USING CONTEXT IN SKETCH RECOGNITION

by

Christopher Frederick Herot

Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

at the

Massachusetts Institute of Technology

May, 1974

Signature of Author_____
Department of Electrical Engineering, May 10, 1974

Certified by_____
Thesis Supervisor

Accepted by_____
Chairman, Departmental Committee on Graduate Students

USING CONTEXT IN SKETCH RECOGNITION

by

Christopher Frederick Herot

Submitted to the Department of Electrical Engineering on May 10, 1974 in partial fulfillment of the requirements for the Degree of Master of Science.

ABSTRACT

A central problem in graphical communication between a human and a computer is the ability of the computer to build an internal representation of the human designer's intentions, something which is not possible if the machine does not have some knowledge of the subject matter being sketched. This thesis describes a system for storing such knowledge in the computer, to be used by the machine in building a model of the user's intentions. This semantic knowledge is stored in a network and matched to the input sketch in a top-down manner, using the knowledge to direct the machine's search for entities in the sketch.

The operation of a simple program using these principles is presented in detail along with an account of the complexities involved in applying the scheme to complex sketches. Methods of representing relationships to be looked for in a drawing are described and examples are provided of some descriptions which should prove useful in the next implementation.

Two implementatons are described, one on a mini-computer and one on a large time sharing system. Included is a discussion of the requirements such a program makes on the programming environment and their implications for the future of inexpensive mini-computer oriented implementations.

THESIS SUPERVISOR: Nicholas P. Negroponte
TITLE: Associate Professor of Architecture

ACKNOWLEDGEMENT

I wish to thank the members of the Architecture Machine Group, especially Nicholas Negroponte, who had the idea that machines could be made to understand the intentions of humans, and managed to convince others of it, including James Taggart, who laid the foundations upon which this work is based. I am grateful to Mike Miller, Jeff Entwisle and Linda Christian for their comments and criticisms and to Andy Lippman without whose efforts this thesis might never have made it to paper. I would also like to thank Berthold Horn for the use of the Artificial Intelligence Laboratory's computer when it was most needed.

This work was supported in part by Project MAC, an MIT interdisciplinary laboratory sponsored by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract number N00014-70-A-0362-0001.

TABLE OF CONTENTS

1. Introduction

1.1 Computer Graphics as a Communications Medium

There are many design problems which could benefit from the application of the computer if it were not for the difficulties in communication between the human and the machine. In all applications the user must state his problem in a form understandable to the machine, a task which too often requires either an expert knowledge of computers or some sort of tedious protocol at a computer terminal. There have been many attempts to use graphics as a means of communication, originating with Sutherland's SKETCHPAD[1], a system which allowed graphical input of descriptions by use of a light pen and function switches. Unfortunately each point and line had to be specified explicitly, so that the input of a complicated description was too tedious to make the computer useful in any application not containing large numbers of repetative elements. Most subsequent graphics systems have been little improvement over Sutherland's program of over ten years ago, with the result that the use of the computer in design usually occurs after the design has been formalized to the state where it is cost-effective to "digitize" a drawing by tracing with a data tablet or even by typing in coordinates from a keyboard.

If the computer could understand, with a minimal

amount of human intervention, the informal sketches common to the early stages of the design, it could be used earlier in the design process, providing feedback to the user about his design, possibly helping him to make better decisions. If computer displays became inexpensive and easier to use, they could replace pencil and paper, allowing the designer to work with three dimensional representations from the start. This would not only aid in visualizing the object being designed but, since these descriptions would be available to the machine, the computer could serve as a partner in the design process. Unfortunately, present day computer recognition of drawings functions at such a low level that interactive, graphical examples of computer aided design are limited to areas which lend themselves to highly stylized input, such as layout of integrated circuits, printed circuit boards and machine parts.

One of the goals of the Architecture Machine Group when it was formed at MIT in 1967 was to develop an inexpensive, unobtrusive means of graphical communication. The HUNCH program, developed by James Taggart[2] was the begining of such a system. Using parameters such as the velocity with which the line was drawn and pressure exerted on the pen, the program translated the data points of a sketch drawn on a Sylvania Data Tablet into a list of lines and endpoints, in effect "straightening" the drawing. To those people accustomed to the rubber-band-line form of

input common to other systems, HUNCH yielded impressive results, but there were severe shortcomings in some of the interpretations it made. For example, the original version attempted to "latch" the endpoint of a line to any nearby line, but the results were often suprising, æs can be seen in Figure 1-1.

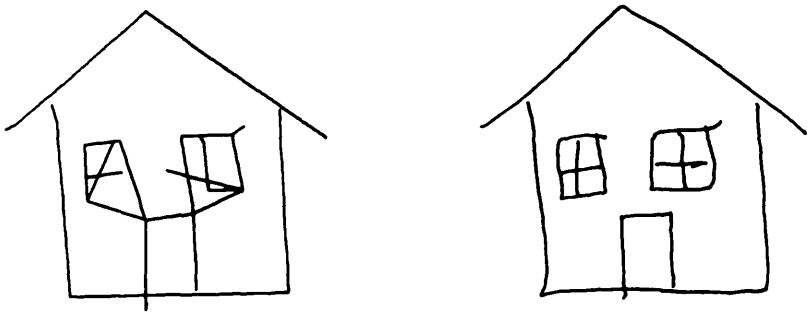

Figure 1-1 "Aunt Fifi's House"

Other problems of interpretation included overtraced lines, which could be taken to mean either emphasis or erasure, and lettering. As Negroponte[3] has argued "sketch recognition is an issue of machine intelligence." The result of this posture was that most researchers abandoned the notion of free-form graphical input as being too fanciful and produced systems with no graphical input or which were no improvement upon SKETCHPAD. At the

Architecture Machine the emphasis of research was shifted into developing better low level descriptors to improve the output of the line finding program and investigating additional representations of drawings to augment the point/line one.

One example of a new representation is illustrated in a floor plan recognition program[4]. The program used a bit map to represent the drawing positionally instead of by the HUNCH sequential line/endpoint representation. The bit map was stored on a fixed head disk and accessed by mapping selected bits into an array in core. By varying the number of bits mapped into one element of the array, a program could control the level of detail in the sample under consideration. The program could find regions which were assumed to represent rooms, with connections between the regions interpreted as doorways. While this method was reasonably successful, the one scale chosen for the entire sketch occasionally caused some rooms to be missed, and the representation required for doors (a gap in a wall) was too restrictive. It became evident that the program would require some knowledge of what to expect in a sketch if it was to be able to find objects by anythingbut blind application of all descriptions to all parts of the sketch.

While it was clear that syntactic descriptions, such as what constituted an endpoint of a line or what was the difference between a straight line and a curve could be

embedded in the programs, semantic descriptions, those of the domain being sketched, needed to be stored outside the program if the system was to handle more than one specific domain and be easily modified. With the advent of new display technologies which promised the capability of more complex output for a lower price, and developments by Artificial Intelligence researchers in methods of structuring knowledge, the time had come to make use of these developments to apply semantic knowledge of the sketching domain in recognizing sketches. This thesis describes a system which attempts to use some of these developments to advance the field of sketch recognition.

The proposed system would operate interactively, using the semantic knowledge to interpret the sketch as it is drawn, building a model of the user's intentions. While the purpose of the model is to allow the user, or a user-written program, to ask questions about the design, the machine will be able to ask questions of the user when it doesn't understand something he has drawn. Although the program described here comes nowhere near meeting these goals, it is hoped that this thesis can provide the first step towards building a program which can. Architecture will be used to illustrate the scheme throughout this thesis, although the scheme is desgned to applyto a wide range of sketching domains. Architecture can serve as a useful paradigm because of the large amount of research

that has been done into the nature of Architectural design, some of it by people at MIT, who can help in the design and testing of the program.

1.2 Overview

Chapter 2 describes the requirements for and the fundamental concepts behind the system. A method is described for storing descriptions of elements expected in a sketch (refered to here as the general description), along with a low level description of the sketch itself (the data description) in a network similar to that used by Winston[5]. The general description is used in a top-down fashion to search for entities in the sketch, in a way inspired by the program of David Waltz[6] which found lines in a collection of blocks by using knowlege of the legal configuration of vertices in the blocks world.

Chapter 3 presents the workings of the system in greater detail, if somewhat naively, describing the pattern matcher and control structure designed to find elements in the sketch and match them to nodes in the network. While Minsky[7] and Fahlman[8] have proposed new methods of structuring knowledge and control to solve such problems, and some of their ideas are used here, the emphasis in this thesis is on the application of tried and tested techniques which are available in currently implemented programming languages such as CONNIVER or, with

minimal modifications, LISP.

Chapter 4 goes on to discuss the implications of applying the system to complex sketches, describing the kind of descriptors which must be provided and the complications which arise when an initial hypothesis made by the program does not work out. This chapter suggests the direction in which future research should procede, and what kinds of data and control structures would be needed in an improved program.

Chapter 5 consists of a detailed example of a description, illustrating the concepts presented in the earlier chapters.

The two implementatons, one done in FORTRAN on a mini-computer and one done in CONNIVER on a large time sharing system, are described in Chapter 6, along with a discussion of the facilities such a program requires from the hardware and the operating system.

Chapters 7 and 8 contain suggestions for future work and thoughts on the implications of the research described here, including the uses for the model built by the program.

While this thesis is not presented as an example of Artificial Intelligence research, it does take advantage of many of the techniques and languages developed for AI. One of the problems plaguing AI researchers is finding a suitable "problem space" to try out ideas, one that is

sufficiently structured to be amenable to current techniques but is still rich enough in complexity and detail to yield rewarding and useful results. It seems that sketches, when confined to a suitable domain, comprise such a problem space. The input data is more definite than that found in vision problems, since each line is explicitly drawn by the user and recorded by the machine.

In addition the program can make use of the sequence in which the lines were drawn, both for finding straight lines and for finding elements in the sketch. Although much research is going on in the area of structuring knowledge and building control structures capable of handling large recognition problems, the emphasis here is on the use of existing techniques. What is unique is the way in which these various techniques are assembled and put to use on a problem much more structured and definite than most AI problems.

A major advantage of the interactive approach offered here results from the fact that we are not so much concerned with simulating human intelligence as we are in providing an intelligent machine to work with a human, allowing us to take advantage of the unique capabilities of both. If the machine arrives at a point where there is no clear cut solution, it can ask the user, which is no worse than current systems which require interaction for every decision. With suitable language capabilities,

the interaction might even approach a dialogue useful to both human and machine, with the machine warning the user of unusual conditions. By using even currently available graphic techniques, a rich set of graphical interactions is possible. Even a barely functioning recognition system would have some value as an improved "line straightener" in an interactive sketching system. As the capabilities of the system improve, the interactions can be either be made more meaningful or can be reduced in number.

2. The Descriptive System

2.1 Overview

The knowledge of the sketching domain must be structured in such a way that it can be used to direct the machine's analysis of the sketch, supplying goals to direct the low level routines in their search for lines, curves, and corners and supplying enough information about what is plausible in the sketch to resolve ambiguities and fill in missing information. This knowledge must also be in a form that can be easily understood and modified. The traditional method of hierarchically organizing the analysis from the bottom up will not suffice because much of the low level information only takes on meaning in the context of the entire sketch, the intentions of the designer, and the domain in which he is working. These three areas taken together will be referred to as the context of the sketch. Since the specification of the context is necessary in order to interpret the sketch, but is also determined by the sketch, we have selected the alternative of specifying the context in advance. This does not rule out a seperate set of programs to recognize context. It is just that context definition is a separate problem from the harder problem of context recognition.

The act of recognizing a sketch involves drawing the sketch for the machine and specifying the "context," thus

avoiding the problem of context recognition. This context specification might be as specific as "suburban homes" or "machine screws" or it might be as general as "structures" or "tools."

2.2 Context Definition

One commonly used representation of a drawing is the hierarchy, with a node at the top for the entire drawing which branches into subnodes describing pieces of it which in turn branch further until, at the bottom level, one finds the raw data. For example, a drawing of a house could be represented by the diagram in figure 2-1, where the lines descending from an object lead to elements that are parts of that object.

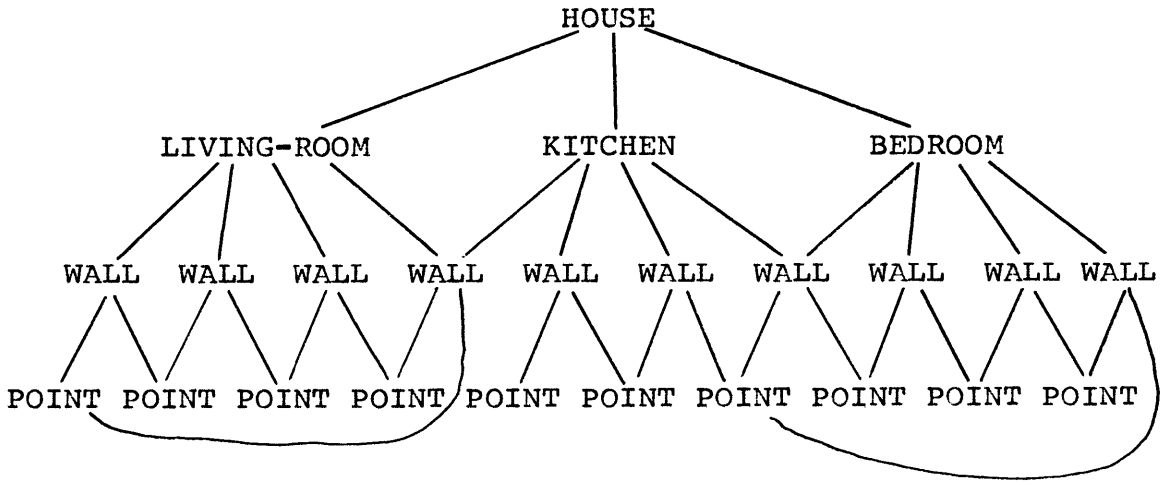

Figure 2-1 - House

We would like to view this diagram as the composition of two separate structures: a context-free structure of points, lines, and areas, which can be generated by the low level routines, and a structure which defines the general case of a house. The data structure includes actual existing physical entities, with no inherent meaning, while the general structure's connections specify what kinds of data are required and what things are permissible in a house. The intent of the system described here is to synthesize the composite structure from the two individual ones.

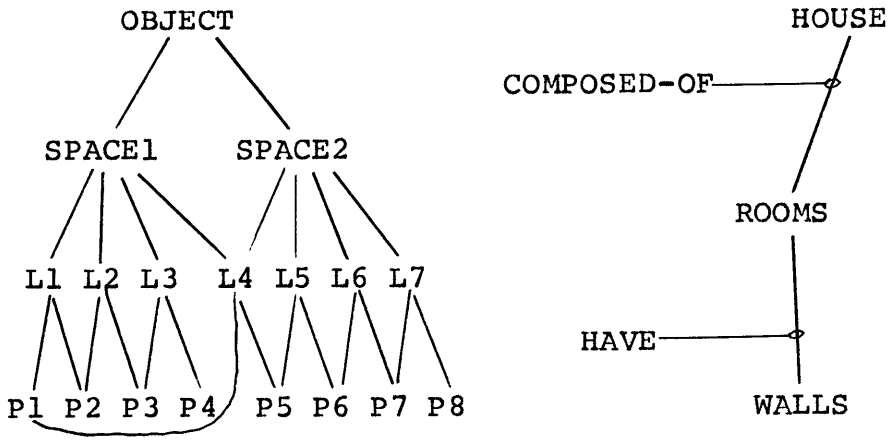

Figure 2-2 - Fragments of the Two Constituent Structures

The general case structure is composed of higher-level descriptions with classes of possible data represented instead of the actual data values. By putting this skeletal

structure into the machine, whether through teaching by example or typing instructions from the console, we can define a context for sketches to be understood by the machine. The context-free data structure, on the other hand, resembles the output of the Architecture Machine's current plan recognition and line straightening programs. The process of machine recognition of a sketch involves combining the two structures, substituting instances of objects found in the actual sketch for the corresponding descriptions of possicle objects in the general case structure, a process we will refer to as instantiation.

