TOOLS AND TECHNIQUES TOR KNOWLEDGE-BASED EXPERT SYSTEMS FOR ENGINEERING DESIGN

by

M.L, Maher, D. Srixam and S.J. Fenves

DRC-12-22^84

December, 1984

TOOLS AND TECHNIQUES FOR KNOWLEDGE-BASED EXPERT SYSTEMS FOR ENGINEERING DESIGN[1]

M. L. Maher, D. Sriram and S. J. Fenves

Department of Civil Engineering
Carnegie-Mellon University
Pittsburgh, PA 15213
U. S. A.

ABSTRACT

Knowledge-Based Expert Systems are emerging as an important tool kit for the development of engineering software. These systems provide a means to solve the ill-structured problems encountered in engineering design. A number of tools exist for the development of these systems. In this paper, the applicability of three such tools • OPS5, SRL, and PROLOG • to engineering design is described. Overviews of knowledge-based expert systems and various problem solving strategies are also provided.

KEY WORDS

Artificial intelligence, Knowledge-based expert systems, Problem solving strategies, Engineering Design, Civil Engineering, Production systems, Frame representation, Logic Programming

[1] To appear in Advances in Engineering Software, UJC

Tools and Techniques for Knowledge Based Expert Systems for Engineering Design

M. L. Maher, D. Sriram and S. J. Fenves
Department of Civil Engineering
Carnegie-Mellon University
Pittsburgh, PA 15213

1. INTRODUCTION

The emergence of Knowledge Based Expert Systems (KBES) provides a means for the engineer to use the computer as an aid in the solution of *ill-structured* problems. KBES are interactive computer programs that incorporate the knowledge and judgment of experts in appropriate domains. These systems promise to introduce changes at least as far-reaching as the entire computer revolution to date. The number of papers published on the applications of KBES to engineering problems in the last decade reflects the interest being shown in the engineering community [31].

The development of a KBES presently involves the cooperative effort between one or more experts who possess the domain-dependent knowledge and a knowledge engineer (KE). The KE elicits the knowledge and uses either an expert system building tool or a general-purpose language to represent and manipulate it. The representation of knowledge in a KBES is dependent on the selection of the tool or language to be used. The KE must make a choice among several available tools before embarking into a major developmental task; the ease of building an expert system depends in part on the choice of the tool.

The purpose of this paper is to survey a number of tools and techniques available for building KBES and discuss their applicability to engineering design. The nature of engineering design is briefly presented in the next section, followed by a review of KBES. Section 4 provides a description of the problem solving strategies employed in current expert systems, and an overview of languages and tools is given in Section 5. A civil engineering design example is discussed in Section 6. Three domain independent tools are further illustrated through the implementation of the design example.

2. ENGINEERING DESIGN

Design may be defined as the process in which an idea is developed, refined, and elaborated into the detailed instructions for manufacturing [24]. The design process is typically divided into phases, where each phase may be handled by different individuals from different disciplines. The design process begins with the identification of a need and ends with the *complete* product specification. Typically this process requires more than engineering design and analysis, for example, the architectural considerations in the design of a building. However, this paper addresses only the engineering aspects of a design. Engineering design may be decomposed into the following tasks.

1. Preliminary Design. The input to this task is the specification of the product in terms of the properties and conditions that it must satisfy. During the preliminary design task several alternative designs are considered with respect to their feasibility and one alternative is selected for further consideration, based on the satisfaction of a few key constraints.

2. Analysis. The goal of the analysis task is to determine the expected response of the design alternative to its intended environment. This task involves the selection of an analysis technique, the representation of the physical model by a mathematical model, the analysis of the mathematical model, and finally the interpretation of the results of the analysis.

3. Detailed Design. The detailed design task involves the specification of the parameters required for the production of the selected design alternative. This task includes insuring the satisfaction of all applicable constraints.

4- Evaluation and Optimization. The detailed design alternative is evaluated. Backtracking to an earlier task may be necessary so as to achieve a feasible, acceptable or optimal design.

The design process, as outlined above, starts with the visualization of the product at the highest abstract level, and as the design progresses, this abstraction is refined into smaller subsystems. Such an approach is referred to as hierarchical planning. Depending on the complexity of the product, the design process may become very complex, requiring different problem solving strategies at different levels of the design. For example, the overall approach to building design requires working from the abstract to the detail, but some aspects of the building design may require proposing details and working toward more general abstractions. Also, the design process rarely follows the indicated order of tasks without backtracking. It is necessary to make assumptions during design that may lead to inconsistencies or contradictions as the design progresses. In such cases it is necessary to backtrack to a previous task and revise the assumption.

In addition to the description of engineering design in terms of the solution process, it is important to describe the design problem in terms of the constraints on the solution. Engineering design is constraint oriented: much of the design process involves the recognition of applicable constraints

and the satisfaction of these constraints. There are many sources of constraints, ranging from subjective constraints imposed by individuals to constraints imposed by the fundamental laws of nature. The efficient and knowledgeable handling of the potentially large number of constraints can expedite the design process (for a detail discussion of constraint handling in engineering design see [18,25]).

3. OVERVIEW OF KBES

The gamut of tasks performed by experts consists of a spectrum bounded by *derivation* and *formation* tasks [1]. In derivation tasks, the problem conditions are described as parts of a solution description; this description is completed by using the rules so that the given facts are well integrated into the solution. On the other hand, in formation tasks the problem conditions are given in the form of properties that the solution as a whole must satisfy; the possible candidate solutions are *generated* and *tested* against the given conditions or constraints. In real life, most tasks fall between these two extreme categories. Tasks normally encountered at the derivation end of the spectrum are: *interpretation, diagnosis, monitoring, control,* and *repair. Planning* and *design* are typical of tasks at the formation end.

KBES are interactive computer programs which attempt to simulate the expert's thought processes, providing advice for a wide range of problems as described above. Since this article focuses on the tools and techniques for implementing KBES, only a brief overview of KBES is included in this section.

KBES typically consists of three major components (see Figure 1).

- Knowledge-Base. The knowledge-base consists of general facts and heuristic (rules of thumb) knowledge. A number of formalisms, such as production rules, frames (concepts), logic and semantic nets, are available for representing knowledge. Expert system frameworks based on these representation formalisms are discussed in later sections.

- Context. The context is a collection of symbols or facts that reflects the current state of the problem at hand. It consists of all the information generated during a particular program,execution.

- Inference machine. The inference machine controls the processing of the program by using the knowledge-base to modify the context. Important problem solving strategies used in some existing systems are discussed in the next section.

The components discussed above form the kernel of most existing expert systems. In addition, there

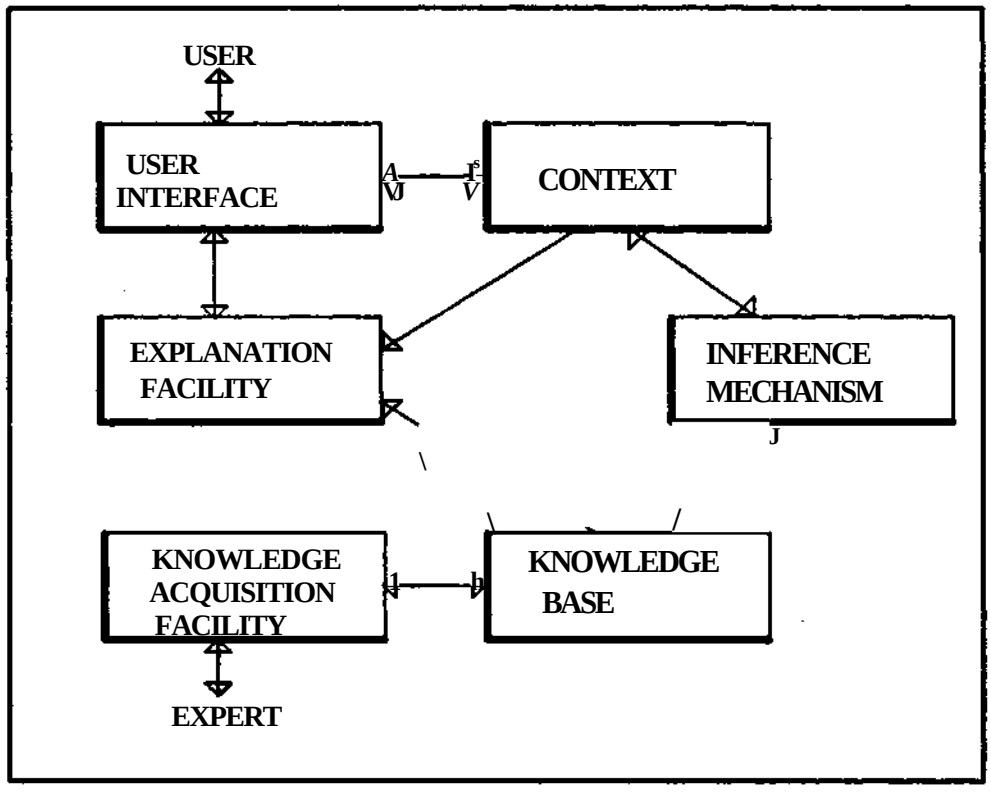

Figure 1: A Schematic View of a Complete Knowledge-Based Expert System [30]

are three modules which are desired in any expert system: the user interface, the knowledge-acquisition, and the explanation modules. A more complete description of knowledge-based expert systems for civil engineering is treated in [29], and good overviews can be found in [9,13,21].

