; y > 0.0);
   —>
   modify iV (x - AV.y; y - 0.0);
  };
```

Figure 3-5: A sample OPS83 rule

The production system is activated by finding rules whose patterns match the elements in working memory and firing those rules. This component of the system is defined as a *recognize-act* cycle. It proceeds in the following fashion:

1. Match: Find all rules whose LHS match the current contents of working memory. All such rules are placed in a *conflict set*
2. Conflict Resolution: Select a rule from the conflict set. If no rules were found, halt.
3. Act: Execute actions specified in the RHS of the selected rule.
4. *Go to step 1.*

***This is one of the differences between OPS83 and the earlier production system languages like OPS5.**

In order to use OPS83, the programmer has to specify a conflict resolution strategy(CRS) according to his needs[3]. We implemented a CRS that selects a rule that has changed most recently, and that has the most number of patterns in its LHS from among rules in the conflict set. Selection of such a strategy provides a direction to the sysetm since it becomes sensitive to more recent tasks and executes them to completion before selecting new ones. This strategy also takes into account the fact that more specific a rule, more specific contexts it is likely to serve and thereby such a rule is given priority over the others. Both criteria seem to follow from human subject's behaviors in solving similar space planning tasks [Akin86b].

3.3. Major Data Types

3.3.1. Design Units

HeGeL comprises of a number of data types or WM elements and a collection of operations or rules that manipulate WM elements. A fundamental concern in our domain is representing spaces in terms of their geometric (e.g. location) and non-geometric (e.g. solid boundary) attributes, and certain relations (e.g. adjacency) among spaces. We decided to limit the scope of our system to deal with only rectilinear spaces that are parallel to the Cartesian coordinate axes. This decision was influenced by the fact that most of the subjects in our experiments came up with solutions that could be modeled within these limitations. A few protocols that deviated from these assumptions did not seem to offer additional evidence towards the validation of the paradigm that we proposed.

```
type design_ur.it - element
  (
   id           : siyrabol;            – site, S, CE, ...
   crig_dims    : array(2: real);     – hcriz. £ vert, dimensions
   cocrds       : array(2: vertex);   – location coords.
   ccr.tains    : array(2: symbol);   – alternate furniture patterns
   orientation  : symbol;             – N, E, S, W
   ...
   alt_locs     : array(30: vertex);– alternate location coords.
  );
```

Figure 3-6: WM type for design units

Declaration of a typical spatial unit consists of a number of attributes as shown in Fig.3-6. A unique identifier is used for each distinct spatial entity required by the program. Essentially, such a declaration provides a template for a particular kind of a WM element, and specific instances of it are distinguished by creating and assigning a unique identifier to each of them. Not all WM elements of one type need to use all the attributes associated with it. For our problem, the site of a design is completely defined a priori and there is no need to compute, for example, its coordinates. On the other hand, a design unit, i.e. a functional area associated with one of the personnel to be accommodated in the site, needs to be first

[3] **In the production languages available previously, the system interpreter took care of this component. This is the second major difference between OPS83 and its predecessors.**

given a size based on the kind of furniture to be placed inside that design unit.

Viewed in this fashion, a WM element type for design units defines a universe of possible attributes (of interest to us), particular instances of which take on different values. For example, we needed to define specific functional areas for the personnel to be accommodated; secretary (S), chief engineer (CE), staff engineers (SEs), and conference (C). A functional area, in a generic sense, defines a universe of furniture placements that are possible. A particular WM element for a design unit, say SEs, may have more than one possible furniture arrangement. Depending on which furniture pattern is asserted during execution, HeGeL takes care of assigning appropriate dimensional requirements for a design unit.[4] A set of possible furniture arrangements for a design unit SEs (staff engineers) is shown in Fig.3-7. Note that each of these patterns incorporate a band of access space from one or more directions. This is to ensure that a design unit will have sufficient area for forming a continuous circulation space when combined with other design units.

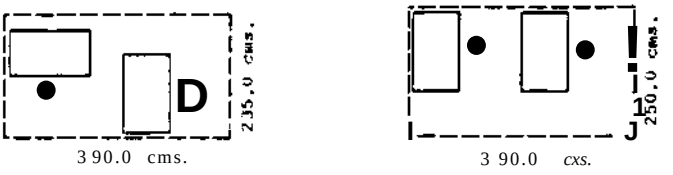


Figure 3-7: Alternate furniture patterns for SEs

In addition, HeGeL should be able to handle all different sites and all different functional areas to be accomodated with the same data structure. There are three different kinds of sites in our problem; a rectangular, a square and an L-shaped site. The first two can be defined within the limits imposed earlier, namely treating them as rectangles. In case of the L-shaped site, we simply defined a larger rectangle with a dummy (transparent) design unit that is already located inside the larger rectangle during the system initialization.

Although HeGeL, at present, does not manipulate each furniture piece separately, such an extension seems to require a generalized declaration for each furniture item. Currently, for each of the furniture patterns, we have declared marginal spaces in all four directions that may or may not be allowed to be overlapped with some other space. A more generalized form could explicitly store each furniture item in the form of: (a) a space occupied by the object (and hence cannot overlap with any other space), and (b) a clearance space in order to make the object usable or accessible (which may be allowed to overlap with other similar clearance spaces). Such individual items can then be collapsed into furniture patterns and used just as HeGeL uses them presently.

[4]This feature of HeGeL in which design units like S, CE, etc. are first assigned an approximate dimension based on a furniture pattern is consistent with behavior of most of the subjects. Given a list of personnel and furniture items, the subjects invariably organized the furniture items that are related to a functional area thereby (a) getting a sense of scale of the space to be accommodated and (b) freeing their attention from very low level details Ike individual furniture placement to a higher level abstraction.

3.3.2. Predicates

Predicates deal with relationships among design units as well as attributes associated with a particular design unit. An example of the former is: *secretary has to be directly accessible from the main entrance;* this predicate stipulates an access relation between S and the main door (Dm). An example of the latter is: *staff engineers need natural light,* this predicate stipulates a spatial attribute for the design unit SEs. A typical declaration of a WM element type for predicates is shown in Fig.3-8.