By comparing a description of the possible structures of a house, previously stored in the machine, to the sketch being analyzed, the machine has a way of directing its search for entities in the sketch, by attempting to instantiate entities described in the generalized structure and by filling in missing or incomplete information from defaults in the general case structure. As information is gathered, it is checked for consistency with the rest of the description. A conflict triggers procedures which further examine the data, revising the new information and possibly the old information until everything fits together. As the models of the user and his sketch become more complete, the decisions become more definite, until the meaning of the entire sketch is determined. At times it may be necessary to ask questions of the user to resolve

a conflict or to include some information which cannot be deduced from the existing data. These questions could be about some specific element of the sketch ("What is that object?") or about the user's intentions ("Are you sure you don't want a living room?"). If the user's concept of what he is designing differs drastically from the machine's, the questions will undoubtedly come quickly at first until the machine has enough information to reorganize its model of the user and of the sketch. While the questions will be primarily of a primitive nature, hopefully the comments from the machine can develop into a useful tool to warn the user of conflicts in his design and things he may have overlooked. In the first implementation the questions and answers will bear a one-to-one correspondence with modifications to the data base, but eventually they will be incorporated into a scheme which allows the machine to learn in a manner similar to that demonstrated by Winston.

Note that even with the simple example above, the sharing of walls between rooms and points between walls requires a network instead of a simple tree. A network also allows storing, with an economy of memory, such useful information as circulation patterns between rooms, acoustical access, visual access and other such alternative ways of describing a house besides a simple hierarchy. The descriptive system described in this thesis consists

of a network of objects and relations together with a program to match the general case part of the network to the data part.

2.3 Use of the Network

Once the machine has a description of the sketching domain, it can use this description to analyze the sketch. Although the description is stored in a network, it can also be viewed as a tree, with the top node being the most general (HOUSE in figure 2-3), and branching out until the nodes at the bottom are the very specific objects found in the raw data, such as lines, points, and regions. The non-hierarchical links of the tree can be thought of as criteria which must be satisfied among the leaves of the tree. The process of recognizing a sketch involves instantiating the variables in the general structure (the nodes labeled N1, N2, N3, etc. in figure 2-3) with actual objects found in the raw data. The end result will be a structure similar to the one in figure 2-3 but with all of the variables instantiated. This new network forms the "recognized" sketch and, as a structured representation of the sketch and the user's intentions expressed through the sketch, can be used by higher level programs which have to know about the design. It could even be used to output a better "straightened" version of the original.

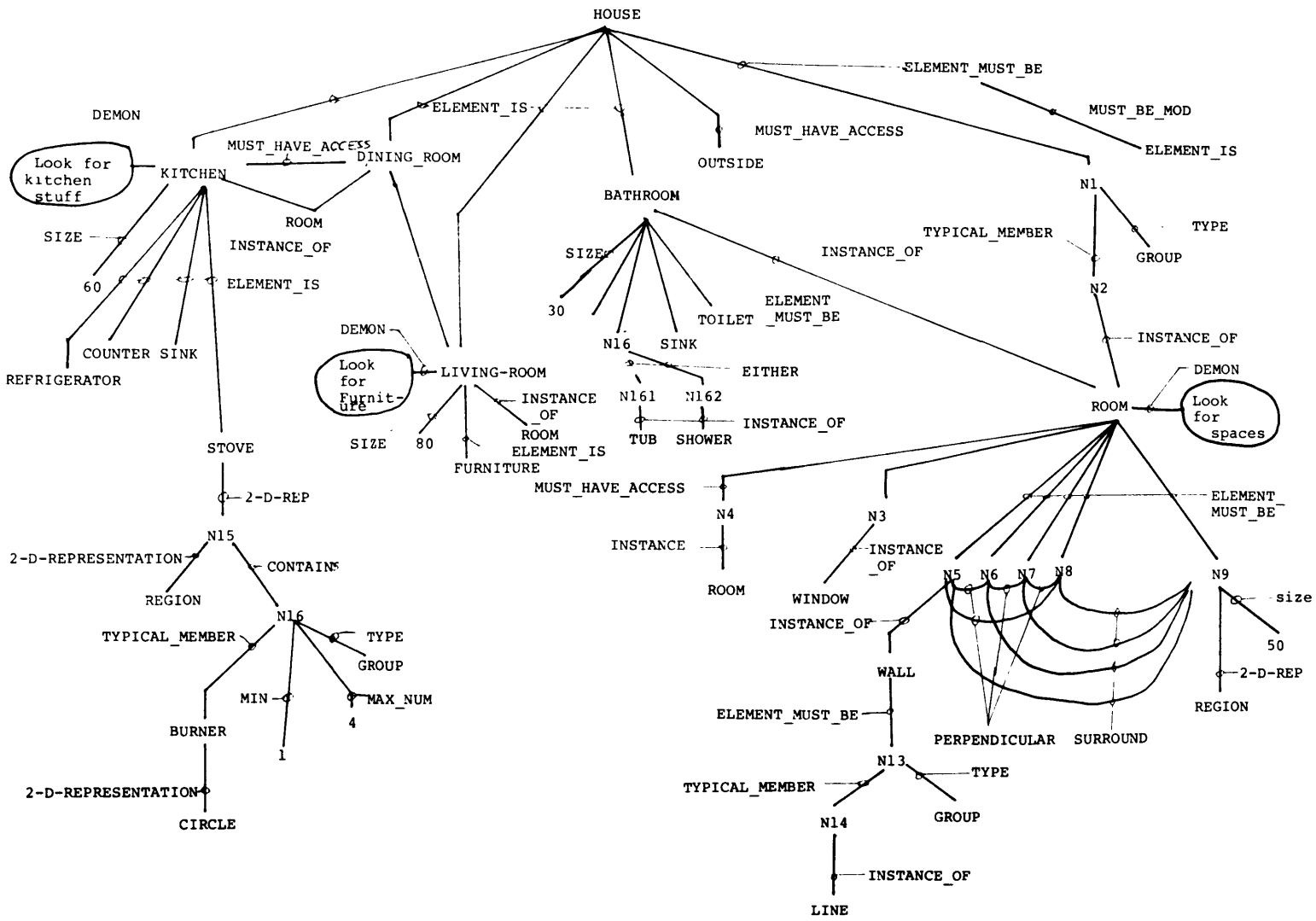
HOUSE
DEMON
Look for kitchen stuff
KITCHEN
MUST_HAVE_ACCESS
DINING_ROOM
ELEMENT_IS
OUTSIDE
MUST_HAVE_ACCESS
ELEMENT_MUST_BE
MUST_BE_MOD
ELEMENT_IS
SIZE
60
INSTANCE_OF
ELEMENT_IS
ROOM
REFRIGERATOR
COUNTER
SINK
STOVE
2-D-REP
N15
2-D-REPRESENTATION
CONTAINS
REGION
N16
TYPICAL_MEMBER
TYPE
GROUP
MIN
1
MAX_NUM
4
BURNER
2-D-REPRESENTATION
CIRCLE
DEMON
Look for Furnit-ure
LIVING-ROOM
SIZE
80
INSTANCE_OF
ROOM
ELEMENT_IS
FURNITURE
BATHROOM
SIZE
30
N16
SINK
TOILET
ELEMENT_MUST_BE
EITHER
N161
N162
TUB
SHOWER
INSTANCE_OF
N1
TYPICAL_MEMBER
TYPE
GROUP
INSTANCE_OF
N2
INSTANCE_OF
ROOM
DEMON
Look for spaces
MUST_HAVE_ACCESS
N4
INSTANCE
ROOM
N3
INSTANCE_OF
WINDOW
ELEMENT_MUST_BE
N5
N6
N7
N8
N9
size
50
2-D-REP
REGION
INSTANCE_OF
WALL
ELEMENT_MUST_BE
PERPENDICULAR
SURROUND
N13
TYPICAL_MEMBER
TYPE
GROUP
N14
INSTANCE_OF
LINE

Figure 2-3

2.4 Types of Nodes and Relations

The nodes in the network can be divided conceptually into variables and descriptions. A description is a node which has many nodes descending from it, called daughters, which form a sub-network describing some object that may be present in many parts of the entire network. ROOM and WINDOW are examples of description nodes. The daughters of description nodes are usually related to them by ELEMENT-IS relations. These daughters will usually be variables, such as N3 and N37 in figure 2-4. Variables stand for individual instances of a description and will be matched to named objects when the network is instantiated. The relation INSTANCE-OF indicates what description must be satisfied to instantiate the variable. There may be any number of variables having an INSTANCE-OF relation to the same description.

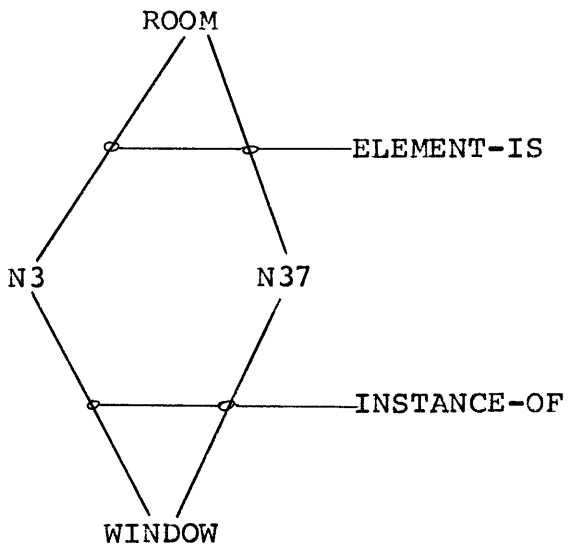

Figure 2-4 INSTANCE-OF

Other relations in the network are criteria which have to be satisfied when looking for an instance of an object. They function like ELEMENT-IS, indicating an element restricted to those satisfying the specified relation.

2.5 Groups

A special type of relation indicates a GROUP. This relation is a special case. It indicates that the node having this relation is an abstraction for a variable number of objects, which are either represented by ELEMENT-IS relations or relations for the minimum and maximum number of elements. Since elements of a group are all of the same type, the TYPICAL-MEMBER relation points

to a description which embodies the concepts common to all the members of the group. This description may contain references to the special node ANOTHER-MEMBER, which indicates another member of the same group. The group description can also specify names for the various members by referencing them through ELEMENT-IS nodes as is done for N3 in figure 2-5.

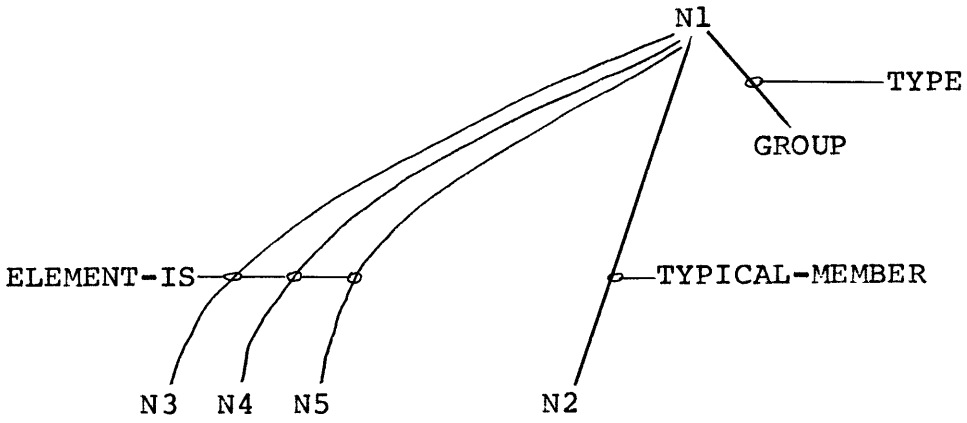

Figure 2-5 Group

The EITHER relation, like the group, is a way of combining several related nodes in one abstraction. It functions like an exclusive-or relation, indicating that just one of the specified relations is true. For example, in Figure 2-6, N16 can be either a tub or a shower.

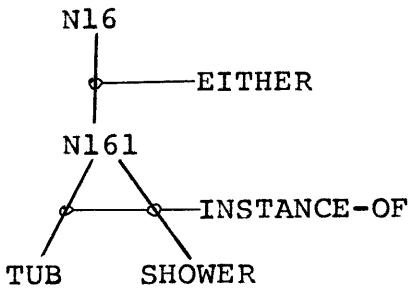

Figure 2-6 EITHER

2.6 Defining Meanings of Relations

The relations between nodes can themselves be treated as nodes, and thus can be related to other relations. For example the relation ELEMENT-MUST-BE in figure 2-3 can have the relationship MUST-BE-MOD to the relation ELEMENT-IS, indicating that it is the imperative form of ELEMENT-IS. The relations themselves have no meaning as far as the data base programs themselves go, but take on meaning through the programs which use the data base. In addition, it is possible to associate a procedure with a relation such that if the specified relationship is not found in the data base, the procedure is called. This facility can be used to make some relationship appear to be in the data base even though it is not explicitly stored there. For example the relationship PARALLEL can have an associated procedure which looks to see if the slopes of the two lines in question are within some increment (modulo 180 degrees)

of each other, eliminating the need to store an explicit PARALLEL relation for each pair of parallel lines. Another kind of "if-needed" procedure can return all the items which match a specified pattern-for example,all of the lines parallel to a given one. This same technique will be used by the matcher in finding objects which do not exist as raw data but are found through application of their descriptions.

2.7 Relations on Relations

The most common types of relations on relations are the MUST-BE-MOD and MUST-NOT-BE-MOD which serve to indicate imperative forms of their objects. The matching program uses the non-imperative form in testing for the relation, using the form of the modification to evaluate the result. In general, not finding a relationship which is a MUST-BE modification or finding one which is a MUST-NOT-BE modification will cause the rejection of the object being instantiated. In addition, it is possible to form other conditionals, such as "only true if X is true" bycreating a modification of a relation and an associated if-needed method which tests for the condition.

3. Operation of the System

3.1 Input

Having described the network in terms of the meanings of the relations, we can proceed with an explanation of how it is used by the matching program. The program begins with a sketch and the specification of the context. The sketch is in the data base in the form of objects such as lines, points, and regions which were found by the low level routines. These objects will form the bottom fringe of the instantiated structure. The lines are not really stored as instances of lines but rather as a list. An if-needed procedure will extract the lines from the line-list, breaking them or joining them if necessary. For example, lines L3 and L8 in figure 3-1 will sometimes be

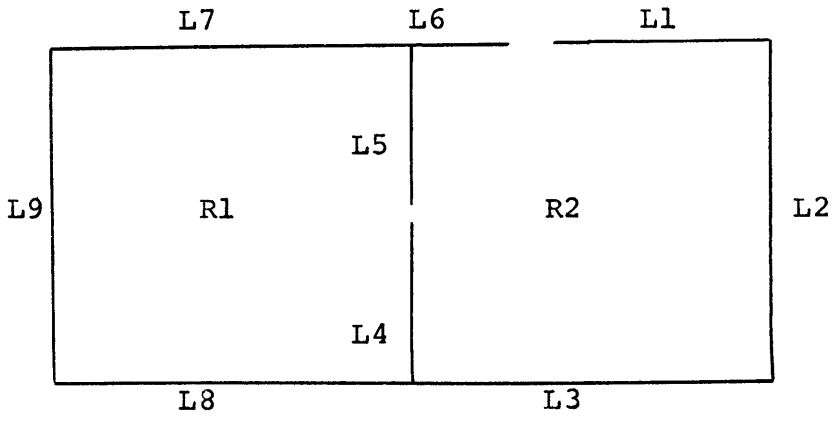

Figure 3-1 Sample Sketch

considered as one line and sometimes as two, but the low-level straightening routines know only about how they were drawn, not what they mean, so the result could be either one line or two, depending on how the sketch was drawn. This information is still kept around, however, since it may prove useful later. Also, parameters such as slope, rate, and pressure are stored as data for the line and not strictly as relations, but discussion of such details will be postponed until section 4.3. The data structure for the sketch in Figure 3-1 is shown in figure 3-2.

