1985

PLANPERT - an expert system for administrative planning

Dershya Song

PLANPERT - AN EXPERT SYSTEM FOR ADMINISTRATIVE PLANNING

Dershya Song

Rochester Institute of Technology
School of Computer Science and Technology

PLANPERT -
An Expert System for Administrative Planning

by
Dershya Song

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirement for the degree of
Master of Science in Computer Science

Approved by :

Professor John A. Biles

Professor Lawrence A. Coon

Professor Michael J. Lutz

June 7, 1985

Title of Thesis: PlanPert – An Expert System for Administrative Planning

I Dershya Song hereby (grant/deny) permission to the Wallace Memorial Library, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

or

I Dershya Song prefer to be contacted each time a request for reproduction is made. I can be reached at the following address:

_____

_____

Date: June 10, 85

1. INTRODUCTION AND BACKGROUND

1.1 What Is An Expert System?

There are lots of tasks that require a great deal of specialized knowledge that most people do not posses. These tasks can only be performed by experts who have accumulated the required knowledge. Examples of such tasks include medical diagnosis, electronic design, and scientific analysis. Computer programs to perform these tasks would be very useful since there is usually a shortage of qualified human experts. Such programs are called expert systems. An expert system, then, is a computing system that embodies organized knowledge concerning some specific area of human expertise at a level sufficient to perform as a skilled and cost-effective consultant. Thus an expert system is a high-performance special-purpose system designed to capture the skill of an expert consultant such as a doctor of medicine, a chemist or a mechanical engineer.

The most important characteristic of an expert system is that it relies on a large data base of knowledge. It reasons with judgemental knowledge as well as with formal knowledge of established theories, because human experts generally possess private knowledge that has not found its way into the published literature. This private knowledge consists largely of rules of thumb that have come to be called heuristics. Thus a large part of what an expert system needs to know is the body of heuristics that specialists use in solving hard problems. An expert system

provides explanations of its line of reasoning and answers to queries about its knowledge. It also integrates new knowledge incrementally into its existing store of knowledge.

Expert systems are often regarded as representing a subclass of artificial intelligence, but they differ from the broad class of AI tasks in several respects. First, they perform difficult tasks at expert levels of performance. Second, they emphasize domain-specific problem-solving strategies over the more general methods of AI. Third, they employ self-knowledge to reason about their own inference processes and provide explanations or justifications for conclusions reached. Finally, they solve problems that generally fall into one of the following categories: interpretation, prediction, diagnosis, debugging, design, planning, monitoring, repair, instruction, and control. As a result of these distinctions, expert systems represent an area of AI research that involves paradigms, tools, and system development strategies. (Hayes-Roth, Waterman,& Lenat, 1983)

1.2 History

The first expert systems, DENDRAL (Lindsay er al. 1980) and MACSYMA (Moses, 1971), emphasized performance, the former in organic chemistry and the latter in symbolic integration. These systems were built in the mid-1960s and were nearly unique in AI because of their treatment real-world problems with specialized knowledge.

The DENDRAL project at Stanford recently entered its sixteenth year. This project produced two systems, DENDRAL and METADENDRAL (Buchanan and Mitchell 1977, and Lederberg 1971; Lindsay et al. 1980). DENDRAL analyzes mass spectrographic, nuclear magnetic resonance, and other chemical experiment data to infer plausible structures of an unknown compound. DENDRAL employs an efficient variant of the generate-and-test strategy in its problem-solving. Its generator can enumerate every possible organic structure that satisfies the constraints apparent in the data by systematically generating partial molecular structures consistent with the data and then elaborating them in all plausible ways. By rapidly eliminating inplausible substructures, it avoids an otherwise exponential search. By systematically generating all plausible structures, it finds even those candidates that human experts occasionally overlook. It surpasses all humans at its task and, as a consequence, has caused a redefinition of the roles of humans and machines in chemical research.

The MACSYMA system (Martin and Fateman 1971), was developed at the Massachusetts Institute of Technology. Like DENDRAL, MACSYMA surpasses most human experts. It performs differential and integral calculus symbolically and excels at simplifying symbolic expressions. Used daily by mathematical researchers and physicists worldwide, MACSYMA incorporates hundreds of rules garnered from experts in applied mathematics. Each rule expresses one way to transform an expression into an equivalent; the solution to a problem requires finding a chain of rules that

transforms the original expression into one that is suitably simplified.

In the 1970s, work on expert systems began to flower, especially in medical problem areas. Examples are INTERNIST (Pople 1977), MYCIN (Shortliffe 1976), and CASENET (Weiss at el. 1979). The issues of making the system understandable through explanations and of making the system flexible enough to acquire new knowledge were emphasized in these and later systems. In early work, the process of constructing each new system was tedious because each was custom crafted. The major difficulty was acquiring the requisite knowledge from experts and reworking it into a form fit for machine consumption, a process that has come to be known as knowledge engineering. One of the most important developments of the late 1970s and early 1980s is the construction of several knowledge engineering frameworks designed to aid in building, debugging, interpreting, and explaining expert systems. Engineering an expert's knowledge into a usable form for a program is a formidable task. Thus, the computer-based aids for system builders are important. Current tools, including EMYCIN (vanMelle, 1980), ROSIE (Fain et al. 1981), EXPERT (Weiss and Kulikowski, 1979) and OPS (Forgy and McDermott, 1977), provide considerable help.

EXPERT is an expert-system-building language that evolved from CASNET, an expert system for consultation in the diagnosis and treatment of glaucoma. EXPERT has been used primarily for building consultation models in ophthalmology, endocrinology, and

rheumatology. EMYCIN is a domain-independent version of MYCIN. It contains all of MYCIN except its knowledge of infectious blood diseases. EMYCIN facilitated the development of related diagnostic applications, such as PUFF (Freiherr 1980). ROSIE, developed at Rand, provides a general-purpose programming system for building expert systems. ROSIE evolved from an earlier programming system named RITA, for Rand Intelligent Terminal agent, both deriving their motivation from the success of MYCIN's rule-oriented explanation facility has for users. ROSIE is the first system designed to support a wide class of new expert system applications. OPS is a production system language developed at Carnegie-Mellon University. The most successful application of it is the R1 expert system which configures DEC VAX computer systems.

1.3 Knowledge Engineering

In an expert system, the fundamental assumption is, "knowledge is power". What is knowledge in a specific domain? Knowledge consists of descriptions, relationships, and procedures in some domain of interest. The symbolic descriptions characterize the definitional and empirical relationships in a domain, and the procedures manipulate these descriptions. As electrical engineering is to electricity, knowledge engineering is the technology that promises to make knowledge a valuable industrial commodity.

Because the power of the expert system does not come principally from the knowledge application mechanism, called the "inference engine", but from the richness, pertinence and redundancy of the knowledge itself, expert systems are sometimes referred to as knowledge-based systems. Knowledge-based systems differ from algorithm-based systems which are the conventional program in that algorithm-based systems are deterministic and possess no redundancy. For any given input in an algorithm-based system there is a single computational path that is always followed, and there is a single mechanism capable of producing the correct output for that input. Knowledge-based systems that mimic human beings, however, are often equipped with big amount of overlapping techniques for handling a problem, so that forgetting one technique does not prevent a solution from being found by other means. Again, an algorithm-based system usually exhibits a sharp distinction between code and data - between the recipes for manipulating structures and the structures themselves. Programs in such systems lack the usual human self-awareness of the techniques being employed and cannot reason about or explain their own mechanisms.

Knowledge-based systems, on the other hand focus on knowledge in the ordinary meaning of the term, ie. facts about the task domain and heuristics or rules of thumb that guide the use of knowledge to solve problems in the domain. Most current knowledge-based systems are divided, not into code and data, but into a corpus of knowledge and a comparatively simple mechanism

for applying the knowledge in an opportunistic way to solve problems. Scientists involved in building these knowledge-based systems have placed considerable importance on expressing the knowledge in a form that is not only usable by the inference engine, but also comprehensible to human beings. These systems can then be understood by digesting the knowledge rather than by tracing the intricate computational paths arising from possible interactions among knowledge elements. In this way it is possible to construct complex systems that are still relatively transparent. When the knowledge elements are also fairly independent, incremental modification and improvement of the system is easy. Finally, if the knowledge is redundant, the absence of one fact or mechanism does not necessarily prevent the system arriving at a result by another route.

Knowledge engineering, a process of building knowledge-based system by way of assembling knowledge, was defined by Edward A. Feigenbaum as the followings:

> The knowledge engineer practices the art of bringing the principles and tools of AI research to bear on difficult applications problems requiring experts' knowledge for their solution. The technical issues of acquiring this knowledge, representing it, and using it appropriately to construct and explain lines of reasoning, are important problems in the design of knowledge-based systems .... The art of constructing intelligent agents is both part of and an extension of the programming art. It is the art of building complex computer programs that represet and reason with knowledge of the world (Feigenbaum 1977).

The technical issues mentioned above are knowledge representation, knowledge acquisition and inference methods. In

knowledge representation the core problem is how the knowledge is to be represented so that it is both usable by the system and comprehensible to human beings. Knowledge acquisition concerns knowledge can be acquired and tested for internal consistency and completeness. The architecture a system should have so the knowledge can be brought to bear on problems in a domain is the central problem for inference methods. Knowledge representation and inference methods will be covered in detail in chapter 2. Knowledge acquisition stages and knowledge of knowledge acquisition will be put off until chapter 4, so that a comparison can be made between methods suggested by other researchers and their application in the PlanPert system.

2. FEATURES OF AN EXPERT SYSTEM

2.1 System Structure

In this section an ideal expert system is presented which is derived from the ideal model in paper (Hayes-Roth's 1983) and most real expert systems. The structure of this ideal expert system is shown in Figure 2-1. No existing expert system contains all the components shown, but one or more components occur in every expert system. Those components can be divided into three categories, the knowledge base, the knowledge manager, and the situation model.

The ideal expert system contains a language processor for problem-oriented communications between the user and the expert system; a situation model which may contain a blackboard or working memory; a knowledge base comprising facts as well as heuristic planning and problem-solving rules; and a knowledge manager using the information contained in the knowledge base to interpret the current contextual data in the situation model.

The language processor mediates information exchanges between the expert system and the human user. Typically the language processor parses and interprets user questions, commands, and volunteered information. Conversely the language processor formats information generated by the system, including answers to questions, explanations and justifications for its behavior, and requests for data.