4. PROBLEM SOLVING STRATEGIES

Problem solving involves the search for a solution through a state space by the application of operators, where the state space (the possible states in the problem solution) consists of an initial state, a goal state and intermediate states. The solution path consists of all states that lead from the initial state to the goal state. Domain independent problem solving strategies are commonly referred to as *weak methods* and may lead to combinatorial explosions. Expert systems can be considered *strong problem solvers* since they employ domain knowledge in the solution strategy. In this section a number of problem solving strategies used in current expert systems are briefly presented and discussed in light of their applicability to engineering design. More detailed descriptions of a number

of problem solving strategies can be found in [23, 28,33].

Forward Chaining. A system is said to exhibit forward chaining (bottom-up, data-driven, antecedent-driven are all equivalent to forward chaining) if it works from an initial state of known facts to a goal state. Here all facts are input to the system and the system deduces the most appropriate hypothesis or goal state that fits the facts. The main drawback of this strategy is that it is extremely wasteful to require as input data all the possible facts for all conditions; in many circumstances all possible facts are not known or relevant. This strategy is useful in situations where there are a large number of hypotheses and few input data. Sometimes the problem solving mechanism is guided by the events occurring during the solution process; this type of forward chaining is called *event-driven.*

The forward chaining strategy is not appropriate for a design problem if possible goal states of the design problem are not easily represented by a discrete number of hypotheses. Forward chaining may be used for certain subtasks of the design process, such as selecting the appropriate modelling options for the analysis of a specified configuration. In this subtask the known facts are specifications of the configuration and the goal states represent the modelling options.

Backward Chaining. A system is said, to exhibit backward chaining (also referred to as consequent-driven, top-down, goal-driven and hypothesis-driven) if it tries to support a goal state or hypothesis by checking known facts in the context. If the facts in the context do not support the hypothesis, then the preconditions that are needed for the hypothesis are set up as sub-goals. Essentially, the process can be viewed as a search in the state space going from the goal state to the initial state by the application of inverse operators (as opposed to operators in forward chaining) and involves a *depth first search.*

Backward chaining, in its pure form, is not appropriate for the engineering design process, since the possible goal states of the design process are not easily represented by hypotheses. However, the concept of backward chaining may be applied to the decomposition of the tasks in engineering design. If the current state of the context is not in the proper form for the completion of a task, the task may be decomposed into subtasks. In this way the overall design task may be decomposed into several subtasks and backward chaining may be applied to each subtask.

Means-ends Analysis. In means-ends analysis, the difference between the current state and the goal state is determined and used to find an operator most relevant to reducing this difference. If the operator is not directly applicable to the current situation then the problem state is changed by setting up subgoals so that the operator can be applied. After an operator has been applied, the current state

corresponds to a modified state. Means-ends analysis utilizes both the forward and backward chaining techniques. However, this strategy is only applicable to those tasks where the measures of differences between the various states and the operators to reduce these differences can be formulated apriori.

The use of means-ends analysis requires the formulation of possible states in the solution path and of the operators required to move from one state to another. Currently, most engineering design processes are not sufficiently formalized for this representation. There may be portions of engineering design in a particular domain that lend themselves to a means-ends analysis; however, more research towards the formalization of design is required for the use of means-ends analysis as an overall strategy.

Problem Reduction. Problem reduction involves factoring problems into smaller subproblems. The problem is represented by means of an AND-OR graph. An AND node consists of arcs pointing to a number of successor nodes, all of which must be solved for the AND node to be *true* (or solved). For an OR node, it is sufficient for one of the successor nodes be solved; an OR node indicates that a number of alternate solutions exist for the problem. In many cases, backward chaining is used to solve the AND-OR graph. A detailed description of an algorithm (AO^*) for finding solutions in an AND-OR graph is given in [28]. AO^* is very useful in tackling large complex problems, where the subproblems have minimum interaction.

The problem reduction strategy is easily applied to the design process, as current design practice typically reduces the design problem into subproblems. The use of this strategy in a KBES for engineering design requires the development of an appropriate graph to represent a particular design problem.

Plan-Generate-Test. The generate-and-test strategy in its purest form generates all possible solutions in the search space and tests each solution until it finds a solution that satisfies the goal condition. The plan-generate-test sequence restricts the number of possible solutions generated by an early pruning of inconsistent solutions. The pruning is achieved by the planning stage, where the data is interpreted and *constraints* are evaluated; these constraints eliminate solutions that are inconsistent.

The plan-generate-test strategy is appropriate for the design process if appropriate tests can be formulated. Typically, there is no unique solution to a design problem; therefore there is no absolute test for a solution. This strategy can be applied to the preliminary design phase if the testing stage

can be recast as a ranking stage, to determine the relative value of the possible solutions generated.

Backtracking. The problem reduction approach is applicable to problems that can be subdivided into a tree of fixed subproblems. However, in a number of practical problems it may not be possible to decompose problems into a fixed set of subproblems. A number of alternate approaches may exist. In backtracking, the problem solver backs up to other nodes, at the same level as the starting node, if no solution is found along the current path. Backtracking is incorporated in many artificial intelligence (AI) languages, such as PROLOG [7]. Bracktracking, in its pure form, poses a number of difficulties. To provide an efficient way of backtracking from wrong guesses, Stallman and Sussman [32] developed the concept of *dependency-directed backtracking (DDB).* In DDB, a record of all deduced facts, their antecedent facts along with their support justifications and the relevant rules are maintained; these records are known as *dependency records.* Support justifications are justifications for any assumptions made during the search. When the problem solver comes to a dead end, it retrieves the antecedents of the *contradiction.* Those facts which give rise to the contradiction are removed from further consideration. This strategy involves a lot of book-keeping. However, this additional book-keeping helps in a number of ways. For example, explanation of the program behavior can be extracted from the dependency records. This concept was further extended by Doyle [8] for systems that incorporate *nonmonotic reasoning.*

The need for backtracking cannot be ignored in the engineering design process. It is unusual that the first solution considered satisfies all applicable conditions and constraints. Some kind of backtracking must be incorporated in a KBES for engineering design. The degree of backtracking required depends on the application and dictates the amount of book-keeping needed. In a system with limited backtracking, the recovery from violated constraints may be handled by heuristics.

Hierarchical Planning & Least Commitment Principle. The concept of hierarchical planning involves developing a plan at successive levels of abstraction. For example, in the design of complex systems the design space is divided into a set of levels, where the higher levels are abstractions of details at lower levels; the problem is hierarchically decomposed into loosely coupled subsystems. A number of solutions may exist for each subsystem. However, enough information may not be available to ascertain various variables of the subsystem. Further, the solution to one subproblem may depend on the decisions (or variable bindings) made in the solution of another subsystem. To minimize this dependency, it is important to defer binding decisions as far as possible. This principle is called the *least commitment principle* because variables are not instantiated (decisions are deferred) until more information about the problem space is available.

The application of hierarchical planning and least commitment principle to engineering design is appropriate for the design of complex systems. Many engineering design problems may be formulated as hierarchical subsystem designs. The interaction of different subsystems may require deferring some decisions on a particular subsystem until information from another subsystem is available. The task of incorporating these principles into a KBES requires formalizing the design problem so as to recognize when decisions should be deferred and when deferred decisions should be reconsidered.