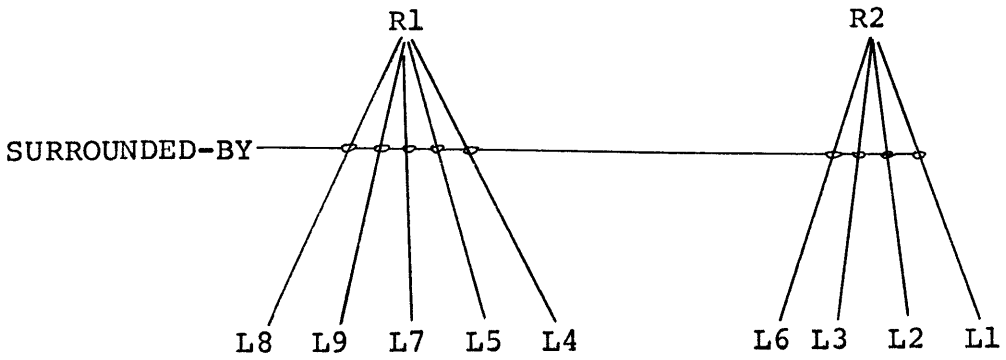

Figure 3-2 Bottom Fringe

In addition, relations such as parallel or perpendicular can be extracted by if-needed methods if necessary.

The context is specified in terms of a network as described in section 2. In the future, it may be possible to recognize which context we are dealing with automatically, but for now we have chosen to avoid the problem of context recognition by requiring the user to specify one, e.g. "houses." For the purposes of illustration, we will use a simplified version of the network in figure 2-3, shown in figure 3-3.

3.2 Finding an Instance

The operation of the matcher is top-down, that is, given the context "houses" it will search the data-base for an instance of a house. At this point there are, of course, no houses in the data base, just a description of what a house looks like and the raw data that might be a house. There is, however, an if-needed method for INSTANCE-OF which is the heart of the network matcher. Asking the data-base for a house will cause this program, which we will call INST, to be invoked, and it will attempt to instantiate a house by matching the general description in the data base to the raw data. The flowchart in figure 3-4 illustrates the program's mode of operation. In trying to instantiate an object, INST will call itself recursively to instantiate that object's components until eventually it gets low enough in the tree that it is looking for some object which is really in the data base, such as a region,

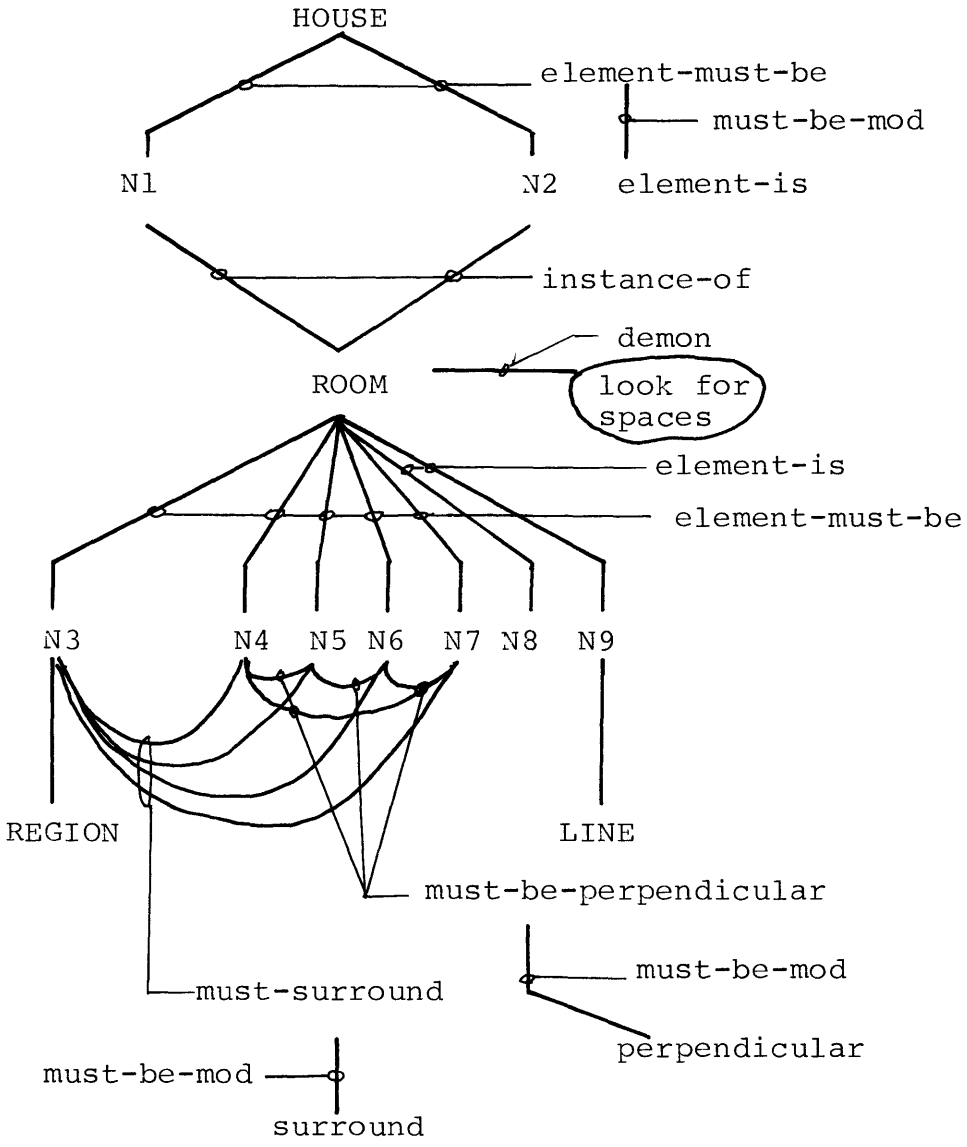

Figure 3-3 Simplified General Case Structure

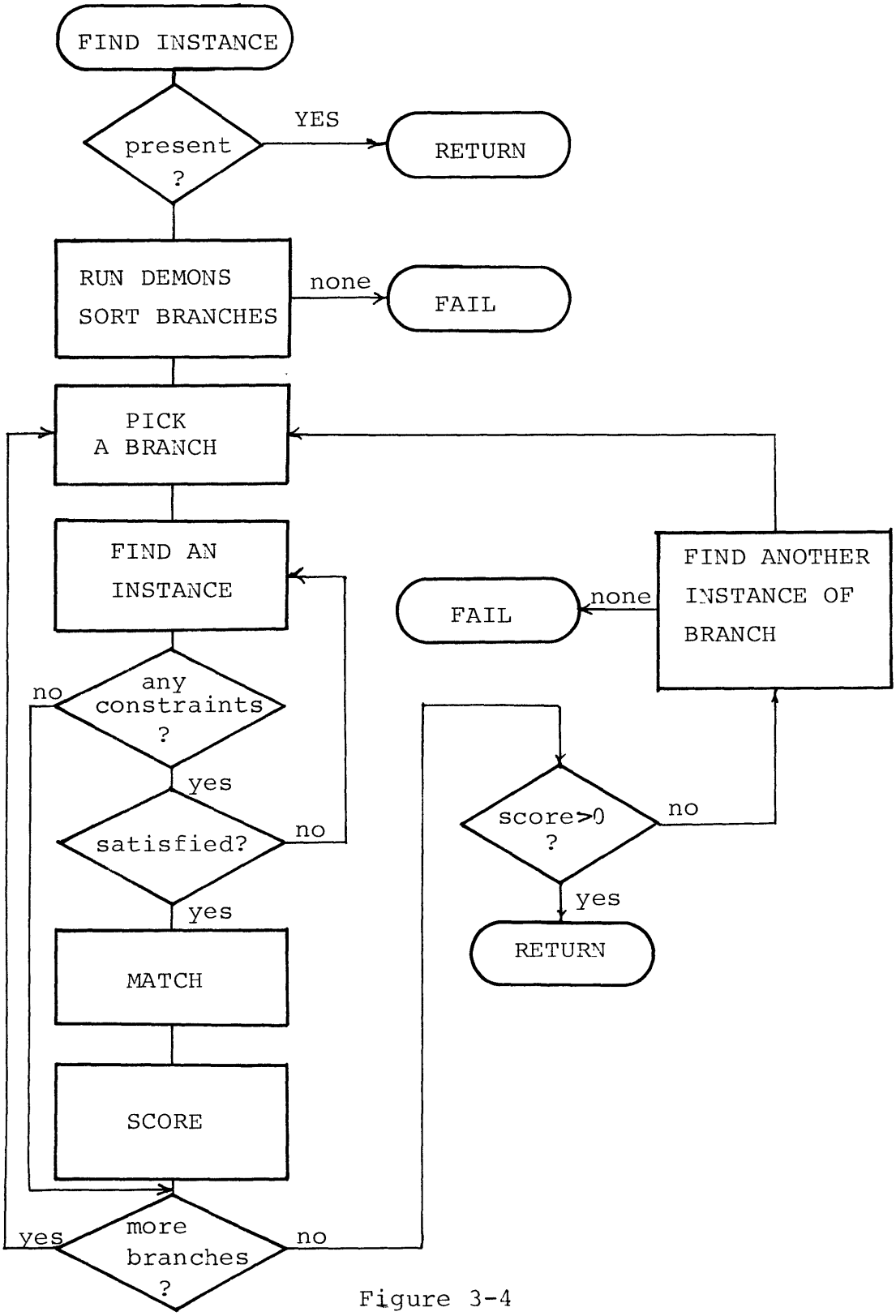
FIND INSTANCE
present ?
YES
RETURN
RUN DEMONS
SORT BRANCHES
none
FAIL
PICK
A BRANCH
FIND AN
INSTANCE
any
constraints
?
no
yes
satisfied?
no
yes
MATCH
SCORE
more
branches
?
yes
no
FAIL
none
FIND ANOTHER
INSTANCE OF
BRANCH
score>0
?
no
yes
RETURN

Figure 3-4

line, or a left-over instance from a previous invocation (see section 3.5). Whenever any object is found, it is stored in the data part of the data base as a match for the searched-for node.

Calls to INST can be divided into two classes. Either the object being sought is a variable which is an instance of something else, for example N1 in figure 3-3, or it is a description which has elements, such as a house or a room, so that the network may be thought of as being composed of fragments like the one shown in figure 3-5.

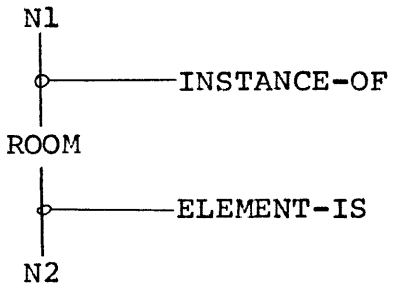

Figure 3-5 Fragment of a General Network

The element N2, representing a part of a room, can match many possible objects, one for each room found in the sketch, but there will be just one for each instance of N1. That is, N1 stands for one instance of one room, while N2 stands for one element in each separate room of the sketch. When we are done, the data base will have

combined the fragment of figure 3-5 with the data network to contain the instantiated fragment of figure 3-6.

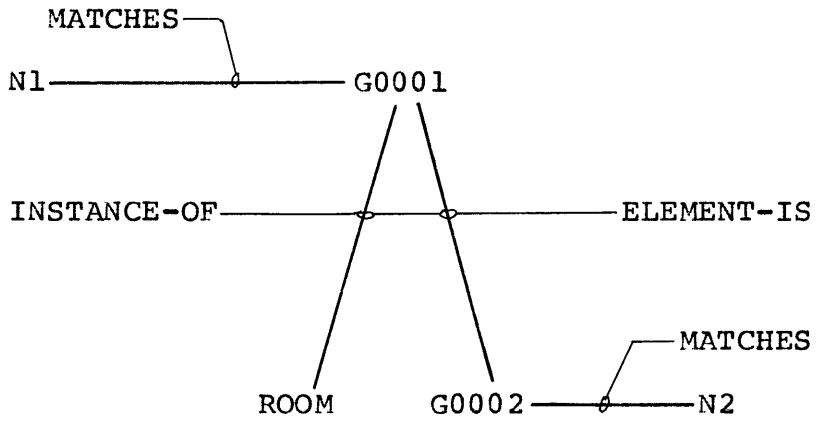

Figure 3-6 Fragment of an Instantiated Network

Although the node G0002 represents an element of a room, it is stored as an element of a specific instance of the room G0001 to distinguish it from other instantiations of N2 in other rooms. If G0002 was instead made a daughter of ROOM, the resulting fragment would look like figure 3-7, where the associations are ambiguous.

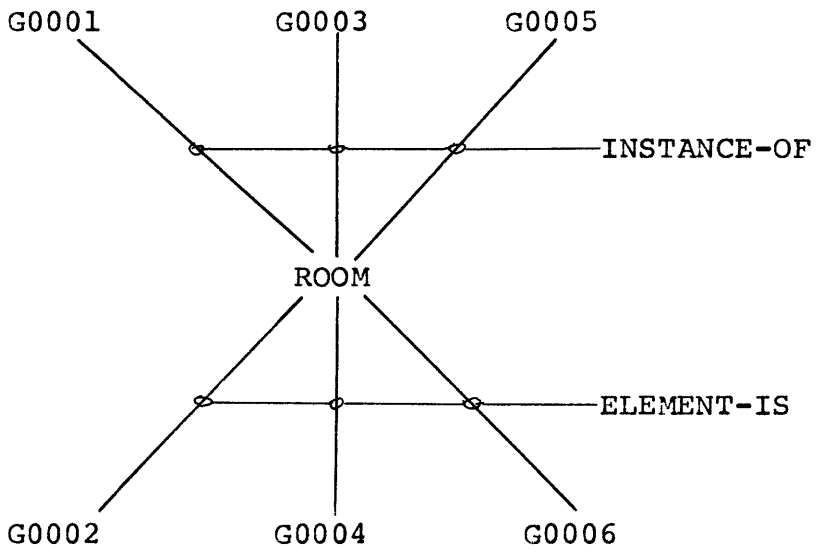

Figure 3-7 Fragment of an Instantiated Network

Since this network is being created by the program, all of the nodes, with the exception of those which are instances of real raw-data objects, such as regions and lines, have to be created by the program and given unique names. This is done every time INST forms an object from its components. It generates the name and adds the match to the network, and adds the ELEMENT-IS relations from the generated name to the nodes returned by the instantiations of the elements, which themselves are usually generated names.

3.3 Demons

Before attempting to instantiate any elements, INST looks to see if there is a demon associated with the node

being instantiated. A demon, as used here, is a procedure which can be attached to any node to be invoked whenever the program is looking for an instance of that node. It provides a procedural means to specify what tests should be performed, what objects to look for, and anything else which should be done before going on, thus offering an "escape hatch" from the declarative format of the network and a way of organizing the search. An example demon for the node ROOM might be a procedure which invokes the region finding program to see if there are any regions in the sketch, since it would obviously be unprofitable to continue if there were none. The demon might use the information it gathers to sort the list of elements to be instantiated, or it might rebind some variables to restrict the match performed by programs it calls. If there is no demon, the default demon merely sorts the relations emanating from the node, with INSTANCE-OF first, then modifications and other relations.

3.4 Example

In matching the structure of figure 3-3 to the house of figures 3-1 and 3-2 there are just two relations emanating from HOUSE. Both are ELEMENT-MUST-BE and specify objects N1 and N2, so the program sets out to find instances of N1 and N2 by searching the data base again. This results in another (recursive) call to INST, this time

with the argument N1. Now N1 has two relations on it, the first one being (INSTANCE-OF ROOM), which takes priority, so another recursive call is made to find an instance of a room. The demon for room directs INST to look for regions first, so another call is made for an instance-of the region in the description, N3, resulting in a search for a region, which this time is an object in the data base.

INST makes a preliminary match between N3 and region R1 and tries to satisfy the criteria on N3, which in this case require N3 to be surrounded by at least four lines, which are found to be lines L8, L9, L7, L5 and L4. In searching for these lines, they are required to have certain relationships of perpendicularity, as specified by the network. Now INST can return R1 as its result for an instance of N3, whereupon it looks for the other elements of a room. Since N4, N5, N6, and N7 have already been matched, all of the elements have been found and INST generates a name, G0002, returning it as an instance of a room, to match N1. The same process is repeated for N2, returning G0003, so that INST can generate a name for the house, G0001, producing the completed network shown in figure 3-8.

