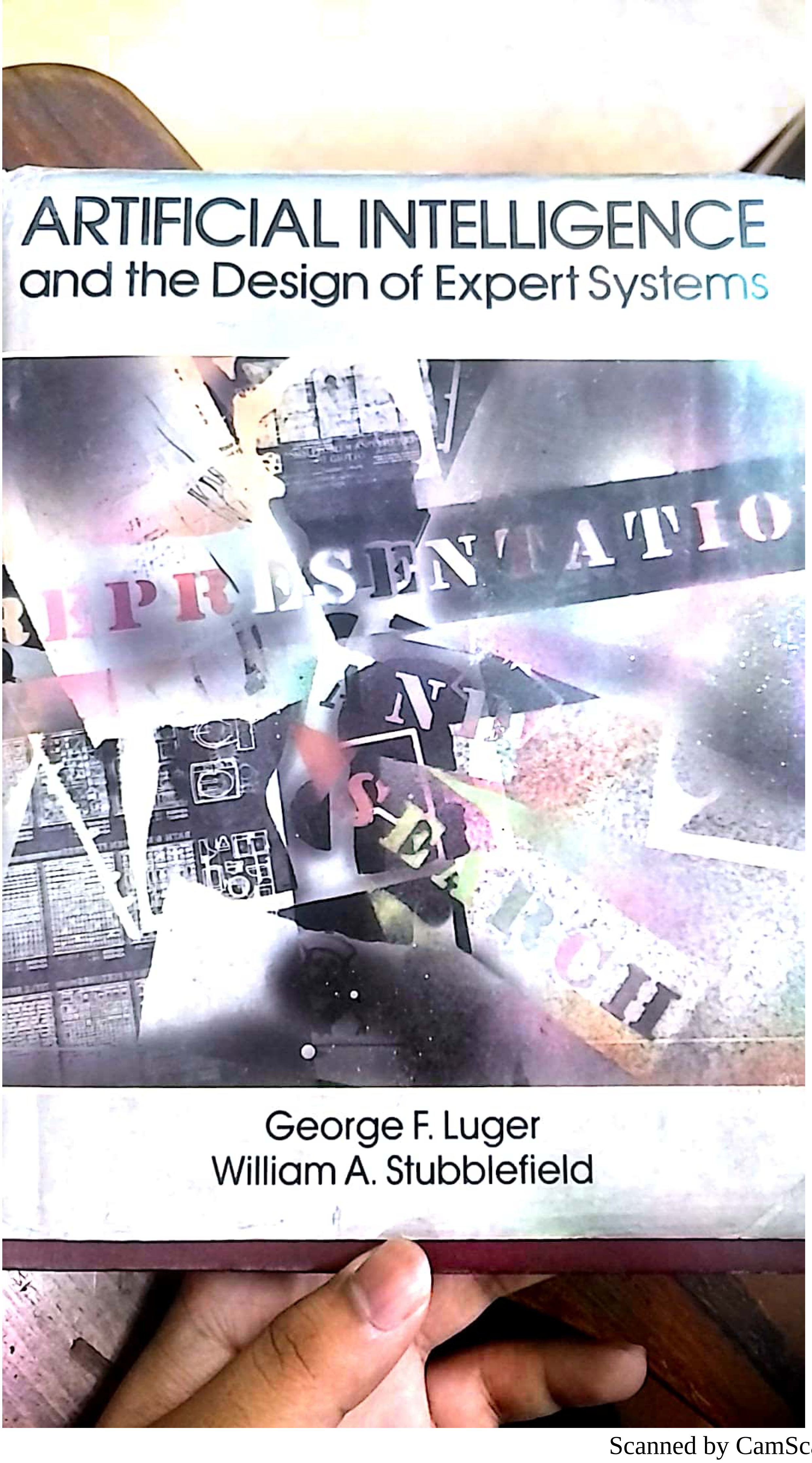


ARTIFICIAL INTELLIGENCE and the Design of Expert Systems

George F. Luger
William A. Stubblefield

8

RULE-BASED EXPERT SYSTEMS

The first principle of knowledge engineering is that the problem solving power exhibited by an intelligent agent's performance is primarily the consequence of its knowledge base, and only secondarily a consequence of the inference method employed. Expert systems must be knowledge-rich even if they are methods-poor. This is an important result and one that has only recently become well understood in AI. For a long time AI has focused its attentions almost exclusively on the development of clever inference methods; almost any inference method will do. The power resides in the knowledge.