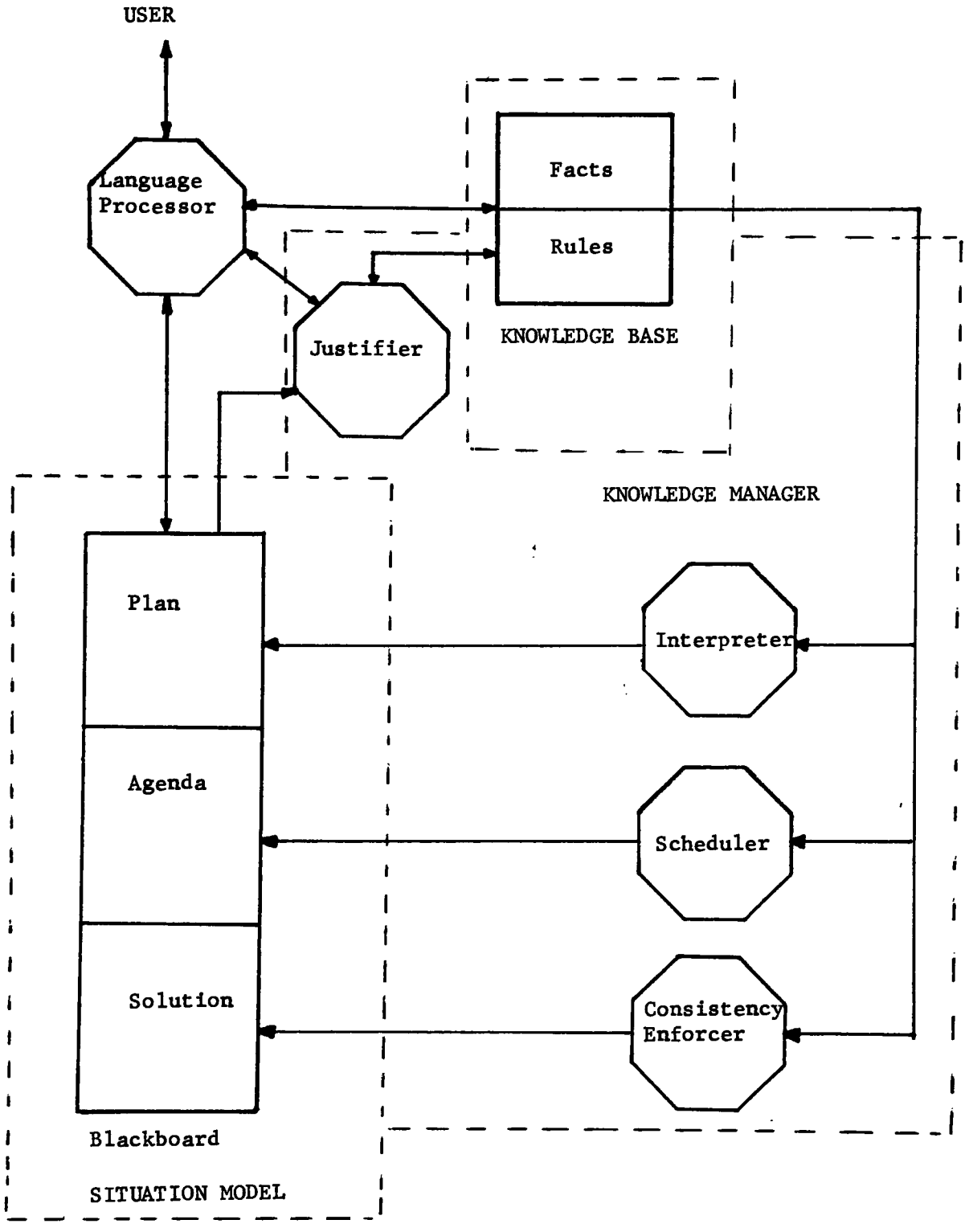


Figure 2-1: Ideal System Structure of An Expert System

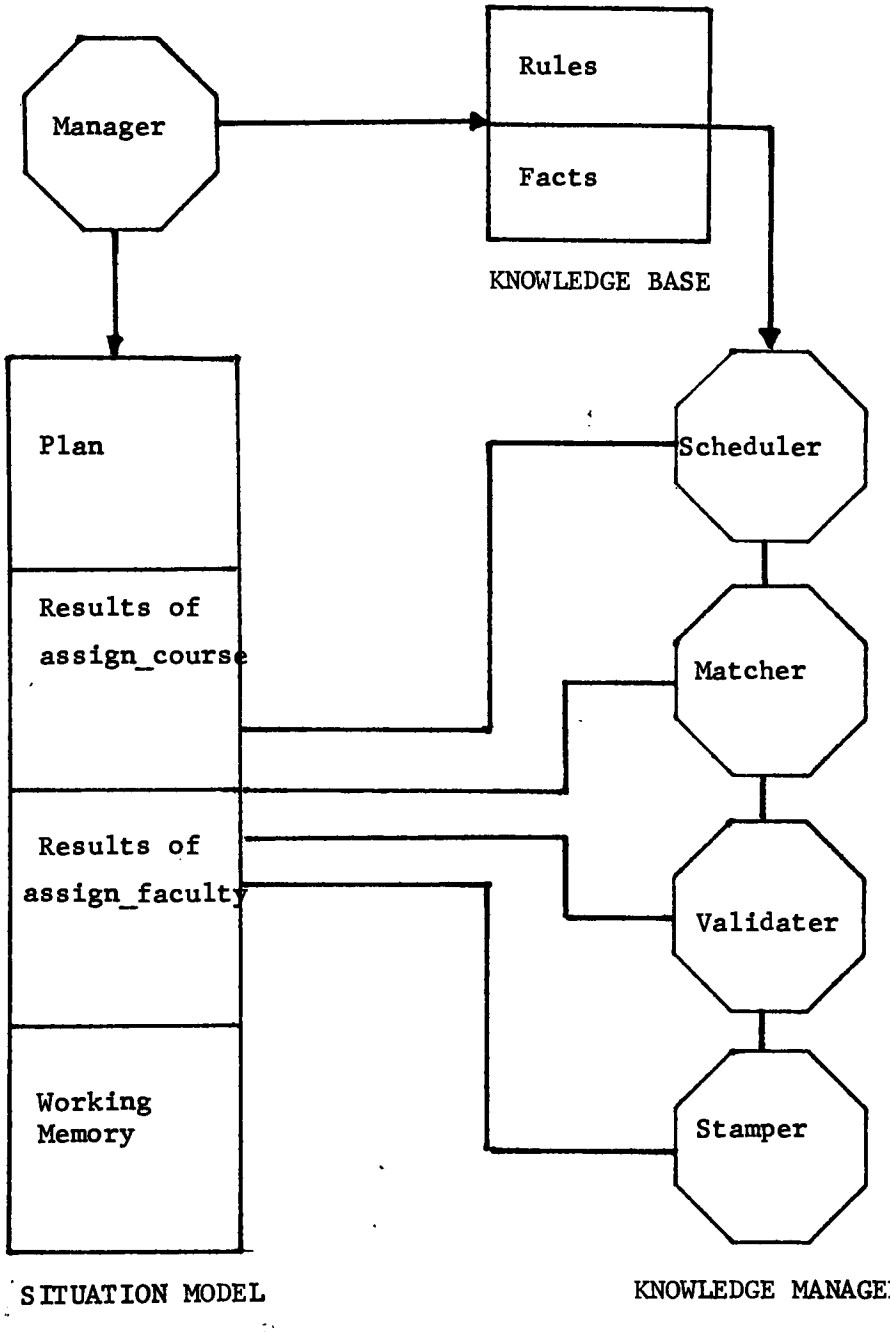


Figure 2-2: Ideal System Structure of PlanPert

The situation model records intermediate hypotheses and decisions that the expert system manipulates. Every expert system uses some type of intermediate decision representation, which may be called working memory, a blackboard or a scratch pad. Fig. 2-1 identifies three types of decisions recorded on the blackboard: plan, agenda, and solution elements. Plan elements describe the overall or general attack the system will pursue against the current problem, including current plans, goals, problem states, and contexts. The agenda elements record the potential actions awaiting execution, which generally correspond to knowledge base rules that seen relevant to some decision placed on the blackboard previously. The solution elements represent the candidate hypotheses and decisions the system has generated thus far, along with the dependencies that relate decisions to one another.

The scheduler maintains control of the agenda and determines which pending action should be executed next. The interpreter executes the chosen agenda item by applying the corresponding knowledge base rule. Generally the interpreter validates the relevance conditions of the rules, binds variables in these conditions to particular solution blackboard elements, and then makes those changes to the blackboard that the rule prescribes. The consistency enforcer attempts to maintain a consistent representation of the emerging solution. The justifier explains the actions of the system to the user. In general, it answers questions about why some conclusion was reached or why some

alternative was rejected.

The knowledge base contains rules, facts, and information about the current problem that may be useful in formulating a solution. The more comprehensive the knowledge base, the less the strain upon the inferential logic inside the knowledge manager when a question has to be answered. This means that the power of the system tends to be defined according to its depth of knowledge rather than its ability to reason.

Figure 2-2 is the ideal system structure of the PlanPert system. The main purpose of this system structure is to help doing the knowledge acquisition more efficiently. The structure consists of a knowledge base, a situation model and a knowledge manager. The knowledge base has two parts, the facts about scheduling domain and the rules the experts use. The knowledge manager supervises the manipulations in the knowledge base and the situation model. It can be divided further into four subtasks which are the scheduler, matcher, validater, and stamper. The scheduler decides which rule to use at a certain time. The matcher scans through the knowledge base to find a proper way to match course, time and faculty. The validater then examines this matching by checking every rule that is associated with it. If the plan made by the matcher is acceptable, the stamper then executes it by updating the scratch schedule in the situation model. The situation model contains the plan, execution of plan, and all intermediate results of any subtasks.

The advantage of adopting this ideal structure is that we can have a systematic and structural way to do the knowledge acquisition. Note that this ideal structure is not the actual structure of PlanPert. It is obvious that there is no such expert who does things structurally or methodically. Why and how, then, is this ideal structure helpful during knowledge acquisition? First of all, it provides an outline and direction for meetings between the expert and the knowledge engineer. The knowledge engineer can easily categorize the information or knowledge collected from the experts. Categorization results in compartmentalization of the knowledge. Some knowledge is the believed facts, some is the applicable rules. There are certain facts and rules associated with certain subtasks. For example, The facts associated with the matcher and stamper are different. The matcher operates on existing facts while the stamper produces new facts. Besides suggesting a good way to start the knowledge acquisition, this ideal structure leads to the completion of knowledge. Like the top-down design of most ideal programming problems, this structure breaks down the knowledge collection into several smaller pieces. Although it has the advantages of simplicity and modularity, it might not be the natural way to view the expert's knowledge, because there is no such clear boundary among the subtasks.

2.2 Knowledge Representation

In order to solve the complex problems encounted in expert systems, one needs both a large amount of knowledge and some

mechanisms for manipulating that knowledge to create solutions to new problems. Because a large amount of knowledge is so critical to the success of an expert system, the question of how that knowledge is to be represented is critical to the design of the system.

A representation is a set of conventions for describing the world. It deals with two different kinds of entities. One is facts which are the truths in some relevant world; these are the things we want to represent. Another is the representations of facts in some chosen formalism; these are the things we will actually be able to manipulate. In order for the representation to be of any interest with respect to the world, there also must be functions that map from facts to representation and from representation back to facts. Usually, these are many-to-many relations rather than functions. Each object in the domain may map to several elements in the range, and several elements in the domain may map to the same element of the range. For example, the sentence 'All dogs have tails' and 'Every dog has a tail' could both represent the same fact.

For all kinds of representation schemes, there are three basic requirements, extensibility, simplicity, and explicitness (Buchanan 1983). The data structures and access programs must be flexible enough to allow extensions to the knowledge base without forcing substantial revisions. The knowledge base will contain heuristics that are built out of an expert's experience. A problem is that experts not only fail to remember all the

relevant heuristics they use, but their experience also gives them new heuristics and forces modification to the old ones. In other words, new cases require new distinctions. Moreover, the most effective way yet found for building a knowledge base is by incremental improvement. Experts cannot define a complete knowledge base all at once for interesting problem areas, but they can define a subset and then refine it over many weeks or months of examining its consequences. All this argues for treating the knowledge base of an expert system as an open-ended set of facts and relations, and keeping the items of knowledge as modular as possible.

The flexibility which was argued for previously requires conceptual simplicity and uniformity so that access routines can be written and modified occasionally as needed. Once the syntax of the knowledge base is fixed, the access routines can be fixed to a large extent. Knowledge acquisition, for example, can take place with the expert insulated from the data structure by access routines that make the knowledge base appear simple whether it is or not. There are two ways of maintaining conceptual simplicity: keeping the form of knowledge as homogeneous as possible and writing special access functions for nonuniform representations. The point of representing much of an expert's knowledge is to give the system a rich enough knowledge base for high-performance problem solving. However, because a knowledge base must be build incrementally, it is necessary to provide means for inspecting and debugging it easily. With items of knowledge represented

explicitly, in relatively simple terms, the experts who are building knowledge bases can determine which items are present and which are absent.