```
type predicate - 6lament
  (
    id       : SXTTU?C1;   -- pl, p2, p3, ...
    status   : symicl;     - active, current, passive
    unitl    : siiriel;    -- design unit- 1
    unit2    : SXTTJ:C1;   -- design unit- 2
    relation : snrJ:cl;    — re.aticr. or attribute,
                           — e.g. arress, rrivacv, ...
  );
```

Figure 3-8: WM element type for predicates

Based on the analysis of protocols, possible relationships were defined in HeGeL For each instance of a relationship, HeGeL creates a unique predicate and assigns proper values to its attributes. Almost all the predicates for our application are concerned with *direct access, easy access, natural light, and privacy.* These in our implementation are interpreted as follows.

Direct access is a symmetric relationship between two design units, D1 and D2, if D1 and D2 are directly adjacent, i.e. one of the edges of D1 is coincident with any one of the edges of D2 (Fig.3-9). Easy access is a symmetric relationship between two design units, D1 and D2, if there is a third design unit D3 such that D1 and D3 are directly accessible and D2 and D3 are directly accessible (Fig.3-9).

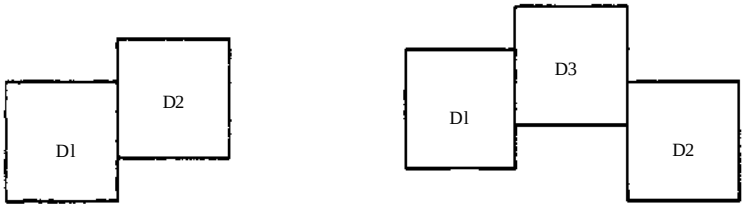


Figure 3-9: Direct and easy access

Natural light defines a spatial attribute of a design unit D1 if one of the edges of D1 has a window opening onto the exterior of the site (Fig.3-10).

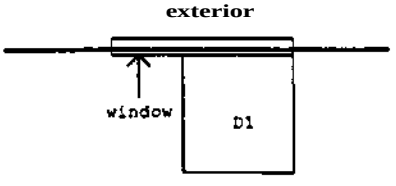


Figure 3-10: Natural Ught

Privacy is a spatial attribute of a design unit D1 and is defined in terms of D1 having solid boundaries. Unless explicitly assigned some other value, all the edges of design units located by HeGeL are considered transparent. If a predicate specifies privacy as an attribute of unit D1 with respect to another design unit D2, then D1 is located as far away from D2 as possible and only those edges of D1 that face D2 are assigned solid boundaries (Fig.3-11). If the predicate does not specify D2 and requires that D1 be private then all the edges of D1 are assigned solid boundaries.

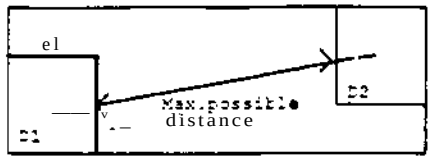


Figure 3-11: Privacy

The subjects in our experiments developed a final solution satisfying a set of relationships that were established as the design progressed. In other words, the design process that we observed and aimed to model was made up of a number of stages or cycles. During each such stage, called *episodes* in the design process, certain relationships were consistently used in generating and testing alternative solutions and these were not modified or completely disregarded until after the end of the episode. In the implemented version of HeGeL, each episode is modeled by a certain number of predicates that are *active* and the remaining ones are *passive.* The generation and test operations recognize only those predicates that are active. At the end of the episode, HeGeL can manipulate both the active and passive predicates to restructure the design problem.

3.3.3. Process History

In order to keep a chronological record of its own design process, HeGeL makes use of a global two-dimensional *history* matrix(Fig.3-12). Each row in the matrix corresponds to an episode or a cycle in the solution development by HeGeL. Each column records the sequence in which design units are located or attempted to be located in each episode. An episode in which all the design units are successfully located will have found a design solution.

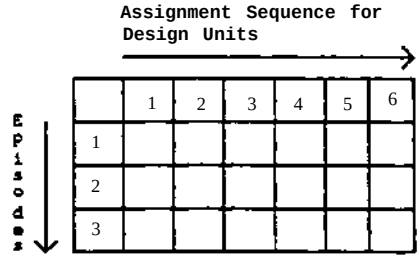


Figure 3-12: Process History

3.3.4. Other Data Types

In addition to the major WM elements described above, other data types are also used in HeGeL Some of them are used to direct execution of HeGeL, e.g. a WM element called *blackboard* is used as a first pattern on the LHS of all productions. This enforces a desired sequence to the execution of HeGeL since, during the recognize-act cycle, only the productions that have successful match for each of the patterns on their LHS become candidates for potential firing. In other words, a WM element like *blackboard Is* used to partition the productions into chunks of process-specific actions like initialization, generation, etc.

Some other data types were also used to store and pass values to productions or to make some operations more straight forward. Since they were used for very narrow or specific purposes, they are not described here.

3.4. Sequence of Operations

3.4.1. Initialization

All the subjects in the experiments were given a specific site and an identical problem statement in terms of the personnel and furniture items to be accommodated. Each subject developed a different solution depending on how he structured the problem. Some of them first established an object hierarchy or a functional hierarchy [Akin87], and then identified a set of relationships that were to be satisfied among the design elements. This process proceeds in cycles, each cycle involves a number of alternatives to be generated and tested. Some of them are developed further, gradually converging towards a final solution.

A typical session with HeGeL starts by setting up the problem definition. In order to simulate the data from a given protocol in HeGeL, first a file is set up that specifies a particular site, design episodes observed in a particular protocol, relationships identified for each episode, and furniture patterns associated with each design unit. At the start of the session, the system is initialized by reading in these data from a file.

As shown in Fig.3-13, an initialization file has three major sections. The first line specifies a site from among three possible ones; rectangular, square or L-shaped. The next section (lines 3-17) specifies the predicates (Section 3.4.3) corresponding to relationships derived from human protocols, each separated by a blankline. Some of these predicates are designated *active,* i.e. only those predicates can affect generation and test operations during a given episode; and the rest are designated *passive.* Passive predicates can be activated just as active ones can be placed on passive status at the end of each episode. The last section (lines 19-23) specifies the design units and alternate furniture patterns for each design unit. In this way HeGeL permits the "playing out" of different episodes, successively.

The initialization process can be described in a different way. All the subjects brought their personal knowledge to bear on developing a solution to a given design problem. The system at the start of a session reflects only a container for such knowledge in all its variations. Once initialized, the system is

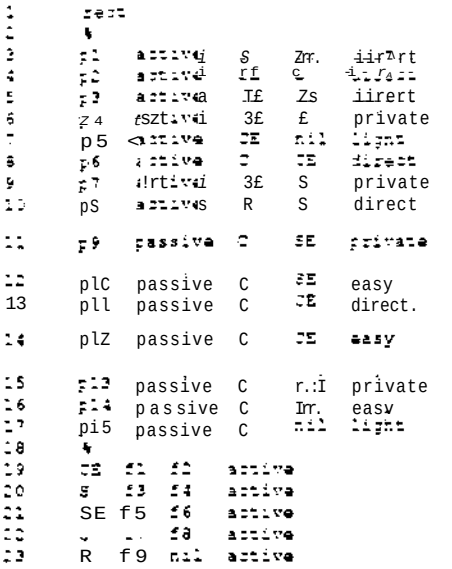


Figure 3-13: A Sample Initilization File

equipped with specific knowledge which will determine its subsequent behavior (Fig.3-14).

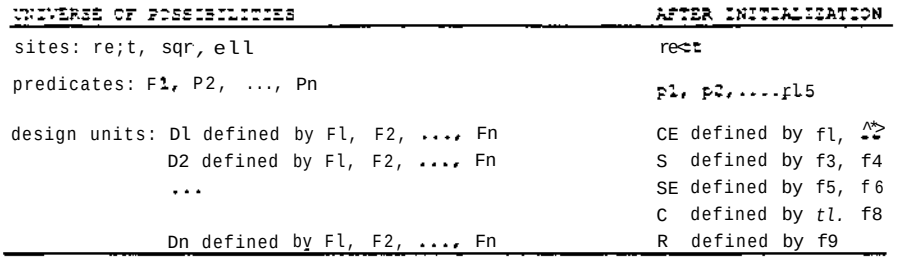


Figure 3-14: Initialization Process

3.4.2. Solution Development

As observed in [Akin87]$_t$ the subjects generate solutions *by pattern* or *by zoning.* The former utilizes object (e.g. furniture items) or functional (e.g. design units) hierarchy. The latter utilizes extant cues in the site (e.g. door and window locations) to first create zones into which the design units are mapped. Presently, HeGeL is capable of generation *by pattern* only. This is purely a circumstantial limitation of HeGeL; we did not have sufficient time to implement generation *by zoning.* But the underlying principles are well analyzed and described in [Akin87]. In the rest of this report, we describe implementation of HeGeL in terms of developing solutions *by pattern* only.[5]

[5]**It should be emphasized that the data structures and initialization of HeGeL are designed to handle both generation by pattern and by zoning.**

3.4.3. Predicate Selection

The design units are assigned in accordance with certain desired relationships which we have termed predicates. They represent relations between design units or attributes of a design unit. Depending on the context, predicates are used as generative *constraints* or evaluative *criteria.* In order to assign a design unit, HeGeL first needs to find a predicate which then will be treated as a generative constraint for that design unit. Selection of a generative predicate can be done in a number of ways, and any one or a combination of the following choices are acceptable to HeGeL

1. By specifying a particular *design unit,* and finding ail the predicates with which the design unit is associated.
2. By specifying a particular *relation* or an *attribute,* and finding all the predicates in which the relation occurs.
3. By specifying that design units are to be assigned in the decreasing order of the number of predicates associated with each of them. This is a form of *most-constrained first* strategy.
4. By specifying a relational predicate between two design units in which at least one of the units is already located. This is a *by reference* strategy.
5. By specifying a predicate in which an attribute of a design unit depends on the given *site elements* (e.g. windows).
6. By selecting at *random.*

To illustrate, if the selection strategy for the initilization file in Fig.3-13 were specified as a combination of CE and privacy, HeGeL would find predicates as illustrated in Fig.3-15. From the set of initialized predicates, a subset is created containing only those predicates that are active. From this subset another subset is created with those predicates in which CE occurs, and so on. In a sense, selection criteria are treated as successive filters which let through only certain predicates insuring, eventually, the identification of one or only a few predicates.

Predicates Initialized	Predicates Active	Selection strategy args. (predicates associated with) CE	privacy
Pi	pl		
?2	?2		
P3	F3	p3	
P*	P4	p4	p4
P5	F5	F5	
p6	p6	p6	
p7	p7		
P8	p8		
P»			
plO			
pll			
pi:			
pl3			
P14			
PIS			

Figure 3-15: Predicate Selection

If HeGeL is moving forward i.e. not backtracking (explained in in Sec.3.4.6), the user interactively inputs predicate selection criteria. If HeGeL is in a backtrack mode, then it will find all pertinent predicates since the process history records the sequence in which design units were assigned previously. In either case, this process may lead to three possibilities for a given predicate selection strategy: (a) no predicate is

found in which case predicate selection process has to start again, (b) a unique predicate is found then HeGeL moves forward to the process of generating alternative locations for a design unit, and (c) more than one predicate are found in which case the user has to interactively select one of those predicates. Currently, HeGeL interacts with the user in either of these three situations. The first and the second possibilities can be trivially automated, while the third possibility requires addition of substantial domain knowledge to HeGeL to full automation. In Sec.4.3.2, we discuss some of the heuristics applicable to this category.

3.4.4. Generation

Once a predicate is selected and designated as a generative constraint for a particular design unit, HeGeL is ready to generate alternate locations for the design unit. As noted previously, a generative constraint (i.e. a predicate) may involve one of the following two situations:

1. The constraint deals with some relationship between the design unit that HeGeL is attempting to locate (DU) and another design unit which we will call the reference design unit (RDU). If the RDU is not yet located then HeGeL cannot proceed, and the user interacts with it to either return to the earlier state of predicate selection (Sec.3.4.3) or directs it to first locate the RDU instead of DU.
2. The constraint deals with some attribute of the unit to be located (DU).

HeGeL generates alternate locations depending on the relationship or attribute specified in the generative constraint. It is important to note that any given predicate may be treated as a generative constraint or an evaluative criterion. In either case, a relationship or an attribute associated with a predicate has the same meaning except that (a) if used in a generative form, a specific relationship or an attribute becomes the prime parameter for generating alternate locations for the DU (i.e. generating locations that satisfy the constraint), and (b) if used in a testing phase, generated locations are checked to see if they satisfy the interpretations associated with a specific relationship or an attribute.

A generative constraint may have two design units DU and RDU. If the constraint deals with direct access, HeGeL projects dimensions of DU from all corner and intermediate vertices of RDU to compute relative coordinates of all possible locations for DU (Fig.3-16).

When the generative constraint requires easy access, HeGeL first generates locations as if the specified relationship were *direct access* as explained above. Next, HeGeL finds ail design units that are directly adjacent to RDU. From all corner and intermediate vertices of each such design unit(AU), HeGeL projects dimensions of DU to compute possible locations (Fig.3-17).

When the generative constraint requires natural light, each corner vertex of the existing windows on the site is taken as a reference point from which dimensions of DU are projected and locations computed (Fig.3-18).

When the generative constraint requires privacy, HeGeL projects dimensions of DU from all corner and intermediate vertices of the site as well as all the design units that are already located. At this point, it is not necessary to consider whether the constraint specifies privacy of DU in reference to another unit RDU

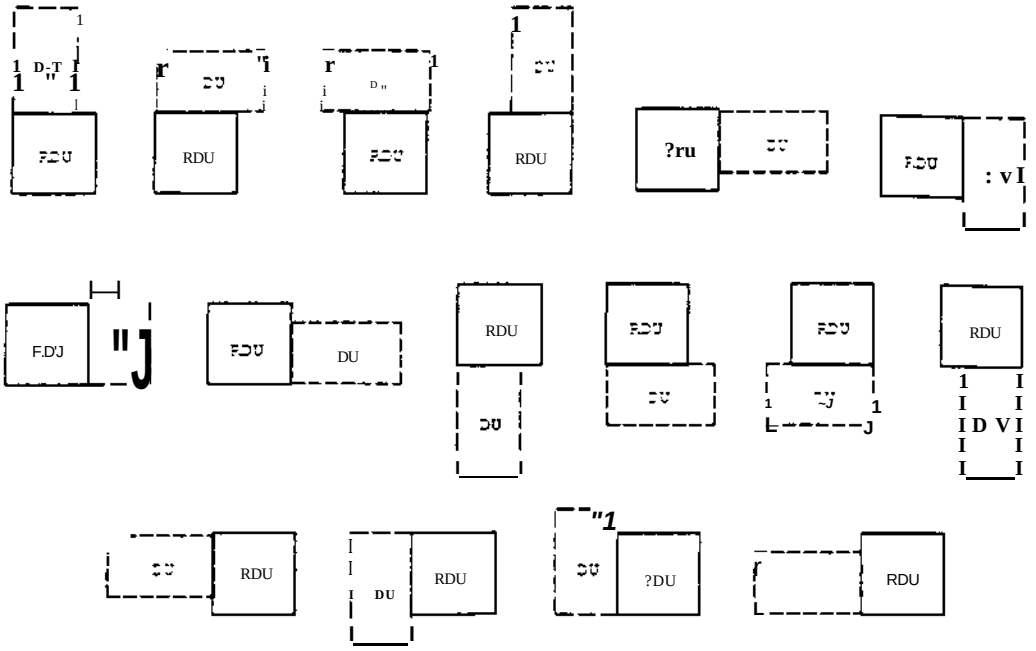


Figure 3-16: Generating locations: Direct access

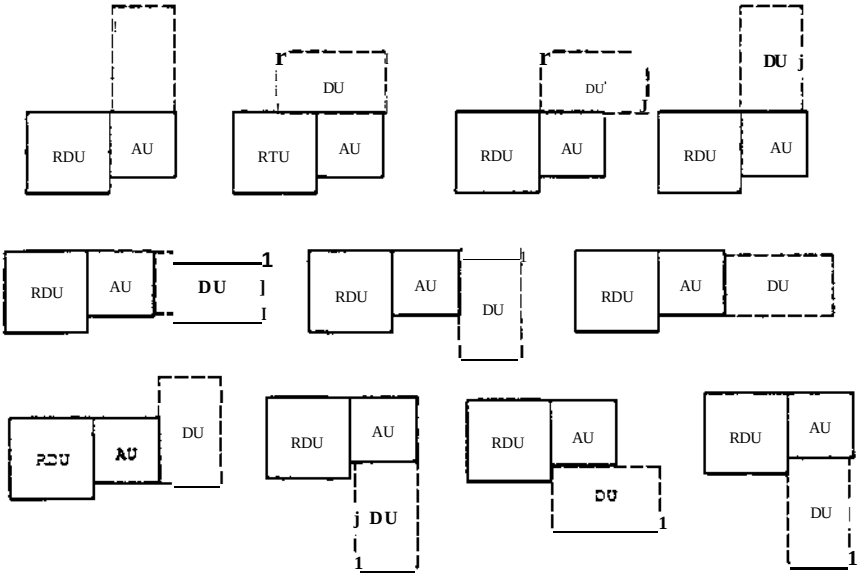


Figure 3-17: Generating locations: Easy access

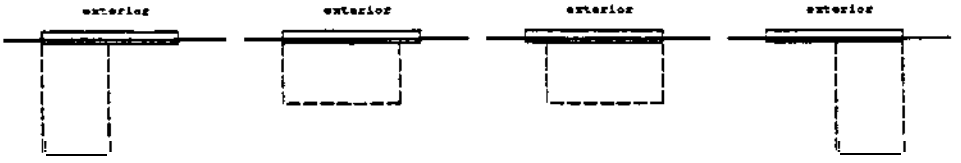


Figure 3-18: Generating locations: Natural light

or not. During the subsequent stage, the test procedures assign solid boundaries depending on whether

an RDU is specified in the generative constraint or not.

All the generated locations are stored in an array. Some of these may be duplicate locations, e.g. if a given DU is square in shape. HeGeL scans the array of possible locations and drops any locations that are duplicated. Next, all the locations that fall outside the site boundaries by more than 10.0 cms. are deleted from the array. Lastly, HeGeL drops any location that overlaps by more than 30.0 cms. with any other design unit located previously. Locations that fall ouside the site by a margin of 10.0 cms. or overlap with another design unit by a margin of 30.0 cms. are permitted since some flexibility in floor area is acceptable in most solutions.