—Edward Feigenbaum, Stanford University

I hate to criticize, Dr Davis, but did you know that
most rules about what the category of
an organism might be
that mention:
    the site of a culture and
    the infection
also mention:
    the portal of entry of the organism?
Shall I try to write a clause to account for this?

—The program Teiresias helping a Stanford physician improve the MYCIN knowledge base

8.0 Introduction

An *expert system* is a knowledge-based program that provides "expert quality" solutions to problems in a specific domain. Generally, its knowledge is extracted from human experts in the domain and it attempts to emulate their methodology and performance. As with skilled humans, expert systems tend to be specialists, focusing on a narrow set of problems. Also, like humans, their knowledge is both theoretical and practical, having been perfected through experience in the domain. Unlike a human being, however, current programs cannot learn from their own experience; their knowledge must be extracted from humans and encoded in a formal language. This is the major task facing expert system builders.

Expert systems should not be confused with cognitive modeling programs, which attempt to simulate human mental architecture in detail. These are discussed in Chapter 15. Expert systems neither copy the structure of the human mind nor are mechanisms for general intelligence. They are practical programs that use heuristic strategies developed by humans to solve specific classes of problems.

Because of the heuristic, knowledge-intensive nature of expert level problem solving, expert systems are generally:

1. Open to inspection, both in presenting intermediate steps and in answering questions about the solution process.
2. Easily modified, both in adding and in deleting skills from the knowledge base.
3. Heuristic, in using (often imperfect) knowledge to obtain solutions.

An expert system is "open to inspection" in that the user may, at any time during program execution, inspect the state of its reasoning and determine the specific choices and decisions that the program is making. There are several reasons why this is desirable: first, if a human expert such as a doctor or an engineer is to accept a recommendation from the computer, they must be satisfied the solution is correct. "The computer did it" is not sufficient reason to follow its advice. Indeed, few human experts will accept advice from another human without understanding the reasons for that advice. This need to have answers explained is more than mistrust on the part of users; explanations help people relate the advice to their existing understanding of the domain and apply it in a more confident and flexible manner.

Second, when a solution is open to inspection, we can evaluate every aspect and decision taken during the solution process, allowing for partial agreement and the addition of new information or rules to improve performance. This plays an essential role in the refinement of a knowledge base.

The exploratory nature of AI and expert system programming requires that programs be easily prototyped, tested, and changed. These abilities are supported by AI programming languages and environments and the use of good programming techniques by the designer. In a pure production system, for example, the modification of a single rule has no global syntactic side effects. Rules may be added or removed without requiring further changes to the larger program. Expert system designers have commented that easy modification of the knowledge base is a major factor in producing a successful program (McDermott 1981).

The third feature of expert systems is their use of heuristic problem-solving methods. As expert system designers have discovered, informal "tricks of the trade" and "rules of thumb" are often more important than the standard theory presented in textbooks and classes. Sometimes these rules augment theoretical knowledge; occasionally they are simply shortcuts that seem unrelated to the theory but have been shown to work.

The heuristic nature of expert problem-solving knowledge creates problems in the evaluation of program performance. Although we know that heuristic methods will occasionally fail, it is not clear exactly how often a program must be correct in order to be accepted: 98% of the time? 90%? 80%? Perhaps the best way to evaluate a program is to compare its results to those obtained by human experts in the same area. This suggests a variation of the Turing test (Chapter 1) for evaluating the performance of expert systems: a program has achieved expert level performance if people working in the area could not differentiate, in a blind evaluation, between the best human efforts and those of the program. In evaluating the MYCIN program for diagnosing meningitis infections, Stanford researchers had a number of infectious-disease experts blindly evaluate the performance of both MYCIN and human specialists in infectious diseases. Similarly, Digital Equipment Corporation decided that XCON, a program for configuring VAX computers, was ready for commercial use when its performance was comparable to that of human engineers.