To achieve these goals, three types of representation frameworks have been used in expert systems. They are production systems, first-order logic statements, and frame systems. Such frameworks are often called representation languages because, as with other programming languages, their conventions impose a rigid set of restrictions on how one can express and reason about facts in the world.

Production Systems

A classical production system has three major components: (1) a global data base that contains facts or assertions about the particular problem being solved, (2) a rule base that contains the general knowledge about the problem domain, and (3) a rule interpreter that carries out the problem-solving process. The facts in the global data base can be represented in any convenient formalism, such as arrays, strings of symbols, or list structures. The rules have the form:

IF <condition> THEN <action>.

In general, the left-hand side or condition part of a rule can be any pattern that can be matched against the data base. It is usually allowed to contain variables that might be bound in different ways, depending upon how the match is made. Once a match

is made, the right-hand side or action part of the rule can be executed. In general, the action can be any arbitrary procedure employing the bound variables. In particular, it can result in the addition of new facts to the data base, or modification of old facts in the data base.

The rule interpreter has the task of deciding which rules to apply. It decides how the condition of a selected rule should be matched to the data base and monitors the problem-solving process. When it is used in an interactive program, it can turn to the user and ask for information or facts that might permit the application of a rule. The strategy used by the rule interpreter is called the control strategy. The rule interpreter for a classical production system executes rules in a "recognize-act" cycle. Here the rule interpreter cycles through the condition parts of the rules, looking for one that matches the current data base and executing the associated actions for some or all rules that do match.

Production rules offer a knowledge representation that greatly facilitates the usefulness and maintenance of an evolutionary knowledge base that supports interactive consultation. One view of a production rule is a modular segment of code (Winograd 1975), which is heavily stylized (Waterman 1970). Such modular, stylized coding is an important factor in building a system that achieves a high level of competence. It also has the following advantage: (1) It is a useful way of searching, (2) It is a good way to model the strong data-driven nature of

intelligent action; as the new inputs enter the data base, the behavior of the system change. (3) New rules can easily be added to account for new situations without disturbing the rest of the system. This is important since no A.I. program is truly complete. Stylization and modularity also result in certain shortcomings, however, in that it is harder to express a given piece of knowledge if it must be put into a predetermined format. Another shortcoming in the formalism arises in part from the backward chaining control structure. It is not always easy to map a sequence of desired actions or tests into a set of production rules whose goal-directed invocation will provide that sequence.

First-order predicate logic

The second representational scheme represents knowledge as assertions in logic, usually first-order predicate logic or a variant of it. This mode of representation is normally coupled with an inference procedure based on theorem proving. It allows quantified statements and all other well-formed formulas as assertions. The rigor of logic is an advantage in specifying precisely what is known and in knowing how the knowledge will be used. A disadvantage is the difficulty in dealing with the imprecision and uncertainty of plausible reasoning. Here there is no distinction between rules and facts. Information about the domain and the problem is represented by logical formulae and augmented by rules of inference. This formulation has been further restricted in PROLOG, a widely used approach in this

direction. In PROLOG all knowledge is reduced to a collection of Horn Clauses of the form:

R <-- A1 ^ A2 ^ ... ^Ax

and the problem to a goal clause of the form:

<-- B1 ^ B2 ^ Bi ^ ... ^ By

where R, A and B are atomic formulae. The semantics of a Horn clause can be thought of as being either declarative "if A1,A2,...Ax all hold then so does R" or procedural "If you want R then do A1, then A2,..., then Ax" thus combining the idea of declaiming eternal verities with the sort of 'do it this way' instruction sequences found in more conventional programming languages. The specialized inference rule used to deduce new information is the following. Given the goal clause and Horn Clause above, if there is a substitution S of terms for variables that will make Bi and R identical, then we can derive the new goal clause.

<--(B1 ^ B2 ^ Bi-1 ^ A1 ^ ... ^Ax ^ Bi+1 ^ ... ^By) S

In production systems, computation stops when a solution is found or no rule has a situation satisfied by the current database. A PROLOG program halts when the empty goal clause is generated or no new goal clause can be formed.

Frame Systems

So far, we have presented two mechanisms that can be used to represent specific events or experiences. There exists a great deal of evidence, however that people do not analyze new situations from scratch and then build new knowledge structures to describe those situations. Instead, according to one view, they have available in memory a large collection of structures representing previous experience with objects, locations, situations, and people. To analyze a new experience, they evoke appropriate stored structures and then fill them in with the details of the current event. A general mechanism designed for the computer representation of such common knowledge is the frame. The word frame has been applied to a variety of slot-and-filler representation structures, mostly following the theory presented in (Minsky, 1975) and discussed in (Kuipers, 1975).

The kernel of the idea is the representation of things and events by a collection of frames. Each frame corresponds to one entity and contains a number of labelled slots for things pertinent to that entity. Slots in turn may be blank, or may be specified by terminals referring to other frames, so the collection of frames is linked together into a network. As an illustration, consider the entity 'a cricket match'. A frame for this might contain slots for the names of the teams, descriptions of the players, the dates played, the result of the match, and so on. Some slots may be blank, such as the result in the case of a future match. Some slots may have default specifications by way of appropriate links, and these default links would only be bro-

ken when they contradict further information. The slots delimit the information that is relevant to the entity. There would be no question of trying to find the weight of a cricket match because no such slot would exist. Frames can be used to represent vastly different things such as facts, objects and concepts, or prototype objects('test'). The interconnections among frames allow information to be 'inherited' by one frame from another, for example from the frame describing a prototype to one describing an object covered by that prototype.

2.3 Inference Methods

Although the performance of most expert systems is determined more by the amount and organization of the knowledge possessed than by the inference strategies employed, every expert system needs inference methods to apply its knowledge. The resulting deductions can be strictly logical or merely plausible; rules can be used to support either kind of deduction. Thus a rule such as:

```
Has(x,features) OR (Able(x,fly) & (Able(x,lay-eggs)) -->
Class(x,bird)
```

amounts to a definition, and can be used, together with relevant facts, to deduce logically whether or not an object is a bird. On the other hand, a rule such as

```
State(engine,won't turn over) & State(headlights,dim) -->
State(battery,discharged)
```

is a "rule of thumb" whose conclusion, though plausible, is not

always correct. Clearly, uncertainty is introduced whenever such a judgemental rule is employed. In addition, the conclusions of a logical rule can be uncertain if the facts it employs are uncertain. Both kinds of uncertainty are frequently encountered in expert systems applications. In this section we consider the methods for using rules when everything is certain and ignore the complications introduced by uncertainty.

Besides logical and plausible inference, there are other control strategies and explicit representations of control knowledge in the inference methods that are used in expert system. Three commonly used control strategies are data driven, goal driven and mixed. Data driven control is very popular, and is also known as bottom up, forward chaining, pattern directed, pattern driven, antecedent reasoning, or condition driven. To use this strategy, one must begin by entering information about the current problem as facts in the data base. Program execution consists sole of a continuous sequence of cycles terminating when some 'halting' action is executed. At each cycle, all rules with conditions that are satisfied by the contents of the database are determined. If there is more than one rule, we have the problem of deciding which one to apply. Data-driven strategies differ greatly in the amount of problem-solving effort they devote to rule selection. A simple and inexpensive strategy is to select the first rule that is encountered. Unfortunately, unless the rules are favorably ordered, this can result in many useless steps. Elaborations intended to overcome such shortcomings can

make data-driven control arbitrarily complex.

The popularity of data-driven control derives largely from the fact that such a program can respond quickly to input from the user, rather than forcing the user to wait until the program gets around to what the user wants to talk about. An excellent example of an expert system that employs this strategy is R1 (McDermott 1980).

An alternative control strategy that is more commonly used in expert systems is goal driven. Goal driven control is also known as top-down, backward chaining, consequent reasoning, action driven, or consequent driven. This strategy focuses its efforts by only considering rules that are applicable to some particular goal. Since we are limiting ourselves to rules that can add simple facts to the data base, achieving a goal G is synonymous with showing that the fact statement corresponding to G is true. In nontrivial problems, achieving a goal requires setting up and achieving subgoals. This also can lead to fruitless wandering if most of the subgoals are unachievable, but at least there is always a path from any subgoal to the original goal. The chief disadvantage of the goal-driven strategy is that it does not allow the user to steer it by volunterring relevant information about the problem. This can make goal-driven control unacceptable when rapid, real-time response is required.

Data driven and goal driven strategies represent two extreme approaches to control. Various mixtures of these approachs have

been investigated in an attempt to utilize their various advantages while minimizing their disadvantages. The basic idea is to alternate between these two phases, using informative volunteered by the user to determine a goal and then querying the user for more information while working on that goal.

Neither data driven, goal driven, nor any particular mixed strategy is good for every problem. Different approaches are needed for different problems. Indeed, one kind of knowledge possessed by experts is knowledge of procedures that are effective for their problems. This knowledge is called control knowledge. It includes knowledge about a variety of processes, strategies, and structures used to coordinate the entire problem-solving process. On the other hand, the control knowledge may be viewed as procedural knowledge that describes sequences of things to do or goals to be achieved. This kind of knowledge is often difficult and cumbersome to describe in a declarative manner (Winograd 1975). In fact, even though some procedural knowledge can be incorporated in pure rule-based systems, it is there only because the rule interpreter executes the rules procedurally in some specified order. This means that procedural knowledge and the establishment of contexts in which a particular inference is valid can only be represented implicitly in the system (e.g., by ordering the clauses of a premise and thus ensuring a particular sequence of evaluation). This can create dependencies and interrelationships that tend to make the knowledge base not quite as modular or flexible as perhaps was

originally intended. Because of the homogeneity of the rule representation, it is not possible to distinguish between those rules for which the order of invocation is important and those for which it is not. Thus control knowledge is not explicitly represented, but is buried in the code. This means that the system cannot easily explain its problem-solving strategy, nor can the system builder easily modify it. Explicit representation of control knowledge has significant advantages both for the acquisition and modification of domain knowledge and for explanations of knowledge used in the expert system.

3. CASE STUDY OF THE R1 EXPERT SYSTEM

The reason R1 was chosen for a case study is that it differs from other systems primarily in its use of matching rather than generate-and-test as its central problem-solving method. Rather than exploring several hypotheses until an acceptable one is found, R1 exploits its knowledge of the task domain to generate a single acceptable solution. This is similar to the PlanPert system. In PlanPert, a single acceptable schedule is generated by properly matching courses, faculty, and time and following the rules provided by the experts.

3.1 Introduction and Background

R1's domain of expertise is configuring Digital Equipment Corportion's VAX-11/780 systems. Its input is a customer's order and its output is a set of diagrams displaying the spatial relationships among the components on the order. These diagrams are used by the technician who physically assembles the system. Since an order frequently lacks one or more components required for system functionality, a major part of R1's task is to notice what components are missing and add them to the order. R1 is currently being used on a regular basis by DEC's manufacturing organization.

John McDermott and Barbara Steele began work on R1 in December, 1978. It was implemented in OPS5, a general purpose rule-based language which provides a rule memory, a global working memory, and an interpreter that tests the rules to determine

which ones are satisfied by a set of the descriptions in working memory. A rule is an IF-THEN statement consisting of a set of conditions (patterns that can be matched by the descriptions in working memory) and a set of actions that modify working memory. On each cycle, the interpreter selects one of the satisfied rules and applies it. Since applying a rule results in changes to working memory, different subsets of rules are satisfied on successive cycles.