Once this process of generating and filtering locations is complete, HeGeL already may have come up with one or more locations for a DU that satisfy a particular generative constraint, and it proceeds to the next stage of testing them against other predicates associated with DU. It may also happen that no locations are generated because either they fall outside the site or they overlap with previously located design units. In this case, HeGeL undertakes backtracking as explained in Sec.3.4.6.

3.4.5. Testing

Once HeGeL has generated alternate locations for a design unit (DU), such locations are tested against all predicates that are pertinent to DU. As noted earlier, a set of predicates in which DU appears is identified during the predicate selection stage. From this set, one predicate is selected as being a generative constraint according to which alternate locations for DU are generated; hence that particular predicate is already satisfied. The remaining predicates in this set are treated as test criteria (Fig.3-19), and the generated locations need to be tested to ensure that they satisfy test criteria.

Predicates Initialized	Predicates Active	Selection strategy args. (predicates associated with) CE	privacy	Generative Constraint	Test Criteria
pl	pi				
?2	p2				
p3	F3	?3			F3
P<	P4	p4	p4	F-4	
p5	p5	p5			p5
F-6	p6	*p6*			p6
P7	P7				
?8	P3				
p9					
plO					
pll					
pl2					
P13					
pi4					
plS					

Figure 3-19: Set of Test Criteria

While testing the locations, HeGeL interprets each criterion as described previously (Sec.3.3.2). If the criterion deals with *direct access* between two design units, DU and RDU, HeGeL attempts to infer if any one edge of DU is coincident with any one edge of RDU. If such an edge is found, that location of DU is considered to satisfy the criterion, else that location is dropped. Similarly, if the test criterion deals with *easy access* between two design units DU and RDU, HeGeL attempts to find a third unit AU such that

each pair of DU and AU, and RDU and AU satisfy the criteron of direct access (thereby ensuring a path between DU and RDU). If the test criterion deals with *natural light,* each edge of DU is compared with locations of available windows. If there is an overlap, the location fulfills the criterion. If the test criterion deals with *privacy* for a design unit DU with respect to another unit RDU, the generated locations are first sorted by decreasing order of the distance between DU and RDU. And HeGeL selects a location for DU that is the farthest from RDU.

Once all relevant tests have been carried out, it may happen that none of the generated locations successfully passed ail the test criteria. In such a situation, HeGeL backtracks as explained in Sec.3.4.6. On the other hand, there may be one or more locations that successfully pass all the test criteria. If a unique location is identified (either because only one location passed through or more than one location passed through but they are ordered by distance) then HeGeL establishes if any edges of DU need to be assigned an attribute *solid* (since one of the predicates associated with a DU may specify privacy). If more than one location are available but they are not ordered by distance, then HeGeL displays all successful locations and the user selects one of them as a final location for the DU. The remaining successful locations, if any, are stored as alternate locations for that particular DU and may be utilized later during backtracking.

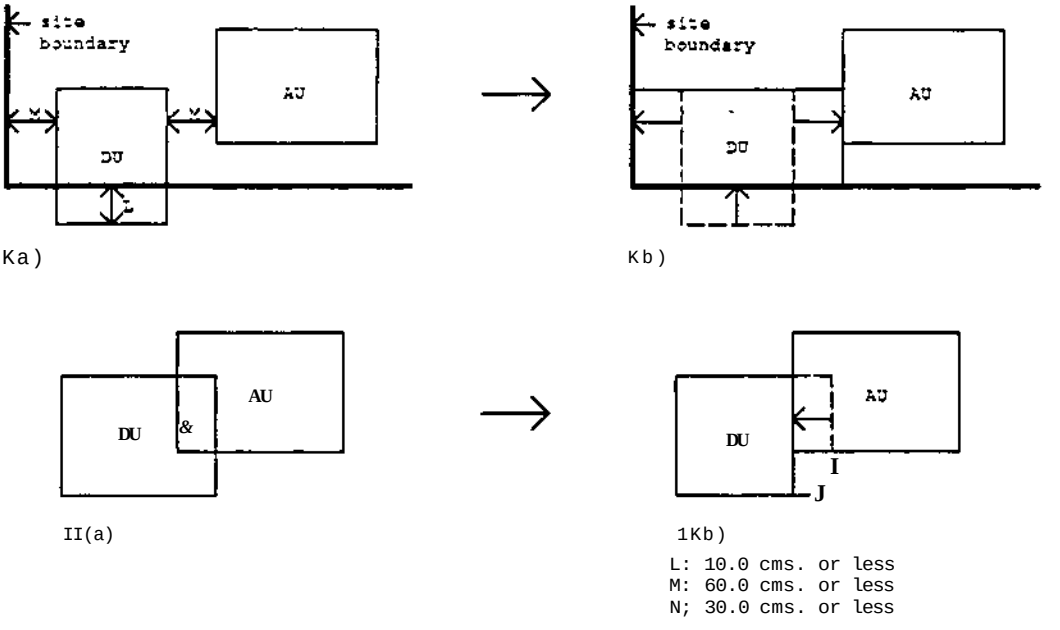


Figure 3-20: Shrink and Stretch Operations

Finally, the DU is assigned the uniquely identified location coordinates. At this point, HeGeL makes appropriate adjustments, if any, to the location coordinates of DU. If a selected location either falls outside the site or overlaps with any other other design unit within allowable margins, HeGeL shrinks the DU. If the DU is located in such a way that there is another design unit or a site boundary only 60.0 cms. away or less, HeGeL expands the DU in that direction (Fig.3-20). Since in our specific design problem all the windows are along the site boundaries, stretching a design unit in relation to windows is subsumed

under stretching operations in relation to the site boundaries and no additional operations need be defined for this purpose.

Once a location for DU is adjusted and assigned, HeGeL updates adjacencies for all the design units located so far. For each design unit, any other unit that is directly adjacent to it in a cardinal direction is stored. In this way, HeGeL maintains an accurate record of assignments as well as other attributes and relations among the located design units. At this point, if any design unit remains to be located, HeGeL returns to the predicate selection stage (Sec.3.4.2). After a number of such generate-and-test cycles, all the design units are located and the task is accomplished. If HeGeL cannot successfully locate a design unit, it undertakes backtracking as explained in the next section.

It should be noted that the order in which test operations are applied to the generated locations is not important since eventually only those locations that pass all the test criteria are considered as candidate locations for a design unit. On the other hand, test operations are applied only to the design unit being located and HeGeL, in its present form, cannot ensure that none of the previously located design units get affected in the process. Although this is a serious concern, we have not dealt with it in the present version of HeGeL.

3.4.6. Backtracking

Whenever HeGeL is unable to locate a design unit during either generation or testing stage, it employs backtracking to find alternate locations for any of the previously assigned units and then reattempts to locate the design unit. In a very strict sense, HeGeL employs a simple chronological backtracking mechanism.

As described earlier, HeGeL maintains a chronological history of the order in which design units are attempted to be located. With each successful predicate selection operation, an additional design unit is inserted into the history list. For each design unit, all generated locations that successfully pass test criteria pertinent to that design unit are stored as alternate locations for that unit. Not all the located design units may have such alternate locations. In essence, combining the history list consisting of the order in which design units are attempted to be located and possible alternate locations for each of the units can be depicted as shown in Fig.3-21.

According to the history list in Fig.3-21, the sequence of assignments of design units reads: S, CE, C. While S shows three alternate locations *(L1, L2, L3),* CE has two such locations *{L4, L5)* and C only one *(L6).* Now if SE were to be located next and no possible locations could be found for SE, backtracking mechanism works as follows. Note that SE would already be inserted in the history list, only it is not yet assigned any location. HeGeL searches backwards through the history list to find the first design unit with alternate locations. When HeGeL cannot find any locations for SE, it will traverse backwards in the history list to find a unit located preceding SE, namely unit C. Since there are no alternate locations associated with this unit- C, HeGeL will backtrack once more to find the preceding unit in the history list, namely unit CE, and it would find an alternate location for CE *(L5).* Once such a unit with alternate locations is found, HeGeL deassigns design units including and following that unit in the

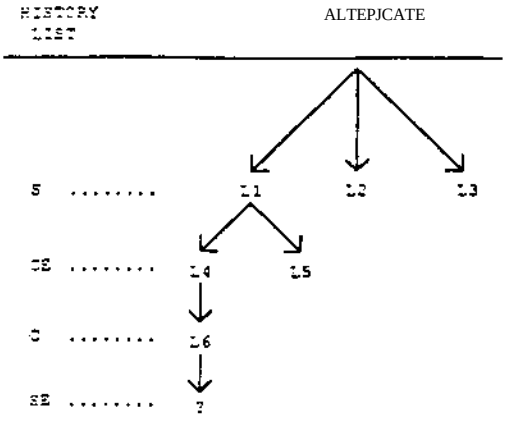


Figure 3-21: Backtracking mechanism

history list. In the above example, CE and C would be deassigned. No action for SE is necessary since it is not yet assigned. Note that this operation removes locations assigned to selected design units and updates adjacencies for other design units that remain unaffected by backtracking. HeGeL does not modify the history list which is left intact. As long as HeGeL performs chronological backtracking, it also need not adjust the dimensions of the remaining units after backtracking. This is due to the fact that shrinking and stretching operations are always performed on the DU that is most recently located. At the same time, HeGeL cannot update the remaining units in terms of their edge attributes like *solid* that may have been previously required by design units with *privacy* relations.

Once this process is complete, HeGeL picks up the alternate location for the selected unit, i.e. location *L5* for CE. Since all alternate locations for a design unit are stored only if they satisfy all test criteria pertinent to that unit, HeGeL does not repeat any tests except in one condition. If the design unit for which an alternate location is found appears in some predicate that specifies *privacy* then the alternate location needs to be checked to find if all or a few edges need to be assigned an attribute *solid.* Next, this location is post-processed for any required shrinking or stretching as described earlier, followed by adjacency update for all the design units located so far. Once a design unit is successfully relocated and all appropriate bookkeeping is complete, HeGeL goes about locating all other design units that were deassigned.

This is accomplished by simply following the history list In the above example, once SE could not be located, the backtracking mechanism deassigned CE and C. Next, CE is relocated and HeGeL scans the history list to find a design unit that follows CE, namely C. HeGeL returns to the stage of predicate selection and finds all predicates associated with C. Once a predicate is designated as a generative constraint for C, generate and test operations take care of locating C. Once again, HeGeL scans the history list to find a design unit following C, namely SE and the cycle continues until all the units in the history list are located. In the end, if all the units are assigned then the task is finished. If HeGeL successfully locates all units in the history list but all units in the design problem are not yet located, it returns to the stage of predicate selection (Sec.3.4.3) as a normal sequence of operations. In that case, the next design unit will be inserted in the history list unlike backtracking mode which does not delete or

insert new design units in the history list.

This is a very simple chronological backtracking mechanism which, if needed, will exhaustively search the entire space of available alternate locations for all the design units. If even after exhaustively searching all possible locations no solution is found, current set of *active* predicates will be interactively modified. This operation corresponds to restructuring of the problem.

3.5. Review: Sequence of Operations

A paradigm for the designers' behavior was presented [Akin87] in terms of two functionalities: *Problem (Re)structuring* and *Problem Solving.* In the preceding sections, we described a computer program-HeGeL, that operationalizes these two functionalities. A complete and detailed sequence of operations carried out by HeGeL is illustrated in Fig.3-22.

At the start of the session, HeGeL is given the problem definition in the form of an initialization file (Fig.3-22, link 1). The initialization file specifies a set of predicates some of which are active and the others passive. Each time the status of a predicate is changed from active to passive or vice versa, HeGeL prompts for a generation mode (Fig.3-22, link 2), the options being generation by pattern or by zoning. If the latter is selected (Fig.3-22, link 4), HeGeL has to undertake solution development by zoning, a component not currently encoded. If the former is selected (Fig.3-22, link-3), HeGeL proceeds to the stage of predicate selection (Fig.3-22, link 5). If HeGeL is not backtracking, predicate selection is interactively carried out based on various criteria, otherwise predicates are selected according to the design units in the history Jist (Fig.3-22, link 6). Depending on the number of predicates identified (Fig.3-22, links 7, 8, 9), HeGeL may be directed to designate a unique predicate to be used as a generative constraint (Fig.3-22, links 11, 12). If no predicate is identified (Fig.3-22, link 10), HeGeL returns to predicate selection phase (Fig.3-22, link 38).

If the generative predicate contains a reference unit that is located, HeGeL generates possible locations for a design unit according to the specified relationship (Fig.3-22, links 14, 17). If the reference unit is not located (Fig.3-22, link 13), HeGeL is directed to locate that unit first by generating alternative locations (Fig.3-22, links 15, 16). If one or more locations are generated (Fig.3-22, link 18), HeGeL tests each of these locations according to test criteria (Fig.3-22, link 20). A successful location is selected and post-processed for any adjustments required (Fig.3-22, links 22, 23, 26, 27), and a design unit is assigned the location coordinates. If another design unit remains to be located as called out in the problem definition, HeGeL returns to the stage of predicate selection (Fig.3-22, links 30, 37, 38). If all the design units are successfully located, HeGeL can be directed to search for another solution (Fig.3-22, links 37, 39) or to stop (Fig.3-22, link 36).

If no location is generated (Fig.3-22, link 19) or if the generated location(s) could not pass all the test criteria (Fig.3-22, link 24), HeGeL backtracks (Fig.3-22, links 21, 25). Currently, HeGeL is equipped to undertake chronological backtracking (Fig.3-22, link 31) by searching through the history list to find a design unit that has alternative locations. If such a unit is found (Fig.3-22, link 32), HeGeL deassigns all

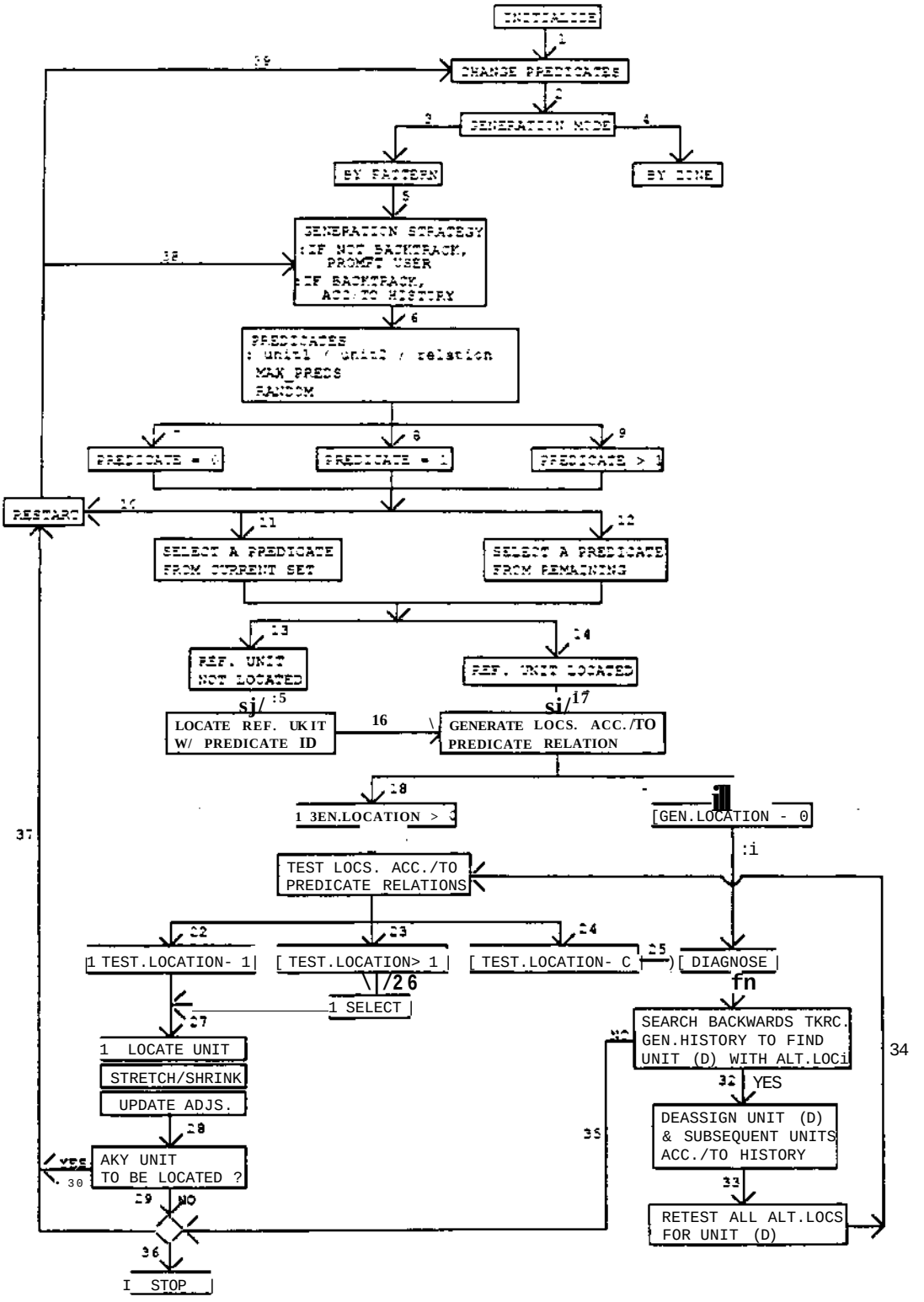


Figure 3-22: Sequence of operations

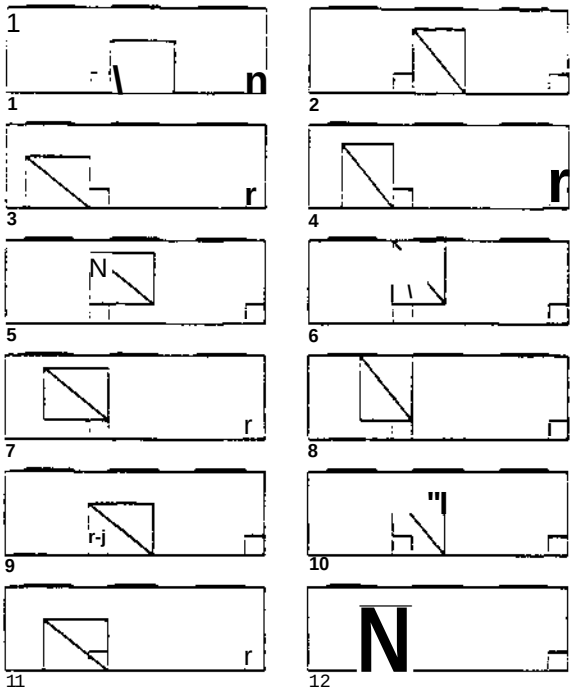
the design units including and following that design unit, picks up an alternative location that is then subjected to test criteria pertinent to that design unit (Fig.3-22, links 33, 34). Having successfully located a

design unit to an alternate location, HeGeL attends to the task of locating the remaining design units by repeating the previous operations (Fig.3-22, links 30, 37, 38). If HeGeL cannot find any design unit with alternative locations (Fig.3-22, link 35), it can be directed to either change the predicate set (Fig.3-22, links 37, 39), or to stop (Fig.3-22, link 36).

3.6. Sample Runs

In order to illustrate various features of HeGeL described in previous sections, a sample run is presented below. The left hand column shows alphanumeric component of HeGeL, letters in bold indicate user input. The right hand column shows current stage of design development in a graphical format.