Expert systems have been built to solve a range of problems in domains such as medicine, mathematics, engineering, chemistry, geology, computer science, business, law, defense, and education. These programs have addressed a wide range of problem types; the following list, adapted from Waterman (1986), is a useful summary of general expert system problem categories. Another excellent survey of expert system applications is Smart and Langeland-Knudsen (1986).

1. *Interpretation*—forming high-level conclusions or descriptions from collections of raw data.
2. *Prediction*—projecting probable consequences of given situations.
3. *Diagnosis*—determining the cause of malfunctions in complex situations based on observable symptoms.
4. *Design*—determining a configuration of system components that meets certain performance goals while satisfying a set of constraints.
5. *Planning*—devising a sequence of actions that will achieve a set of goals given certain starting conditions.
6. *Monitoring*—comparing the observed behavior of a system to its expected behavior.
7. *Debugging and Repair*—prescribing and implementing remedies for malfunctions.
8. *Instruction*—detecting and correcting deficiencies in students' understanding of a subject domain.
9. *Control*—governing the behavior of a complex environment.

8.1 Overview of Expert System Technology

8.1.1 Design of Rule-Based Expert Systems

Figure 8.1 shows the most important modules that make up a rule-based expert system. The user interacts with the expert system through a user interface that makes access more comfortable for the human and hides much of the system complexity (e.g., the

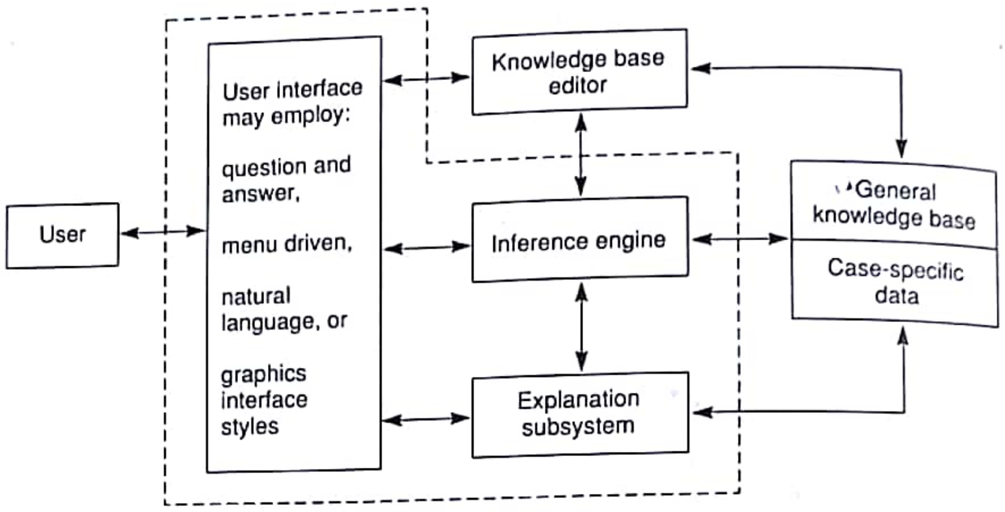


Figure 8.1 Architecture of a typical expert system.

internal structures of the rule base). Expert systems employ a variety of interface styles, including question and answer, menu-driven, natural language, or graphics interfaces.

The program must keep track of *case-specific data*, the facts, conclusions, and other relevant information of the case under consideration. This includes the data given in a problem instance, partial conclusions, confidence measures of conclusions, and dead ends in the search process. This information is separate from the general knowledge base.

The *explanation subsystem* allows the program to explain its reasoning to the user. These explanations include justifications for the system's conclusions (*how queries*, as discussed in Section 8.2.2), explanations of why the system needs a particular piece of data (*why queries*), and, in some experimental systems, tutorial explanations or deeper theoretical justifications of the program's actions.

Many systems also include a *knowledge base editor*. Knowledge base editors can access the explanation subsystem and help the programmer locate bugs in the program's performance. They also may assist in the addition of new knowledge, help maintain correct rule syntax, and perform consistency checks on the updated knowledge base. An example of the Teiresias knowledge base editor is presented in Section 8.4.5.

The heart of the expert system is the general knowledge base, which contains the problem-solving knowledge of the particular application. In a rule-based expert system this knowledge is represented in the form of *if . . . then* rules, as in the auto battery/cable example of Section 8.2. (Other ways of representing this knowledge are discussed in chapters 9 and 14.)