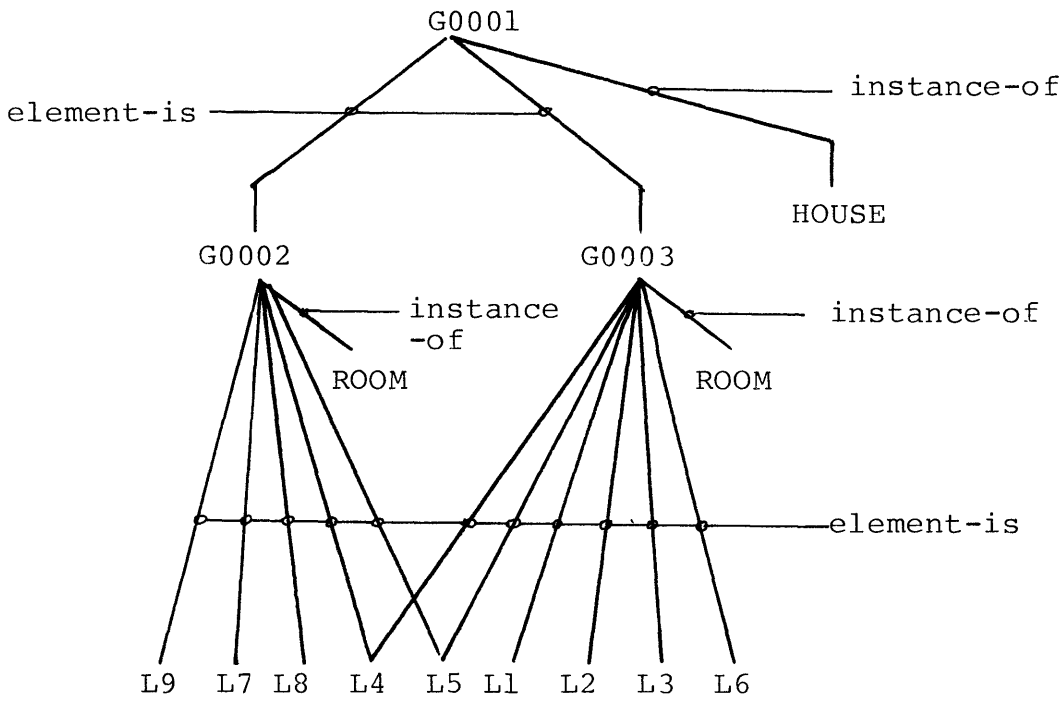

Figure 3-8 Instantiated Data Network

3.5 Scoring and Failure Mechanisms

In the example above, each match made by the program was the correct one, which is what one would hope for as a result of the demon organizing the search on the basis of the data, but things are usually not so simple. Often some relations are satisfied and not others, or some conditions hold that are prohibited. The solution used in the present scheme is to provide a numerical score for each instantiation. For each successful match the score is incremented, and for each unsuccessful match it is decremented. When all matches are done that can be done, the score is compared to some threshold value, and if greater, the matches and data network which have been built up for this instantiation are "finalized" into the returned result. Otherwise they are flushed and the program tries again with a new instance or a different combination of elements. If they are exhausted, the failure is returned to the caller who tried to get the instance. However, all of the effort that went into generating the instance is not wasted. Once an instantiation is added to the data base it stays there, even if the program to which it is returned does not find it satisfactory. Thus a window or a room which was instantiated but not used remains in the network as a disowned entity, so that a later program looking for the same thing can take advantage of the work already done. Of course the instance can be deleted if it

is later really found to be something different.

This scheme, while adequate for the simple examples tried so far, will have to be extended if it is to work with complicated sketches, at least as far as building more than the two must and must-not modifications. There are some relations which, if not present, absolutely rule out the possibility of the instantiation being correct, such as the requirement that a room have some means of entering and leaving. Other relationships may not be as important, or may correspond to something the user hasn't drawn yet, and thus should not completely rule out the match. For example, windows almost always occur in exterior walls, but in some cases there will be windows between rooms, or between a room and a porch. A first step could be to expand the modifications to include SHOULD-BE and SHOULD-NOT-BE, but a more important issue is how to make use of the knowledge gained by the failure to suggest the next step. Should the program try some other description or should it take a closer look at the low-level data, perhaps asking the line straightening program to reconsider one of its decisions? These questions are explored further in the next chapter.

3.6 Cleanup

One of the tests performed before the matcher returns to top-level is to make sure that all of the data in the

sketch is accounted for. As each item is matched to a node in the general description, it is marked as TAKEN, that is, unavailable to be used in another instantiation. At the end of the program if all of the data is not taken then the program must have missed something, probably because the left over items did not fit in the general network's model of what components were needed. If the left over items are scattered, they could be extra parts of already "complete" objects which might still have uninstantiated daughters. If many unused objects are grouped together, some form of demon will have to be invoked to try to make sense out of them and fit them into the plan.

4. Representation of Complex Structures

4.1 Introduction

The preceding chapter sets forth some of the basic elements of a knowledge-based sketch recognition system, elements which served as the basis for a simple computer program to do hierarchical top-down analysis of sketches. (The implementation is described in section 6.) While the program demonstrated the feasibility of the scheme, it came nowhere near satisfying the requirements of a sketch recognition system. While an implementation based solely on the preceeding section might be criticized as premature, it did provide a useful grounding in reality, and a way of crystalizing some ideas about what should be in a sketch recognition system. More importantly, once the outline of the scheme has been decided upon, it can serve as a framework for discussing the kinds of descriptors and procedures necessary in using semantic knowledge to recognize a sketch. This section offers some suggested methods for representing knowledge about a sketching domain, including the use of sequence, locality, and sharing of entities. Better methods are suggested for representing currently available low-level information such as lines and points. The chapter concludes with a discussion of the use of the hypothesise and test method to explore the implications of decisions and a mechanism

for dealing with, and avoiding as much as possible, backup from failures.

4.2 Low Level Descriptors

The current implementation of STRAIN[9], the Architecture Machine's line straightening program, outputs a sequential list of lines and their associated endpoints. Each point has its X and Y coordinates, while each line's data block contains the rate, pressure, length, and (implied by its position in the sequence) the time. In addition, each line has a pointer back to the original raw data points which generated it. These points are stored in a disk file or in the case of large drawings, magnetic tape. (It must be kept in mind that with the data tablet producing up to 400 points per second, and each point of raw data requiring 6 bytes of storage, that a 15 minute drawing occupies 400 x 60 x 6 x 15 =2 megabytesof data.) While the straightened data is more accessible, it is not always in a form most useful to the network matching program described in section 3. Often the network will specify a corner, but when the user has drawn what looks like a corner on the tablet, the data base does not contain what we would like to define as a corner, that is, an intersection of two lines latched together. The same object can have several different representations depending on the manner in which it was drawn, even if the

two-dimensional displays look the same. Sometimes we can take advantage of this phenomenon to resolve the user's intentions but sometimes it can get io the way. Another common problem occurs when a line is overtraced-that is, the machine encounters several lines where one was expected. One attempt was made to solve the problem by the use of a positional representation (see section 1.1) which turned overtraced lines into one fat line. Since it did not preserve the sequence of the lines, it often turned two intentionally seperate lines into one just because they were close together. The positional system did prove advantageous in locating regions however, and is discussed in section 4.4.

4.3 A Knowledge Based Line Straightener

The original HUNCH program[10] was very bold in its attempts to straighten lines, and even more daring in its latching of lines to nearby points. When it became obvious that the problem could not be solved by adjusting the global parameters of the latching program, one solution that was proposed was to combine a bold line "proposer" with a conservative line "criticizer," an idea which later lead to the hypothesize and test paradigm. Without any good ways of implementing the criticiser, the STRAIT program was shorn of its latching subroutine and rechristened STRAIN. When the issue of curves presented

itself, the straightening routines where made even more cautious so that curves would not be "straightened." Even if STRAIN could make "perfect" decisions, though, many problems would still remain, for the decision as to where is the best place to break a line ultimately depends on its context. Take figure 4-1 as an example.

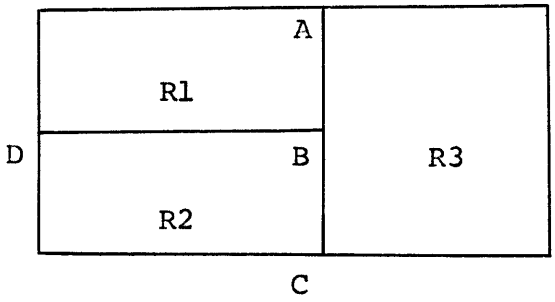

Figure 4-1 Example Plan

For the definitions of regions R1 and R2 we would like to represent the line AC as two lines, AB and BC, but for the definition of region R3 we would like to think of AC as one line segment. The problem may be further compounded if the line DB does not actually touch AC at point B, for then the low level routines have no way to even create point B. While the problem could be avoided by defining a room so that it could be bounded by a part of a line, this does not simplify the problem of finding the line. Besides, we would like to be able to refer to lines by

their endpoints and to be able to refer to the corner of a room. If we can refer to the lines as discrete lines, we can refer to them in local descriptions to allow the program to make use of their length and position in describing the room. We would like the machine to use its knowledge of architecture by using the search for components of the room to help define the lines making up the walls.

The "heuristic" line straightener would work in two parts, a sequential information processing phase and a knowledge-directed phase. The first phase is basically the present day STRAIN. The straightened, unlatched lines, without intersections noted, are added to the data base in the order they are drawn. The lines might be grouped together according to slope and position, so that overtraced lines are coalesced into "meta-lines" if the low-level program determines they are obviously intended to represent the same line. When a program needs an instance of a line to instantiate some part of a description, the second phase of the line finding routine is invoked. The parameters passed to this program can include the desired slope, position, and length of the line, or the region it is supposed to bound. If a line is found which passes through the desired points, it is chopped to its desired length and returned to the caller, who has the choice of using the truncated line, the

original line, or some modification thereof. If no line exists in the desired space, the program can look for a series of lines it can concatenate together or, failing that, it can look in the positional representation to see if the desired space is occupied by any points at all. If the points are there, the original raw data can be re-examined by what could be called a goal-driven STRAIN, a program which returns a line given the endpoints, like the line finder in Binford and Horn's vision system[11].

One pleasant result of the scheme just proposed is that it not only solves the problem of line straightening, but solves the problems of latching and finding intersections as well, by delaying decisions until the points are needed. If a pretty "STRAINed" output of the sketch is desired, a program could build new straightened lines out of all instances of lines in the completed network. Overtraced lines would become one line, and stray points would be eliminated.

4.4 Regions

A problem similar to finding lines is that of finding regions. Most simple region finding programs trace a path along the various lines of the scene until a complete circuit has been made, the edges of the region being defined by the path. In many sketching situations, including architecture, this definition is insufficient.

We would like a region to approximate a room with all of the inherent ambiguity, that is some space which is enclosed in some loose sense of the word, allowing for doors and corners which do not quite meet. It would be nice if the matcher could look for them the same way it looks for lines, exploring a part of the sketch where a region is expected and looking for the enclosing lines, but there is not usually as much information about where to look. One heuristic would be to look for regions bordering all known regions, but we need some more global information to know where to look for the first one. This could be done in a brute force manner by starting at some arbitrary place, but the positional representation of the sketch provides a convenient way of locating potential regions. By examining the bit map at a coarse scale, small gaps and irregularities such as doors and windows can be blurred, leaving only the gross features such as regions.

This technique was used in the plan recognition program described in section 1.1 and sufficed to find most of the rooms in a floor plan, but occasionally missed on small or irregularly shaped rooms, due to the global manner of application. Using the approach outlined here, the void finding program could use the grid first to find most of the regions and then then look locally, with either the grid or the straightened lines, at and around each region for other regions it missed. If the knowledge-based part

still needed another region, the program could take a closer look, since, as with line straightening, the parsing of a sketch into regions is dependent on the context. For example, in figure 4-2, it would be premature to divide

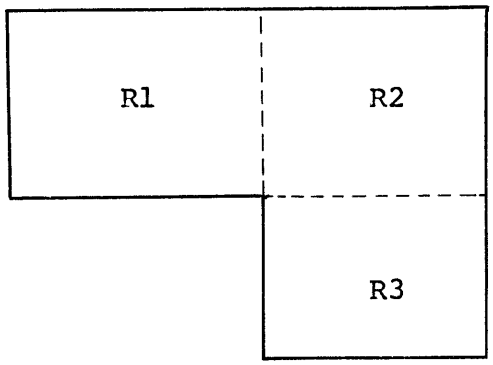

Figure 4-2 L-shaped room

the L-shaped room into one, two, or three regions until there was a need for them. The region finding program could make use of the various entities which divide rooms, so that, depending on the need, a room like the one in figure 4-3 can be looked upon as either one room or two.

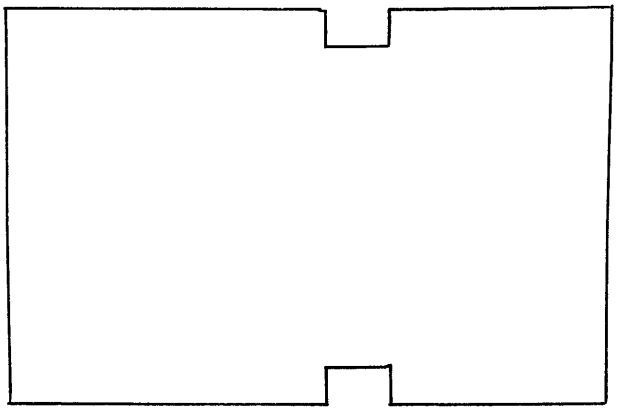

Figure 4-3

The problem becomes even more complex when the sketch describes a three-dimensional object. Perhaps a concept more useful than region or void is one of containment, which could be defined in terms of its various two and three dimensional representations, or as Michael Frieling[12] has suggested, objects should be defined in terms of their function. For example, a door would be defined as a way of getting from one place to another, while a room could be defined as someplace which contained, protected or isolated some space. This notion is especially appealing to architects, who are often uncomfortable with the thought of defining houses in terms of rooms, prefering instead to think of the functions and relationships between spaces. Such definitions could be

incorporated into the network, but will have to wait for someone to devise a suitable representation.

4.5 Locality

One of the original reasons for using semantic knowledge in a sketch recognition system was to provide a means of directing the search for entities in the sketch. No matter how good the pattern matcher is, it will have a hard time finding windows in a sketch if it doesn't know that they only occur in walls, usually exterior ones at that. Very often the same combination of lines appearing in a different context can have a completely different meaning. Also, a brute force search of every line in the drawing for a combination matchng a given pattern could be very time consuming, and since sketches are often incomplete, unsuccessful. Therefore part of the description of every object is some information about what other objects it is usually near or part of. This is done implicitly for objects which are elements of other objects, although the actual specification of what lines to look at to find some object will probably have to be specified procedurally. For example, a program to look for doors, given a room to look in, would first scan each wall, probably using the grid, for some evidence of "busy-ness." The line list could then be re-bound to just those lines in the busy area, and the matcher put to work looking for a door in

the small collection of lines.

4.6 Sequence

One of the properties of a computer-recorded sketch which looked most promising but has been the least exploited has been the sequence in which the lines were drawn. Sequence seems to be a natural way of separating items in a sketch, on the theory that the user will complete one object, or at least part of it, before drawing another one, or that the outline of something will be drawn before the details are filled in. There seems to be considerable room for variation here, possibly with more variation between individual users than between different objects.* This finding seems to indicate that sequence is a low level feature, and like straightening lines or isolating regions, its effect is relative to the context. Perhaps sequence can be embedded in an if-needed type of procedure for use in comparing the sequence of two complex objects, although a concept of sequence can be used in many parts of the system. Sequence is already used implicitly in the line straightening program, since by default it

---

* One striking example is in the way people draw faces. A novice will often start with the outline of the face, followed by the features, while an artist will usually start with the nose and eyes, gradually filling in the picture until the outline of face eventually appears. An initial study of human skeching behavior is reported by Cavannaugh and Markowitz in ref[13].

returns lines in the order in which they were drawn. Another use is in the instantiation program (INST of section 3.2) which could use the indication that two objects (or two lines) which are close together in space but distant in time might not be part of the same object, or could be different parts of it. For instance, an object would probably be drawn before any text that labels it.

Sequence could easily be encoded explicitly in the description of an object merely by providing an ordering to the elements, but the definitions probably would have to be stored individually for different users as was done in Teitleman's character recognition program[14].