Constraint Handling. If the subgoals in hierarchical planning do not interact with each other, they can be solved independently. However, in practice these subgoals do interact. The interaction between subgoals can be handled by *constraint satisfaction* methods. Constraint satisfaction methods involve the determination of problem states that satisfy a given set of constraints. Essentially, constraint satisfaction methods utilize constraints to determine the values of parameters in a *completely specified* problem. Stefik [34] proposed an extension to the classical constraint satisfaction method (see [17] for a review of constraint satisfaction methods) by integrating it into hierarchical planning. This method, known as *constraint posting*, involves three stages.

1. Constraint formulation is the operation of adding new constraints representing restrictions on variable bindings. The constraints contain increasing detail as design progresses.

2. Constraint propagation is the creation of new constraints from formulated constraints. This operation handles interactions between subproblems through the reformulation of constraints from different subproblems.

3. Constraint satisfaction is the operation of finding values for variables so that the constraints on these variables are satisfied.

Constraint handling is an important part of the engineering design process. Much of the work that has been done in this area could well be applied to the formal mathematical constraints encountered in engineering design, such as the causal constraints imposed by the laws of nature. There are constraints applicable to engineering design that are not yet formalized, such as constraints imposed by individual preference. These informal constraints require special consideration in the constraint handling process.

Agenda Control. When a human problem solver is required to perform a number of tasks at one time, he gives a priority rating to these tasks. The task with the highest priority rating is performed first. In other words, he prepares an *agenda* of tasks. A list of justifications and a priority rating can be associated with each task. This type of control can be used for complex tasks that require focusing

attention on certain parts of the problem. Agendas can also be used in systems that require several independent sources of expertise to communicate with each other.

This strategy is appropriate for a KBES for the design of a complex system. As the complexity of the problem increases, due importance should be given to the selection of an appropriate solution strategy. The agenda control strategy allows the most flexible approach to the design problem solving strategy.

5. LANGUAGES AND TOOLS FOR BUILDING KBES

A number of languages and tools are currently available for building KBES. These tools can be grouped into three categories [13].

General Purpose Programming Languages. AI projects are usually implemented in a high-level language. These high-level languages need some novel features, such as facilities for experimentation with large chunks of knowledge, tentative modifications, planning and reasoning strategies. In addition, these languages need powerful abstraction mechanisms with which other higher level constructs can be built so as to make programming flexible and easy. Current expert system frameworks have been built using a number of languages, of which LISP [37] and PROLOG seem very popular among AI researchers. Bobrow [4] discusses some of the languages used in AI research.

General Purpose Representation Languages. General purpose representation languages are programming languages developed specifically for knowledge engineering. These languages are not restricted to implementing any particular control strategy, but facilitate the implementation of a wide range of problems encompassing the derivation-formation spectrum. Some general purpose languages are: SRL [38], RLL [12], KEE [14], OPS5 [11] (OPS5 has an inbuilt control strategy, but is more general than the domain independent frameworks), ROSIE [10], LOOPS [5], and AGE [22].

Domain Independent Expert System Frameworks. A domain independent expert system framework provides the system builder with an inference mechanism, from which a number of applications can be built by adding domain specific knowledge. Such systems also provide knowledge-acquisition and explanation modules to simplify the construction of the expert systems. These frameworks normally have evolved out of domain specific KBES. Hence, their control strategies are restricted to those provided in the original system. Systems under this category include: EMYCIN [35], KAS [26],

TOOL/LANGUAGE	DEVELOPER	REPRESENTATION SCHEME	IMPLEMENTATION LANGUAGE
OPS5	Carnegie-Mellon University	Rules	LISP/BLISS
EMYCIN	Stanford University	Rules	INTERLISP
HEARSAY-HI	USC-ISI	Rules	INTERLISP
EXPERT	Rutgers University	Rules	FORTRAN
ROSIE	Rand Corporation	Rules	INTERLISP
KS300	Tecknowledge Inc.	Rules	INTERLISP
AGE	Stanford University	Rules	INTERLISP
KAS	SRI International	Rules & Smantic nets	INTERLISP
KMS	University of Maryland	Rules/Frames	LISP
KEE	IntelliGentics Inc.	Rules & Frames	INTERLISP
RLL	Stanford University	Frames	INTERLISP
PSRL	Carnegie-Mellon University	Rules & Frames	FRANZLISP
LOOPS	Xerox-PARC	Rules & Frames	INTERLISP-D
KL-ONE	Rand Corporation	Semantic nets	INTERLISP
C-PROLOG	University of Edinburgh	Logic	C

HEARSAY-III [2], EXPERT [36], and KMS [27] (currently marketed as KES).

A number of widely used languages and tools is shown in Figure 2. A detailed description of these tools is beyond the scope of this paper and the reader is referred to Part V of [13]. In the following sections, OPS5, SRL, and PROLOG are illustrated with an engineering paradigm.

6. AN ENGINEERING DESIGN PROBLEM

The applicablity of various tools to engineering design is illustrated through a civil engineering example, which involves the synthesis of alternate floor systems for the preliminary structural design of a building. This is the first part of the preliminary design phase, described in Section 2. The next step would involve the selection of a floor system from among these alternatives, followed by analysis and detailed design phases; these steps are not addressed here.

The input to this design problem is a two dimensional rectangular grid defining a typical bay in the building. The dimensions of the bay are defined by b, the larger dimension, and d, the smaller dimension (see Figure 3). It is assumed that the floor slab is continuous over adjacent grids. In the following implementations, an alternate floor system is synthesized from three levels of information: 1) slab action; 2) type of material; and 3) type of support condition. The problem solving strategy is modelled after the problem reduction strategy described in Section 4, with constraints used to reduce the search space; the design problem is reduced to the three subproblems of selecting an alternative from each level of information.

The first level is the selection of the appropriate mode of behavior or *action* of the slab. A slab tends to resist the bending moment induced by the gravity load in two-way action (bending in two orthogonal directions) when the aspect ratio is near unity; the slab is designed for half of the total bending moment in each direction. A slab with a large aspect ratio behaves in one-way action; the total gravity load is resisted by bending in the short direction. For two way action, the larger dimension is conservatively taken as the span; for one-way action, the smaller dimension is the span.

The constraints on the selection of one-way or iwo-way action used in the following implementations are[2]:

[2]The representation used here may not lead to the most efficient implementation. However, it was chosen to illustrate the design example.

```
IF  b/d <- 1.6
THEN two-way action 1s valid
IF  b/d > 1.2
THEN one-way action 1s valid
```

The upper bound in the first rule is given by experience [16]. The lower bound in the second rule is somewhat arbitrary. It is an illustration of the least commitment principle: oneway action may still be preferable for low aspect ratios if other components of the floor system not yet synthesized can provide an improvement.

The second level of information for the synthesis of alternate floor systems is material selection. The materials considered in the following implementations are: reinforced concrete, prestressed concrete, steel deck, and prefabricated concrete panels. There are two types of constraints for each of the above materials. The first set of constraints defines the range of economic spans for each material, taken from [16].

```
IF   span <- 12 feat
THEN steel deck 1s valid

IF   span <• 16 feet
THEN, prefabricated concrete panel 1s valid

IF   10 feet <- span <- 26 feet
THEN reinforced concrete slab 1s valid

IF   20 feet <- span <- 40 feet
THEN prestressed concrete slab 1s valid
```

The second set of constraints represents the interaction between slab action and material. These constraints are dictated by the nature of the slab material; for example, a steel deck is made from concrete cast on top of a ribbed steel form, and therefore the slab can only resist bending moment efficiently in the direction of the ribs. The constraints used in the following implementations are:

```
IF   one-way action
THEN steel deck 1s valid  AND
     prefabricated concrete panel 1s valid  AND
     reinforced concrete slab 1s valid  AND
     prestressed concrete slab 1s valid

IF   two-way action
THEN reinforced concrete slab 1s valid AND
     prestressed concrete slab 1s valid
```

The third level of information for the synthesis of alternate floor systems is the selection of the number of slab edges to be supported by beams. The alternatives at this level are:

- 0 edges indicating support by columns only (flat slab construction) (Figure 3 a);
- 2 edges indicating beam support on two opposite edges (Figure 3 b); and
- 4 edges indicating beam support on all four edges (Figure 3 c).