The *inference engine* applies the knowledge to the solution of actual problems. It is the interpreter for the knowledge base. In the production system, the inference engine performs the recognize-act control cycle. The procedures that implement the control cycle are separate from the production rules themselves. It is important to maintain this separation of knowledge base and inference engine for several reasons:

1. The separation of the problem-solving knowledge and the inference engine makes it possible to represent knowledge in a more natural fashion. "If . . . then" rules, for example, are closer to the way in which human beings describe their own problem-solving techniques than a program that embeds this knowledge in lower-level computer code.
2. Because the knowledge base is separated from the program's lower-level control structures, expert system builders can focus directly on capturing and organizing problem-solving knowledge rather than on the details of its computer implementation.
3. The separation of knowledge and control, along with the modularity of rules and other representational structures used in building knowledge bases, allows changes to be made in one part of the knowledge base without creating side effects in other parts of the program code.
4. The separation of the knowledge and control elements of the program allows the same control and interface software to be used in a variety of systems. The *expert system shell* has all the components of Figure 8.1 except that the knowledge base (and, of course, the case-specific data) contains no information. Programmers can use the "empty shell" and create a new knowledge base appropriate to their application. The broken lines of Figure 8.1 indicate the shell modules.
5. As illustrated in the discussion of production systems (Chapter 4), this modularity allows us to experiment with alternative control regimes for the same rule base.

The use of an expert system shell can reduce the design and implementation time of a program considerably. As may be seen in Figure 8.4, MYCIN was developed in about 20 person-years. EMYCIN (Empty MYCIN) is a general expert system shell that was produced by removing the specific domain knowledge from the MYCIN program. Using EMYCIN, knowledge engineers implemented PUFF, a program to analyze pulmonary problems in patients, in about 5 person-years. This is a remarkable saving and an important aspect of the commercial viability of expert system technology. Expert system shells have become increasingly common, with commercially produced shells available for all classes of computers.

It is important that the programmer choose the proper expert system shell. Different problems often require different reasoning processes: goal-driven rather than data-driven search, for instance. The control strategy provided by the shell must be appropriate to the new application. The medical reasoning in the PUFF application was much like that of the original MYCIN work; this made the use of the EMYCIN shell appropriate. If the shell does not support the appropriate reasoning processes, its use can be a mistake and worse than starting from nothing. As we shall see, part of the responsibility of the expert system builder is to correctly characterize the reasoning processes required for a given problem domain and to either select or construct an inference engine that implements these structures.

Unfortunately, shell programs do not solve all of the problems involved in building expert systems. While the separation of knowledge and control, the modularity of the production system architecture, and the use of an appropriate knowledge representation language all help with the building of an expert system, the acquisition and organization of the knowledge base remain difficult tasks.

8.1.2 Selecting a Problem for Expert System Development

Expert systems tend to involve a considerable investment in money and human effort. Attempts to solve a problem that is too complex, poorly understood, or otherwise unsuited to the available technology can lead to costly and embarrassing failures. Researchers have developed an informal set of guidelines for determining whether a problem is appropriate for expert system solution:

1. *The need for the solution justifies the cost and effort of building an expert system.* For example, Digital Equipment Corporation had experienced considerable financial expense because of errors in configurations of VAX and PDP-11 computers. If a computer is shipped with missing or incompatible components, the company is obliged to correct this situation as quickly as possible, often incurring added shipping expense or absorbing the cost of parts not taken into account when the original price was quoted. Because this expense was considerable, DEC was extremely interested in automating the configuration task; the resulting system, XCON, has paid for itself in both financial savings and customer goodwill. Similarly, many expert systems have been built in domains such as mineral exploration, business, defense, and medicine where there is a large potential for savings in either money, time, or human life. In recent years, the cost of building expert systems has gone down as software tools and expertise in AI have become more available. The range of potentially profitable applications has grown correspondingly.