4.7 Sharing - revisited

Section 3.6 mentioned the TAKEN descriptor and how it was used to indicate that an item had already been matched. Since an item in a sketch can be matched to more than one node in the general description, e.g. a wall shared by two rooms, some more complex mechanism is necessary to indicate which items can be shared and by how many different nodes. A wall, by definition, can only be shared by two rooms, a hallway can be shared by a arbitrary number of rooms, while a sink can only be in one room. This information could be stored either in the description of the shared object or in the node to be matched to it. The first solution offers a greater economy of storage, since only

one copy is needed, but the second offers the flexibility of some instances being shared and some not, so the solution adopted here is to allow both, as illustrated in figure 4-4. The definition of wall includes the restriction that it can only be shared by two nodes. The node represented by N9 is restricted to match something which nothing else matches, while the node N4 is content to share its instantiation with the entire world. It is also possible to specify sharing with a specific object, as with N10 in the figure which can only be shared with an instance of of a DINING-ROOM.

4.8 Adding Knowledge

One would hope that the task of adding new knowledge to this system will be an easy one. This should be the case, since the method used to encode the knowledge is independent of the subject matter. New descriptors need only be related to other relations or given if-needed methods to define their meanings. Adding new criteria merely requires that they be added to the data base. The pattern-directed invocation of procedures allows procedural knowledge to be added without affecting the old ones. One intriguing possibility is that the program could modify the descriptions dynamically, alllowing it to "learn." While the question of how machines can best be programmed to learn is a complex one, one basic requirement, a good

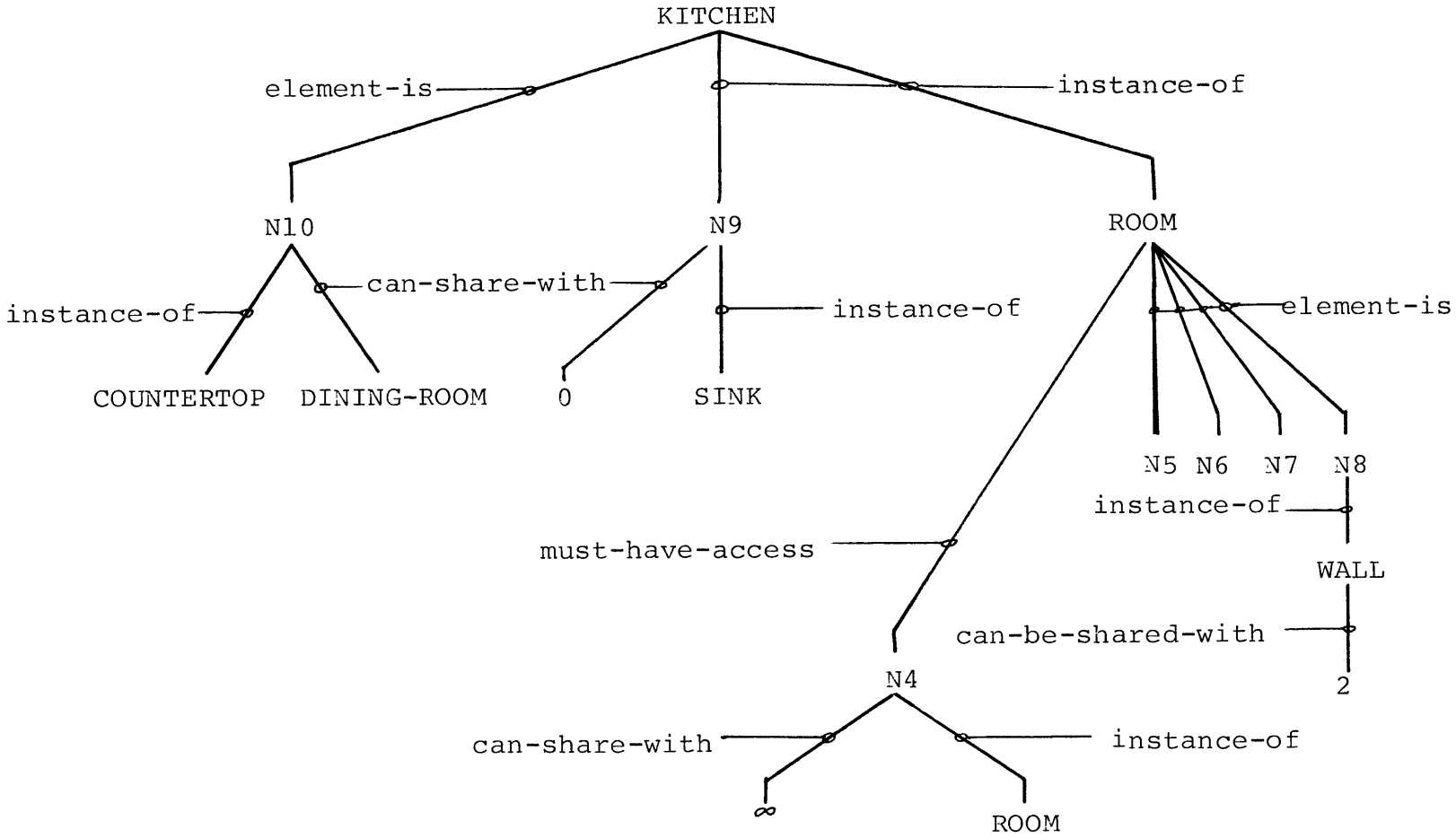

Figure 4-4 Sharing

descriptive mechanism, is present. What is needed for learning is some way of comparing the machine's model to the one provided by the teacher, perhaps a user trying to his explain his concept of "house" or even "comfort" to the system.

4.9 Hypothesis, Testing, Backup, and Failure

The instantiation program works by matching nodes in the general description to items in the database. If a mechanism exists to make temporary matches, such as the context levels of Conniver or QA4[15], these matches can be used as hypotheses with tests being performed on them to learn about the consequences. If the results of the tests are used merely to decide whether to accept or reject the original match, the result resembles the backup mechanism of PLANNER, which has proven itself inadequate for complex problems. We would like to make use of the consequences of the hypothesis, not just to decide what to do next, but to learn something about the data, as Minsky has suggested as part of his Frame Systems[7]. The hypothesis and test paradigm has the advantage that the machine always has some goal that it is trying to instantiate, and thus somewhere to store the information it aquires about the sketch, in this case the network it is building to return to the user. Every fact coming in can be used to either confirm or deny the hypothesis. If

it contains a contradiction, the mismatch can provide advice on what hypothesis would prove better. For example, if a program had assumed a particular room was a bedroom and was looking for a bed, and then discovered a stove, it could change the original hypothesis from a bedroom to a kitchen. This transformation is achieved through the use of what Minsky calls similarity pointers between objects having similar descriptions, and difference demons, programs which look for details that would indicate a match for one of the similarity pointers.

While the frame concept is appealing, there is no currently working program which embodies these principles, nor is there a language which can be used to easily implement them. While research goes on in these areas, we will have to be content with a simpler system. The instantiation program has two basic modes of disproving hypothesis. The first and most common occurs when one or more criteria have been violated: a vital component is missing or a prohibited one is present. For each of these criteria, there can be a pointer to what match (new hypothesis) would be a better one. An example might be that a window which does not open onto the outside might be a door. The pointer could indicate a demon procedure which will be run to determine what to do next.

The other way a hypothesis can fail is if it can not explain all of its data: some objects remain in the scene

which have no place in the general description. This problem is a little more difficult since our top-down approach has failed us here. If we are not looking for something it is harder to decide what the unknown objects are; they could be just about anything. One solution is to have similarity pointers to descriptions in other parts of the network that might be tried. If this new description matched, the program could see what elements matched the new description and try them out. For example, if the instantiation program thought it was in the dining room looking for a table, but the table had four circles in it, the program would follow a similarity pointer to STOVE and, if successful, change the initial assumption from dining room to kitchen. Of course by this time other bedroom objects might have been matched and need to be explained by following their similarity pointers as well.

A third possibility is that while the hypothesis has been disproved, no satisfactory alternates are present, necessitating the return of a failure to the calling program which can try another possibility or return failure itself. This isn't too bad if there aren't too many choices to be dealt with, and it will probably be the dominant mode of operation until the appropriate formalisms have been developed for more intelligent failure handling. The intention throughout is that failures will be avoided when at all possible by the use of the demons which can

sit at any node in the network, making sure that the conditions are favorable before a match is attempted. These same demons should be able to make use of the "failure messages" in order to make better choices.

Another interesting way the program can get itself out of an otherwise untenable situation is to ask the user for help since, after all, he is sitting at the computer terminal drawing his sketch or waiting for the computer to say something. Since this system is intended to run while the sketch is being drawn, the "failure" reported by the matcher might not be a failure on the machine's part but a misunderstanding on the part of the user. An anomaly caught by the machine early in the design might save the user from a costly or embarassing mistake later. Because the concepts in the network are those of the domain being sketched, the machine can use those concepts to communicate with the user, perhaps indicating graphically the appropriate part of the sketch. The program might ask the user a question like: "Why is this stove in the bedroom?" To which the user might respond:

1. The object is not a stove; it is a _____
2. The room is not a bedroom; it is a _____
3. I'm sorry, please delete the stove.
4. Because _____ _____

The first two responses indicate a mistake by the machine

and offer a way to correct it. The third indicates the designer's mistake and tells the computer to modify the description of the sketch. The fourth response indicates a misunderstanding between the user and the machine: the human insists on something the machine thinks is not possible. While the problems of natural language understanding can be avoided by providing a multiple-choice type of response, a more flexible mechanism would allow many more actions. Note that even simple responses like number 1 require some knowledge of language to allow for the possibility that the object in question is part of another object. One can envision a dialogue between the human and the computer, each pointing out pieces of the drawing and perhaps even drawing example sketches for each other. The emphasis of this thesis is on understanding the sketches, but with advances in understanding language, the combination of language understanding and sketch understanding should prove quite powerful.

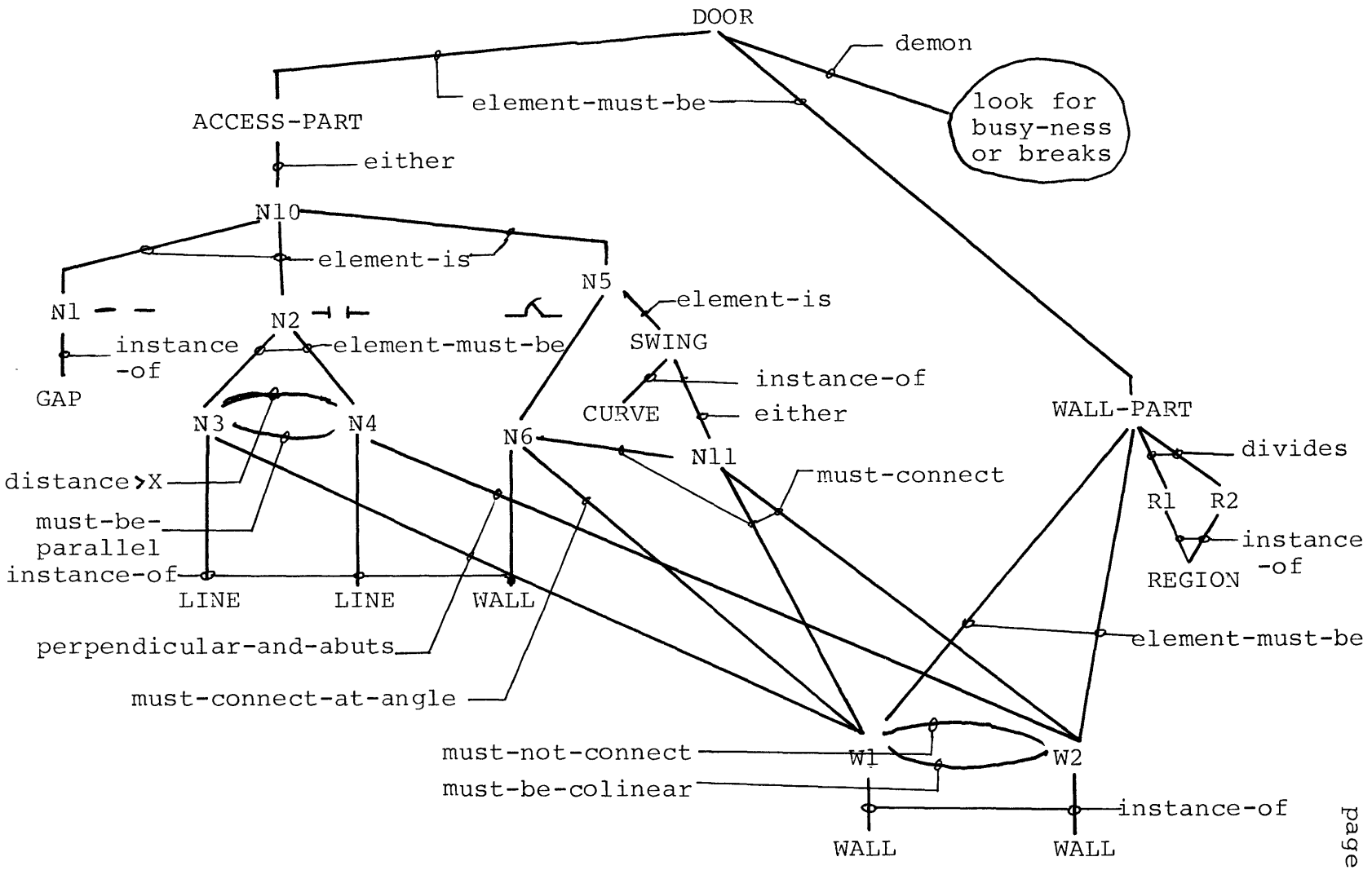

Figure 5-1 Description of a Door

5. An Example

This section offers a specific example to illustrate some of the concepts discussed in the preceding chapters. Figure 5-1 contains a general description intended to match a variety of doors as part of a larger network. Although it divides doors into three types, as illustrated by the sketches next to nodes N1, N2 and N5, all three types depend on a common notion that a door is something which divides two rooms and occurs as part of a wall. The top node has a demon associated with it which looks for door-like features like a gap or busy-ness (i.e. a group of small lines) along the walls in question. Because the top-down nature of the matcher insures that the description of a door will only be used when the matcher has a room needing a door, the demon knows which walls to look at and can search the list of lines making up the walls or look on the grid for places where the wall is thicker than usual. When the demon finds a likely spot on one of the walls, it matches the short line segments to the ACCESS-PART in such a way that they will be looked at first when the matcher tries to instantiate the elements of that node. The longer, co-linear lines are matched to the WALL-PART, such that the ACCESS-PART lines fall between them.

At this point the matcher can proceed in the usual manner, since the demon has found a likely spot for a door.

Had it attempted to find a door by matching the description to every combination of lines, it might have become hopelessly lost. After testing the criteria for the WALL-PART, that is, that the two lines must divide two regions, not touch each other and must be co-linear, it can try to instantiate the ACCESS-PART. This node is split three ways. The simplest type is just a gap in a wall, which is sufficient to define a door. The second description, N2 requires two parallel lines perpendicular to the wall and touching the lines of the WALL-PART. The third description, N5 is for a door with an optional swing, represented by a curved line. In all of the descriptions, the ACCESS-PART is assured to relate properly to the WALL-PART by the actions of the demon.

This example illustrates the importance of the demon procedure in ordering the search. It is through this method that the matcher knows to look for doors as part of the walls of certain rooms, eliminating the need for a depth-first search of the data base for the right combination of lines. By allowing the line list to be re-bound before trying to match the ACCESS-PART, the search for the appropriate lines can be drastically reduced from the entire data base to only those lines local to the door. Although there are only three descriptions of ACCESS-PART in this sample network, they are general enough to match a varity of doors. This looseness need not pose the

problem of finding doors where none exist since the top-down demon-ordered matching precludes applying the description where it is inappropriate.

6. Implementations

This section describes two separate computer programs which demonstrate the ideas presented in this thesis. The first one was done on the mini-computer facility of the Architecture Machine using a locally written disk-resident data base system, while the second implementation was done in the Conniver language, first on MIT's Multics time sharing system and later on the MIT Artificial Intelligence Laboratory's PDP-10 system.