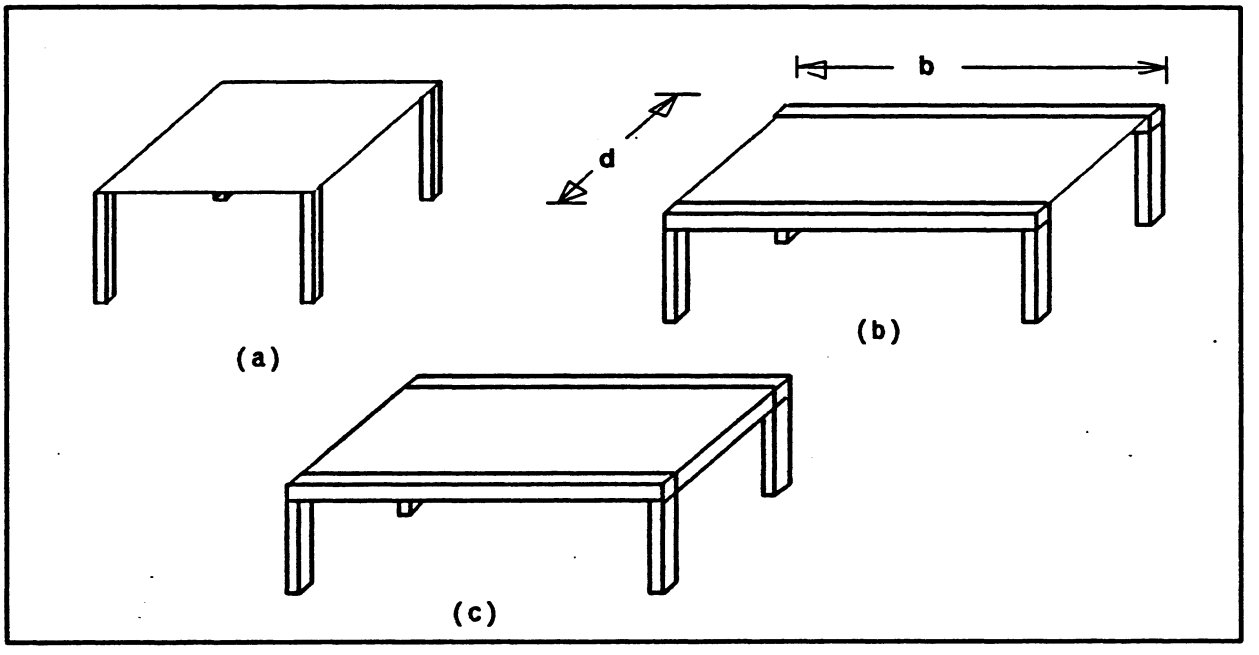

Figure 3: Support Conditions for the Slab

There are two types of constraints at this level; one representing the dependence of the support selection on the material, and the other on the type of action. These constraints are given below.

```
IF   material is steel deck  OR
     material is prefabricated concrete panel
THEN 2 edge support is valid

IF   material is reinforced concrete slab  OR
     material is prestressed concrete slab
THEN 0 edge support is valid  AND
     2 edge support is valid  AND
     4 edge support is valid
```

```
IF   one-way action
THEN 0 edge support 1s valid  AND
     2 edge support 1s valid

IF   two-way action
THEN 0 edge support 1s valid  AND
     4 edge support 1s valid
```

The first two constraints are dictated by the nature of the material. A steel deck or prefabricated panel result in one-way action as described before and therefore needs support by beams on 2 edges. A reinforced or prestressed concrete slab is not constrained with respect to the support. The last two constraints specify the acceptable support conditions for one-way and two-way action; i.e. a one-way slab does not require support on 4 edges.

A tree of all possible alternative solutions for the design example is shown in Figure 4. The result of the synthesis is the specification of a set of feasible alternate floor systems.

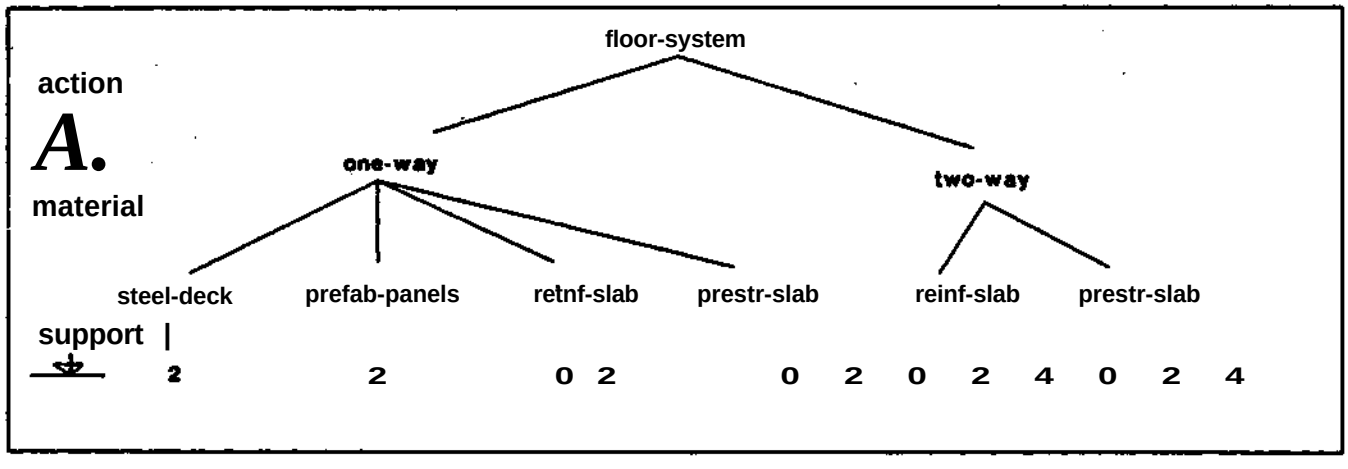

Figure 4: The Tree **of** Possible Alternative **Solutions**

7. PRODUCTION SYSTEM PROGRAMMING - OPS5

Production system programming is the most widely used style of programming in current expert systems. A production system consists of a set of rules manipulated by an inference engine. In a **pure** production system, commonly referred to as a *rule-based* system, the entire knowledge base is

encoded in IF-THEN rules.

Typically a production system has three parts: a working memory (context), a production memory (knowledge-base), and an inference engine. Working memory is a store of data representing the facts and assertions about the problem. The production memory contains a set of rules, referred to as production rules. A rule has a left hand side (LHS) and a right hand side (RHS); the LHS is the condition part and the RHS is the action part. The inference engine is based on a *find-select-execute* cycle, as shown in Figure 5 [6]. In the *find state,* the machine finds all rules whose conditions are satisfied by the data in working memory. In the *select state,* a rule is selected from the set of rules found in the find state. In the *execute state,* the selected rule is executed. Typically the execution of a rule modifies working memory and the find-select-execute cycle begins again.