```
% HeGeL

 Name cf initialization file : D»ta3.r»a

 Generate by 'pattern' or 'z:ne' ? pattern

 Enter ;an. strategy to select predicates,
finishing with 'end' : S %nd

 Predicates identified: pi p2 p4 p7 p3
Select: <ID> (predicate from current set)
             (predicate not in current set)
        F(ire new generation strategy) : pi

 Net assigned yet. Select rr.ain aocr : Dl

 Passing generated locations to test_procs.

Goal: test_locs Status: pi Locating: S

More than one locations are possible.

1: 150 480 390 780
2: 90 480 390 720
3: 150 90 390 390
4: 90 150 390 390
5: 60 390 300 €90
6: 0 390 300 630
7: 60 180 300 480
8: 0 240 300 480
9: 150 390 390 690
10: 90 390 390 630
11: 150 180 390 480
12: 90 240 390 480

Select a location : 8

 Located design unit S 0 240 300.480
```

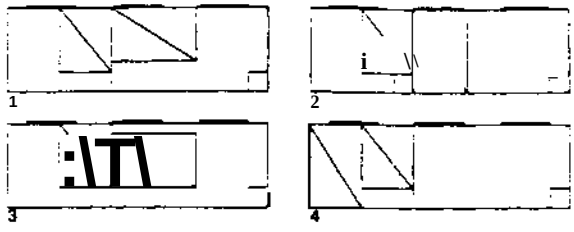


Altarnat* location* for dasign unit- S

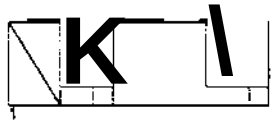
Figure 3-23: Locating unit S

HeGeL reads in data from an initialization file (Fig.3-13), followed by a selection of the solutio development strategy- by pattern or by zoning. Next, HeGeL identifies all predicates associated with 5 i.e. *p1, p2, p4, p7, p8.* From this set, *p1* is selected as a generative constraint based on which possib locations are generated, while the rest are used as criteria for testing generated locations. Befc generating locations for S, HeGeL prompts for assignment of main door(Dm) since predicate *p1* specif *direct access* between S and Dm. From 12 acceptable locations, unit S is assigned location #8 (F 3.23.8). Note that any other location is equally valid and they all are stored as alternate locations for S.

```
Inter ;9r.. strategy to ssle:: predicates,
finishing with 'end' : SI «ad

Predicates identified: p7 p2
Select: <I2> (predicate from current set)
             (predicate net in current set)
        F(ire new çer.eraticn strategy) : p2

Passing generated loraticr.s to test_prccs.

Seal: test_lccs Status: p2 Locating: SE

Mere than one locations are possible.

1: 0 480 250 S"0
2 : 0 430 290 "30
3: 50 480 300 S~0

4: 0 -10 390 240

Select a location : 4

L:cated design unit SE 0 0 29C 240
Enter gen. strategy to se.ect predicates,
finishing with 'end' : CS «nd

Predicates identified: p3 p5 p6 p4
Select: <Z2> (predicate from current set)
             (predicate not in current set)
        F(ire new generation strategy) : p3

Passing generated locations to test_procs.

Goal: test_locs Status: p3 Locating: CE

Located desian unit CE 0 910 300 1200
```



Alternate location* for d*aign unit.- SB



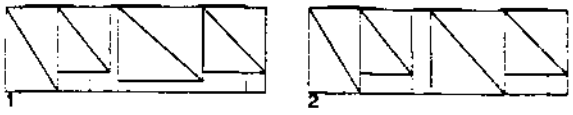
Alt«raat« location* for design unit- CX

Figure 3-24: Locating units SE and CE

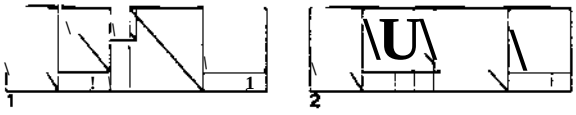
Having located S, HeGeL generates and tests locations for unit SE. There are four acceptable locations from which location #4 is selected for SE. Note that location #4 falls outside the site by 10 cms. (which is within the acceptable margin of 30 cms.). Before actually locating SE, HeGeL shrinks the area so as to bring it within legal bounds of the site as reflected in the final locational coordinates of SE. The other locations are stored as alternate locations for SE.

Next, HeGeL locates unit CE in a similar fashion. In this case, HeGel does find a unique solution after generating locations and testing them against all applicable test criteria. Since there is only one acceptable location, HeGeL takes the proper action of assigning coordinates to CE without requiring user interaction.

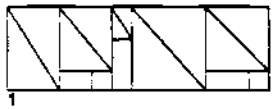
```
£r/.er gen. strategy :: s^lert predicates,
finishing with 'end'  :  C and

 Predicates identified: p6
Select: <ID>  (predicate from current set)
              (predicate not in current set)
        F(ire new generation strategy)  :  p6

 Passing generated locations to tes:_pr::s.

Goal: test_iocs Status: p6 Locating: c

More than cne locations are possible.

1: 0 520 340 910

Z: 0 5"0 390 9-10

Select a location :  2

 Located design unit Z 0 5^0 390 910
Enter gen. strategy to seleot predicates,
finishing with 'end'  :  R •nd

 Predicates identified: p8
Select: <IC>  (predicate from current set)
              (predicate net in current set)
        F(ire new generation strategy)  :  p8

 Passing generated locations to test_procs.