6.1 Small Machine Approach

One of the goals of the Architecture Machine Group has been to develop a low cost, easy to use design system, one that could be used by architects and others who usually can not afford the cost of a computer. The answer at the time (around 1968) was to buy time on a time sharng system, but the kind of high bandwidth input and output inherent in graphics dictated a dedicated computer, leading to the concept of a mini-computer to handle the I/O and perform the preliminary processing of the data. The HUNCH program compressed the 400 points per second of raw data into a much smaller number of "straightened" lines, which could be sent to Multics over a 1200 baud telephone line. The response time of the Multics system proved to be too slow and the costs too high, however, and with the emergence of more powerful mini-computers, more and more processing

was done locally.

6.2 The Relational Data Base

One of the problems encountered with using mini-computers was the small (maximum 64K bytes) memory size, hardly enough to store a reasonable sized drawing if any programs were to occupy the same memory. Since paging hardware was not available, the only solution was to write programs to store the data in a file, the most recent of which became known as the Relational Data Base(RDB)[16]. The RDB is a set of FORTRAN or (eventually) LISP callable functions which allow data to be stored in the form of named objects, called atoms, and relations between them, similar to the methods used in PLANNER, CONNIVER, QA4, and several earlier systems developed by groups like IBM[17,18].

Data is added to the RDB in the form of ordered triplets, where each element of the triple is an atom. The second element of the triple is assumed to be the relation, but only by convention. The RDB itselfhas no built-in knowledge of the meanings of the atoms. Data is retrieved by giving a Planner-style pattern to a function. Once the pattern has been specified, a function call provides the matching relations one at a time, as in QA4 or Conniver until they are exhausted. In addition, a block of arbitrary data can be associated with each atom. This

is typically used to store large volumes of numerical data. Another feature is the use of contexts, or validation levels, which allow atoms, data blocks, and relations to be added provisionally. By "popping" a context the temporary additions can be removed all at once, restoring the data base to its previous state.

When a straightened sketch is added to the RDB, an atom is created for each point containing its coordinates in the associated data block, and an atom is created for each line, with the data block containing the slope, length, etc. Relationships are added to indicate which lines correspond to which endpoints, and a limited form of disk-based list processing is used to build lists of lines and points. A display of a simple drawing with the lines labeled is shown in figure 6-1. The relationships for that sketch are shown in figure 6-2.

The first step in implementing the sketch recognition program was the creation of a system for pattern directed procedure invocation, of which there are two varieties. For example, if there is a procedural representation of parallel, the user may wish to know if two specific lines are parallel or he may have a line and want all of the lines parallel to it. The Fortran implementation uses a table to match patterns to procedures, and each procedure can create a data block to store its internal state between calls. The output of a fetch for parallel lines is shown

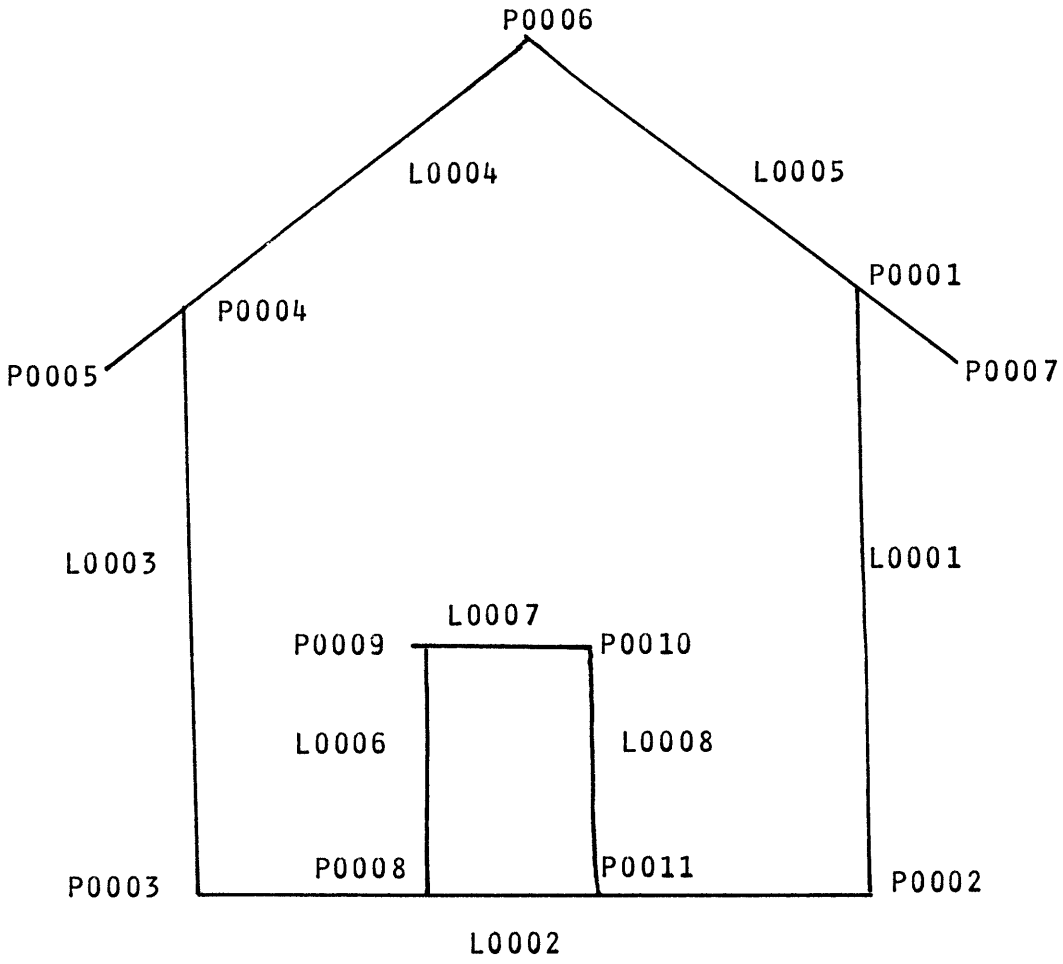
P0006
L0004
L0005
P0001
P0004
P0005
P0007
L0003
L0001
L0007
P0009
P0010
L0006
L0008
P0008
P0011
P0003
P0002
L0002

Figure 6-1

```
LINE_LIST
    OWNS
        L0001 L0002 L0003 L0004
        L0005 L0006 L0007 L0008

POINT_LIST
    OWNS
        P0001 P0002 P0003 P0004
        P0005 P0006 P0007 P0008
        P0009 P0010 P0011

L0001
    HAS_ENDPOINT
        P0001 P0002

L0002
    HAS_ENDPOINT
        P0002 P0003

L0003
    HAS_ENDPOINT
        P0003 P0004

L0004
    HAS_ENDPOINT
        P0005 P0006

L0005
    HAS_ENDPOINT
        P0006 P0007

L0006
    HAS_ENDPOINT
        P0008 P0009

L0007
    HAS_ENDPOINT
        P0009 P0010

L0008
    HAS_ENDPOINT
        P0010 P0011
```

Figure 6-2 Relationships for Figure 6-1

```
FETCH L0003 PARALLEL_TO ?
L0003 PARALLEL_TO L0001
L0003 PARALLEL_TO L0006
L0003 PARALLEL_TO L0008
```

Figure 6-3 Parallel Relations for Figure 6-1

in figure 6-3. This method works fairly well for one level generators, where an item can be fetched directly from the data base and tested, but since Fortran is a non-recursive language, the generator can not invoke another generator. Unfortunately, Lisp is not much help here either. While Lisp has a recursive control structure, it is not possible to preserve a part of the stack, making co-routines impossible. Several solutions to this problem have been proposed, including the "spagetti stack" suggested by Bobrow and Weigbreit[19] which will soon be implemented in INTERLISP[20]. In the meantime, a different approach was attempted.

6.3 Conniver

Coniver[21] is a Lisp-like language with a built in data base (similar in outward appearance to the RDB) and a facility for constructing arbitrary control structures. This "hairy control structure" facility implements stacks with Lisp list structures, known as CFRAMES which also contain values of local variables, permtting generators to be nested to any level. The interpretive Lisp environment also makes adding procedures much simpler than in Fortran.

The Conniver implementation took a different approach than the RDB implementation. Rather than starting at the bottom with actual points and lines, the Conniver

implementation was begun from the top, to handle general descriptions like houses. The lines were added to the data base with explicit assertions as to which lines were perpendicular and which regions were surrounded by which lines. The program followed the design of sections 3 and 4 except that it was simplified to eliminate the concepts of sharing and groups.

Since Conniver allows a tree of contexts rather than just a stack like the RDB, separate contexts were used for the general description and the data description. The relationships among the various contexts are illustrated by figure 6-4 where the upper

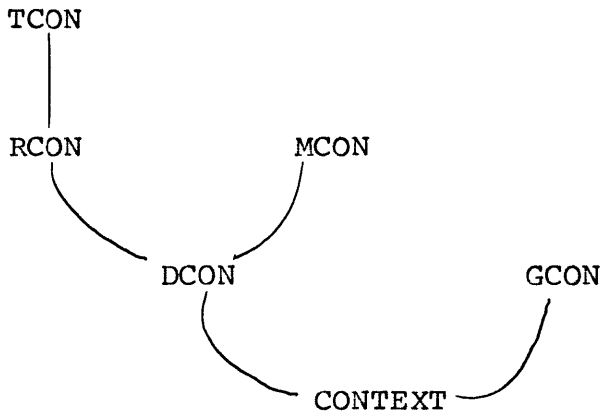

Figure 6-4 Context Tree

contexts contain any assertions present in inferior contexts. For example, any assertion in DCON will also

be true in RCON and MCON, but an assertion added to RCON or MCON will not be present in DCON. The lowest level is CONTEXT which is the context level provided when the system is initialized and, in this application, contains only the if-needed procedures. GCON and DCON are the two input contexts. GCON contains the general description, e.g. what a house should look like, while DCON contains a representation of the actual sketch under consideration. MCON is used by the instantiation program to store temporary matches while it is trying to instantiate an object. TCON and RCON are used to build the synthesised structure which will eventually be returned as the result.

The heart of the system is an if-needed method, INST which is called whenever an instantiation is not found in the data base. It is called with the argument TYPE to specify of what object an instance is needed. INST creates a new layer of MCON, superior to the old MCON every time it is called. This context will be used to store temporary matches for nodes of the description. Section 3 stated that a demon procedure would be called on entry to INST to determine what to look for. In this implementation, the demon is simply a program which finds all daughters (nodes descending from the one in question) and sorts them such that the INSTANCE-OF, if present, comes first, followed by MUST-BE modifications, MUST-NOT-BE modifications, and non-imperative relations, respectively.

Relations in imperative form are split into their non-imperative form and the type of modification. For example, a request for an instantiation of node N4 in figure 6-5 would result in the following sorted list of daughters:

```
((INSTANCE-OF LINE NIL INSTANCE-OF NIL)
 (MUST-SURROUND N3 MUST SURROUND NIL)
 (MUST-BE-PERP N5 MUST PERP NIL))
```

The last element of each sub-list, initially NIL, will be used to store the possibilities list, a Conniver device for storing the status of a fetch so that subsequent calls will return new elements. Thus if a given instance of some daughter is unsatisfactory, the possibilities list for that daughter allows a new one to be generated. If INSTANCE-OF is the first element of the list, an instance is fetched from the data base, possibly causing one to be generated by another call INST, and a match between the new instance and the corresponding node in the general description is added to both MCON and TCON. In addition, a notation is added to TCON to indicate that the fetched object is TAKEN so that some future effort to instantiate some object will not make use of the same components. Future implementations will have to process sharing relations here, but for now the program does not allow for sharing.

If no INSTANCE-OF appears in the list, it is because INST is expected to synthesize an instance. It GENSYMs a

unique name and asserts in TCON that it is an instance of the requested node. This would happen if INST were asked to find an instance of a general type like ROOM. Since ROOM does not have an INSTANCE-OF daughter but is instead composed of several elements, INST creates a new node with these elements as its ELEMENT-IS daughters.

In either case, the program tries to instantiate the rest of the daughters, testing the relations provided and keeping a numerical score. If a test succeeds, the score is incremented; if not, it is decremented. When the list has been exhausted, the score is compared to an arbitrary threshold to decide if the object is worthy of returning to the caller. If so, TCON is copied into RCON making the matches part of the result. If not, TCON is cleared and the process starts over, using the saved possibilities lists to find new instances. If the first daughter was an INSTANCE-OF, a new instance is fetched and the tests repeated. Otherwise, a new combination of components is tried. A successful instantiation results in an AU-REVOIR, the Conniver primitive to suspend the generator, save the stack and all local variables and return to the program which requested the instance. The next request will start the generator exactly where it left off, to generate a new instance.