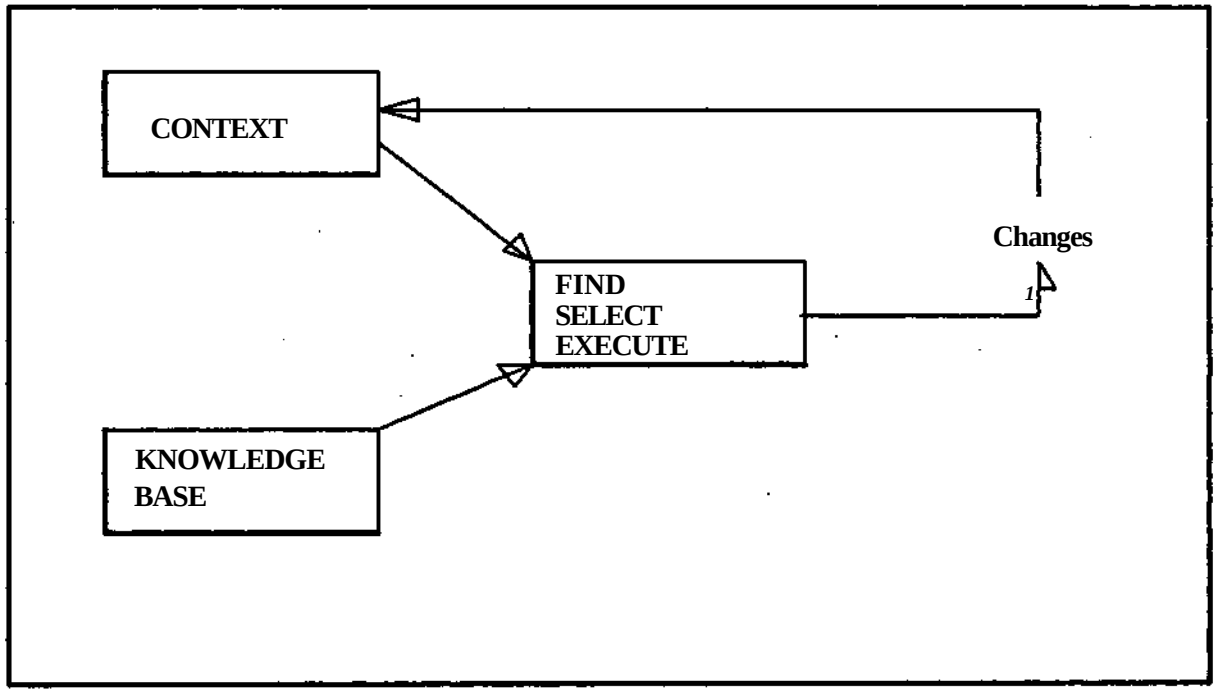

Figure 5: Execution Strategy in a Production System

OPS5 [11] is one of the series of OPS (Official Production System) languages developed at C-MU. OPS5 has been used to develop R1 [19], one of the few KBES used in industry, and a number of other systems in engineering and cognitive psychology. OPS5 is a pure production system, with facilities for calling LISP functions. The LISP functions may be used to represent procedural information or mathematical computations not easily handled by OPS5 rules.

The find-select-execute cycle implemented in OPS5 is referred to as the *recognize-act* cycle. A situation is recognized by matching working memory elements with rule conditions, as in the find state described above. When more than one rule matches in a single recognize-act cycle these rules are placed in a conflict set. The selection of one of the rules in the conflict set is called conflict resolution. OPS5 provides the user with two conflict resolution strategies: LEX (LEX refers to the lexicographic order of the sorted condition elements according to their recency) and MEA (MEA refers to Means-Ends-Analysis). Basically, the LEX strategy gives priority to the rule with the most recently created (or modified) condition element, and the MEA strategy gives priority to the rule with the most recently created (or modified) *first* condition element; this time is determined through a time tag that is attached to each element. The recognize-act cycle continues until the elements in working memory do not satisfy the condition part of any production rules.

In OPS5 all objects and attributes must be declared before their use in a rule. This declaration is done with a literalize statement as shown below:

```
(literalize grid b d bd-ratio)
```

This literalize statement declares an element class with the name **grid** and attributes **b**, **d**, and **bd-ratio**. The attribute **bd-ratio** is used by the constraint rules as shown below.

A rule in OPS5 consists of a unique name, one or more condition elements, and a sequence of actions. A sample rule is shown below.

```
(p compute-ratio
     (goal ↑name compute-ratio)
     { <grid> (grid ↑b <b> ↑d <d> ↑bd-ratio nil)}
     -->
     (modify <grid> ↑bd-ratio (compute <b> // <d>)))
```

The name of the above rule is **compute-ratio**. The LHS contains two condition elements, one is a goal and the other is a grid. The RHS modifies the grid element using the compute function. There are several RHS actions available to the user. The three actions that affect working memory are: *make* a new working memory element, *modify* an existing element, and *remove* an element. There are other RHS actions such as *bind* a value to a variable, *write* a message to the terminal, and *compute* an arithmetic expression.

A goal oriented approach to the solution of the floor system design problem was implemented by selecting the MEA strategy for conflict resolution. The first condition element of a production rule is

typically a goal. The order in which the goals are completed is controlled by the recency of the creation or modification of the goals. There is one rule that controls the top level order of tasks, this is given below in OPS5 syntax:

```
(p beg1n-des1gn
     (goal tnaroe start-design)
     —>
     (make goal tname select-support)
     (make goal tname select-material)
     (make goal tname select-action))
```

The goals are made in reverse of the order in which they are to be executed because the last goal created will have the most recent time tag.

The information for each level of synthesis is represented in working memory elements with *vector attributes;* attributes with a value list. This information is placed in working memory with the use of make statements:

```
(make action ttypes one-way two-way)
(make material ttypes steel-deck prefab-panels reinf-slab
                      prestr-slab)
(make support ttypes 0 2 4)
```

The synthesis of a configuration is similar for each level of information. The following rule illustrates the selection of one-way or two way action.

```
(p select-action
   (goal tname select-action)
   { <act1on> (action ttypes <f1rst> <> nil) }
   -->
   (bind <n> (Htval types))
   (bind <n> (compute <n> + 1))
   (modify <act1on> ttypes (substr <act1on> <n> 1nf))
   (make floor-sys taction <f1rst>)
   (make goal tname check-action))
```

This rule will match if there is a **goal** element in working memory with the **name** attribute **of select-action** and **the types** attribute **of** the **action** element is not equal to nil. The first three RHS actions result in the modification of the working memory, action, so that the vector attribute **types** contains the current value list without the first element. The next RHS action places a floor system alternative in working memory with an action attribute. The last RHS action makes a goal to check the selected action with the constraints at this level. The other rules for the synthesis of the floor system are given below.

```
(p select-material
   (goal ↑name select-material)
   { <mat> (material ↑types <first> <> nil) }
   { <floor> (floor-sys ↑action <> nil ↑material nil) }
   -->
   (bind <n> (litval types))
   (bind <n> (compute <n> + 1))
   (modify <mat> ↑types (substr <mat> <n> inf))
   (modify <floor> ↑material <first>)
   (make goal ↑name check-material))

(p select-support
   (goal ↑name select-support)
   { <sup> (support ↑types <first> <> nil) }
   { <floor> (floor-sys ↑action <> nil ↑material <> nil
                        ↑support nil) }
   -->
   (bind <n> (litval types))
   (bind <n> (compute <n> + 1))
   (modify <sup> ↑s types (substr <sup> <n> inf))
   (modify <floor> ↑support <first>)
   (make goal ↑name check-support))
```

The constraints are represented as elimination constraints; if one of the rules matches then the associated constraint is not satisfied and working memory is modified. The following rules represent the constraints at the action level.

```
(p check-action::two-way
  { <goal> (goal ↑name check-action) }
  (grid  ↑bd-ratio > 1.5)
  { <floor> (floor-sys ↑action two-way) }
  -->
  (remove <floor>)
  (remove <goal>))

(p check-action::one-way
  { <goal> (goal ↑name check-action) }
  (grid ↑bd-ratio < 1.2)
  { <floor> (floor-sys ↑action one-way) }
  -->
  (remove <floor>)
  (remove <goal>))
```

Sample constraint rules from the other two levels of synthesis are given below. If one of these rules matches, the floor system alternative is modified so that the most recent selection is removed.

```
(p check-material::st-deck
   { <goal> (goal tname check-material) }
   { <floor> (floor-sys tmateHal steel-deck) }
   (grid td > 12)
   -->
   (modify <floor> tmaterial nil)
   (remove <goal>))

(p check-support::0-edges
   { <goal> (goal tname check-support) }
   { <floor> (floor-sys tmaterial steel-deck tsupport 0) }
   -->
   (modify <floor> tsupport nil)
   (remove <goal>))
```

The implementation described above finds one feasible floor system configuration. The result is a working memory element named floor-sys that has attributes action, material, and support. The select rules could be expanded to find all feasible floor system configurations but these rules would be more complex. The OPS5 program was run using a grid element shown below:

```
(grid tb 30.0 td 13.0)
```

The result was a working memory element as shown below:

```
(floor-sys taction one-way tmateHal prefab-panels tsupport 2)
```

The use of a production system representation for a design problem has several advantages. The domain dependent knowledge is represented in chunks called rules. After the individual is comfortable with OPS5 syntax these rules can be easily understood and modified. The use of the MEA strategy for conflict resolution forces a goal oriented approach to the solution. The advantage to this approach is that it results in organizing the production rules so that it is easy to determine what aspect of design a rule addresses.

There are two major disadvantages to the use of OPS5 for engineering applications. The first arises from the restriction that complex computations must be done by externally defined functions. External functions have only limited access to working memory and the process required for changing working memory is cumbersome. The second disadvantage is that a production rule can typically encode a very small chunk of knowledge, resulting in a large number of rules to do a relatively simple task.