Goal: test_locs Status: p8 Locating: R

More than, one locations are possible.

1: 0 480 150 600
2: 150 480 300 600

Select a location :  1

 Located design unit R 0 480 150 570

All units located. Finished ...!
G«n.Cycl«:<l>: S SE CE C R
```



JLLtarnat* locations for **daaign** unit- C



Alt*xn»t« **locations for design** xuxit- **R**



final Solution

Figure 3-25: Locating units C and R

Here, HeGeL finds two acceptable locations for C, out of which location #2 is selected and the other is stored as alternate location for C. Similarly, two acceptable locations are found for unit R. Note that both locations for R overlap with unit C within allowable margin (30 cms. or less) and hence are considered acceptable. Once location #2 is selected for unit R, it is shrunk in the direction of overlap with C and adjusted coordinates are assigned to R.

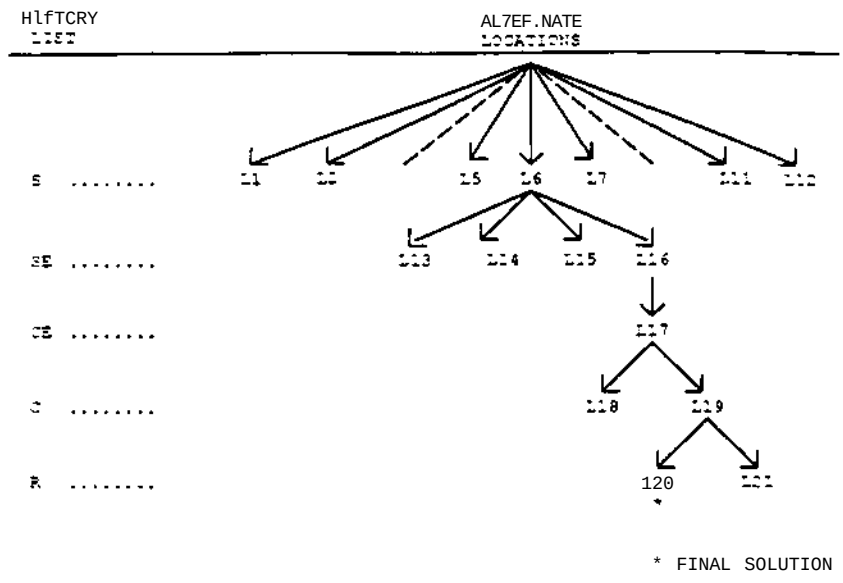


Figure 3-26: Search space generated

In the preceding session with HeGeL, a number of alternate locations were available for most design units. Topology of the search space explored in this particular session is illustrated in Fig.3-26. Note that even if some other node in the search tree were selected, backtracking mechanism would have found the current final solution since the search space would not be altered as long as the set of *active* predicates is not altered. To illustrate, in Fig.3-26, locations *L1 through L12* for the unit S satisfy all the *active* predicates that are related to the unit S. Although the unit S is assigned location *L6,* the remaining locations are stored as alternate locations. If a location other than *L6* for S were selected and subsequently other units could not be located for some reason, HeGeL would backtrack and the unit S would be assigned the next available location from the list of alternate locations. Eventually, HeGeL would assign location *L6* to the unit S and the solution illustrated in Fig.3-26 would be found. It should be stressed that the alternate locations for a design unit are in reference to a particular set of *active* predicates. As long as this set of *active* predicates is not changed, backtracking mechanism by performing exhaustive search will find a solution if one exists in the current search space.

At this point, all the units are located and an acceptable solution has been found. HeGeL can be directed to stop here or search for another solution as shown in the next segments. This feature of HeGeL corresponds to the behavior of subjects found in our protocol studies. Some subjects, after having found an acceptable solution, attempted to search for another, possibly better solution. Some subjects searched for another solution without changing any of the predicates asserted previously while some others changed the predicates. Additionally, if even after an extensive search a solution was not found, subjects (and HeGeL) searched for a solution by changing the predicates asserted. This amounts to restructuring the design problem. We call each such (re)structuring an *episode* in the design process. Figures 3-23, 3-24 and 3-25 together constitute one episode; the following segments constitute another such episode.

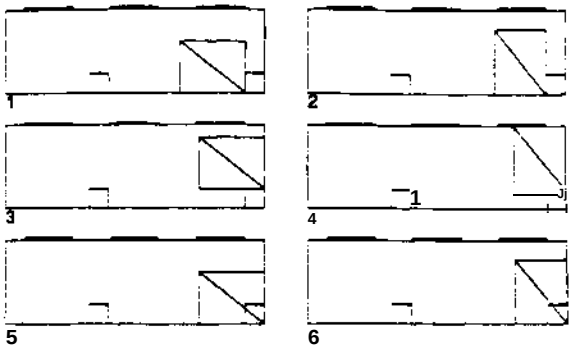
```
Search frr ar. :tr.«r s-rlutier.? ;y, r.." : y

Change main door? [y, n]: y

New main door: DC.

Initialized set :f predicates contains:
?R£D_ID: p9 passive C SE private
PR£D_ID: plO passive C SE easy
FR£D_ID: pll passive C CE direct
FR£D_ID: p!2 passive C CE easy
PPZD_ID: p!3 passive C nil private
FRED_ID: pi4 passive C Dm easy
?RED_I2: pi5 passive C nil light
FRED_ID: pi active £ Dm direct
FP£D_ID: p2 active SE £ direct
?RED_ID: p7 active SE S private
FF£D__ID: p4 active CE S private
FRED_ID: p5 active CE r.ii light
FR£D_ID: p3 active CE Ds direct
?RED_ir: p6 active C CE direct
FR£D_ID: ?& active R S direct

Change predicates- 'active' to 'passive'.
Enter <ID's>, finishing with 'end': p3 •&d

Changed predicate p3 to passive.

Change predicates- 'passive' to 'active'.
Enter <ID's>, finishing with 'end': p9 end

Changed predicate p9 to active.

Restructured predicate set.

Generate by 'pattern' or 'zone' ? pattaxn
```

Figure 3-27: Restructuring

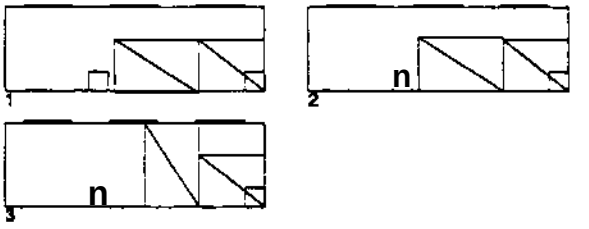
In order to search for another solution, some problem parameters are changed as shown in Fig.3-27. Two global changes are executed in this segment. First, the main door, i.e. entrance into the office is changed from the previous episode. Second, one of the active predicates- *p3* is changed to passive, whereas predicate *p9* is activated. In this example only two predicates are changed, i.e. their status is modified, however, it is possible to modify any predicate in the initialized set of predicates.

Additionally, HeGeL prompts for a solution development strategy- by pattern or by zoning. This feature corresponds to the behavior of some subjects who, after pursing one strategy, decided to switch to another development strategy. Following this selection, HeGeL continues in a fashion similar to the earlier example.

```
Enter ger.. strategy t: scle:: rrs-iratss,
finishing with 'end' : S and

Predicates identified: pi p2 p" p4 pS
Select: <!!:> (predicate frrm currant set)
             (predicate not in current set:
        F'ire new generati:n strategy; : pi

Passing generated Irrations to :*st_cr::s.

3cal: test_locs Status: pi Lrrating: S

More than cne locations are possible.

1: 150 S10 290 1110
C: 90 S~0 290 1110
2: 60 900 300 1200
4: 0 960 300 12 00
5: 150 900 390 1200
6: 90 960 390 1200

Select a location : 5

Located design unit S 150 900 390 1200

Enter gen. strategy to select predicates,
finishing with 'end' : SB «nd

Predicates identified: p9 p7 p2
Select: <ID> (predicate from current set)
             (predicate not in current set)
        F(ire new generation strategy) : p2

Passing generated locations to test_procs.

Goal: test_locs Status: p2 Locating: SE

Mor« than on© locations ar« possible.

1: 150 510 400 900
2: 140 510 390 900
3: 0 650 290 900

Select a location : 1

Located design unit SE 150 480 3 90 900
```



Alternate location* for d*«ign unit- S

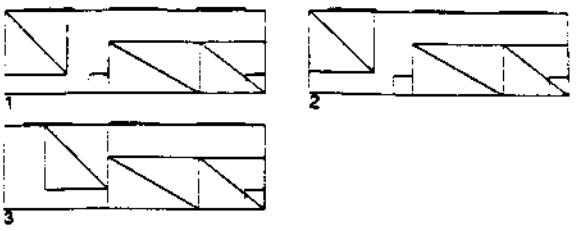


Xlt«r&at« location* for d*»ign unit- SI

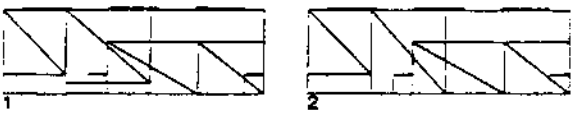
Figure 3-28: Locating units S and SE

Having restructured the problem parameters, HeGeL develops acceptable locations for S and SE according to pertinent constraints and criteria for each unit. Next, HeGeL assigns coordinates to both these units according to the selected location while the others are stored as alternate locations. Unlike the previous episode wherein main door was not assigned initially and had to be specified before locations for S can be generated, here HeGeL has made pertinent modifications for both doors during the restructuring stage (Fig.3-27).

```
Enter ;sr.. strategy to selert predicates,
finishing with 'end'  :  CX aod

Predicates identified: p5 p6 p4
Select: <ID>  (predicate from current set)
              (predicate not in current set)
        F{ire new generation strategy)  :  p4

Passing generated loratirns to test_proos.

Goal: test_locs Status: p4 Locating: CE

Located design unit CE 0 0 300 290

Enter gen. strategy to select predicates,
finishing with 'end'  :  C *nd

Predicates identified: ;9 p€
Select: <I2>  (predicate from current set)
              (predicate not in current set)
        F{ire new generation strategy)  :  p€

Passing generated locations to test_prors.

No locations are generated.
```



Xlt«znat« locations for d*«ign unit- CX



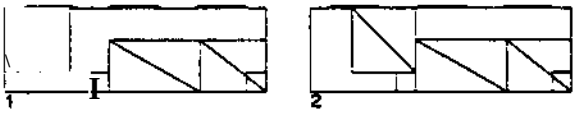
Unacceptable location* for d*aign unit- C

Figure 3-29: Locating units CE and C

Next, HeGeL finds three acceptable locations for CE which are first sorted in decreasing order of distance from unit S as required by predicate *p4.* Once these locations are ordered, HeGeL assigns CE to the first location (which is the farthest from S), the other two locations are stored as alternate locations.

Next, HeGeL attempts to generate locations for C without success. This is because both possible locations that can be generated overlap with SE exceeding the allowable overlap margin. This forces HeGeL to backtrack as seen in the next segment. Note that the unit C is not yet located but it is already inserted into the history list.

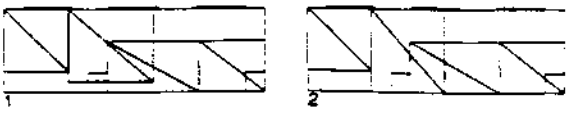
```
Backtracking to search for alternate locations.

Child & Parent: C CE

Goal: try_alt_loc3 Status: CE

Exited PROC: find_alt_locs...YES.

Located design unit CE 0 0 290 300
```



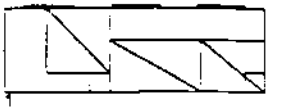
Alternate looatioas for d*sign unit- CX

Figure 3-30: Relocating unit CE

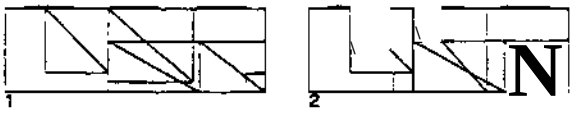
```
Generate arocrdmg :: Hist:ry_list: C

Predicates identified: p£ ;S
Seleot: <ID> (predicate fr:ir. current set)
             (predicate net in current set)
        Fiire new generation strategy) : p€

Passing generated locations :: :est_pro:s.

Uo locations are generated.
Backtracking to search for alternate locations.