2. *Human expertise is not available in all situations where it is needed.* Much expert system work has been done in medicine, for example, because the specialization and technical sophistication of modern medicine have made it difficult for doctors to keep up with advances in diagnostics and treatment methods. Specialists with this knowledge are rare and expensive, and expert systems are seen as a way of making their expertise available to a wider range of doctors. In geology, there is a need for expertise at remote mining and drilling sites. Often, geologists and engineers find themselves traveling large distances to visit sites, with resulting expense and wasted time. By placing expert systems at remote sites, many problems may be solved without needing a visit by a human expert. Similarly, loss of valuable expertise through employee turnover or pending retirement may justify building an expert system.

3. *The problem may be solved using symbolic reasoning techniques.* This means that the problem should not require physical dexterity or perceptual skill. Although robots and vision systems are available, they currently lack the sophistication and flexibility of human beings. Expert systems are generally restricted to problems that humans can solve through symbolic reasoning.

4. *The problem domain is well structured and does not require commonsense reasoning.* Although expert systems have been built in a number of areas requiring specialized technical knowledge, more mundane commonsense reasoning is well beyond our current capabilities. Highly technical fields have the advantage of being well studied

and formalized; terms are well defined and domains have clear and specific conceptual models. Most significantly, however, the amount of knowledge required to solve such problems is small in comparison to the amount of knowledge used by human beings in commonsense reasoning.

5. *The problem may not be solved using traditional computing methods.* Expert system technology should not be used to "reinvent the wheel." If a problem can be solved satisfactorily using more traditional techniques such as numerical, statistical, or operations research techniques, then it is not a candidate for an expert system. Because expert systems rely on heuristic approaches, it is unlikely that an expert system will outperform an algorithmic solution if such a solution exists.

6. *Cooperative and articulate experts exist.* The knowledge used by expert systems is often not found in textbooks but comes from the experience and judgment of humans working in the domain. It is important that these experts be both willing and able to share that knowledge. This implies that the experts should be articulate and believe that the project is both practical and beneficial. If, on the other hand, the experts feel threatened by the system, fearing that they may be replaced by it or that the project can't succeed and is therefore a waste of time, it is unlikely that they will give it the necessary time and effort. It is also important that management be supportive of the project and allow the domain experts to take time away from their usual responsibilities to work with the program implementers.

7. *The problem is of proper size and scope.* It is important that the problem not exceed the capabilities of current technology. For example, a program that attempted to capture all of the expertise of a medical doctor would not be feasible; a program that advised MDs on the use of a particular piece of diagnostic equipment would be more appropriate. As a rule of thumb, problems that require days or weeks for human experts to solve are probably too complex for current expert system technology.

Although a large problem may not be amenable to expert system solution, it may be possible to break it into smaller, independent subproblems that are. Or we may be able to start with a simple program that solves a portion of the problem and gradually increase its functionality to handle more of the problem domain. This was done successfully in the creation of XCON: initially the program was designed only to configure VAX 780 computers; later it was expanded to include the full VAX and PDP-11 series (Bachant and McDermott 1984).

8.1.3 Overview of "Knowledge Engineering"

The primary people involved in building an expert system are the *knowledge engineer*, the *domain expert*, and the *end user*.

The knowledge engineer is the AI language and representation expert. His or her main task is to select the software and hardware tools for the project, help extract the necessary knowledge from the domain expert, and implement that knowledge in a correct and efficient knowledge base. The knowledge engineer may initially be ignorant of the application domain.

The domain expert provides the knowledge of the problem area. The domain expert is generally someone who has worked in the domain area and understands its problem-in reducing the design time. This is perhaps the most important reason for current commercial successes.

8.2 A Framework for Organizing and Applying Human Knowledge

8.2.1 Production Systems, Rules, and the Expert System Architecture

The architecture of rule-based expert systems may be understood in terms of the production system model for problem solving presented in Part II. In fact, the parallel between the two entities is more than a simple analogy: the production system was the intellectual precursor of modern expert system architecture. This is not surprising; when Newell and

Simon began developing the production system model, their goal was to find a way to model human problem solving.

If we regard the expert system architecture in Figure 8.1 as a production system, the knowledge base is the set of production rules. The expertise of the problem area is represented by the productions. In a rule-based system, these condition-action pairs are represented as rules, with the premises of the rules (the if portion) corresponding to the condition and the conclusion (the then portion) corresponding to the action. Case-specific data are kept in the working memory. Finally, the inference engine is the recognize-act cycle of the production system. This control may be either data driven or goal driven.