8. FRAME REPRESENTATION LANGUAGES - SRL

A number of languages have been developed that fall under the category of frame-based representation languages. A partial list of such languages is Units, RLL, SRL, KRL, and most recently KEE (more information about these languages is given in Figure 2.). A frame-based representation language provides facilities for developing a data structure particularly suited for the representation of symbolic knowledge. There is a basic data element, referred to as a frame by Minsky [20], that may be assigned a name and several attributes. These data elements may be linked together in order to share information; this is typically referred to as inheritance. Procedural information may also be associated with a data element, as well as information concerning *when* the procedural information is relevant. The languages vary in the details of the construction and use of this basic data element.

SRL [38], Schema Representation Language, is a language whose basic data element is called a schema. A schema has a *name,* a number of *slots,* and each slot may have a number of *facets.* A slot in a schema may simply be an attribute or it may be a relation. A relation slot is used to link two schemas together. SRL provides two basic relations and allows the user to define his own relations and their associated inheritance specifications. The inheritance specifications define how much information is allowed to be inherited from one schema to another. A facet is an attribute of a slot and typically contains meta-information about the slot. There are some standard facets provided by SRL and the user may define his own facets. The use of the facets provided by SRL include range checking, general book-keeping, and procedural attachment. The facet used for procedural attachment is called a demon. A demon allows the user to associate a LISP function with a slot and specify when this function is to be evaluated.

The knowledge-base in a KBES implemented in SRL contains schemas and LISP functions. The schemas serve as templates for the design solution. The LISP functions provide procedural knowledge about how the schema templates are to be instantiated for a particular problem. The templates define the names and attributes of the elements in the design solution as well as information about the relationships between these elements.

The floor system design solution is represented by a hierarchy of schemas linked by relations that allow all slots and slot values to be inherited. There are four templates that represent the different levels of information for the floor system design problem, as shown below.

```
{ "grid"
    "b" :
    "d" :
    "bd-rat1o" :
           "demon" : "compute-bd-rat1o" }

{ "floor-action"
    "1s-alt" :
      "range": (type 1s-a "grid")
    "fIoor-act1on-descr1pt1on" :
      "range" : (or one-way two-way)
    "constra1ned-by" :
      "range" : (or "one-way-constraint" "two-way-constraint")
    "f!oor-act1on-alt" : floor-material
    "constraint-status" :
      "range": (or t nil)
 }

{"floor-material"
   "1s-alt" :
     "range" : (type 1s-a "floor-action")
   "fIoor-mater1al-descr1pt1on" :
     "range" : (or steel-deck prefab-panels
                   re1nf-slab prestr-slab)
    "costra1ned-by" :
     "range" : (or "steel-deck-constraint" "prefab-panels-constraint*'
               "re1nf-slab-constra1nt" "prestr-slab-constra1nt")
    "fior-mateMal-alt" : floor-support
    "cbstraint-status" :
     "range" : (or t nil)
}

{ "support-floor[11]
    "1s-alt" :
    "floor-support-descr1pt1on" :
       "range" : (or 0-edges 2-edges 4-edges)
    "constra1nted-by" :
       "range" : (or "0-edges-constra1nt" "2-edges-constra1nt"
                     "4-edges-constraint[11]")
    "floor-support-alt" : nil
    "constraint-status[19]" :
       "range": (or t nil)
}
```

In the above templates, the "is-alt" slot is the relational link between schemas representing alternative configurations. This slot is filled as alternative configurations are generated. The "range" facet is used to define the possible values for the associated slot. The "constrained-by" slot denotes the constraint schema applicable to the current instance of the template. For example, if the "floor-action-description[>f] has the value *one-way* then the applicable constraint would be an instance of "one-way-constraint". If the constraint schema, from which the value of

"constraint-status" is determined, evaluates to *nil* then the alternative configuration is not valid and the next alternative is tried. The **"floor-action-alt"** provides the template for the descendant schema.

The constraint schemas represent valid combinations of selections at each level of synthesis. A sample constraint schema is shown below:

```
{ "one-way-constraint"
     "constrains"* :
     "no-constraints" : 1
     "cond1t1on-l" : (lambda (x) ((valuegl x "bd-rat1o") > 1.2))
}
```

The "constrains" slot is a relational link to the alternative being considered. This relation allows the constraint schema to inherit slot value from the schema to which it applies. The "no-constraints" determine the number of conditions that have to be satisfied for the constraint to be true. The predicates that have to be satisfied are placed in the condition slots in the schema.

The inference mechanism is provided by user defined LISP functions. This implementation required three LISP functions. One function instantiates the schema templates, using the range facet, in a depth first search for valid alternative configurations. A second function checks the appropriate schemas by evaluating the condition slots in the constraint schema. A third function, evaluated through the bd-ratio demon, computes the b/d ratio to be stored in the grid schema.

The context representing the design solution takes the form of a tree. The schemas in the context tree are instances of the schema templates in the knowledge-base. The context tree given in Figure 6 shows the solution for a 30.0 by 13.0 grid. There are three possible floor system configurations, each described by a single path through the tree.

The use of SRL for a design problem has the major advantage of the high level constructs for the representation of the design solution. The structure of this representation is defined by the schema templates in the knowledge base. The LISP functions that manipulate this data structure are simplified by the inheritance and range checking facilities provided by SRL. SRL and other frame-based languages facilitate *object-oriented* programming, which provides a number of constructs for the development of KBES for design (see [3] for a discussion on the use of object-oriented programming for engineering design).