Every rule in R1's knowledge encapsulates one set of constraints on the way in which components can be associated. Initially, working memory contains just a list of the names of the components ordered. After a description of each of these components is retrieved from a data base, R1 constructs a configuration by associating components in ways that satisfy the constraints held in the rules. R1 is recognition-driven; that is, each of its rules recognizes a different situation in which the creation or extension of a partial configuration is called for. R1 does almost no backtracking because its knowledge is sufficient to lead it unerringly from the set of components ordered to an acceptable configuration.

3.2 Knowledge Representation

R1 was implemented as a production system using the production system language OPS5. An OPS5 production system consists of a set of productions held in production memory and a set of data elements held in working memory. A production is a conditional

statement composed of conditions and actions has the form:

```
Pi (C1 C2 ... Cn --> A1 A2 ... Am)
```

An English translation of a sample rule is shown below.

```
PUT-UB-MODULE-6
  IF: THE CURRENT CONTEXT IS PUTTING UNIBUS MODULES
      IN BACKPLANES IN SOME BOX
    AND IT HAS BEEN DETERMINED WHICH MODULE TO
      TRY TO PUT IN A BACKPLANE
    AND THAT MODULE IS A MULTIPLEXER TERMINAL
      INTERFACE
    AND IT HAS NOT BEEN ASSOCIATED WITH ANY PANEL
      SPACE
    AND THE TYPE AND NUMBER OF BACKPLANE SLOTS
      IT REQUIRES IS KNOWN
    AND THERE ARE AT LEAST THAT MANY SLOTS
      AVAILABLE IN A BACKPLANE OF THE APPROPRIATE
      TYPE
    AND THE CURRENT UNIBUS LOAD ON THAT BACKPLANE IS
      KNOWN
    AND THE POSITION OF THE BACKPLANE IN THE BOX IS
      KNOWN
  THEN: ENTER THE CONTEXT OF VERIFYING PANEL SPACE FOR
      A MULTIPLEXER
```

A rule can be viewed as state transition operator. The conditional part of each rule describes features that a state must possess in order for the rule to be applied. The action part of the rule indicates what features have to be added in order for a new state that is on a solution path to be generated. Each rule is a more or less autonomous piece of knowledge that watches for the generation of a state that it recognizes. Whenever that happens, it can affect a state transition. If all goes well, this new state will, in turn, be recognized by one or more rules; one of these rules will affect another state transition, and so on until the system is configured.

The working memory consists of four types of elements:

(1) Component descriptions.

(2) Elements that define partial configurations.

(3) Elements that indicate the results of various sorts of computations.

(4) Context symbols.

Initially, working memory is empty. It grows, during the course of configuring a system, to contain descriptions of the components ordered and, as various components are associated, to contain descriptions of partial configurations as well as other component information required to do the configuration task. An element that defines a partial configuration contains a description of the relationships among two or more components. Typically, these elements indicate either that one component is to be connected to another by means of a cable or, in the case of a component that is a container, it gives the spatial relationship between the container and each of the components it contains. An element that indicates the result of some computation contains a symbol identifying the computation and one or more values indicating the result. The component descriptions, together with the elements that define partial configurations and the elements that indicate the results of various computations, constitute the component information. A context symbol contains a context (subtask) name and an indication of whether or not the context is

active.

The third memory in OPS5 is the data base, which contains descriptions of each of the 420 components currently supported for the VAX. Each data base entry consists of the name of a component and a set of eight or so attribute/value pairs that indicate the properties of the component that are relevant to the configuration task. Figure 3-1 shows three of the entries in the data base. The RK711-EA is a bundle of components; it contains a 25 foot cable (70-12292-25), a disk drive(RK07-EA*), and a bundle of components (RK611) which itself consists of three continuity boards (G727), a unibus jumper cable (M9202), a backplane (70-12412-00), and a disk drive controller (RK611*). The RK07-EA* is a single port disk drive; it is a unibus device that requires an RK611* as its controller, and it is connected to that controller either directly or via another disk drive with a 70-12292 cable of some length.

```
RK711-EA
     CLASS: BUNDLE
     TYPE: DISK DRIVE
     SUPPORTED: YES
     COMPONENT LIST: 1 070-12292-25
                     1 RK07-EA*
                     1 RK611

RK07-EA*
     CLASS: UNIBUS DEVICE
     TYPE: DISK DRIVE
     SUPPORTED: YES
     FLOOR RANK: 8
     DEPTH: 28 INCHES
     WIDTH: 24 INCHES
     UNIBUS MODULE REQUIRED: RK611*
     PORTS: 1
     VOLTAGE: 120 VOLTS
     FREQUENCY: 60 HERTZ
     CABLE TYPE REQUIRED:   1 070-12292 FROM A DISK DRIVE
                              UNIBUS MODULE
                         OR 1 070-12292 FROM A DISK DRIVE
                              UNIBUS DEVICE

RK611
     CLASS: BUNDLE
     TYPE: DISK DRIVE
     SUPPORTED: YES
     COMPONENT LIST: 3 G727
                     1 M9202
                     1 070-12412-00
                     1 RK611*
```

Figure 3-1: Some representative items from the data base

In addition to containing descriptions of VAX components, the data base also contains a few cabinet templates. A cabinet template describes what space is available in a particular cabinet type. These templates serve two purposes: (1) they enable R1 to know, at any point in the configuration process, what container space is still available, and (2) they enable R1 to assign a specific location (i.e., coordinates) to each component that it places in a cabinet. Figure 3-2 shows the tem-

plates for the cpu cabinet. The components that may be ordered for the cpu cabinet are sbi modules, power supplies, and an sbi device. The template for the cpu cabinet contains descriptions of the space available for each of these classes of components and specifies what can be put where.

```
CPU-CABINET
    CLASS: CABINET
    HEIGHT: 60 INCHES
    WIDTH: 52 INCHES
    DEPTH: 30 INCHES
    SBI MODULE SPACE: CPU NEXUS-2 (3 5 23 30)
              4 INCH-OPTION-SLOT 1 NEXUS-3 (23 5 27 30)
                      MEMORY NEXUS-4 (27 5 38 30)
              4-INCH-OPTION-SLOT 2 NEXUS-5 (38 5 42 30)
              4-INCH-OPTION-SLOT 4 NEXUS-5 (42 5 46 30)
              3-INCH-OPTION-SLOT NEXUS-6 (46 5 49 30)
    POWER SUPPLY SPACE: FPA NEXUS-1 (2 32 10 40)
              CPU NEXUS-2 (10 32 18 40)
              4-INCH-OPTION-SLOT 1 NEXUS-3 (18 32 26 40)
              MEMORY NEXUS-4 (26 32 34 40)
              4-INCH-OPTION-SLOT 2 NEXUS-5 (34 32 42 40)
              CLOCK-BATTERY (2 49 26 52)
              MEMORY-BATTERY (2 46 26 49)
    SBI DEVICE SPACE: IO (2 52 50 56)
```

Figure 3-2: A sample templates

3.3 Knowledge Analysis

In this section, the knowledge of R1 is analyzed in four aspects: the compartmentalizability, the level of uncertainty, the applicability, and the thickness. The purpose of this analysis is to extract knowledge that is useful for the knowledge acquisition of PlanPert.

According to (McDermott 1983), R1 has about 2400 rules distributed among 280 subtasks. On the average, then, there are 9

rules (pieces of domain knowledge) associated with each subtask. Although there is some variation in the amount of knowledge associated with each subtask, most of the subtasks have more than 30 rules associated with them. Typically 2 or 3 of the rules associated with each subtask recognize when other subtasks must be performed and enable those subtasks by depositing the subtask names in working memory. Almost none of R1's knowledge is relevant to more than one subtask. This compartmentalizability is good for constructing an expert system, because that there is very little knowledge associated with each subtask. It is possible to estimate how much knowledge is associated with each subtask and statically impose a structure on that knowledge. It is also possible to determine dynamically when to apply which pieces of knowledge.

Because the configuration task as originally defined for R1 did not include direct interaction with a salesperson or customer, the only data available to R1 was that provided at the beginning of the task. This input consists of a list of quantity/component-name pairs and sometimes other information describing customer-specific configuration requirements. The major uncertainty associated with the data is whether the set of specified components are orderable, play together, and are complete; and these uncertainties are precisely those which the task is there to resolve. Any changes to the order, or additional information that a salesperson or customer might want to provide, occur after the task has been performed. Such changes define a

new configuration task, rather than reflecting uncertainty in data for the initial task.

Though R1 typically has about 9 rules that are potentially relevant at any given time, ordinarily only 2 or 3 of those rules have conditions that are fully satisfied. Almost all of R1's rules have an applicability factor of 1. This means that if a rule's conditions are satisfied and if that rule is not dominated by some other rule on the basis of OPS5's conflict resolution strategies, applying the rule will result in a transition to a state that is on a solution path. Those pieces of R1's knowledge that have an applicability factor less than 1 bear on unibus configuration. An applicability factor less than 1 means that performing the action will result in a transition to a state on a solution path, but backtracking may be necessary. In order for R1 to have unibus configuration knowledge whose applicability factor is 1, it would have to have knowledge that explicitly identified all valid or all invalid unibus configurations. This is simply not feasible.

The role for applicability factors is to make searching more efficient and serve as indirect measures of the likelihood that the goal has been achieved. For tasks that use only knowledge whose applicability factor is 1, the accomplishment of the task is certain, because every transition is on the path to the goal. For tasks that use some knowledge whose applicability factor is less than 1, extensive search may be needed. If there is enough knowledge to make sure that the transition are on a path leading

to the goal, however, the achievement of the goal is under control.

The thickness of a body of knowledge is the amount of different pieces of knowledge that are relevant in the same situation. For the most part, R1 deals with the world at a single level of abstraction. Its knowledge is primarily about configurable components and their possible interrelationships. It does understand that the task it performs has a hierarchical decomposition, and it performs a variety of abstract tasks, but the knowledge it uses to make its way around in those abstract tasks is still knowledge of configurable components. Moreover, its knowledge is at a single level; it does not know why its rules are valid. Though many of its rules could be justified on the basis of more general engineering knowledge, a significant number could be fully justified only by appealing to custom. If R1 had thicker knowledge, it would reduce the number of intractable problems; it also would reduce the amount of searching.

4. BUILDING AN EXPERT SYSTEM

Knowledge acquisition is a bottleneck in the construction of an expert system. It comprises two main phases. The first phase involves identifying and conceptualizing the problem. Identification includes selecting and acquiring an expert, knowledge sources, and resources, and clearly defining the problem. Conceptualization includes uncovering the key concepts and relations needed to characterize the problem. The second phase deals with the formalization, implementation, and testing of an appropriate architecture for the system, including constant reformulation of concepts, redesign of representations, and refinement of the implemented system. Revision results from the expert's criticisms and suggestions for improving the system's behavior and competence. Much time is spent in the latter phase as the system evolves, but accurately scoping the problem and carefully attending to the task types and strategies in the initial phase can dramatically affect the outcome (Buchanan 83).

The knowledge engineer's job is to act as a go-between to help an expert build a system. Since the knowledge engineer has far less knowledge of the domain than the expert, however, communication problems impede the process of transferring expertise into a program. In PlanPert's case, communication was never a problem. Because almost everyone understands scheduling and can do it to some extent. The only difference is that no one can do it as well as an expert does. Thus, the domain of PlanPert was simpler than that of many other expert systems.