In a goal-driven expert system, the goal expression is initially placed in working memory. The system matches rule conclusions with the goal, selecting one rule and placing its *premises* in the working memory. This corresponds to a decomposition of the problem into simpler subgoals. The process continues, with these premises becoming the new goals to match against rule conclusions. The system thus works back from the original goal until all the subgoals in working memory are known to be true, indicating that the hypothesis has been verified. Thus, backward search in an expert system corresponds roughly to the process of hypothesis testing in human problem solving.

In an expert system, subgoals are often solved by asking the user for information. Some expert system shells allow the system designer to specify which subgoals may be solved by asking the user. Other inference engines simply ask about all subgoals that fail to match with the conclusion of some rule in the knowledge base; i.e., if the program cannot infer the truth of a subgoal, it asks the user.

Many problem domains seem to lend themselves more naturally to forward search. In an interpretation problem, for example, most of the data for the problem are initially given and it is often difficult to formulate a hypotheses (goal). This suggests a forward reasoning process in which the facts are placed in working memory and the system searches for an interpretation in a forward fashion.

As a more detailed example, let us create a small expert system for diagnosing automotive problems:

Rule 1:

if

the engine is getting gas, and

the engine will turn over,

then

the problem is spark plugs.

Rule 2:

if

the engine does not turn over, and

the lights do not come on

then

the problem is battery or cables.

Rule 3:
if
the engine does not turn over, and
the lights do come on
then
the problem is the starter motor.

Rule 4:
if
there is gas in fuel tank, and
there is gas in carburetor
then
the engine is getting gas.

To run this knowledge base under a goal-directed control regime, place the top-level goal, the problem is X, in working memory as in Figure 8.5. X is a variable that can match with any phrase; it will become bound to the solution when the problem is solved.

There are three rules that match with the expression in working memory, rule 1, rule 2, and rule 3. If we resolve conflicts in favor of the lowest-numbered rule, then rule 1 will fire. This causes X to be bound to the value spark plugs and the premises of rule 1 to be placed in the working memory as in Figure 8.6. The system has thus chosen to explore the possible hypothesis that the spark plugs are bad. Another way to look at this is that the system has selected an or branch in an and/or graph (Chapter 3).

Note that there are two premises to rule 1, both of which must be satisfied to prove the conclusion true. These are and branches of the search graph representing a decomposition of the problem (finding if the spark plugs are bad) into two subproblems (finding

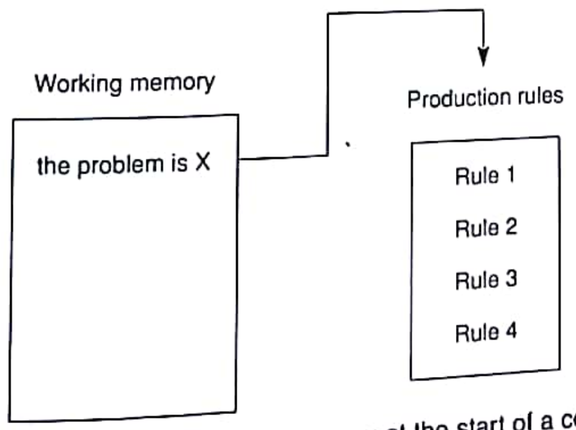


Figure 8.5 Working memory at the start of a consultation in the car diagnostic example.

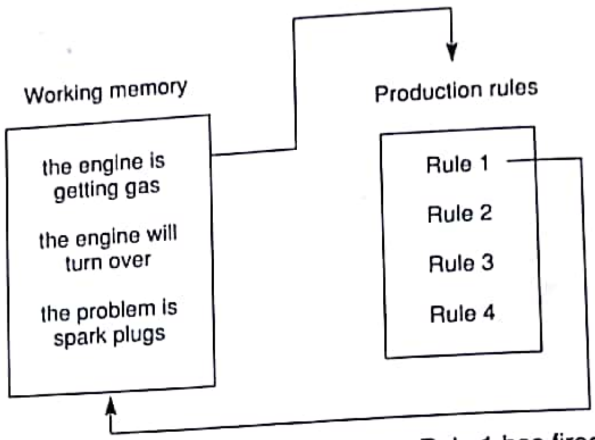

Working memory
the engine is getting gas
the engine will turn over
the problem is spark plugs
Production rules
Rule 1
Rule 2
Rule 3
Rule 4

Figure 8.6 Working memory after Rule 1 has fired.