Child i Parent: C CE

Seal: try_alt_locs Status: CE

Edited ?ROC: find_alt_locs ... YES.

•Located design unit CS 0 190 300 480

Generate according to Histcry_list: C

Predicates identified: p9 p6
Select: <ir> {predicate from current set)
             (predicate not in current set)
        F(ire new generation strategy) : p6
Passing generated locations to test_procs.

No locations are generated.
Backtracking to search for alternate locations.

Child & Parent: C CE
Goal: try_alt_locs Status: CS

Exited PROC: find_alt_Iocs...NO.
```



Unacceptable locations for d»»ign unit- C



Altarnat* locations for daaiga unit- **CS**



Unacceptable locations for d*«ign unit- C

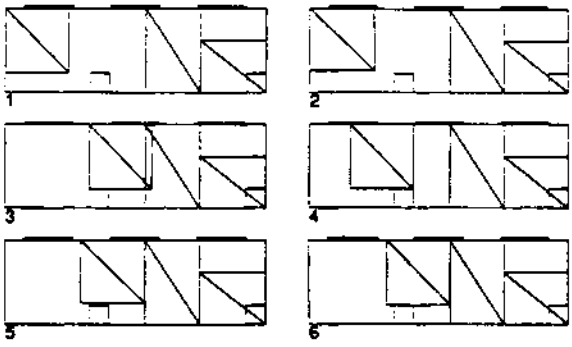
Figure 3-31: Relocating units CE and C

During backtracking the HeGel tries to determine if the preceding unit- CE, has any alternate locations. Note that CE, as shown in Fig.3-29, has alternate locations that are already ordered by distance between CE and S. Once an alternate location is found, HeGeL assigns CE to the new location and updates data regarding all the units assigned so far.

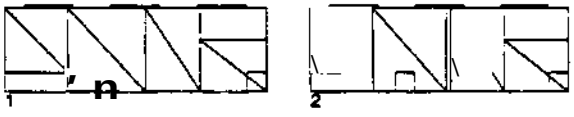
Next, HeGeL retrieves unit C from the history list, which is yet to be located. Once again, HeGeL attempts to find possible locations for C without any success and backtracks as before. After going through a similar cycle, HeGeL exhausts all alternate locations for CE without being able to locate C. At this point, HeGeL backtracks one more level to the unit preceding CE, namely SE as shown in the next segment.

```
 Child « Parent: CE SE

E:-:ite<S?ROC:fir.d_alt_lzzs...YES.

Deleting adjs.from ur.it: SE ref. : CE

Deleting aṡjs.fr^m ur.it: S r<=f.: SE

Mere than cr.e Icrati-r.s are possible.

1: 140 510 390 900
2: 0 650 390 900

Select a locatirn : 2

 Located dasijr. unit SE 0 65C 390 900

Generate acccrding tc Kistrry_list: CE

Predicates identified: p4 ?5 ç6
Select: <ID> (predicate frcm current set)
             (predicate net in current set)
        F(ire new generation strategy) : p4

 Fassing generated locations to test_procs.

Goal: test_locs Status: p4 locating: CE

Located design unit CE 0 0 300 290

Generate according to Histcry__list: C

Predicates identified: p9 p6
Select: <ID> (predicate from current set)
             (predicate not in current set)
        F(ir« new generation strategy) : p<

 Passing generated locations to test_procs.

Goal: test_iocs Status: p6 Locating: C

Located design unit C 0 290 390 650
```

*X±t*mmf* locations for daaign xiait- SB



Alt«zn*t* looationa for d*«ign unit- CB

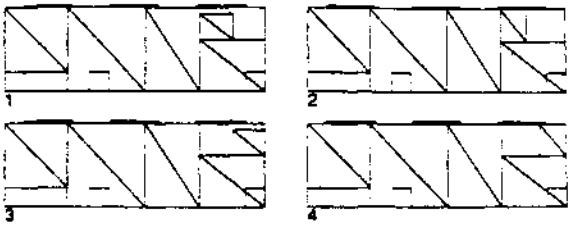


Xlt*rnat« locatioas for d*«ign unit- C

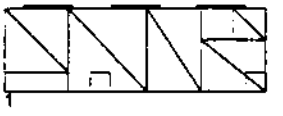
Figure 3-32: Relocating units SE, CE and C

When HeGeL backtracks to unit SE, it finds two alternate locations for SE. Once a location is selected for SE, it is relocated and HeGeL updates data regarding all the units assigned so far. Next, HeGeL picks up CE (the last unit to deassigned) and locates it as described earlier. This is followed by an attempt at locating the next unit in the history list, i.e. C. This time HeGeL successfully locates the unit C and there are no more units left in the history list that remain to be located.

```
 Generate according :: Hist:ry_list:

Enter gen. strategy to select predicates,
finishing with 'end'  :  R and

 Predicates identified: p8
Select: <I^> (predicate from current set)
             •predicate nrt in current set)
        F!ire new generation strategy)  :  p8

 Passing generated locations to test_procs.

Goal: test__lccs Status: pð locating: R

More than one locations are possible.

1: 30 9C'O 150 1050
2: 0 90: 150 1C20
3: 30 i:50 150 i:00
4: 0 1050 150 1200

 Select a location :  3

 Located design unit R 0 1C50 150 1200

All units located. Finished ...!
3en.Cycle:<l>: S SE CE C R
Gen.Cyde:<2>: S SE CE C R