The vocabulary initially used by an expert to talk about a problem domain with a novice is often inadequate for problem solving. Thus the knowledge engineer and the expert must work together to extend and refine it. Although there was no communication problem is discussing PlanPert's domain, the expert and the knowledge engineer spent lots of time together conceptualizing and refining the knowledge. In most cases, knowledge was refined by making it more specific and more compartmentalized to facilitate designing and programming, (which might not be true, but it will be discussed in chapter 5) and the expert would then have to validate the refined knowledge. We can see this in terms of a knowledge engineer who speaks of the program and an expert who speaks of the expertise. An unobtainable goal is to make the program perform with knowledge that is exactly the same as that of the expert.

4.1 Major Stages of Knowledge Acquisition

There are two valuable studies that discuss knowledge acquisition (Buchanan 83, McDermott 83). Buchanan concentrates on the major stages of knowledge acquisition while McDermott focus on the knowledge about knowledge acquisition. In the following two sections these two papers will be reviewed, and the applications of the ideas that were presented in them will be described and discussed in detail in chapter 5.

Before producing an expert system, the knowledge engineer proceeds through several stages that can be characterized as

problem identification, conceptualization, formalization, implementation, and testing. These stages are simply rough characterizations of the complex and ill-structured activities that takes place during knowledge acquisition.

(I) Identification Stage

This stages involves identifying the following things: (1) Participants - Who are knowledge engineer and domain expert? (2) Problem - What class of problem will the expert system be expected to solve? How can these problems be characterized or defined? What are important subproblems and partitionings of tasks? What are the data? What are important terms and their interrelations? What does a solution look like and what concepts are used in it? What aspects of human expertise are essential in solving these problems? What is the nature and extent of "relevant knowledge" that underlies the human solutions? What situations are likely to impede solutions? How will these impediments affect an expert system? (3) Resources - How much knowledge resource, time, computing facility (both hardware and software), and money are available? (4) Goal - What is the goal for the expert system to achieve?

(II) Conceptualization Stage

This stage involves repeated interactions between the knowledge engineer and the domain expert that are important, difficult and time consuming. During this stage the knowledge engineer has to think about how to formalize the knowledge being

gathered - that is, what architecture would best organize the knowledge. Before the knowledge engineer proceeds with this stage, there are a list of questions that must be answered.

(1) What types of data are available?

(2) What is given and what is inferred?

(3) Do the subtasks have names?

(4) Do the strategies have names?

(5) Are there identifiable partial hypotheses that are commonly used? What are they?

(6) How are the objects in the domain related?

(7) Can one diagram a hierarchy and label causal relations, set inclusion, part-whole relation, etc.? What does it look like?

(8) What processes are involved in problem solution?

(9) What are the constraints on these processes?

(10) What is the information flow?

(11) Can the knowledge needed for solving a problem be identified and separated from the knowledge used to justify a solution?

(III) Formalization Stage

The formalization process involves mapping the key concepts and relations into a formal representation suggested by some expert-system-building tool or language. The knowledge engineer must select the language and, with the help of the expert, represent the basic concepts and relations within the language's framework. Three important factors in this stage are the hypothesis space, the underlying model of the process, and the characteristics of the data. One must formalize the concepts and determine how they link to form hypotheses in order to understand the structure of the hypothesis space.

The nature of the hypothesis space can also be derived from the following concepts: whether or not it is finite, whether it consists of prespecified classes or must be generated from concepts by some procedure, whether or not it is useful to consider hypotheses hierarchically, whether or not there will be uncertainty or other judgemental elements related to the final and intermediate hypotheses, and whether or not diverse levels of abstraction would be useful.

The underlying model of the process may include both behavioral and mathematical models. If the expert uses a simplistic behavioral model when reasoning or justifying reasoning, analyzing it may yield numerous important concepts and relations. If there is a mathematical model underlying part of the conceptual structure, it may provide enough additional problem-solving information to be included directly in the expert system, or it may serve merely to justify the consistency of the causal

relations in the expert system's knowledge base.

Understanding the nature of the data in the problem domain is also important in formalizing knowledge. If the data can be explained directly in terms of certain hypotheses, it is useful to know if the nature of this relation is causal, definitional, or merely correlational, because this may help explain how hypotheses that directly explain data can be related to other, high-level hypotheses and how these hypotheses relate to the structure of goals in the problem-solving process.

(IV) Implementation Stage

Implementation involves mapping the formalized knowledge from the previous stage into the representational framework associated with the tool chosen for the problem. As the knowledge in this framework is made consistent and compatible and organized to define a particular control and information flow, it becomes an executable program. The knowledge engineer evolves a useful representation for the knowledge and uses it to develop a prototype expert system. The domain knowledge made explicit during the the formalization stage specifies the contents of the data structures, the inference rules, and the control strategies. The tool or representation framework specifies their form. Local consistency of the problem-solving primitives will already have been worked out in previous stages, but this does not guarantee an executable program, since there may be global mismatches between data structures and some rule or control specification.

Such inconsistencies must be eliminated to ensure rapid development of the prototype expert system.

(V) Testing Stage

The testing stage involves evaluating the prototype system and the representational forms used to implement it. Once the prototype system runs from start to finish on two or three examples, it should be tested with a variety of examples to determine weaknesses in the knowledge base and inference structure. The experienced knowledge engineer will elicit from the domain expert those problems likely to challenge the system's performance and reveal serious weakness or errors. The elements usually found to cause poor performance because of faulty adjustment are the data acquisition and conclusion presentation, inference rules, control strategies, and text examples.

4.2 Knowledge about Knowledge Acquisition

Over the past several years a great deal of research has been done that emphasizes techniques applicable to narrow domains. What has not yet been achieved, however, is much of an understanding of why these techniques work or of the limits of their usefulness. In this section the knowledge which a knowledge engineer should have when doing knowledge acquisition is introduced. In order to understand the roles knowledge plays in any particular task domain, it is necessary to know the following things about the task:

(1) Compartmentalizability of the task knowledge.

(2) Level of uncertainty of the task information.

(3) Applicability of the task knowledge.

(4) Thickness of the task knowledge.

How and to what extent task knowledge can be compartmentalized depends almost exclusively on the structure of the task; the more compartmentalized the knowledge, the narrower its scope and the simpler the choice of what pieces of knowledge to apply. Knowledge operates on information made available by the task environment; the more uncertain that information, the less control the knowledge has. Each piece of knowledge has limited applicability; the power of a piece of knowledge depends on the set of situations to which it is applicable. Finally, many different pieces of knowledge may be relevant in the same situation;

the thicker the task knowledge, the more knowledge there is to bring to bear at any given time.

(McDermott 83) presented eight hypotheses concerning knowledge:

(1) Only a relatively small amount of an expert's knowledge is potentially relevant in any given situation.

(2) Subtasks serve as the dominant organizing principle for an expert's knowledge.

(3) An expert's task is almost always, at least in part, one of data validation.

(4) Tasks which are defined in such a way that the environment cannot intrude after the task has begun should always be redefined to allow intrusion.

(5) If the situations in which an expert can find itself evoke a relatively small number of different behaviors, the pieces of knowledge that the expert uses to determine how to act in those situations will have an applicability factor of 1.

(6) The narrower the domain, the more likely it is that the expert will almost never search.

(7) The more abstract a piece of knowledge is relative to other knowledge about the same situation, the lower its applicability factor will be.

(8) Thick knowledge reduces the number of intractable problems; it also can reduce the amount of search. Though the hypotheses lack substance, they do suggest the sort of advantages substantive hypothesis could confer. First, having a partial description of a role that knowledge can play gives us something to refine and exploit or falsify; we can focus our efforts on pushing and shaping promising insights. Second, substantive hypotheses also provide an avenue of discovery; given a set of hypotheses to reflect on, we are in a position to discover approaches that have not been tried. In next chapter we will see both how this knowledge affects knowledge acquisition, and the benefits of taking it into account during the process of extracting knowledge from experts.

5. THE CONSTRUCTION OF PLANPERT

The first step in building PlanPert was to choose an administrative expert to provide expertise on scheduling courses. After the participants were selected, their roles in the knowledge acquisition process were defined. This process was informal, with the expert expressing things in his own way and the knowledge engineer bringing up questions for the expert to answer. Notes were taken by the knowledge engineer whose job was to reorganize the notes to try to find useful knowledge that could be implemented in a program.

In the initial meeting, the following issues were addressed:

(1) The problem domain would be scheduling courses, faculty and times for undergraduate Computer Science courses.

(2) In order to do scheduling one has to have a list of courses and sections that are offered, and a list of faculty with their preference for teaching specific courses.

(3) The job of the expert system would be put these courses and faculty into time slots so that an acceptable schedule is arrived at. Nine time slots from 8:00 am to 4:00 pm were decided on.

(4) Some courses, (likes 355 and 610) were excluded, because they usually were offered during the evening by specific adjunct instructors.

(5) The knowledge engineer would schedule numerous meetings with the expert to uncover the basic concepts, primitive relations, and definitions needed to talk about the problem and its solutions. During these meetings the knowledge engineer would attempt to understand what concepts are important and relevant to the problem by asking the expert to explain and justify reasoning used to deal with specific types of scheduling problems.

During the knowledge acquisition meeting, the knowledge engineer listened to the expert in order to characterize the expertise in terms of a few broad kinds of knowledge that have been encountered when developing expert systems. In addition to recording terms that the expert used in a well-defined manner, the knowledge engineer also noted other organizational methods that the expert seemed to use. A second kind of knowledge the knowledge engineer listened for was the basic strategies the expert used when performing the task. What facts does the expert try to establish first? What kinds of questions does the expert ask first? Does the expert make initial guesses about anything based on tentative information? In what order does the expert pursue each of the subtasks, and does this order vary across different situations?

Appendix A records the interaction between the expert and knowledge engineer in three different stages. At the early stage of the knowledge acquisition process, most of the questions were definitional. For example, what is the faculty's course

preference list? What does the ranking in the preference mean? What are the pair courses (courses usually taken in the same quarter by a student) ? How are upper level and lower level courses defined? As the process continued, more and more "reasoning" questions were asked. Toward the end of the process, the knowledge engineer was able to justify and validate the knowledge used in scheduling.

PlanPert must have two sorts of knowledge. First, it must have information about each course, (e.g. when is a course offered? How many sections are offered? Is it a programming course? Does it have prerequisite courses?). Second, PlanPert must have rules that enable it to arrange all components (i.e. courses, faculty, and time) to form a partial schedule and assemble partial schedule to form an acceptable one. The rules must indicate what course can be put in which time slot and what constraints must be satisfied in order for these arrangements to be acceptable.

5.1 Development of the Knowledge Base

The knowledge base of PlanPert has three parts, facts, constraints, and heuristics. Facts are things that are always true. Constraints are things that one must be satisfied in order to make a correct decision. Heuristics are methods that can be used to get something done. In this section the conceptualization and implementation of facts are discussed. There are two kinds of facts, one associated with course (C-Facts) and the other asso-

ciated with faculty (F-Facts). Two examples of C-Facts represented in English are shown below.

```
COURSE NUMBER: 241
PREREQUISITE: NONE
PAIR: 202
CONCENTRATION: NONE
SECTION/QUARTER: 9 (851)
                 4 (852)
                 1 (853)
ATTRIBUTES: PROGRAMMING
            LOW-LEVEL
```

```
COURSE NUMBER: 580
 PREREQUISITE: 450
PAIR: NONE
CONCENTRATION: 520, 540
SECTION/QUARTER: 1 (852)
                 1 (853)
ATTRIBUTES: PROGRAMMING
            HIGH-LEVEL
```

There are total 37 C-Facts entries in PlanPert's knowledge base. Each entry in the knowledge base gives the following information about a course:

(1) Course Number.

(2) What its prerequisites are?

(3) Pair Courses: Whether it is a course usually taken with another course.