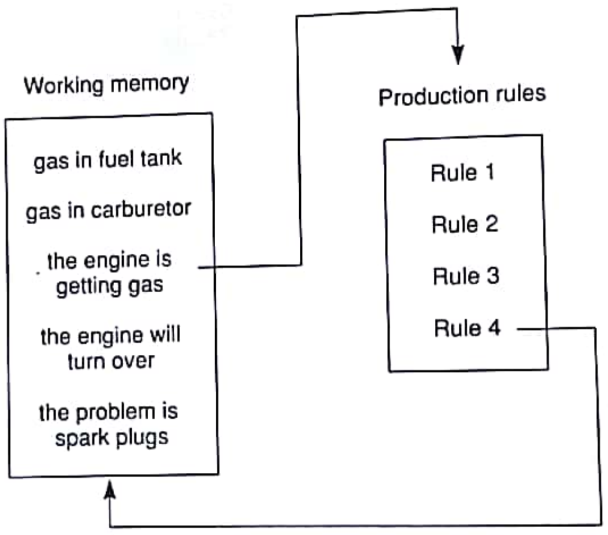
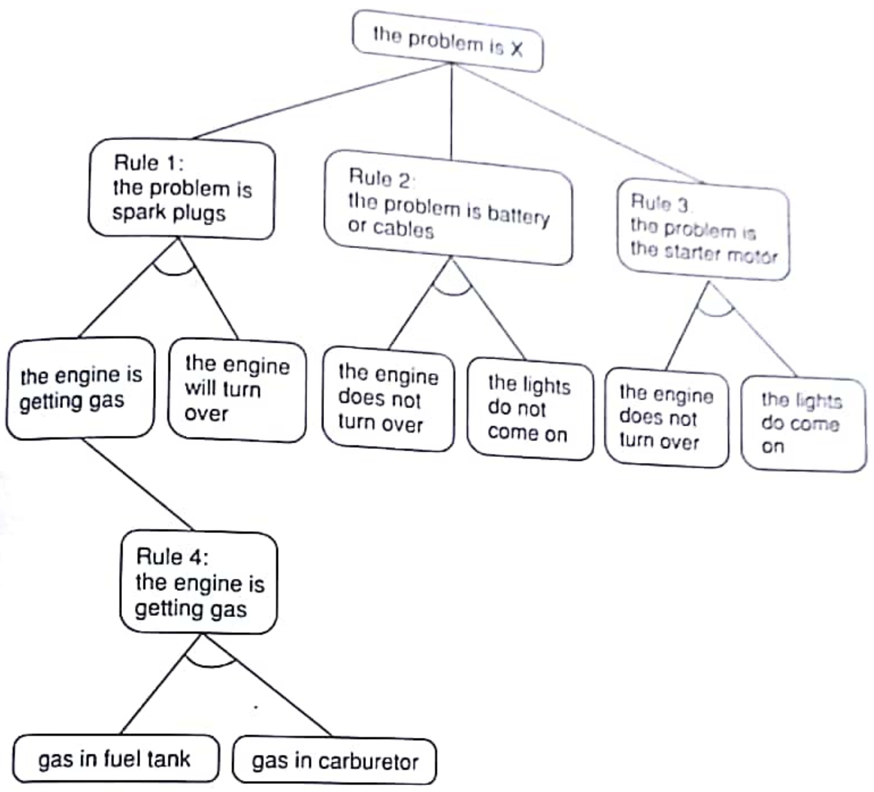


Figure 8.7 Working memory after Rule 4 has fired.

if the engine is getting gas and if the engine is turning over). We may then fire rule 4, whose conclusion matches with "engine is getting gas," causing its premises to be placed in working memory as in Figure 8.7.

At this point, there are three entries in working memory that do not match with any rule conclusions. Our expert system will query the user directly about these subgoals. If the user confirms all three of these as true, the expert system will have successfully determined that the car will not start because the spark plugs are bad. In finding this solution, the system has searched the and/or graph presented in Figure 8.8.