9. LOGIC PROGRAMMING - PROLOG

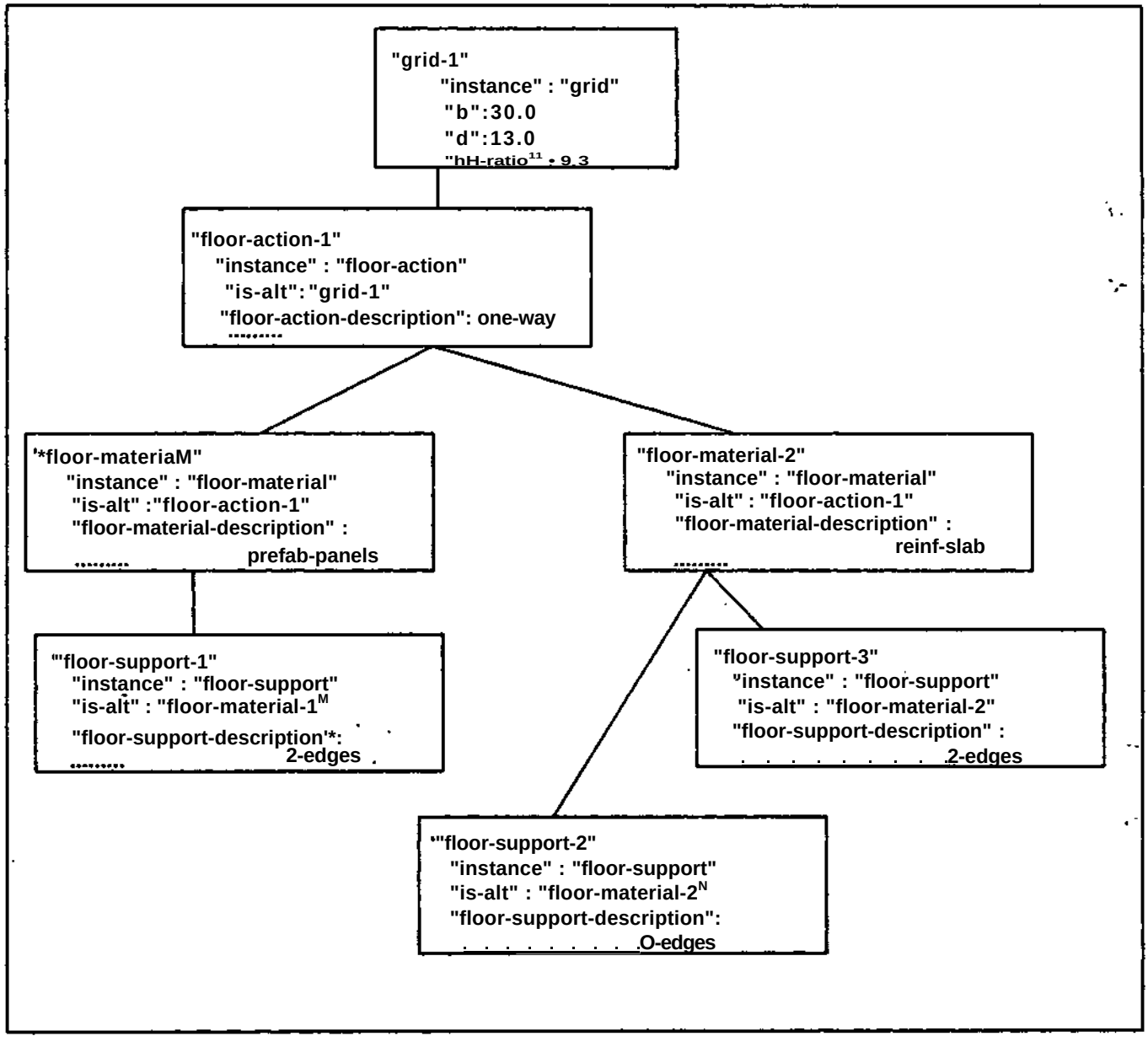

Figure 6: Context Tree In SRL

Logic programming is gaining increasing acceptance among AI researchers because of its elegance, simplicity, and sound mathematical basis. First order predicate calculus can be used to represent a wide range of real world facts. The language of predicate calculus consists of a number of components, such as *predicate symbols, variable symbols, function symbols,* and *constant symbols* [23]. For example! the fact

The material of slab 1s steel-deck

can be represented by the following atomic formula or wff (in logic the phrase *well formed formula* (wff) is used to denote a legitimate expression):

nater1a1-of(slab$_9$ steel-deck)

where the predicate symbol material-of consists of the constants slab and steel-deck. Atomic formulas are combined by using connectives, such as A (and), \/ (or) and --> (implication), to form more complex wffs (atomic formulae and rules are also referred to as *clauses).* For example, the statements

```
IF      The material of slab 1s steel-deck, and
        The slab action 1s one-way
THEN    The support condition for slab 1s 2 edges
```

can be translated into:

```
•ater1al-of(slab,steel-deck)  A  act1on-type(s1ab,  one-way)
     —>  support-type(slab,  2)
```

The truth value of the implication a --> b is equivalent to ~ a \/ b. This transformation is useful for theorem proving. The *resolution* principle, which is a rule of inference for determining particular facts from other known facts, provides an automatic way of proving theorems from axioms. Using this principle one can establish the fact b from a data base of facts a and a --> b. The *unification* principle is used to match variables in predicate formulae. For example, if there is a fact support-type(slab,2) and a goal support-type(slab, X), where X is a variable, then the application of the *unification* principle matches X to 2. For a more detailed review of these principles see [15,23].

The resolution principle can be effectively used in a large number of theorem proving tasks, if the clause representation of the facts and rules can be recast into Horn-clauses. A Horn-clause consists of a set of literals with at most one unnegated literal. For example, the Horn-clause representation of the above rule would be

```
~«ater1al-of(s1ab,steel-deck) \/ ~act1on-type(slab$_9$ one-way)
                    \/ support-type(s1ab,2)
```

which is formally represented as:

```
(-material-of(slab,steel-deck)  ~act1on-type(slab,  one-way)
support-type(slab,2))
```

PROLOG was developed as a computer programming language to manipulate *objects* and *relations.* PROLOG can also be viewed as a logic based programming language since it is based on the idea of a *theorem prover.* Theorem proving in PROLOG involves recasting the rules in the database in the form of Horn-clauses and applying resolution and unification on these clauses. There are some built-in predicates in PROLOG which cannot be expressed in predicate calculus. However, the use of these predicates can be restricted to a small set of clauses, thus retaining the elegance of logic programming.

The knowledge-base or database in a PROLOG program consists of sets of rules representing the relationship between objects. These rules are of the form:

$$R_o \leftarrow R_1 \ \& \ R_2 \ \& \ R_3 \ .. \ R_n, \text{ where } n \geq 0$$

Each Rj in the above rule is called a *term.* Each term can have any number of arguments, which may be *atomic constants, variables,* or *Jerms.* In C-PROLOG, the UNIX[3] version, constants begin with a lower case letter, while variables begin with a upper case letter. When n = 0, the above rule becomes a *fact.* Alternatively, a PROLOG knowledge-base (database) can be viewed as a collection of *facts* and *rules* about objects and relationships.

The inference engine in PROLOG is essentially a theorem prover, which tries to prove the goal (LHS of a rule (R_Q)) by proving each of the subgoals (conditions of RHS) starting from the leftmost subgoal in a *depth first* manner. Hence, the *depth first* strategy is built into the control mechanism. However, other problem solving strategies can be easily progammed. If any of the subgoals are not satisfied for a particular binding of variables, then the system *backtracks* and the program continues with a new set of variable bindings. These variable bindings are available in the database; the variable bindings can also be provided through some user defined functions.

The use of PROLOG for the design example is illustrated by representing most of the constraint knowledge in a factbase and the selection rules for the action type, material type, and support type in a rule base, as shown below.

---

[3] a trademark of BeN labs

FACTBASE:

```
act1on_type(one_way,1.2,20.0).
act1on_type(two_way,1.0,1.6).
mater 1a1__type(stee1_deck, 0,12).
mater1al_type(prefab_panels,0,16).
mater1al~type(re1nf_slab,10,26).
mater1al_type(prestr_slab,20,40).
mater1al_act1on(one_way,steel_deck).
mater1al_action(one_way,prefab_panels).
mater1al_act1on(one_*ay,re1nf_slab),
mater1al_act1on(one_way,prestr_slab).
mater1a1_act1on(two_way,re1nf_slab),
mater1al_act1on(two_way,prestr_slab).
support_type(re1nf_s1ab,0).
support_type(re1nf_slab,2).
support_type(re1nf_slab_t4).
support_type(steel_deck_#2).
support_type(prefab_pane1s,2).
support_type(prestr_slab,0).
support_type(prestr_slab,2).
support_type(prestr~slab,4).
support_allowable(one_way_t0).
support_allowable(one_way,2).
support_allowable(two_way,0).
support_allowable(two_way,2).
support_a11owable(two_way,4).
```

RULEBA5E:

```
slab_parameter(B,D,Action,Material^Support) :-
          slab_action(B,D,Action),
          slab_span(Action,Span,B,D),
          $lab_material(Action,Span,Material),
          slab_support(Act1on,Material.Support).


slab_act1on(B,D,Action) :-
          Ratio 1s B / D_t
          action_type(Act1on,X,Y),
          Ratio > X,
          Ratio -< Y.


slab_mater1al(Act1on,Span_fX) :-
         •aterial.typeCX.Y.Z),
         Span >- Y,
         Span -< Z_f
         nater1al_act1on(Act1on_fX).
```

```
slab_support(Action,Material,Support) :-
            support_type(Material,Support),
            support_allowable(Action,Support).


slab_span(one_way, D, B, D).

slab_span(two_way, B, B, D).
```

Once the factbase and the rulebase are defined, the parameters for the slab are selected by asking PROLOG the question:

```
?- slab_parameter(30.0, 13.0, Action, Material, Support).
```

The flow of control in PROLOG to answer the above question is described below.

The factbase is searched to satisfy the goal **slab_action**. Since the **b/d** ratio is greater than **1.2**, the goal is satisfied by the fact **action_type(one_way, 1.2, 20.0)**. Since the action of the slab is **one_way**, the **Span** is evaluated to be **13**, through **slab_span(one_way, 13.0, 30.0, 13.0)**. The **slab_material** goal is satisfied by the facts **material_action (one_way, prefab_panels)** and **material__type(prefab__panels, 0, 15)**. The support type is determined by the fact **support_type(prefab_panels,2)**, which implies that the **prefab_panels** can have 2 supports, and the constraint fact **support_allowable(one_way,2)**. Hence the values of the variables **Action, Material**, and **Support** are instantiated to **one_way, prefab_panels**, and **2**. The user can continue with the selection by typing a semicolon after the answer is returned. The system will provide other feasible systems: a **reinf_slab** supported on **0** or **2** edges, which correspond to the right half of the tree shown in Figure 6 for the SRL paradigm.

From the above example, one can see that PROLOG can be used for the synthesis part of engineering design. However, its interface to other engineering software is not clearly defined. Since PROLOG searches the database starting from the first fact it may be very slow for problems having very large databases (factbases); some implementations provide novel hashing schemes for faster access to data. With the increased interest being shown in the use of this language by the fifth generation computing researchers, a number of additional features may be added to make PROLOG a valuable tool for engineers.