(4) Concentration courses: Whether it is a course that is one of a series of specialized courses.

(5) Section/Quarter: How many sections of it are offered in particular quarter.

(6) Attributes: Whether the course requires programming projects. What's its level of difficulty?

F-Facts contain the following:

(1) Faculty Name.

(2) Courses that he may teach.

(3) Preference rankings for each course. 1 means that the faculty wants to teach it. 2 means that the faculty is willing to teach it. 3 means that the faculty does not wish to teach it. 4 means the faculty never wants to teach it.

(4) Whether the faculty is ready to teach it or not?

(5) How many sections does a faculty teach for a certain quarter?

Example of F-Facts in the knowledge base are shown below.

```
FACULTY NAME: BAKER
COURSE-NUMBER: 202
PREFERENCE: 2
READY TO TEACH: NO


FACULTY NAME: BAKER
SECTIONS: 2
QUARTER: 851
```

Facts mentioned above are direct input to the system. There are other facts that can be derived from the original ones. For example, the number of faculty that are willing and ready to teach a course.

```
FACULTY-AVAILABLE    COURSE-NUMBER: 202
                     FACULTY-AVAIL:   9
                     QUARTER: 851
```

To derive this fact, searching through the F-Facts and counting the number of faculty whose preference rank is 1 or 2, and whose ready-to-teach value is 'yes' are necessary. This fact is important because the number of available faculty must be greater than or equal to the number of sections of a course.

During this conceptualization stage, the knowledge engineer was also thinking about how to formalize the knowledge being gathered - that is, what architectures would best organize the knowledge. This task involved picking the organization and tool or programming environment to use for the particular application. The language chosen for PlanPert's implementation was PROLOG, which is an unusual but exceptionally simple language. It is an excellent language for expert system implementation for the following (Pereira, Sabatier, & d'Oliveira, 1982):

(1) The various components of an expert system are integrated into the same simple formalism: natural language processing, knowledge base, explanation facility, relational data base, metaknowledge (information about how to use knowledge), and interpreters for specialized control.

(2) Compactness of expression, together with an efficient implementation, permits programs as complex as expert systems to be used practically.

(3) The dual semantics, declarative and procedural, facilitates the development of metaknowledge features.

The knowledge base in PlanPert can be viewed as a relational data base. Two basic relations are Course and Faculty as shown in Figure 5-1. Course is a relation of degree 6 containing six domains which are set of values representing, respectively, the course number, number of sections, programming attribute, prerequisites, concentration courses, and pair courses. Faculty is a relation of degree 4 containing four domains which are the faculty name, the course, the preference rank, and the ready-to-teach flag.

COURSE

Course	Sect	Attru	Preq	Concnt	Pair
202	5	-	-	-	241
241	9	prog	-	-	202
242	1	prog	241	-	-
243	3	prog	242	-	-
305	6	prog	243	-	-
306	1	prog	305	-	-
315	3	-	305	-	325
325	3	prog	243	-	315
440	3	-	325	-	420
540	1	prog	440	580	-

FACULTY

Faculty	Course	Rank	Ready
baker	202	2	no
baker	440	1	yes
baker	560	3	no
carithers	241	2	yes
catithers	360	3	yes
carithers	440	1	yes
carithers	570	2	no
comte	202	3	no
comte	560	3	no
coon	400	3	no
coon	440	2	no
coon	560	1	yes
coon	570	2	yes

Figure 5-1

The goal of PlanPert is to derive a new relation from relation FACULTY and COURSE. The new relation is called SCHEDULE, and an example is shown in Figure 5-2.

SCHEDULE

Time	Course	Faculty
9:00	202	Wolf
10:00	202	Hammerton
11:00	202	Lasky
12:00	202	Hammerton
2:00	202	Wolf
4:00	306	Carithers
9:00	307	Carbin
9:00	315	Comte

Figure 5-2

In PROLOG the basic unit, the clause, is a production rule, which has emerged, independently of logic programming, as the favorite formalism of knowledge engineering. Moreover, in logic programming the natural way of activating clauses is by pattern matching. Also, clauses are so general that they encompass as a special case the relational data base model. As a result of this, clauses can always be regarded as specifying, explicitly or implicitly, a relational data base. In this way the usual distinction between program and data base disappears, which is especially attractive for knowledge engineering. There is no distinction between the knowledge base and the situation model, either. The clauses belonging to knowledge base remain unchanged

through the processing, and the clauses belonging to the situation model are being constantly asserted and retracted. The transformation of some simple facts from English to PROLOG are shown below.

C-Facts (English)

```
COURSE NUMBER: 440
PREREQUISITE: 315, 325
PAIR: 420, 450
CONCENTRATION: NONE
SECTION/QUARTER: 3 (851)
                 2 (852)
                 1 (853)
ATTRIBUTES: NONE PROGRAMMING
            HIGH-LEVEL
```

PROLOG:

```
course_section_quarter(440,3,851).
course_section_quarter(440,2,852).
course_section_quarter(440,1,853).
prerequisite(440,315).
prerequisite(440,325).
pair(440,420).
pair(440,450).
attribute(440,high-level).
```

F-Facts (English)

```
FACULTY NAME: ELLIS
COURSE NUMBER: 560
PREFERENCE: 3
READY TO TEACH: NO
```

PROLOG:

```
preference_ready(ellis,560,3,n).
```

5.2 Development of the Knowledge Manager

There are two kinds of constraints, one associated with assigning courses (C-Constraints), and the other associated with assigning faculty (F-Constraints). C-Constraints must be

satisfied when assigning courses to time cabinets. From now on, the term 'time slot' will be replaced by 'time cabinet'. A time cabinet means a block of time; for example, time cabinet 9 is a block of time from 9:00 am to 10:00 am. Because there are usually several courses offered at the same time, the 'cabinet' simply means that a time container contains a number of courses. The following are constraints and heuristics that were extracted from the expert.

C-Constraints

(1) There is no course offered at 1:00 pm.

(2) Avoid assigning courses to time cabinet 8 and 4.

(3) The maximum number of courses in one cabinet is 10.

(4) If a course has more than five sections, assign no more than two sections to one cabinet.

(5) If course has less than five sections, assign only one section to one cabinet.

(6) It is better not to assign pair courses to the same cabinet.

(7) It is better not to assign concentration courses to the same cabinet.

(8) It is better to assign a course and its prerequisites to the same cabinet.

(9) It is better to assign programming courses to the same cabinet.

F-Constraints

(1) Each faculty has no more than two sections for a given course.

(2) Do not assign a faculty either three sections of the same course or three different courses.

(3) A faculty gets only one section of a high level course.

(4) A faculty is assigned two sections of a low level course.

(5) A faculty is assigned two sections of multiple section course.

(6) Whoever has a higher preference rank for a course has the higher priority to be assigned first to that course.

(7) If one faculty has three courses to teach, one of them should be a high level course and the other two, low level.

(8) A faculty does not teach at three consecutive time slots, for example from 9:00 am to 12:00 straight.

(9) A faculty does not teach at every other time, for example 10:00 am, 12:00 am and 2:00 pm.

There are two kinds of heuristics. C-Heuristics are used to assign courses to cabinets. F-Heuristics are used to assign faculty to cabinets.

C-Heuristics

(1) Most-Section-First (MSF) : Start processing with the course that has most sections.

(2) Least-Cabinet-First (LCF) : Assign courses to the time cabinet with the fewest courses in it.

F-Heuristics

(1) High-Level-First (HLF) : Start processing from highest level course.

(2) Least-Faculty-First (LFF) : Assign faculty to a course that has the least available faculty first.

These constraints and heuristics are associated with the subtasks in PlanPert. The hierarchical organization of subtasks in PlanPert is shown in Figure 5-3. At the top level, there are four subtasks, initialize, assign_course, assign_faculty, and preassign.

Subtask Initialize. The first subtask is to set up the working memory in the situation model, derive important facts from the original ones in the knowledge base, and sort part of the knowledge base for the sake of efficiency. There are 9 time cabinets needed, ranging from 8:00 am to 4:00 pm. The set up is done by asserting the following facts into the working memory.

```
cabinet_count(9,0).
cabinet_count(10,0).
cabinet_count(11,0).
cabinet_count(12,0).
cabinet_count(2,0).
cabinet_count(3,0).
cabinet_count(4,0).
```

Cabinet 8, 1, and 4 are excluded in order to satisfy the C-Constraints (1) and (2). Two new facts are derived from the original knowledge base, one is total_section_offer and the other is faculty_available. Total_section_offer is the total number of sections offered in a certain quarter. Faculty_available is the total number of available faculty for a certain course. Because the heuristics used by subtask assign_course and assign_faculty all deal with ordering, sorting the required facts in the

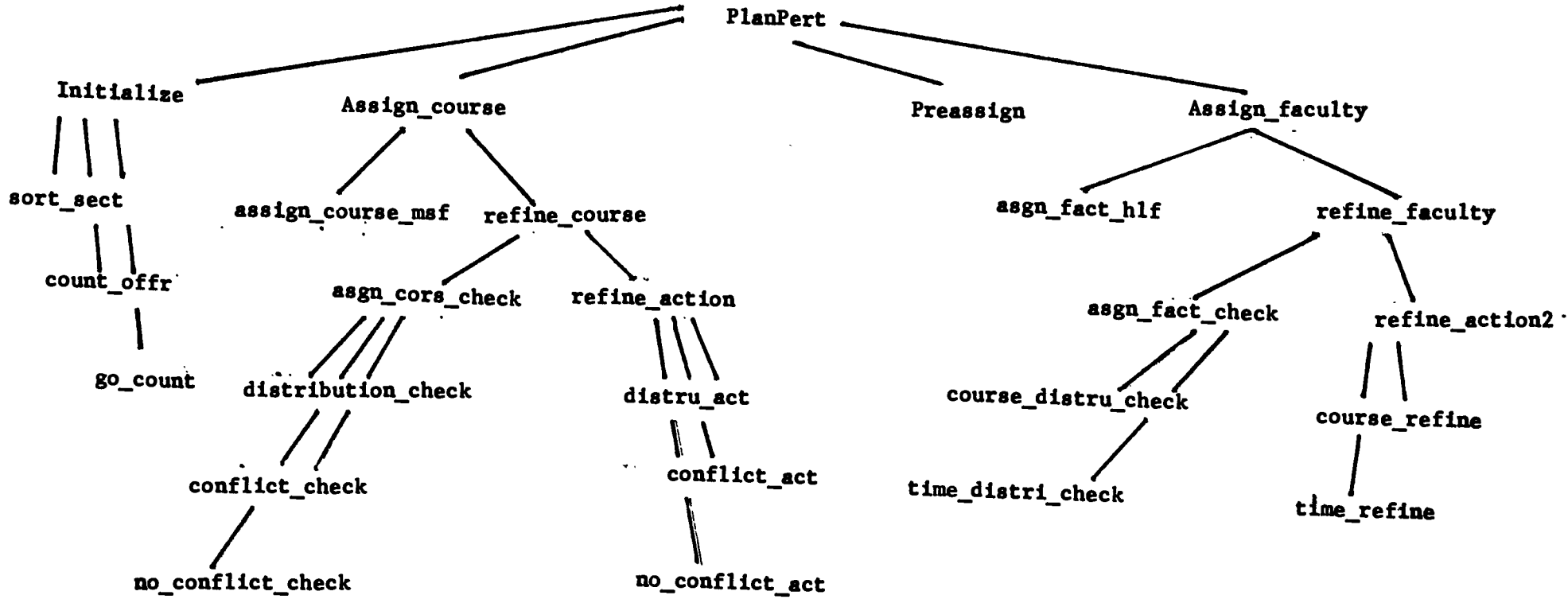


Figure 5-3: Hierarchical Organization of Subtasks in PlanPert

knowledge base eases the implementation of these heuristics.