This is, of course, a very simple example. Not only is its automotive knowledge limited at best, but also a number of important aspects of real implementations are



the problem is X
Rule 1: the problem is spark plugs
Rule 2: the problem is battery or cables
Rule 3: the problem is the starter motor
the engine is getting gas
the engine will turn over
the engine does not turn over
the lights do not come on
the engine does not turn over
the lights do come on
Rule 4: the engine is getting gas
gas in fuel tank
gas in carburetor

Figure 8.8 And/or graph searched in the car diagnostic example.

ignored. The rules are phrased in English, rather than a formal language. On finding a solution, a real expert system will tell the user its diagnosis (our model simply stops). Also, we should maintain enough of a trace of the reasoning to allow backtracking if necessary; in our example, had we failed to determine that the spark plugs were bad, we would have needed to back up to the top level and try rule 2 instead. Notice that this information is implicit in the ordering of subgoals in working memory of Figure 8.7 and in the graph of Figure 8.8. This example of expert systems underscores their foundation in the production system and and/or graph search models of earlier chapters.

At the beginning of this chapter, we emphasized that an expert system needed to be open to inspection, easily modified, and heuristic in nature. The production system architecture is an important factor in each of these requirements. Ease of modification, for example, is made possible by the syntactic independence of production rules: each rule is a chunk of knowledge that can be independently modified. The heuristic nature of these rules is in the knowledge obtained from the domain expert and, as seen in Section 8.3, from use of confidence measures. The next section discusses the way in which this architecture supports explanations and inspection of the reasoning process.

8.2.2 Explanation and Transparency

The production system implements a graph search; the explanation routines of Figure 8.1 keep track of the graph search and use a trace of the search to answer user queries. The production rules themselves document each step of the reasoning process.

In general, the two questions supported by rule-based expert systems are why? and how? In a why query, the user has been asked for information by the program and responds with why, meaning "why did you ask that?" The answer is the current rule that the production system is attempting to fire. The answer to "how did you get that result?" is the sequence of rules that were used to conclude a goal.

Suppose our automotive expert system has an explanation facility and assume that the consultation has reached the stage represented by the working memory configuration in Figure 8.7. The following dialogue begins with the computer asking the user about the goals present in working memory. User responses are in this type.

gas in fuel tank?
yes

gas in carburetor?
yes

engine will turn over?
why

Here the user enters a why query, i.e., why do you want to know that?

It has been established that:

1. engine is getting gas,

therefore if

2. engine will turn over,

then the problem is spark plugs.

Since the user answered yes to the first two questions, rule 4 was able to fire, verifying the first premise of rule 1. When the why query was made, the system was trying to solve the second premise of rule 1. Note that the explanation is just a restatement of the current rule, but it provides an adequate explanation for why the program wants to know if the engine will turn over. Continuing with the dialogue:

how engine is getting gas

Here the user asks how the program knew that the engine is getting gas. The response is a trace of the reasoning that led to this conclusion, working back from the goal along the rules that support it to the user responses:

This follows from Rule 4:

if

gas in fuel tank, and

gas in carburetor

then

engine is getting gas.

gas in fuel tank was given by the user

gas in carburetor was given by the user

This simple example illustrates the explanations that expert systems provide the user. Why queries are answered by showing the current rule that the system is attempting to fire. How queries are answered by giving a trace of the reasoning that led to a goal. Although the mechanisms are conceptually simple, they can exhibit remarkable explanatory power if the knowledge base is organized in a logical fashion.

The production system architecture provides an essential basis for these explanations. Each cycle of the control loop selects and fires another rule. The program may be stopped after each cycle and inspected. Since each rule represents a complete chunk of problem-solving knowledge, the current rule provides a context for the explanation. Contrast this with more traditional program architectures: if we stop a Pascal or FORTRAN program in midexecution, it is doubtful that the current statement will have much meaning.

If explanations are to behave logically, it is important not only that the knowledge base get the correct answer but also that each rule correspond to a single logical step in the problem-solving process. If a knowledge base combines several steps into a single rule or if it breaks up the rules in an arbitrary fashion, it may get correct answers but seem arbitrary and illogical in responding to how and why queries. This not only undermines the user's faith in the system but also makes the program much more difficult for its builders to understand and modify.

8.2.3 Heuristics and Control in Expert Systems