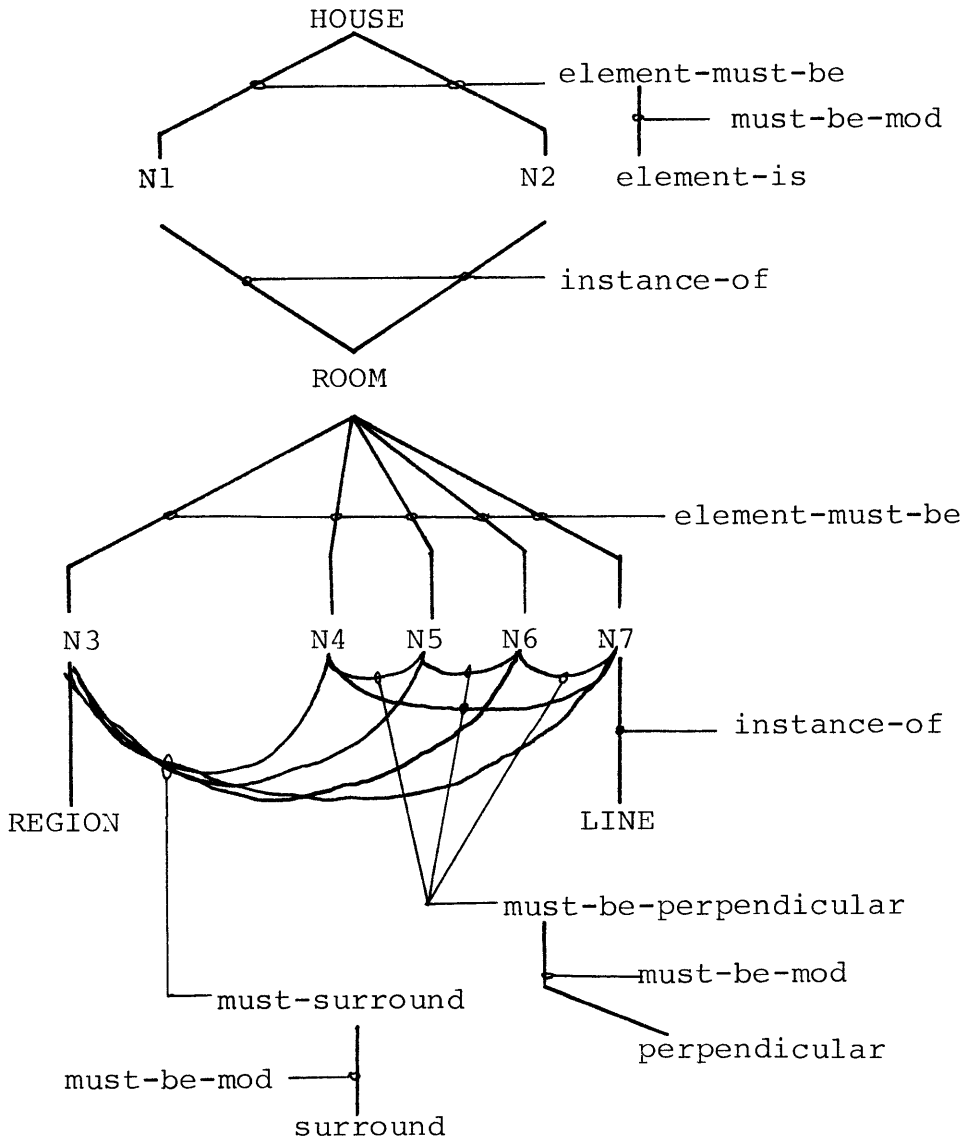

Figure 6-5 Sample Description of a House

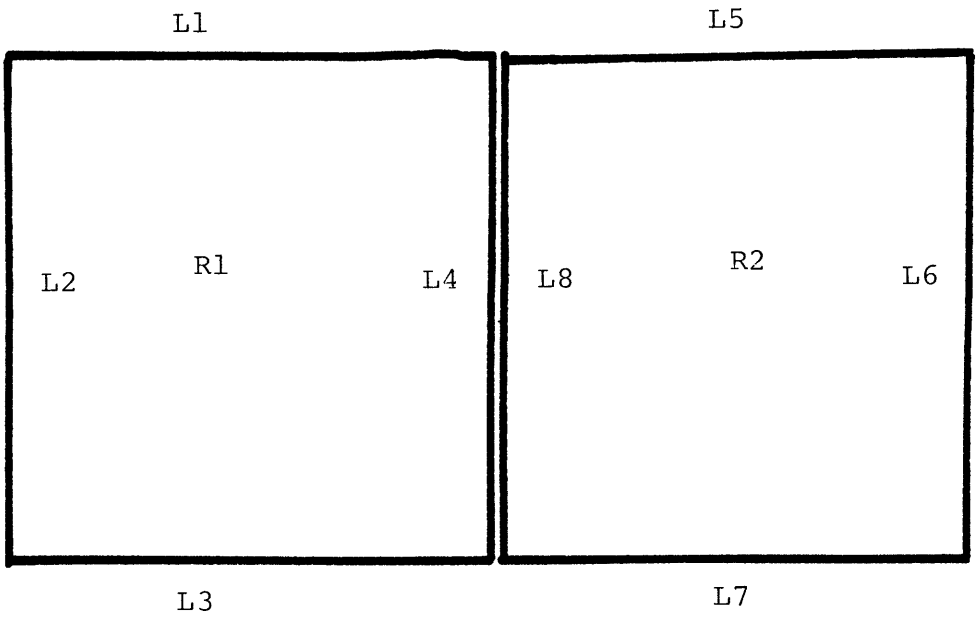

Figure 6-6 Sample Sketch

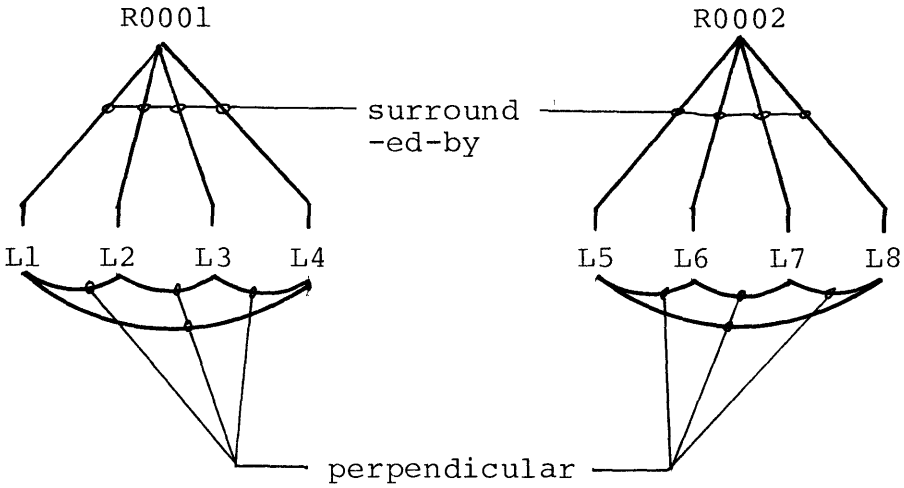

Figure 6-7 Diagram of Figure 6-6

```
((*POSSIBILITIES (!>X INSTANCE-OF HOUSE)) *IGNORE
 (*METHOD INST ((TYPE HOUSE) (ANSWER !>X))
 ((X *UNASSIGNED))  (!>X INSTANCE-OF HOUSE)))

-

(LOOKING FOR A HOUSE)

(LOOKING FOR A N2)

(LOOKING FOR A ROOM)

(LOOKING FOR A N5)

(TRYING SCORE 3, ANSWER L0001 FOR N5)

(RETURNING L0001)

(LOOKING FOR A N4)

(TRYING SCORE 1, ANSWER L0002 FOR N4)

(RETURNING L0002)

(LOOKING FOR A N6)

(TRYING SCORE 3, ANSWER L0003 FOR N6)

(RETURNING L0003)

(LOOKING FOR A N7)

(TRYING SCORE 1, ANSWER L0004 FOR N7)

(RETURNING L0004)

(LOOKING FOR A N3)

(TRYING SCORE 1, ANSWER R0001 FOR N3)

(RETURNING R0001)

(TRYING SCORE 5, ANSWER G0013 FOR ROOM)
```

Figure 6-8 Trace

```
(RETURNING G0013)
(TRYING SCORE 1, ANSWER G0013 FOR N2)
(RETURNING G0013)
(LOOKING FOR A N1)
(LOOKING FOR A ROOM)
(LOOKING FOR A N5)
(TRYING SCORE 3, ANSWER L0005 FOR N5)
(RETURNING L0005)
(LOOKING FOR A N4)
(TRYING SCORE 1, ANSWER L0006 FOR N4)
(RETURNING L0006)
(LOOKING FOR A N6)
(TRYING SCORE 3, ANSWER L0007 FOR N6)
(RETURNING L0007)
(LOOKING FOR A N7)
(TRYING SCORE 1, ANSWER L0008 FOR N7)
(RETURNING L0008)
(LOOKING FOR A N3)
(TRYING SCORE 1, ANSWER R0002 FOR N3)
(RETURNING R0002)
(TRYING SCORE 5, ANSWER G0014 FOR ROOM)
(RETURNING G0014)
(TRYING SCORE 1, ANSWER G0014 FOR N1)
```

Figure 6-8 Trace (continued)

```
(RETURNING G0014)
(TRYING SCORE 2, ANSWER G0012 FOR HOUSE)
(RETURNING G0012)
(*NOTE ((X G0012)))
```

Figure 6-8 Trace (continued)

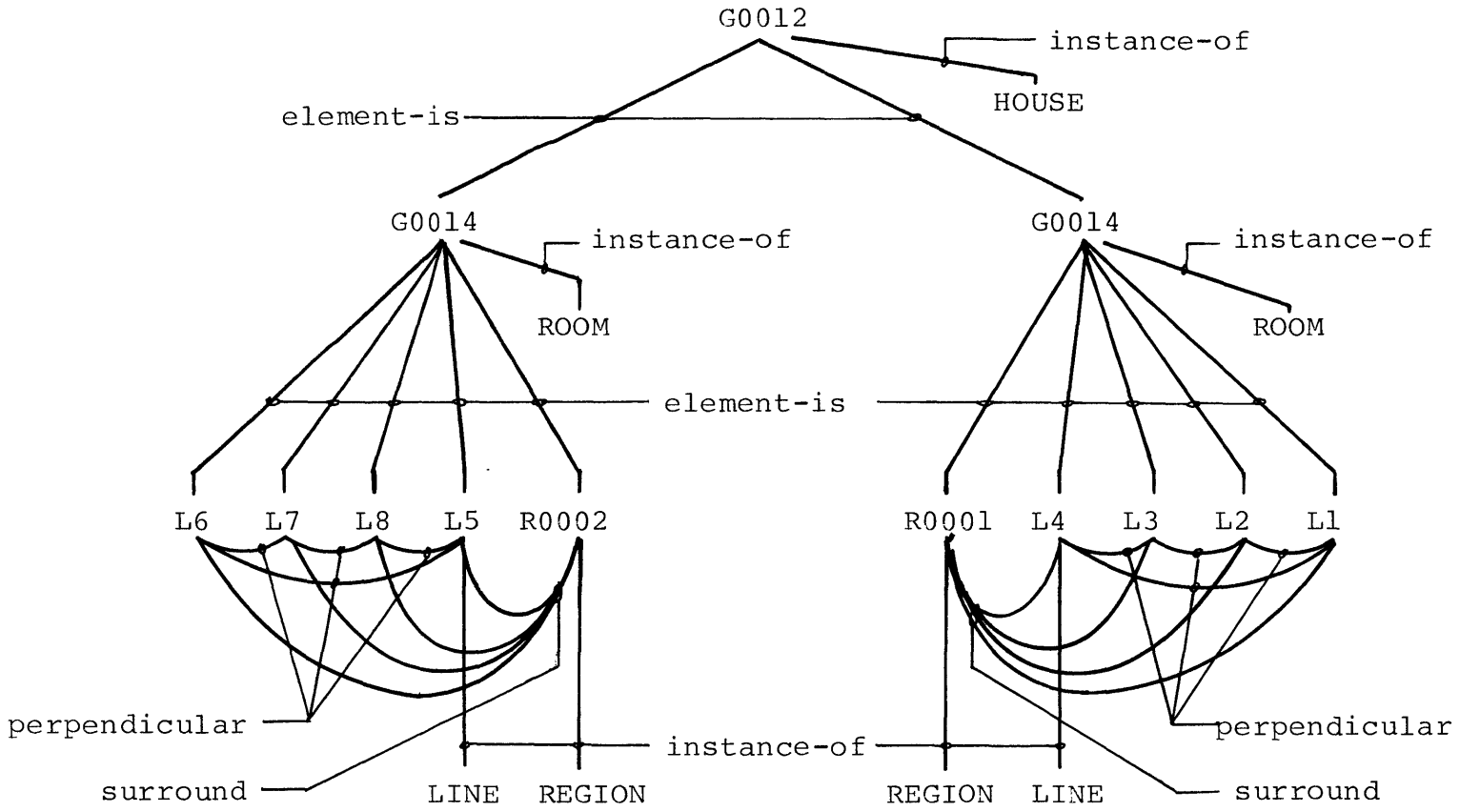

Figure 6-9 Instantiated Network

6.4 Example of Program Operation

A sample description (figure 6-5) of a house was added to GCON in the form of Conniver assertions and a sample sketch (figures 6-6 and 6-7) was added to DCON. A trace printed by the program is shown in figure 6-8. The function SORT-PD is the demon which sorts the daughters and returns their modifications. The output left inRCON is diagramed in figure 6-9.

6.5 Implications

Any program of the type described here places considerable demands on the supportng computer system. The large amount of information required for even a simple structure and a simple drawing necessitates that the mini-computer store it on a disk, resulting in a loss of speed. Even more important is the control structure required to support the type of pattern directed invokation described here. The simple recursive control structures of Lisp and PL/1 are simply not enough to implement the generator functons in a clean, concise, and understandable manner. While the same effect can be achieved through the explicit saving of variables, the resulting program would become so complex that progress would be impossible. Although Conniver is the answer insofar as providing these facilities as primitives of the language, implementation of its run-time stack in Lisp is so inefficient as to make

Conniver too expensive and slow to be of practical use. In addition, the context mechanism has recently come in for some criticism for being too inefficient and not giving the user enough control over the fetching process [22,8], the same kind of criticism the inventors of Conniver had for Planner[23]. Unfortunately, most of the proposed solutions have called for even more complex control structures and data bases, so the implementation problem is not likely to become easier in the near future.

The present problem though, is how to efficiently implement the system described in this paper on a machine of modest cost. The very minimum seems to be a simple Lisp with a facility for modifying control structures. The data base and context mechanism need not be as complex as that of Conniver, but some method must be devised for dealing with large descriptions, whether within Lisp or as part of an external file system. The control structure problem can probably be solved with a spagetti-stack as proposed by Bobrow and Weigbreit although, in any case, the available memory will have to be larger than the 64K bytes available on most mini-computers and used in the first experiment.

7. Discussion

This section presents a scenario following the developments described in this thesis. In its most elementary form, the system could be used in an operator-based form like SKETCHPAD, where the user could draw in his sketch, perhaps labeling those objects the machine couldn't understand. Then he could perform the usual editing operations on it, moving objects around the sketch until it was satisfactory, perhaps by means of macro operations like "delete the bedroom." Since the system would know how to separate elements of the sketch, merely pointing at a part of an object or naming it would suffice; it would not be necessary for the user to specify what lines belonged to what object. As the system became more advanced, it could be used to answer questions about the design or feed the results to another program which could estimate costs, do area take-offs, structural evaluations, and even try to improve on the design. Yona Freidman has suggested a design system for urban settings where the computer would serve as an intermediary among groups with competing interests[24]. As the system improved further, the interactions with the user could become more useful, informing him of possible errors, even things that he might not have seen. Eventually, if the machine could build a sufficiently good model of the designer himself, the

machine could even do some of the design for him, filling in lines or even entire parts of the sketch for him.

8. Conclusions

While some of the scenarios outlined here may seem somewhat fanciful, and the goal of the project may seem overly ambitious, it is hoped that the partial results achieved along the way have more than theoretical significance. Even a partial success in understanding sketches would be an improvement over present graphical input systems. The real doubts about achieving the stated goal of a reasonable understanding of sketches stem from the complexities involved in dealing with such a potentially contradictory and incomplete form of input. The first question that arises about the design of the system is whether it will be able to avoid getting lost in the complexities of the real world, or more specifically, if the hierarchical, top-down approach offered here is adequate for the task of understanding complex sketches.

The answer has to be "probably not," but then one has to start somewhere. The top-down approach is the strength of the whole system and seems to offer the most hope, and the ability to make use of partially successful results, even if it alone is not sufficient. If necessary, the form of input can be restricted at first, since even a good contextual symbol recognizer would be beneficial.

The research described in this thesis is only the

beginning. Just as the first crude implementation pointed out many areas of vagueness in the original concept, resulting in the improvements outlined in Chapters 4 and 5, the next implementation is expected to present many more complications. A major difficulty in any complex programming task is that something which appears very simple on the surface can prove very complicated when the question of implementation arises. This is especially true here, where such a structured interpretation is attempted of large amounts of data.

REFERENCES

[1] Ivan Sutherland, "SKETCHPAD: A Man-Machine Graphical Communications System," Proceedings of AFIPS, Spring, 1963.

[2] James Taggart, Reading a Sketch by Hunch, Master's Thesis, MIT Department of Electrical Engineering, May, 1973.

[3] Nicholas Negroponte, Soft Architecture Machines, MIT Press, forthcoming.

[4] Nicholas Negroponte, et. al., Machine Recognition and Inference Making in Computer Aids to Architecture, Proposal to National Science Foundation, MIT Department of Architecture, 1973.

[5] Patrick Winston, Learning Structural Descriptions from Examples (PhD Thesis), Technical Report AI-TR-231, MIT Artificial Intelligence Laboratory, September, 1970.

[6] David Waltz, Generating Semantic Descriptions from Drawings of Scenes with Shadows (PhD Thesis), Technical Report AI-TR-271, MIT AI Laboratory, November, 1972.

[7] Marvin Minsky, Frame Systems, in progress.

[8] Scott Fahlman, A Hypothesis-Frame System for Recognition Problems, Working Paper 57, MIT AI Laboratory, December, 1973.

[9] Nicholas Negroponte, "Recent Advances in Sketch Recognition," Proceedings of the AFIPS, NewYork, 1973.

[10] James Taggart, "HUNCH - An Experiment in Sketch Recognition," in Computer Graphics, edited by W. Giloi, Berlin, 1971.

[11] Berthold Horn, The Binford-Horn Line Finder, Vision Flash 16, MIT AI Laboratory, June 1971.

[12] Michael Freiling, Functions and Frames in the Learning of Structures, Working Paper 58, MIT AI Laboratory, December, 1973.

[13] Joe Markowitz and Kevin Cavanaugh, "The Cavanaugh Experiment," appendix to Machine Intelligence and Inference Making in Computer Aids to Architecture, Proposal to National Science Foundation, MIT Department of Architecture, Spring, 1974.

[14] Warren Teitelman, "Real Time Recognition of Hand-Drawn Characters," Proceedings of AFIPS, Fall, 1964.

[15] J. Rulifson, J. Derksen, and R. Waldinger, QA4: A Procedural Calculus for Intuitive Reasoning, AIC Technical Note 73, Stanford Research Institute, November, 1972.

[16] Chris Herot and Michael Miller, Relational Data Base Programming Manual, MIT Architecture Machine Group, January, 1974.

[17] M. Crick et. al., A Data-Base System for Interactive Applications, IBM Cambridge Scientific Center Report G320-2058, July, 1970.

[18] M. Crick, A. Symonds, A Software Associative Memory for Complex Data Structures, IBM Cambridge Scientific Center Report G320-2060, August, 1970.

[19] Daniel Bobrow and Ben Wegbreit, A Model and Stack Implementation of Multiple Environments, BBN Report Number 2334, Bolt, Beranek, and Newman, Inc.