Subtask assign_course. The second subtask involves assigning courses to time cabinets. It has two subtasks named assign_course_msf and refine_course. Assign_course_msf assigns courses to cabinets by using the C-Heuristics MSF (Most-Section-First). Refine_course checks the satisfaction of constraints associated with assign_course (C-Constraints) and takes proper action to modify an unsatisfied assignment. C-Constraints can be grouped as the following:

(1) Distribution Constraints: If a course has more than five sections assign no more than two sections to a time cabinet, else assign only one section to a cabinet. This includes the C-Constraints (4) and (5).

(2) Conflict Constraints: This includes C-Constraints (6) and (7). To satisfy this constraint the courses mentioned in C-Constraints (6) and (7) should be assign to the same cabinet.

(3) No-Conflict Constraints: This includes C-Constraints (8) and (9). To satisfy this constraint the courses mentioned in C-Constraints (8) and (9) should not be assign to the same cabinet.

The first part of refine_course checks every assignment done by assign_course_msf and asserts the unsatisfied one to the working memory. The second part of refine_course reads in the unsatis-

fied assignment and changes it. The first part, in PROLOG, is:

```
refine_course_check:-
    distribution_check,
    conflict_check,
    no_conflict_check.
```

Each of the three checks concerns a group of constraints. For example, distribution_check examines whether or not the distribution_constraints are satisfied. If the constraint is not satisfied, facts like the following will be asserted to the working memory.

```
upper_over_load(9,241,3).
lower_over_load(2,560,2).
```

The first one says that in cabinet 9 there are three sections of course 241, the C-Constraints (4) is not satisfied. The second one says that in cabinet 2 there are two sections of course 560, and C-Constraints (5) is not satisfied.

Subtask preassign. The third subtask assigns faculty to courses without using any heuristics. The result of preassign does not have to satisfy any constraints, so this subtask can be viewed as input from the expert.

Subtask assign_faculty. The fourth subtask involves assigning faculty to courses. It also has two subtasks, one that assigns faculty to courses by using the F-Heuristics HLF (High-Level-First) and LFF (Least-Faculty-First), and the other that refines the results. Assign_faculty is similar to assign_course. They both can be broken down into smaller piece, and each pieces

deal with a certain number of constraints. There are 9 F-Constraints which can be divided into two groups:

(1) Course-distribution constraints. These include F-Constraints (1), (2), (3), (4), (5), (6), and (7). They all concern the distribution of courses for a faculty.

(2) Time-distribution constraints. These include F-Constraints (8) and (9). A faculty does not teach at three consecutive time slots and at isolated time slots.

The refine part of assing_course does the same thing as refine_course. It first checks the unsatisfied assignment, asserts it into the working memory, and then takes proper action.

The subtasks communicate with one another through the working memory. The advantages of grouping constraints is to reduce the complexity of the subtasks and make it easier to implement.

6. CONCLUSIONS

6.1 PlanPert's Knowledge Analysis

PlanPert has 18 constraints and 4 heuristics among 25 subtasks. The 18 constraints are further divided into 5 groups. There are 17 subtasks that are need to deal with these constraints. They are subtasks within refine_course and refine_faculty. On the average, there are 2 constraints associated with each of the 7 subtasks. This compartmentalizability makes it possible to estimate how much knowledge is associated with each subtask and determine dynamically when to apply which pieces of knowledge. Because there is only a small amount of knowledge associated with a subtask, it also makes the implementation of PlanPert easier.

There is no uncertainty in PlanPert's task. All the information the knowledge operates on is available, so the knowledge of PlanPert would be able to have complete control over the scheduling task.

Not all of the constraints in PlanPert have an applicability factor of 1. For example, F-Constraints (3) (Faculty gets only one section of a high level course) has an applicability factor less than 1. This means that it is not necessary to always satisfy F-Constraints (3). Since PlanPert's job is to assist an administrator and not to replace him, the administrator's own knowledge would be able to handle this unsatisfied constraint. He might persuade the faculty who is assigned with 2 sections of

a high level course to accept this assignment. Once the faculty accept it, it is unnecessary to satisfy that constraint. The reason PlanPert would have some constraints with applicability factor less than one is the flexibility of the nature of the scheduling task. The scheduling task not only deals with material objects (courses) but also deals with "spiritual" objects (human beings). There is no certain pattern that human beings can be forced into, and schedule itself is not a law, or something permanent, it is just something that both the administrator and the faculty agree with. Their considerations about accepting a schedule can always change.

The last aspect of analyzing PlanPert's knowledge is the aspect of thickness. Compared with R1, PlanPert certainly has thicker knowledge. This is also due to the nature of the scheduling task. An expert can start assigning courses to cabinets and switch to assign faculty to those courses he just assigned to the cabinets. This means that an expert could do things dynamically and in any way he wants. From the point of PlanPert's subtask structure, an expert can do any subtask in levels 3 and 4 without following any sequence. The subtasks in an expert are not ordered at all. Because of this lack of ordering, any of the 18 constraints can get involved in any situation. The interactions among the subtasks, then, happen at very low level. The thickness of PlanPert's knowledge results in a limitation of its performance and an increase in interactions between the administrator and the program. To enhance PlanPert, more

knowledge acquisition and an extension of the system organization should be done.

6.2 Suggestions for Future Extension

In order to enhance PlanPert, two things have to be done.

(1) Change the sequential processing to parallel. Let the subtasks within PlanPert operate concurrently.

(2) Set up a communication among these parallel subtasks.

One approach can be adopted is the blackboard developed in the context of the HEARSAY speech understanding project (Erman, 1980). The idea behind the blackboard approach is simple. The entire system consists of a set of independent modules, called knowledge sources, that contain the domain-specific knowledge in the system, and a blackboard, a shared data structure to which all knowledge sources have access. A concurrent PROLOG will be needed to do the parallel processing. All the subtasks can access to the blackboard by using some queuing technique. In this way the operation of PLANPERT will be similar to the one of the experts. There are several things expected to happen when doing so, more time-consuming, more complex, and of course higher performance.

Knowledge Acquisition forms an very important part of building an expert system. Expert knowledge comes in two forms: declarative and procedural. Acquisition and representation of diverse forms of declarative knowledge seem reasonably well

understood, however, a significant component of expertise takes a procedural form. The problem of acquiring and representing procedural knowledge is still a major research endeavor. In the development of PlanPert, the knowledge engineer was instructed in the domain by the expert, and then translated that domain-specific knowledge into the particular representation system chosen. For the future extension, a better approach to knowledge acquisition would be having the domain-experts themselves build the knowledge bases. The reasons for providing for knowledge entry by the domain experts are numerous.

First both accuracy and completeness of the knowledge base suffer when domain knowledge passes through the filter of a knowledge engineer, who is not an expert in the chosen domain. Much of the knowledge is complex and subtle, and its purpose may not be immediately apparent to the non-expert. Second, a large knowledge base may be built more quickly when non-experts do not have to be intimately involved in describing each object and rule in the knowledge base. Finally, in building a knowledge base for a program to be used by other experts in the field, an element of trust, as embodied in the name of a known authority, is essential. In addition to extracting knowledge directly from the domain experts, it is also possible to extract the nondomain-specific parts from existing expert systems and use them as tools for building new systems in new domains.

Another extension that could be make would be to improve the learning ability of an expert system. The major failing of all

current expert systems is that they can not learn from experience. This means they may make the same mistakes many times. We can accept that human experts sometimes make mistakes, however, we can generally expect a person to learn from an error and not repeat it. Despite these failings, all of which are being researched, the current generation of expert systems offers a higher level of performance than more traditional programming techniques. We can expect to see their increasing use in more and more situations where expert advice is widely needed and in short supply.

Appendix A

The following are three partial dialogues from three different knowledge acquisition stages. The purpose of this is to show how the knowledge is gradually extracted from an expert by a knowledge engineer. We use K to represent the knowledge engineer and E for expert.

*** Stage 1 *** (Identification Stage)

K : What kinds of information or data do you need for scheduling?

E : Basicly I would need a list of courses and sections that are offered and a list of faculty's course prereference.

K : What would you do first?

E : I would spread the multiple-section course out through the whole day. Usually we do not want to have course at 8:00 am and 4:00 pm. And there is no course at 1:00.

K : What do you mean the multiple-section course ?

E : They are courses with more than 4 sections.

K : When you start assinging course to time what types of rules or things you would consider ?

E : I might have 2 sections of a course meet at the same time. Later on if the number of student is too few for two classes I will cancel one and move all the students to another one. By doing this there is no change or effect on the student's

schedule. For each time slot we can have maximum 10 courses. I will always assign course to the time slot which has the fewest number of courses in it.

K : Do you consider some attribute of the courses for example the prerequisite or co-course ?

E : Yes, I would try to arrange a course and its prerequisite course at the same time slot. In this way I increase the number of students who follow the standard sequence of taking courses. By the way, what are the co-courses?

K : They are the courses that you would have to take them at the same quarter.

E : We do'nt have co-course here. But we have some pair course that student usually take them at the same quarter. Examples are course 201 and 241, course 360 and 315, and course 315 and 325. For those course I would try not to put them at the same time slot. Thus the students wo'nt have conflict schedule problem.

K : .....

*** Stage 2 *** (Conceptualization Stage)

K : We had talked about some general information of making a schedule. There are lots of things we have to consider and lots of rules or heuristics that you use. Let us use an example to actually do the schedule from the very beginning. During the process let us try to make those rules, facts, and heuristics more specific and more obvious. In other words, let us try to name them and describe them.

E : All right. When I start doing it the first thing I will do is to lay out the time slot for courses.

K : You can use the 851 course offering list to show me how you do it.

E : First I will choose course 241.

K : Why choose it ?

E : I start from the course with the most sections. Spread them evenly through the day. Choose course by their descending order of number of sections.

K : Why ?

E : Because single section course is very easy to find a slot for it. And I do the difficult ones first and leave the easy one last. If I do the single section course first, at the end I might have problem to stuff those multiple-section courses into the schedule.

K : So, this method can be treated as a heuristic.

E : Why not ?

K : Remember that we have a rule says that we can only have maximum 10 courses per hour. How does this influence the scheduling ? And do you take the attributes of courses into account ?

E : Since we got 8 time slot. I would put counter on each of them and keep track of how many courses had been assigned to one slot. Assign course to the slot which has the fewest number of courses in it.

K : Another heuristic ?

E : Yes.

Talking about the attributes I always put the programming courses at the same slot. Because most of the students only take one or two programming courses at one quarter. In our undergraduate courses there are several concentration courses. Student usually take them at the same quarter. And I would put m at the different time so the student would be able to take all of them at one quarter.

K : Let me repeat the idea I got from you. And tell me whether or not they are correct. For a curse we can define several attributes like multiple-section, programming, pair courses, prerequisite and concentration. For each attribute there is one or more associated rules you have to consider. For example, we want the programming courses and prerequisite courses meet at the

same time and the the concentration and pair courses at different time.

E : Correct!

K : Then here comes the problem. Which rules you have to strictly follow and which you do'nt ? How to decide the attributes of the rules, say that a rule must be followed or might be followed or better be followed.

E : Yes, I have thought about this problem too. My suggestion for the program would be that assign these rule priority or probability.

K : .....

*** Stage 3 *** (Conceptualization and Formalization Stage)

K : This time we are going to do the assignment for faculty. Let us assume that we already have a schedule which has only courses in it. What would you do first ? Or how do you start ?

E : I would start with the course which has the fewest number of faculty that are willing to teach it.

K : What do you mean they are willing to teach it?