 Search for another solution? [y, n]: a

No satisfied rules
```



Alternate location* for d*«ign unit- R



Final Solution

Figure 3-33: Locating unit R

After having located all the units in the history list HeGeL checks if any units specified in the design problem remain to be located. Having found one, namely R, HeGeL goes through the generation and testing cycle for R, finding four acceptable locations. Location #3 is selected for R which is first stretched since a site boundary exists within the allowable stretching margin. Finally, the adjusted coordinates are assigned to unit R. All the units are assigned and a successful solution has been found during this episode and HeGeL is directed to stop.

4. Evaluation

The current version of HeGeL performs the task of space planning as envisaged. In comparison with performance of the human designers recorded in our protocol experiments, it seems to approximate and arrive at similar final solutions. It is appropriate to ask: (a) does HeGeL validate the paradigm for the designers' behavior proposed earlier, (b) how well does HeGeL perform in terms of the kinds of solutions developed, (c) how could HeGeL be made more efficient, and (d) is a production system model appropriate.

4.1. Validity of the Paradigm

Based on the analysis of protocol experiments, a paradigm for the designers' behavior was proposed earlier [Akin87]. According to the proposed paradigm, the design development process moves back and forth between two phases: *problem (re)structuring,* when problem parameters are established or transformed, *and problem solving,* when these parameters are satisfied in a design solution. HeGeL, was developed in order to validate this paradigm.

The performance of HeGeL leads us to believe that the proposed paradigm is well-founded in its fundamental assumption, namely that a designer has a vast amount of knowledge that is incrementally brought to bear on spatial design problems. The paradigm organizes this incremental process as problem (re)structuring and problem solving. HeGeL organizes these processes in terms of a universe of objects and spaces, and predicates (relationships among objects) which are selectively instantiated and satisfied in generation and testing of design alternatives until a successful solution is found. While the current version of HeGeL validates the proposed paradigm, some additional issues concerning our methodology need to be addressed before the paradigm can be accepted as a theoretical model of the design process.

First, the paradigm was developed based on analysis of protocol experiments of the subjects solving an office layout problem. The paradigm and HeGeL both have been studied and evaluated in a very specific problem context. With a degree of caution and reservation, we believe that the paradigm is general enough to be applied to a wide spectrum of design problems. HeGeL, on the other hand, may not be as general as the paradigm. The paradigm, being highly descriptive and abstract, is concerned with generic concepts while HeGeL is developed to flesh out generic concepts in a specific design context.

Second, since HeGeL is modeled after the paradigm, a successful run of the computer program may not seem to offer any substantial validation. Such a view is only partially correct. A descriptive paradigm is composed of concepts whose precise character may or may not be fully evident a priori. In order to explicate such concepts, they need to be examined under a number of similar situations and a consistent and parsimonious interpretation established. While a descriptive paradigm presented in literal symbols may suffice in specific cases, a computer implementation forces one to completely disambiguate such concepts. A successful execution of the implementation demonstrates predictive power of the underlying model within the limits of assumptions or interpretations incorporated in computational operations. To illustrate, while initially posing the paradigm, we identified *privacy* as one of the predicates or relationships. It was only when translating a concept like *privacy* into a computational operation that we

were forced to define a precise interpretation for it.

Third, HeGeL is automated to a specific degree of detail, e.g. structuring and restructuring phases are partially interactive; HeGeL assigns only design units defined as furniture templates and not individual furniture items; it works in a two-dimensional world consisting of rectangular objects; etc. Although not fully implemented and hence serving only as possible avenues for further validation, in the following sections, we discuss the role of spatial representation, a richer knowledge base and heuristics to overcome some of these limitations.

4.2. Final Solutions

In most cases, HeGeL finds a final solution that is very close at a certain level of detail to the solution developed by the subjects in our protocol experiments. As noted earlier (Sec.3.3.1), HeGeL primarily develops solutions that are composed of design units defined as certain dimensional areas subject to certain requirements. Subjects attend to this task and, in addition, also attend to assigning and organizing individual furniture items in each of the design units. HeGeL, at present, is not capable of assigning furniture items and hence does not reflect a finer degree of resolution in its solutions that would otherwise be required. Currently, each design unit is defined in terms of a certain dimensional area by associating each unit with alternate furniture patterns. This approach resembles the behavior of more experienced designers who retrieve stored templates of design units from their experience (either in terms of a collection of furniture items or an approximate area in which appropriate furniture items are then located). Although this feature is lacking at the moment, it can be added as a separate collection of rules to handle furniture assignment for each design unit in a fashion almost identical to the operations implemented in HeGeL for assigning functional areas.

While such operations can accommodate most furniture placements as recorded in our protocols, a few solutions developed by the human subjects pose some intricate problems regarding furniture placements in a functional area. A major representational problem is dealing with some furniture items that are not entirely stationary. To illustrate, a desk is more fixed than a chair that can be moved around in order to open a desk drawer. It is such almost fluid spatial considerations that HeGeL is not equipped to deal with. Another characteristic of some human designers observed in our protocols is an opportunistic tendency to find purposes once a receiving space is generated in response to entirely different objectives. To illustrate, some subjects used bookshelves to double as space dividers. Additionally, the back of the shelves were assigned the role of pin-up surfaces for displaying drawings. Such post-facto assignment of purposes to objects and spaces is more evident in case of experienced designers and this facet of design expertise needs further investigation.

HeGeL is capable of developing solutions in which design units are either contiguous with each other or attached with one or more site boundaries. This is a direct consequence of the underlying representation and operations used, namely that possible locations for a design unit are projected from reference vertices. For the specific task of designing an office layout, this approach does not cause any severe limitations since the site area and the combined area of all the design units are roughly equal. On the

other hand, if HeGeL were to be given a problem with a more generous site area, it will still attempt to pack design units adjacent to each other. If a design unit were to be placed but no specific relationship were given, HeGeL at present cannot locate a unit as a free floating area not attached to any other unit. This, however, is a case which is not of interest to us almost by definition since subjects generate required relationships for given design units from memory almost independently of requirements called out in the problem description. One possible solution to problems of this kind is anticipated in HeGel. By modifying the tolerance values (30 cm, 10 cm, etc.) for stretching of functional areas layouts of HeGel can be easily adapted to larger sites. This requires the parametric declaration of these values. Other approaches to problems of this kind is to make representational distinctions between topological and dimensional properties of design solutions and use an appropriate control strategy to make design decisions. A number of such approaches are described by Steadman [Steadman83] and Flemming [Flemming86].

Although HeGeL does utilize qualitative information about design units, e.g. attributes of boundaries around each design unit, such information in its present form does not allow for more than simplistic inferences. To illustrate, some subjects in our experiments, even if they were working with a two-dimensional layout, categorically stated if a *partition wail* is intended to be going up all the way to the ceiling or only upto a certain height, and if such a partition is intended to serve visual or acoustic purposes. Once such decisions are made or intended, the subjects can select appropriate material for a partition wall. At present, HeGeL does not have a rich knowledge base that would enable it to attend to finer details of solutions.

4.3. Role of Heuristics

As noted previously, there are times when HeGeL is guided interactively. Three major stages are: (a) initializing a set of *active* predicates, (b) selecting a *generative* predicate and (c) selecting a location from a set of generated alternate locations. One additional situation that is not interactive but may be made more purposeful is *backtracking* which currently employs simple chronological backtracking. Lastly, in order to search for another solution after having found either one or none, HeGeL is presently guided by the user to interactively restructure the problem parameters. The following discussion is concerned with highlighting heuristic means by which **a** system like HeGeL can be made more purposeful and substantially automated.

4.3.1. Initializing Active Predicates

As observed in [Akin87], more experienced architects rely on functional patterns- *scenarios,* retrieved from their personal experience of having solved a variety of spatial problems. An *hierarchical* office or a *participatory* office represent examples of such scenarios, and each scenario represents a set of desirable relations among objects or spaces of interest. This is a heuristic strategy by which the designers can impose a global structure on the design problem where none may be given a priori, and thereby focus their attention in solving the problem. HeGeL currently handles atomic relationships in the form of predicates. Although associating a set of predicates with a distinct WM element like a *scenario*

seems straight forward, a fundamental issue needs further exploration: is there a unique mapping between a set of predicates and a scenario ? A scenario like *an hierarchical office* may be interpreted in terms of the status hierarchy of the personnel to be accommodated in the office. An office layout that reflects such a scenario may be achieved through spatial qualities of each functional area separately (e.g. CE may have the most furnished office) or by other physical or visual cues that reflect differing status of each functional area. For a variety of design problems, a unique mapping between a set of predicates and a scenario may not be available, and it may be established only upon further study.

4.3.2. Selecting a Generative Predicate

Given a set of active predicates and a generation strategy, HeGeL identifies relevant predicates to instantiate one as a generative constraint for a specific design unit. Currently, the generation strategy is interactively specified in terms of a number of possibilities (Sec.3.4.3). A possible way to fully automate this process is by using a heuristic *planner* that can propose either partial plans as the design develops or a complete plan which is then executed. Such a plan may comprise of a *ordered by priority* sequence of generation strategies. A few such heuristic design development strategies are:

1. Assign design units that are constrained by some existing site element (e.g. a door or a window).
2. Assign design units in the decreasing order of number of predicates associated with each unit.
3. If more than one design unit are competing for assignment, select the unit that is more significant than others (and hence more inflexible).
4. Assign design units that are associated by a predicate with some other design unit already located.
5. Assign design units not specifically related to any site elements or other design units.

Additionally, each predicate is concerned with a relation or an attribute associated with a specific design unit. Each such relation or attribute may be given different weights. Such an approach may prove useful for discriminating by priority among competing predicates associated with a design unit.

4.3.3. Selecting a Promising Location

HeGeL generates alternate locations for a design unit based on a predicate. At times, HeGeL, even after applying all the test criteria pertinent to that design unit, comes up with more than one acceptable location. One such situation is shown in Fig.4-1. HeGeL has found eight acceptable locations for S and, at this point in time, HeGeL has no obvious way to further discriminate among these locations. The subjects in our experiments sometimes reached similar situations; only they deliberated for a while on possible locations and then selected one that *looks* most promising.

While we have not developed a general framework that accounts for such selectivity on the part of human designers, some heuristic strategies more than others proved to be useful in our runs. Whenever HeGeL came up with a number of acceptable locations, we selected one that seemed most promising and, at the same time, we tried to literally reason why other locations do not look as promising. For example, in Fig.4-1, we have eight acceptable locations for placing S. Location *L5* can be adjusted to its left but

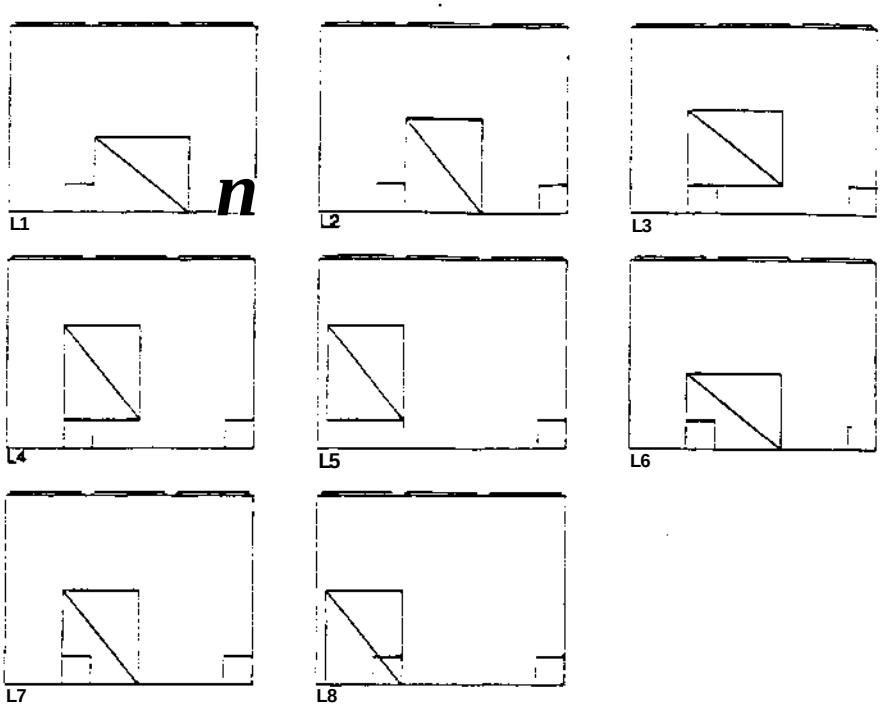


Figure 4-1: Selecting a promising location

wastes a strip of area to its bottom. Locations labeled *L3* and *L4* can be dropped since both contain a narrow strip of area towards the bottom that will be wasted, and area to the left is not sufficient for accommodating any other design unit except R for which it is too large. Locations *L1, L2, L6,* and L7are good enough to accommodate additional design units either to the left or to the right but not both, hence wasteful towards one side and can be dropped from consideration. Thus location *L8* seems to require the least adjustment (i.e. enlargement to its left) without wasting any space towards the interior of the site. So we can be relatively confident that location labeled LS, seems to be the most promising *at this point in time.* Whether it actually turns out to be a fortunate or an unfortunate choice can be confirmed only later on while assigning other design units. It should be noted that with an exhaustive backtracking mechanism, HeGeL will eventually find a solution if one exists. The preceding heuristic strategies will save not only time but also lend a sense of purpose to HeGeL Some heuristic strategies for selecting a promising location from among many acceptable locations are given below:

1. Select a location that has least left-over area on all sides. Here the left-over is defined as a strip of minimum dimension less than the least room dimension.
2. Select a location that requires least adjustment, i.e. least amount of stretching or shrinking on all sides.
3. Select a location that leaves unassigned site area as compactly as possible.
4. Select a location that makes a continuous circulation path of minimum possible length when design units are assigned contiguously.
5. Above four strategies should be interpreted in context of the given design problem and site configuration (i.e. if a design is being developed on a site with a fixed envelope or none, etc.).

4.3.4. Backtracking

One last component of HeGeL that could use some heuristic knowledge is backtracking mechanism. A simple chronological backtracking suffices for a small design problem in which only a small number of active predicates are involved. Once a larger problem is attempted or a large number of relationships are specified in predicates, chronological backtracking may prove highly inefficient. Such inefficiency derives from the fact that backtracking mechanism deassigns certain design units, each of which has to be reassigned. Even if one design unit is not affected by another design unit, HeGeL currently deassigns that unit because it does not have any notion of dependencies among assigned design units.

In Fig.4-2 and Fig.4-3, design units S and SEs are assigned in that order, generative constraints being that S and Dm, and S and SEs should be directly accessible (i.e. adjacent) to each other. Subsequently, when HeGeL tries to assign R so that it is directly accessible to S, it cannot generate any acceptable locations since all such locations overlap with SEs (Fig.4-4) and backtracking mechanism starts up.