10. CONCLUSION

The development of a KBES for engineering design requires the identification of the portion of the design process the system will address. Once the problem has been identified, a solution strategy (or combination of strategies) must be formulated. The selection of a tool to aid in the development depends on the nature of the design problem and the intended solution strategy.

Three languages, OPS5, SRL, and PROLOG, were discussed by implementing an engineering design paradigm. A simple problem was chosen to illustrate various features of these languages; the problem itself could have been easily coded in FORTRAN or any other procedural language. The three languages were chosen to illustrate different representation formalisms - rules, frames, and logic - commonly used in the development of expert systems. While OPS5 and PROLOG provide the user with a primitive inference mechanism, SRL requires the user to write the inference mechanism in LISP; it was easier to implememnt the example in OPS5 and PROLOG than in SRL. However, the inheritence mechanism provided in SRL is very useful for implementing the hierarchical design process.

Full-scale practical implementations of engineering KBES will be greatly facilitated by higher-level expert system development tools, notably in the areas of user interfaces, explanation capabilities and knowledge-acquisition capabilities. Furthermore, practical implementations may also require a combination of the languages described, and a close coupling to databases and algorithmic application programs. Languages such as PSRL and LOOPS offer the advantage of combining rule-based and frame-based programming paradigms. An important consideration in the selection of a tool for engineering design problems is that the tool should provide adequate interfaces to algorithmic programs, such as a finite element program. From our experience with various operating systems and tools, we feel that the UNIX-like operating system with a PSRL-type language offers an adequate environment for the development of KBES for engineering design problems.

11. ACKNOWLEDGMENTS

The help rendered by Vijay Saraswat in making this document more readable and his assistance with PROLOG programming is greatly appreciated. We would also like to thank Elaine Kant and Mike Rychener for providing a number of useful hints on programming in OPS5 and SRL.

12. BIBLIOGRAPHY

[I] A mar el, S., "Basic Themes and Problems in Current Al Research,[11] *Proceedings of the Fourth Annual AIM Workshop,* Ceilsielske, V. B., Ed., Rutgers University, pp. 28-46, June 1978.

[2] Balzer, R, Erman, L D. London, P. and Williams, C, "Hearsay-Ill: A Domain Independent Framework for Expert Systems,[11] *Proceedings of the First Annual National Conference on Artificial Intelligence.,* pp. 108-110,1980.

[3] Barbuceanu, M., "Object-Centered Representation and Reasoning: An Application to Computer-Aided Design,[11] *SIGART Newsletter,* pp. 33-39, January 1984.

[4] Bobrow, D. G. and Raphael, B., "New Programming Languages for Al Research," *Computing Surveys,* Vol. 6, No. 3, pp. 153-174,1974.

[5] Bobrow, D. and Stefik, M., *The Loops Manual,* Xerox Corporation, 1983.

[6] Brownston, L. et. al., Personal Communication, 1984.

[7] Clocksin, W. F. and Mellish, C. S., *Programming in Prolog,* Springer-Verlag, Berlin Heidelberg New York, 1981.

[8] Doyle, J., *Truth Maintenance Systems for Problem Solving,* Technical Report AI-TR-419, MIT, 1978, [Master's Thesis].

[9] Dym, C. L., "Expert Sysptems: New Approaches to Computer-Aided Engineering," *Proceedings Twenty-fifth AIAA-ASME-ASCE-AHS Structures,* May 1984.

[10] Fain, J., Gorlin, D., Hayes-Roth, F., Rosenschein, S. J., Sowizral, H., and Waterman, D., *The ROSIE Reference Manual,* Technical Report N-1647-ARPA, Rand Corporation, Santa Monica; California 90406., 1981.

[Il] Forgy, C. L., *OPS5 User's Manual,* Technical Report CMU-CS-81-135, Carnegie-Mellon University, July 1981.

[12] Greiner, R. and Lenat, D. B., "A Representation Language Language," *Proceedings of the First Annual National Conference on Artificial Intelligence,* pp. 165-168,1980.

[13] Hayes-Roth, F., Waterman, D. A., and Lenat, D.. B., Eds., *Building Expert Systems,* Addison-Wesley Publishing Company, Inc., 1983.

[14] *KEE: Knowledge Engineering Environment,* IntelliGentics, Inc., California, 1984.

[15] Kowalski, R. A., *Logic for Problem Solving,* North Holland Elsevier, 1979.

[16] Lin, T. Y. and Stotesbury, S. D., *Strucutral Concepts and Systems for Architects and Engineers,* John Wiley & Sons, 1981.

[17] Mackworth, A. K., "Consistency in Networks of Relations," *Artificial Intelligence,* Vol.8, pp. 99-118,1977.

[18] Maher, M. L., *HI-RISE: An Expert System For The Preliminary Structural Design Of High Rise Buildings,* 1984, Forthcoming Ph. D. thesis, Department of Civil Engineering, Carnegie-Mellon University.

[19] McDermott, J., *R1 : A Rule-Based Configurer of Computer Systems,* Technical Report CMU-CS-80-119, Carnegie-Mellon University, 1980.

[20] Minsky, M., "A Framework for Representing Knowledge," in *Psychology of Computer Vision,* Winston, P., Ed., McGraw Hill Book Company, 1975.

[21] Nau, D. S., "Expert Computer Systems,[11] *Computer,* Vol. 16, pp. 63-85, February 1983.

[22] Nii, H. P. and Aiello, N., "AGE (Attempt to Generalize) : A Knowledge-Based Program for Building Knowledge-Based Programs.," *Proceedings Sixth IJCAI,* pp. 645-655,1979.

[23] Nillson, N. J., *Principles of Artificial Intelligence,* Tioga Publishing Company, Palo Alto, California, 1980.

[24] Preiss, K., "Data Frame Model for the Engineering Design Process," *Design Studies,* Vol. 1, No. 4, pp. 231 -243,1980, [IPC Bussiness Press].

[25] Rasdorf, W. J., *Structure and Integrity of a Structural Engineering Design Database,* unpublished Ph.D. Dissertation, Department of Civil Engineering, Carnegie-Mellon University, April 1982.

[26] Reboh, Rene[1], *Knowledge Engineering Techniques and Tools in the Prospector Environment,* Technical Report 8172, SRI International, June 1981.

[27] Reggia, J. A. and Perricone, B. T., *KMS Manual,* Department of Mathematics, University of Maryland, January 1982.

[28] Rich, E., *Artificial Intelligence,* McGraw Hill, 1983.

[29] Sriram, D., Maher, M., Bielak, J. and Fenves, S., *Expert Systems for Civil Engineering - A Survey,* Technical Report R - 82 - 137, Department of Civil Engineering, Carnegie-Mellon University, July 1982.

[30] Sriram, D., Maher, M. L. and Fenves, S. J., *Knowledge-based Expert Systems in Structural Design,* 1984, To be presented at NASA conference on Advances in Structural Mechanics, October 1984, Washington, D. C.

[31] Sriram, D., *A Bibliography on Knowledge-Based Expert Systems in Engineering,* 1984, July, 1984, SIGART newsletter.

[32] Stallman, Ft., and Sussman, G. J., "Forward Reasoning and Dependency-directed Backtracking in a System for Computer-Aided Circuit Analysis," *Artificial Intelligence,* Vol. 9, pp. 135-196,1977.

[33] Stefik, M. and Martin, N., *A Review of Knowledge Based Problem Solving as a Basis for a Genetics Experiment Designing System,* Technical Report STAN-CS-77-596, Computer Science Department, Stanford University, March 1977.

[34] Stefik, M., *Planning With Constraints,* Technical Report STAN-CS-80-784, Computer Science Department, Stanford University, January 1980.

[35] van Melle, W., "A Domain Independent Production-Rule System for Consultation Programs," *Proceedings Sixth IJCAI,* pp. 923-925, August 1979.

[36] Weiss, S. M. and Kulikowski, C. A., "EXPERT: A System for Developing Consultation Models,[11] *Proceedings Sixth IJCAI,* pp. 942-947,1979.

[37] Winston, P. H. and Horn, B. K. P., *LISP,* Addison-Wesley Publishing Company, Massachusetts, 1981.

[38] Wright, J. M. and Fox, M. S., *SRL/1.5 User Manual,* Technical Report, CMU Robotics Institute, June 1983.