E : Oh! I forgot to tell you that once a while each faculty has to fill in a prereference list with rank from 0 to 4 and one question for ready to teach or not. If a course has rank 0 that means the faculty definitely will teach it. 1 means that they want to teach it. 2 means that they are willing to teach it. 3 means that they do not want to teach it. And 4 means that if you assign this course to them they are going to kill you. The number of faculty that are willing to teach means they has rank 1 or 2 and have 'yes' for the answer about ready to teach or not.

K : Then what is the next step ?

E : After I pick up one faculty I might assign another 2 or 1 courses to him. Then I am all done with this faculty. Sometime I might do another way. Finish all the assignment for one course and pick up another one. Because for some course I always assign to some faculty. I do these base upon my experience. So the process does not follow a certain route.

K : We discuss these two methods later. Tell me what facts and rules you would use when assign a faculty to a course.

E : There are several of them. Each faculty has no more than 2 sections of one course. They do'nt want to be assigned with either the same three courses or the difference ones. For low level courses I would assign two sections to a faculty each time. And for high level course I only give one faculty one section.

K : How do you define low and high level courses ?

E : Courses that are above 400 level is high otherwise is low. Usually I would assign one high level course and two low level courses to a faculty. Just try to balance their life. There are other things about the time too. For faculty has three courses I would put two of the courses on consecutive time .

K : When you do the assignment, the object you process on is changing all the time. Sometime it is a course and sometime it is a faculty. When you pick up one course that has the fewest number of faculty that are willing to teach it the object is course. When you choose one faculty among those fewest number of faculty you process on a faculty. And you try to finish all the course assignment for this faculty the object becomes courses again. When I say object I mean the target that you are working on. What I am trying to say is shown on this diagram. (Fig. A-1) There are two ways to approach this. Diagram A is the way you do it. In this method you have to keep track of how many faculty are available at different time and how many courses and what are

they has been assigned. The object is dynamically change. There is no certain pattern or route at all. But it probably more efficient. We will find out this later. Diagram B is another way which is linear and simpler. You only have to keep track of how many courses has been assigned to a faculty. Compare these two let us see which of them is fit to the program better.

E : I would say the second one will be lots of easier to implement it.

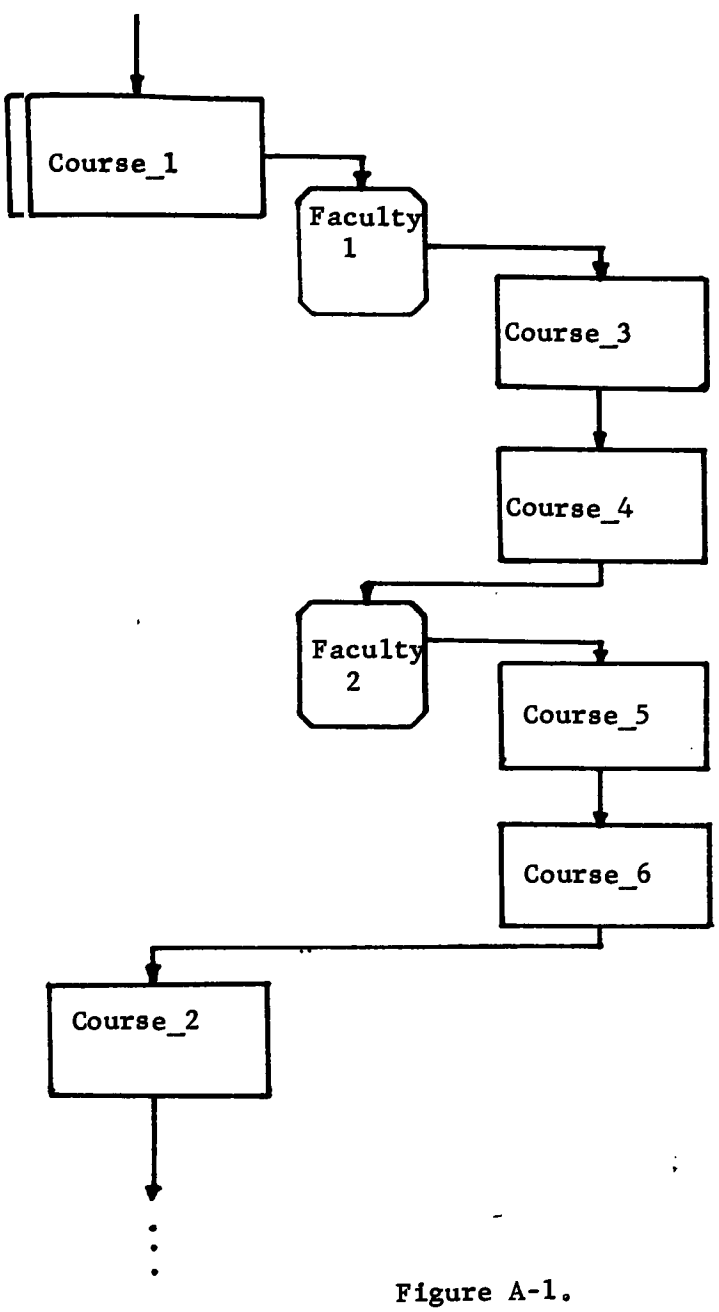
K : If we use the second method then how do we decide the order of courses to be processed. Can we use the fewest number of faculty ?

E : Yes, we can. But there is another ways to do it probably better. May be we can use a ratio. A ratio of number of faculty to the number of sections (R = #F / #S). The ratio would tell us how many number of faculty are available to one section. And we can start assinging faculty to the course which has the lowest ratio. If there are two courses one has ratio 8/9 which means there are 8 faculty and 9 sections and another has ratio 1/1 which means there is only one faculty and one section. In this case we will process the one with ratio 8/9 first which is not always true. Because the only one faculty who teach that course might had been assigned to some other courses by the time we process it. The way we should do is to assign this only one faculty to that only one section first.

K : Since the ratio is not always reliable. And if we only consider the course with the fewest number of faculty first we might have problem about finding faculty for high level courses. For example if we have two courses, a low level course and a high level one. The low level one has fewer sections than the high level one. We should process the high level one first. Because there are less faculty who can teach a high level course. And almost all the faculty can teach the low level one. May be we can combine this attribute with the number of faculty that are available together to make the decision of which course to be processed first.

E : I think we can do this way. We start from the high level courses and within the high level courses we choose the one that has the fewest number of faculty that are available first.

K : .....


Course_1
Faculty 1
Course_3
Course_4
Faculty 2
Course_5
Course_6
Course_2

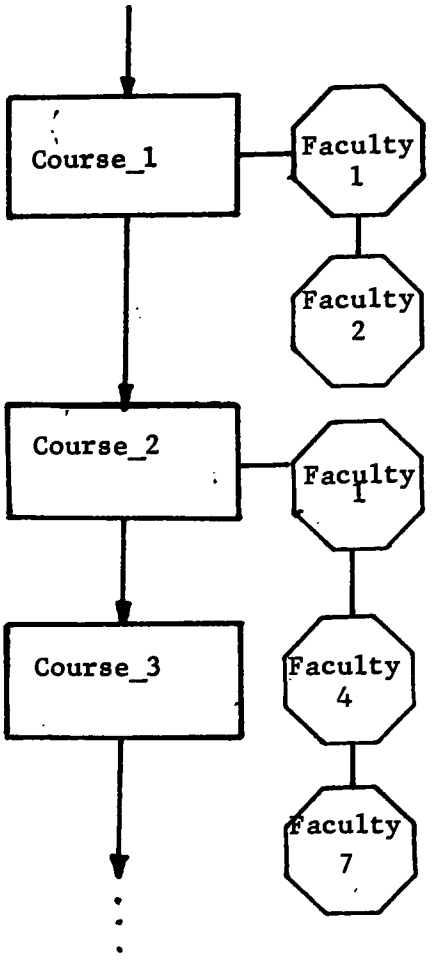

Course_1
Faculty 1
Faculty 2
Course_2
Faculty 1
Course_3
Faculty 4
Faculty 7

Figure A-1.

REFERENCE

(1) Addis, T.R. (1980). Towards an expert diagnostic system. ICL Technical Journal, May, pp. 79-105.

(2) Brachman, R.J. (1977). What's in a concept: Structural foundations for semantic networks. Int. J. Man-Mach. Stud. 9,127-152

(3) Buchanan, B.G. et al. (1976). Applications of Artificial Intelligence for chemical inference XXII, automatic rule formation in mass spectrometry by means of the Meta-DENDRAL program. J. Amer. Chem. Soc.,98 pp.6168-6178

(4) Buchanan, B.G. and Mitchell, T. (1978). Model-directed learning of production rules.

(5) D'Agapeyeff A, Expert Systems, Fifth Generation and UK Suppliers, NCCC Publications 1983

(6) Davis, R., Buchanan, B.G.;and Shortliffe,E.H.(1977). Production rules as a representation of a knowledge-based consultation program. Artificial Intelligence 8, pp 15-45.

(7) Duda, R., Gaschnig, J. and Hart, P. (1979). Model design in the PROSPECTOR consultant system for mineral exploration. ESMA, pp. 153-167

(8) Engelmore, R. and Terry, A. (1979) Structure and function of CRYSALIS system. IJCAI79, pp. 250-256

(9) Erman, L. D., F. Hayes-Roth, V. R. Lesser, & D. R. Reddy, "The hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty." Computing Surveys, Vol. 12, No. 2, June 1980

(10) Genesereth, M.R. (1979). The role of plans in automated consultation. IJCAI79. pp. 311-319

(11) Goldstein, I.P. and Roberts, B.(1979). Using frames in scheduling. In artificial intelligence : an MIT perspective 1, (Winston and Brown, eds.), MIT press.

(12) Lindsay, R.,Buchanan, B.G., Feigenbaum, E.A.,and Lederberg,J. (1980) "Applications of Artificial Intelligence for Organic Chemistry: The DENDRAL Project." McGraw-Hill, New York.

(13) McDermott, J. (1980) R1: a rule-based configurer for computer system. Computer Science Department. Carnegi-Mellon university.

(14) Minsky, M. (1975). A framework for representing knowledge. In "The Psychology of Computer Vision". (P. Winston, ed.), McGraw-Hill, New York

(15) Mitchell, T.M. (1979). Version Spaces: An approach to concept learning.

(16) Pereira, L. M., Sabatier, P., and d'Oliveira, E. 1982. ORBI - An expert system for environmental resource evaluation through natural language, Report FCT/DI-3/82, Department of Information, University Nova de Lisboa, Lisbon, Portugal.

(17) Pople, H.E., Myers, J.D. and Miller, R.A. (1975). DIALOG : a model of diagnostic logic for internal medicine. IJCAI75, pp.848-855.

(18) Quinlan, J.R., (1982) Fundamentals of the knowledge engineering problem. The Rand Corporation, Santa Monica, California.

(19) Shortliffe, E.H. (1976) Computer-Based Medical Consultations : MYCIN. New York, American Elsevier/North Holland.

(20) Simons, GL, (1983) Towards Fifth-Generation Computers, The National Computing Centre.

(21) Weiss, S., Kulikowski, C. (1977). A model-based consultation system for the long-term management of glaucoma. IJCAI77, pp. 826-833

(22) Weiss, S., Kulikowski, C. (1979). EXPERT : a system for developing consultation models. IJCAI79, pp. 942-47

(23) Winograd, T., (1975). "Frame Representations and the Declarative Procedural Controversy", in Bobrow, D. and Collins, A.(Eds.). Representation and Understanding, Academic Press, New York.

(24) Winston, P.H., (1983). Artificial Intelligence. Addison-Wesley Pub. com.