[20] Warren Teitelman et. al., Interlisp Reference Manual, Xerox Palo Alto Research Center, 1974.

[21] Drew McDermott and Gerald Sussman, Son of Conniver: The Conniver Reference Manual, MIT AI Lab, 1973.

[22] William Martin, OWL: A System for Building Expert Problem Solving Systems Involving Verbal Reasoning (draft), Spring, 1974.

[23] Gerald Sussman and Drew McDermott, Why Conniving is Better Than Planning, AI memo 255A, MIT AI Laboratory, April 1972.

[24] Yona Friedman, "Information Processes for Participatory Design," appendix Machine Intelligence and Inference Making in Computer Aids to Architecture, 1973.

APPENDIX:
The Conniver Program

```
;;;             ********    INST     ***********
;;;
;;;
;;; INST TRIES TO FIND AN INSTANCE OF AN ITEM
;;; OF TYPE TYPE BY FINDING ITS COMPONENTS.
;;;
;;;
(ADD (IF-NEEDED INST (!<ANSWER INSTANCE-OF !>TYPE)
          "AUX" ((SCORE 0)  REL COMPONENT MOD-TYPE
                 (INSTFLAG T) (NEWINSTFLAG NIL) (MCON (PUSH-CONTEXT MCON))
                 ELEM PDLIST PDLIST-SAVE PD)



          (TPRINT !"(LOOKING FOR A ,TYPE))

;;;THE FUNCTION SORT-PD FETCHES ALL DAUGHTERS OF TYPE, SORTS THEM, AND
;;;RETURNS A LIST OF ITEMS OF THE FORM:
;;;       (RELATION COMPONENT MOD-TYPE MOD-OP PLIST)
;;;WHERE:
;;;  RELATION      IS THE RELATION FETCHED FROM THE GENERAL DESCRIPTION
;;;  COMPONENT     IS THE OBJECT OF THE RELATION
;;;  MOD-TYPE      IS EITHER MUST-BE MUST-NOT-BE OR NIL
;;;  MOD-OP        IS THE SAME AS RELATION UNLESS IT IS A MODIFICATION IN  WHICH
;;;                CASE IT IS THE NON-IMPERITIVE FORM OF THE RELATION
;;;  PLIST         IS INITIALY NIL AND IS A PLACE WHERE THE POSSIBILITIES
;;;                LIST CAN BE STORED
;;;
          (OR (CSETQ PDLIST-SAVE (SORT-PD TYPE))
              (PROG (TPRINT !"(NO DAUGHTERS FOR ,TYPE))
                    (ADIEU)))
:RESTART  (CSETQ PDLIST PDLIST-SAVE)  ;NEW COPY
          (CSETQ PD (CAR PDLIST)
                 PDLIST (CDR PDLIST)
                 COMPONENT (CADR PD)
```

```
                MOD-TYPE (CADDR PD)
                REL (CADDDR PD)
                SCORE 0)
;;;
;;; FIRST WE MUST MAKE A TENTATIVE MATCH - FIND SOME OBJECT
;;; TO SERVE AS A HYPOTHESIS
;;;
          (COND((EQ REL 'INSTANCE-OF)
                (CSETQ NEWINSTFLAG T)  ;INDICATE THIS IS AN INSTANCE-OF
                (RPLACA (CDDDDR PD)
                      (FETCH !"(!>ANSWER INSTANCE-OF ,COMPONENT) DCON))
!TAKEN1         (TRY-NEXT (CADDDDR PD) '(ADIEU)) ;GET ONE, IF NONE -> LOSE
                (AND (PRESENT !"(TAKEN ,ANSWER) TCON) (GO 'TAKEN1))
                (ADD !"(,ANSWER INSTANCE-OF ,COMPONENT) TCON) ;FOR LATER
                (CSETQ SCORE (ADD1 SCORE)))
               (T (CSETQ ANSWER (GENSYM)) ;IF NOT AN INSTANCE, MAKE ONE
                  (GO 'GOON)))          ;SKIP OVER NEXT PD
;;;
!NEXTC    (AND (NULL PDLIST) (GO 'EV-SCORE))  ;IF ALL GONE, EVALUATE
          (CSETQ PD (CAR PDLIST)
                PDLIST (CDR PDLIST)
                COMPONENT (CADR PD)
                MOD-TYPE (CADDR PD)
                REL (CADDDR PD))
;;;
;; NOW WE TRY TO FIND THE OBJECTS WHICH RELATE TO THIS ONE
;; TESTING THEM TO SEE IF THEY SATISFY THE CONSTRAINTS
;;;
 !GOON    (COND  ((EQ REL 'ELEMENT-IS)
                 (AND INSTFLAG (RPLACA (CDDDDR PD)
                       (FETCH !"(!>ELEM INSTANCE-OF ,COMPONENT)
                                                    DCON))))
!TAKEN2          (TRY-NEXT (CADDDDR PD) '(GONE))
                 (AND (PRESENT !"(TAKEN ,ELEM) TCON) (GO 'TAKEN2))
                 (ADD !"(TAKEN ,ELEM) TCON))
```

```
                ( T             ;OTHERWISE WE HAVE A SPECIFIC RELATION TO SATISFY
                  (AND (PRESENT !"(,COMPONENT MATCHES !>ELEM) MCON)
                       (TRY-NEXT (FETCH !"(,ANSWER ,REL ,ELEM) DCON) *(GO *NO))
                       (GO *YES))
                  (AND INSTFLAG (RPLACA (CDDDDR PD) (FETCH !"(,ANSWER ,REL !>ELEM) DCON)))
*TAKEN3           (TRY-NEXT (CADDDDR PD) *(GO *OTHER))
                  (AND (PRESENT !"(TAKEN ,ELEM) TCON) (GO *TAKEN3))
                  (GO *YES)
          *OTHER  (AND INSTFLAG (RPLACA (CDDDDR PD)
                                 (FETCH !"(!>ELEM INSTANCE-OF ,COMPONENT) DCON)))
*GET-OTHER (TRY-NEXT (CADDDDR PD) *(@ONE))
          (AND (PRESENT !"(TAKEN ,ELEM) TCON) (GO *GET-OTHER))
               (TRY-NEXT (FETCH !"(,ANSWER ,REL ,ELEM) DCON) *(GO *GET-OTHER))))
;;;
*YES    (ADD !"(,COMPONENT MATCHES ,ELEM) TCON) ;FOR LATER
          (COND ((EQ TYPE COMPONENT)  ;IF THIS SEARCH FOR COMPONENT
                 (ADD !"(,COMPONENT MATCHES ,ELEM) (POP-CONTEXT MCON)))
                (T              ;OF SAME TYPE, SAVE FOR NEXT MATCH
                   (ADD !"(,COMPONENT MATCHES ,ELEM) MCON)))
        (ADD !"(,ANSWER ,REL ,ELEM) TCON) ;FOR LATER
          (COND ((EQ MOD-TYPE *MUST-BE) (CSET@ SCORE (ADD1 SCORE)))
                ((EQ MOD-TYPE *MUST-NOT-BE) (CSET@ SCORE (SUB1 SCORE))))
          (GO *NEXTC)
;;;
*NO       (COND     ((EQ MOD-TYPE *MUST-BE) (CSET@ SCORE (SUB1 SCORE)))
                    ((EQ MOD-TYPE *MUST-NOT-BE) (CSET@ SCORE (ADD1 SCORE))))
          (GO *NEXTC)

*EV-SCORE
          (TPRINT !"(TRYING SCORE ,SCORE ANSWER ,ANSWER FOR ,TYPE))
          (COND ((> SCORE 0)
                 (TPRINT !"(RETURNING ,ANSWER))
                 (FINALIZE TCON) ;DUMP INTO DATA CONTEXT
                 (AU-REVOIR (INSTANCE)))
```

```
            (T (TPRINT *FAILED)))
          (CSET@ TCON (PUSH-CONTEXT (POP-CONTEXT TCON)))    ;RESET
          (CSET@ INSTFLAG NEWINSTFLAG)  ;IF DOING AN INSTANCE-OF, GET A NEW ONE
                                        ;AND RE-DO THE POSSIBILITIES LISTS.
                                        ;IF DOING@ SOMETHING@ COMPOSED OF ELEMENTS
                                        ;KEEP POSSIBILITIES TO GET NEW SET OF ELEMENTS
          (GO *RESTART) ))


(DEFUN CADDDDR (F00)
   (CAR (CDDDDR F00)))

(CDEFUN @ONE()       ;SEE'S IF SCORE IS NE@ATIVE...TIME TO @UIT
          (COND      ((MINUSP SCORE)
                      (TPRINT !"(RAN OUT OF INGREDIENTS FOR ,TYPE))
                      (ADIEU))
                     (T (GO *NO))))


(CSET@ TFLAG T)      ;TRA@E FLAG

(CDEFUN TPRINT (SKEL)
          (AND TFLA@
               (@PRINT SKEL)))



;;;                 ********** SORT-PD **********
;;;                 THIS IS THE DEMON BOX
;;;
;;; SORT-PD LOOKS UP ALL BRANCHES EMENATING FROM A SPECIFIED NODE AND
;;;RETURNS A LIST OF ITEMS OF THE FORM:
;;;       (RELATION COMPONENT MOD-TYPE MOD-OB PLIST)
;;;WHERE:
```

```
;;;   RELATION      IS THE RELATION FETCHED FROM THE GENERAL DESCRIPTION
;;;   COMPONENT     IS THE OBJECT OF THE RELATION
;;;   MOD-TYPE      IS EITHER MUST-BE MUST-NOT-BE OR NIL
;;;   MOD-OB        IS THE SAME AS RELATION UNLESS IT IS A MODIFICATION IN  WHICH
;;;                 CASE IT IS THE NON-IMPERITIVE FORM OF THE RELATION
;;;   PLIST         IS INITIALY NIL AND IS A PLACE WHERE THE POSSIBILITIES
;;;                 LIST CAN BE STORED
;;;
;;;
(DEFUN SORT-PD (TYPE)
   "AUX"  ((POSSIBILITIES-LIST (FETCH !"(,TYPE !>REL !>COMPONENT) GCON))
           (INSTANCE NIL) (MUST-BE NIL) (MUST-NOT-BE NIL) (OTHER NIL)
            REL COMPONENT MOD-TYPE MOD-OB)

!MORE      (TRY-NEXT POSSIBILITIES-LIST '(GO 'DONE))
           (COND ((EQ REL 'INSTANCE-OF)
                  (CSETQ INSTANCE
                         (NCONS (LIST 'INSTANCE-OF COMPONENT NIL 'INSTANCE-OF NIL ))))
                 ((PRESENT !"(,REL MUST-BE-MOD !>MOD-OB) @CON)
                  (CSETQ MUST-BE
                         (CONS (LIST REL COMPONENT 'MUST-BE MOD-OB NIL) MUST-BE)))
                 ((PRESENT !"(,REL MUST-NOT-BE-MOD !>MOD-OB) GCON)
                  (CSETQ MUST-NOT-BE
                         (CONS (LIST REL COMPONENT 'MUST-NOT-BE MOD-OB NIL)
                                                          MUST-NOT-BE)))
                 (T
                  (CSETQ OTHER
                         (CONS (LIST REL COMPONENT NIL REL NIL) OTHER))))
           (GO 'MORE)


!DONE      (RETURN @(APPEND ,INSTANCE
                            (APPEND ,MUST-BE
                             (APPEND ,MUST-NOT-BE
                              (APPEND ,OTHER NIL))))) )
```

```
;;;                  ********** DSYS.1 **********
    (PROG (CSETQ GCON (PUSH-CONTEXT CONTEXT)) ;GENERAL DESCRIPTION
          (CSETQ DCON (PUSH-CONTEXT CONTEXT)) ;DATA CONTEXT
          (CSETQ RCON (PUSH-CONTEXT DCON)) ;RESULTS
          (CSETQ TCON (PUSH-CONTEXT RCON)) ;TAKENS
          (CSETQ MCON (PUSH-CONTEXT DCON)) ;MATCHES
          (RETURN '(DESCRIPTIVE SYSTEM)))


;;;  IF-NEEDED METHODS FOR LINES AND REGIONS

;;;    LINES

(ADD (IF-NEEDED GETLIN (!<LINES INSTANCE-OF LINE)
          "AUX"  ((TLINE-LIST LINE-LIST))
 :LOOP    (OR TLINE-LIST (ADIEU))        ;IF NO LINES...LOSE
          (CSETQ LINES (CAR TLINE-LIST)
                 TLINE-LIST (CDR TLINE-LIST))
          (AU-REVOIR (INSTANCE))
          (GO 'LOOP) ))


;;;    REGIONS

(ADD (IF-NEEDED GETREGION (!<REGIONS INSTANCE-OF REGION)
          "AUX" ((TREGION-LIST REGION-LIST))
:LOOP     (OR TREGION-LIST (ADIEU))
          (CSETQ REGIONS (CAR TREGION-LIST)
                 TREGION-LIST (CDR TREGION-LIST))
          (AU-REVOIR (INSTANCE))
          (GO 'LOOP) ))
```

```
;;;        DESCRIPTION OF A HOUSE

  (PROG    (ADD '(HOUSE ELEMENT-MUST-BE N1) GCON)
           (ADD '(HOUSE ELEMENT-MUST-BE N2) GCON)
           (ADD '(N1 INSTANCE-OF ROOM) GCON)
           (ADD '(N2 INSTANCE-OF ROOM) GCON)
           (ADD '(ROOM ELEMENT-MUST-BE N4) GCON)
           (ADD '(ROOM ELEMENT-MUST-BE N5) GCON)
           (ADD '(ROOM ELEMENT-MUST-BE N6) GCON)
           (ADD '(ROOM ELEMENT-MUST-BE N7) GCON)
           (ADD '(ROOM ELEMENT-MUST-BE N3) GCON)
           (ADD '(N3 INSTANCE-OF REGION) GCON)
           (ADD '(N4 INSTANCE-OF LINE) GCON)
           (ADD '(N5 INSTANCE-OF LINE) GCON)
           (ADD '(N6 INSTANCE-OF LINE) GCON)
           (ADD '(N7 INSTANCE-OF LINE) GCON)
           (ADD '(N4 MUST-BE-PERP N5) GCON)
           (ADD '(N5 MUST-BE-PERP N6) GCON)
           (ADD '(N6 MUST-BE-PERP N7) GCON)
           (ADD '(N7 MUST-BE-PERP N4) GCON)
           (ADD '(N4 MUST-SURROUND N3) GCON)
           (ADD '(N5 MUST-SURROUND N3) GCON)
           (ADD '(N6 MUST-SURROUND N3) GCON)
           (ADD '(N7 MUST-SURROUND N3) GCON)
           (ADD '(ELEMENT-MUST-BE MUST-BE-MOD ELEMENT-IS) GCON)
           (ADD '(MUST-BE-PERP MUST-BE-MOD PERP) GCON)
           (ADD '(MUST-SURROUND MUST-BE-MOD SURROUND) GCON)
           (RETURN 'HOUSE-DESCRIPTION))
;;;                ********** ONE.ROOM **********

    (PROG  (CSETQ LINE-LIST '(L0001 L0002 L0003 L0004 L0005 L0006 L0007 L0008))
           (CSETQ REGION-LIST '(R0001 R0002))
           (ADD '(L0001 PERP L0002) DCON)
```

```
(ADD *(L0002 PERP L0003) DCON)
(ADD *(L0003 PERP L0004) DCON)
(ADD *(L0004 PERP L0001) DCON)
(ADD *(L0005 PERP L0006) DCON)
(ADD *(L0006 PERP L0007) DCON)
(ADD *(L0007 PERP L0008) DCON)
(ADD *(L0008 PERP L0005) DCON)
(ADD *(L0001 SURROUND R0001) DCON)
(ADD *(L0002 SURROUND R0001) DCON)
(ADD *(L0003 SURROUND R0001) DCON)
(ADD *(L0004 SURROUND R0001) DCON)
(ADD *(L0005 SURROUND R0002) DCON)
(ADD *(L0007 SURROUND R0002) DCON)
(ADD *(L0008 SURROUND R0002) DCON)
(ADD *(L0006 SURROUND R0002) DCON)
(RETURN *(TWO ROOM HOUSE)))
```