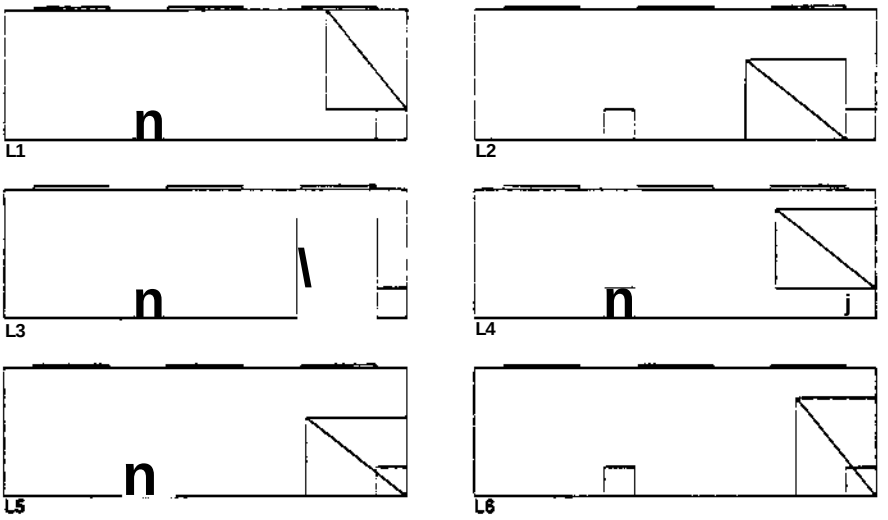


Figure 4-2: Acceptable locations for unit: **S**

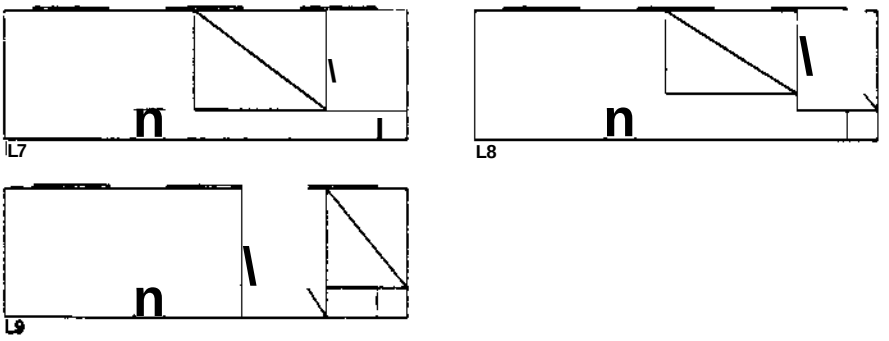


Figure 4-3: Acceptable locations for unit: SEs

At present, HeGeL backtracks to find that the design unit located just preceding the current situation was SEs, and that SEs has alternate locations *(L7, L8,* L9). Having found alternate locations, it will reassign SEs from location *L7* to *L8* and then reattempt to find possible locations for R. As long as HeGeL

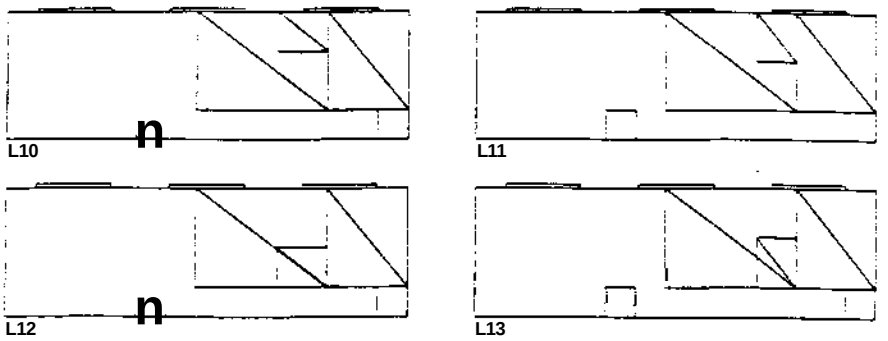


Figure 4-4: Unacceptable locations for unit: R

attempts to relocate SEs and R, it will not find any location that satisfies the requirements. Unless S itself is relocated to an alternate location(L2 *or L5),* HeGeL will waste time in backtracking operation. This implies that the heuristic strategy of selecting a location (in this case location *L1* for unit S) such that there is least left-over area on ail four sides, may not be truly effective in all situations. On the other hand, backtracking mechanism may be made capable of passing precisely such feedback to the heuristic knowledge base concerned with the selection of a promising location. Although it seems intuitively (or rather visually) obvious to us, it is not entirely clear as to how such dependencies among and facts about design units already assigned can be meaningfully stored and incorporated into the backtracking mechanism.

4.3.5. Restructuring

Situations in which HeGeL cannot find a complete solution even after backtracking require *restructuring* the design problem. Additionally, after successfully finding a solution, HeGeL may be directed to search for another, possibly better solution. Both these situations involve modifying a few or all of the previously asserted relationships or introducing additional ones. In the present version of HeGeL, predicates are interactively changed by the user, in order to automate this process, extensive experimentation will be required but a few heuristic strategies are suggested below.

1. Alter the furniture pattern associated with a design unit in order to alter the dimensions of a design unit. Although this is a restricted and localized form of restructuring, when combined with some of the following heuristic startegies, may make resolution of the design problem easier. To illustrate, some subjects had difficulty organizing a conference room with respect to the available area. They resolved this problem by reducing the seating capacity of C, i.e. organizing the furniture of a design unit in response to the available area.

2. Take advantage of the flexibility offered by site elements. To illustrate, in our specific design problem for an office layout, the given site has two doors. Depending on which door is designated as being the primary entrance for the office, the resultant layout gets substantially affected. Since the given site elements like doors, windows, etc. may not be relocated, their potential impact on alternate ways of satisfying the relationships called for in the design problem should be fully explored.

3. Situations in which the relationships mandate a topological configuration of spaces that cannot be geometrically realized, try to combine spaces into a larger unit. This strategy is useful in the sense that when two design units are combined into one area, there usually is some savings in terms of aggregate circulation and thereby requires manipulation of a smaller area than would be otherwise. Although HeGeL works in a rectilinear world, this strategy may be more effective for systems which are capable of handling complex polygons since a complex shape with less area may be more conducive to accommodating

individual furniture items than a rectangle of the same area.

4. When faced with conflicting relationships, relax one that is less important. If CE can either be placed in a *private office* or have a *private entrance* but not both, then it may be better to satify the latter as it involves a site element, e.g. a door, that is given and hence more limiting while the relaxed relationship, e.g. privacy, can still be satified using one of the following restructuring strategies.
5. Adopt alternate interpretations for satisfying a given relationship. To illustrate, when some subjects in our experiments could not locate S and CE so that they are directly accessible to each other, they suggested the use of intercom as a means to substitute for physical proximity. Alternate interpretations may also be achieved by means of assigning multiple purposes to given design elements. Some subjects used bookshelves to double as space dividers, a strategy that is at once space saving as well as architecturally more exciting.
6. Use alternate scenarios for predicate selection. This involves high level restructuring of the problem as it may drastically change the kind of solutions developed. Although the design brief may already specify a scenario, designers usually develop alternate solutions based on entirely different concepts in order to better understand the tradeoffs involved in various solutions.

4.4. Appropriateness of Production Systems

HeGeL is developed as a production system in OPS83. A production system is organized around (a) a global database called *working memorrfWM)* represented as WM *elements{WME)*$_t$ (b) a collection of *if-then* rules, and (c) a *conflict resolution strategy{CRS).* WM elements resemble data structures in procedural programming languages, and similarly the rules specify operations just as procedures and functions. While in traditional procedural languages the sequence of operations is explicitly defined by the sequence in which procedures and functions are encoded in the program, in the case of production systems, the sequence of operations is determined by CRS.

One major benefit of using a production system model over procedural languages has been the ease with which we could add knowledge to HeGeL in an incremental fashion. Initially we identified major components of HeGeL; subsequently each component was fleshed out as a collection of rules. Whenever we needed to add more rules, we had to only specify the contextual conditions in which a rule was to be fired without requiring us to reorder the rule-base. CRS takes care of finding all potential rules for a given context and also selecting the one rule that is to be fired. This enabled us to initially focus our attention on externalizing domain specific knowledge. It proved extremely helpful in developing computational operations for concepts like *privacy* for which a number of interpretations were possible but each interpretation differed slightly from others in terms of either the conditions to be matched or the actions to be carried out. Each new session with HeGeL, after adding a new interpretation (or a rule), demonstrated if HeGeL had adequate knowledge to perform the required actions.

Such incremental development of the system is extremely easy with a production system model. A side effect of such system development is that we developed a precise understanding of interactions among the rules and the flow of execution. Once the rule base became large, sometimes HeGeL did not perform as intended. This situation happens when a number of rules become potential candidates for execution in a given set of conditions but there exist dependencies among such rules that have not yet been detected.

In such a situation, we introduced additional conditions to different rules so as to impose an ordering through contextual specificity. This leads us to believe that the production system model is a good vehicle with which to start for those applications in which a formal body of knowledge is not readily available.

There are some other advantages of working in OPS83. Unlike interpreted programming languages (e.g. Lisp), OPS83 programs are compiled and hence are faster to execute. Also, modular compilation facilitates incremental program development. A substantial advantage in using OPS83 derives from the use of functions and procedures embedded directly in the RHS of rules. This feature helped us to a great extent in collapsing a number of rule-based expressions into a straight forward function or a procedure.

While the production system model has been useful to us for the development of HeGeL, one aspect of it has sometimes turned out to be very bothersome and expensive. The use of WM elements facilitates structuring certain domain information. This brings in restrictions peculiar to production systems. The WM elements are not readily accessible nor can they be manipulated unless explicitly brought in as one of the patterns on the LHS of rules. This results not only in lengthy code but also sometimes necessitates introduction of artificial conditions to some rules in order to make them more specific.

In retrospect, a production system model seems beneficial for applications for which a formal body of knowledge is not readily available. Once a production system is developed and most interactions of a given problem are well understood, it may be more productive and efficient to switch to a procedural programming language.

5. Conclusion

HeGel simulates behaviors of human designers in information processing terms. Human designers start with a set of requirements stated in the given problem, which in this case is that of arranging a number of functional spaces in a given envelope. They use procedural and declarative knowledge which they have acquired through experience either as designers or users of such spaces in solving the problem. Procedural knowledge deals with how to assign the resources of the designer to the task of producing specific design results. The overall process consists of manipulating the design units in such a way that constraints are used to generate alternative solutions and criteria are used to select from among them. Whenever the selection of alternatives is not possible procedural knowledge is used to reformulate the declarative knowledge so that selection becomes possible.

HeGel approaches the problem in a similar fashion. It starts with the same problem as human designers. A set of design units and explicit relationships with respect to proximities and functional requirements are predefined for HeGel. Using a simple generate and test strategy it produces alternative solutions using some of the desired relationships and tests them against others which are not used in their generation. Based on the ultimate number of relationships satisfied and functions allocated, HeGel can be directed towards a modified set of relationships thus simulating the reformation function observed in humans.

In summary, HeGel is a simulation environment in which:

1. Solutions produced are like the solutions produced by subjects.
2. Knowledge used is derived from human protocols and resemble the facts and procedures underlying human behavior.
3. Procedures observable in HeGel's behavior look like those of human subjects.

The purpose behind HeGel stems from a number of factors. One of the primary motivations is to show that the information processing paradigm described as the interchange between problem solving and problem structuring is sufficient to account for human behavior, at least in the context of the task studied. Simulation in the computer is one of the standard techniques which helps in developing paradigms to describe intuitive design accurately and formally.*

A second motivation is to extract generalizable principles underlying the paradigm proposed, especially those related to predicates, solution generation, alternative testing, backtracking, restructuring, and so on. In the simulated environment it is possible to measure the sensitivity of the designs to the various aspects of the paradigm as programmed in HeGel. In the true sense the components of the paradigm can be experimentally manipulated permitting an empirical examination of consequences of such manipulation. Similarly, HeGel provides an experimental medium within which the problem structuring heuristics of the designer can be calibrated, described, and illustrated.

In this iteration, the work has accomplished several ends. One result is that in its general form the paradigm, as implemented through HeGel, is sufficient to show all instances of problem states generated by the human subjects. A second result is that some of the patterns illustrated in HeGel's behavior stem from generalizable rules regarding constraint based generate and test, backtracking, and problem structuring. A third result shown by the work is that the simulation proves to be a useful experimental medium. It allows the systematic testing of alternative search and problem structuring strategies and calibrating their affect on the design solutions produced.

At the same time the work leaves a number of research questions unanswered. These provide motivations for future work still to be undertaken. One obvious extension of the work is the systematic manipulation of search and structuring strategies. The expected results of such an effort would be the empirical derivation of the strategies which are most effective-either in terms of imitating humans or in terms of accomplishing commensurate or even better results.

Another extension of the work would envision the full automation of updating procedures for problem state during the backtracking phase. This would require keeping track of all minor adjustments to the data base - i.e., shrinking and expanding of spaces. In this way, extraneous information left over from earlier iterations would not have the potential of confounding later iterations.

Presently, HeGel has no facility to deal with furniture arrangements within each functional area once they are allocated. In theory, furniture allocations can be dealt with in the identical way that the functional zones are allocated. This envisions a hierarchic nesting of some of the procedures of HeGel, if not all of

them, in order to create a lower level of operations dealing with furniture exclusively.

Finally, it is our intention to automate problem specification strategies so that HeGel can restructure the design problem as it works on it. This requires the codification of relevant heuristic rules through systematic experimentations with HeGel's problem structuring function, which is at this time fully manual. By correlating these rules with the effectiveness of the resulting designs it would be possible to formally describe problem structuring strategies that can improve over the performance of human designers.

References

[Akin78] Akin,O.
How do architects design?
In Latombe,J-C. (editor), *Artificial Intelligence and Pattern Recognition in Computer-Aided Design.* North-Holland, New York, 1978.

[Akin86a] Akin, Omer.
A Formalism for Problem Restructuring and Resolution in Design.
Planning and Design 13:223-232, 1986.

[Akin86b] Akin.O, Chen.C. Dave,B. and Pithavadian.S.
A Schematic Representation of the Designers' Logic.
In *Proceedings of the International Joint Conference on CAD and Robotics in Architecture and Construction.* Marseilles, 1986.

[Akin86c] Akin.O.
Psychology of Architectural Design.
Pion Limited, London, 1986.

[Akin87] Akin.O., Dave.B. and Pithavadian,P.
A Paradigm for Problem Structuring in Design.
IFIP Conference MIT.
October, 1987

[Baykan84] Baykan.C.
Heuristic Methods for Structuring Architectural Design Problems.
Unpublished manuscript.
1984

[Baylor71] Baylor,G.W.,Jr.
A treatise on the Mind's-eye: An Empirical Investigation of Visual Mental Imagery.
PhD thesis, Dept. of Psyhology, Carnegie-Mellon University, Pittsburgh, 1971.

[Eastman70] Eastman.C.
On the analysis of intuitive design processes.
In Moore.G.T. (editor), *Emerging Methods in Environmental Design and Planning.* MIT Press, Cambridge, 1972.

[Flemming86] Flemming.U.
On the representation and generation of loosley packed arrangements of rectangles.
Environment and Planning B: Planning and Design 13:189-205, 1986.

[Forgy85] Forgy,C.L
The OPS83 User's Manual
Production System Technologies, Inc., Pittsburgh, 1885.

[Foz73] Foz,A.
Observations on Designer Behavior in the Parti.
DMS-DRS Journal: Design Research & Methods 7(4):320-323, 1973.

[Freeman71] Freeman,P.A., Newell.A.
A model for functional reasoning in design.
In *Proceedings of the Second International Joint Computer Conference on Artificial Intelligence,* pages 621-640. British Computer Society, London, 1971.

[Friedland85] Friedland.P.
Introduction: Special Section on Architectures for Knowledge-Based Systems.
Communications of the ACM 28(9):902-903, September, 1985.

[Moran70] Moran.T.P.
A model of a multilingual designer.
In Moore.G.T. (editor), *Emerging Methods in Environmental Design and Planning.* MIT Press, Cambridge, 1972.

[Reitman64] Reitman.W.R.
Heuristic decision procedures, open constraints and structure of ill-defined problems.
In Shelly.M.W. and Bryan,G.L (editors), *Human Judgements and Optimaiity,* John Wiley, New York, 1964.

[Simon73J Simon,H.
Structure of Ill Structured Problems.
Artificial Intelligence 4(3-4):181-201, 1973.

[Steadman83] Steadman.J.P.
Architectural Morphology.
Pion Limited, London, 1983.

[Wright83] Wright,J. and Fox.M.
SRU1.5 User manual
1.5 edition, The Robotics Institute, Carnegie-Mellon University, Pittsburgh, 1983.