COMPUTER ENGINEERING HANDBOOK

C. H. Chen, Editor-in-Chief
Department of Electrical and Computer Engineering
University of Massachusetts Dartmouth

McGRAW-HILL, INC.
New York St. Louis San Francisco Auckland Bogotá
Caracas Lisbon London Madrid Mexico Milan
Montreal New Delhi Paris San Juan São Paulo
Singapore Sydney Tokyo Toronto
1992

CHAPTER 9
RULE-BASED EXPERT SYSTEMS

George F. Luger and William A. Stubblefield
Department of Computer Science
University of New Mexico, Albuquerque

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 rule to account for this?

TEIRESIAS program helping a Stanford physician improve MYCIN knowledge base

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 a 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. Expert systems neither copy the

This chapter is adapted from George F. Luger and William A. Stubblefield, *Artificial Intelligence and the Design of Expert Systems*. Palo Alto, Calif.: Benjamin Cummings, 1989, with permission.

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, in both adding and 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 human experts such as doctors or engineers are 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 *artificial intelligence* (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, e.g., 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 [17].

Third, expert systems use 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 percent of the time? 90 percent? 80 percent? 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 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 disease (see Sec. 9.4.4). 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 [23], is a useful summary of general expert-system problem categories. Another excellent survey of expert-system applications is by Smart and Langeland-Knudsen [21].

1. *Interpretation:* Formation of high-level conclusions or descriptions from collections of raw data
2. *Prediction:* Projection of probable consequences of given situations
3. *Diagnosis:* Determination of the cause of malfunctions in complex situations based on observable symptoms
4. *Design:* Determination of 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 of the behavior of a complex environment

9.1 OVERVIEW OF EXPERT-SYSTEM TECHNOLOGY

9.1.1 Design of Rule-Based Expert Systems

Figure 9.1 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 internal

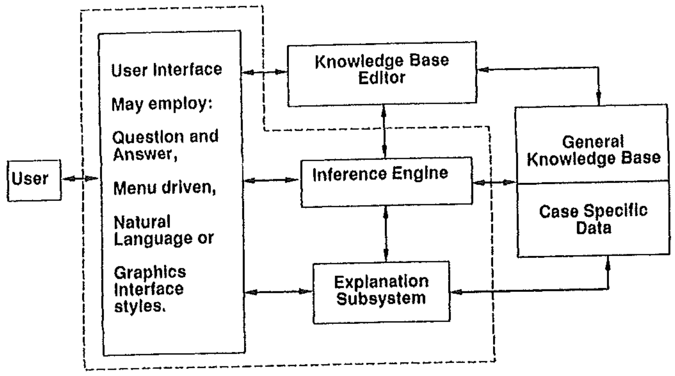

FIGURE 9.1 Architecture of a typical expert system.

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 Sec. 9.2.2), explanations of why the system needs a particular piece of data (why queries, as in Sec. 9.2.2), and, in some experimental systems, tutorial explanations 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 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 Sec. 9.4.5.

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 automobile battery cable example of Sec. 9.2.

The *inference engine* applies the knowledge to the solution of actual problems. It is the interpreter for the knowledge base. In the production system [16, chap. 4], 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, e.g., 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 Fig. 9.1 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 applications. The broken lines of Fig. 9.1 indicate the shell modules.
5. The modularity of the inference process also 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 Fig. 9.4, 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 savings 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 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.

9.1.2 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 (DEC) 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 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, e.g., 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 by 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 by using traditional computing methods.* Expert-system technology should not be used to reinvent the wheel. If a problem can be solved satisfactorily by 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, however, the experts feel threatened by the system, fearing that they may be replaced by it or that the project cannot 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 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 [2].

9.1.3 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-solving techniques, including the use of shortcuts, handling imprecise data, evaluating partial solutions, and all the other skills that mark a person as an expert. The domain expert is primarily responsible for spelling out these skills to the knowledge engineer.

As in most applications, the end user determines the major design constraints. Unless the user is happy, the development effort is wasted by and large. The skills and needs of the user must be considered throughout the design cycle: What level of explanation does the user need? Can the user provide correct information to the system? Is the user interface appropriate? Are there environmental restrictions on the program's use? An interface that required typing, e.g., would not be appropriate for use in the cockpit of a jet fighter. Will the program make the user's work easier, quicker, more comfortable?

Like most AI programming, expert-system development requires a nontraditional development cycle based on early prototyping and incremental revision of the code. This exploratory programming methodology is complicated by the interaction between the knowledge engineer and the domain expert.

Generally, work on the system begins with the knowledge engineers attempting to gain some familiarity with the problem domain. This helps in communicating with the domain expert. This is done in initial interviews with the expert or by reading introductory texts in the domain area. Next, the knowledge engineer and expert begin the process of extracting the expert's problem-solving knowledge. This is often done by giving the domain expert a series of sample problems and having her or him explain the techniques used in the solution.

Here, it is often useful for the knowledge engineer to be a novice in the problem domain. Human experts are notoriously unreliable in explaining exactly what goes on in solving a complex problem. Often they forget to mention steps that have become obvious or even automatic to them after years of work in the field. Knowledge engineers, by virtue of their relative naivete in the domain, should be able to spot these conceptual jumps and ask for clarification.

Once the knowledge engineer has obtained a general overview of the problem domain and gone through several problem-solving sessions with the domain expert, he or she is ready to begin actual design of the system: selecting a way to represent the knowledge, such as rules or frames; determining the basic search strategy, such as forward or backward; and designing the user interface. After making these design commitments, the knowledge engineer builds a prototype.

This prototype should be able to solve problems in a small area of the domain and provides a test bed for preliminary design assumptions. Once the prototype has been implemented, the knowledge engineer and domain expert test and refine its knowledge by giving it problems to solve and correcting its shortcomings. Should the assumptions made in designing the prototype prove correct, it can be built into a complete system.

Expert systems are built by progressive approximations, with the program's mistakes leading to corrections or additions to the knowledge base. In a sense, the knowledge base is "grown" rather than constructed. This approach to programming was proposed by Seymour Papert with his LOGO language [19]. The LOGO philosophy argues that watching the computer respond to the improperly formulated ideas represented by the code leads to their correction (being debugged) and clarification with more precise code. Other researchers have verified this notion that design proceeds through a process of trying and correcting candidate designs, rather than by such neatly hierarchical processes such as top-down design.

It is also understood that the prototype may be thrown away if it becomes too cumbersome or if the designers decide to change their basic approach to the problem. The prototype lets program builders explore the problem and its important relationships by actually constructing a program to solve it. After this progressive clarification is complete, they can often write a cleaner version, usually with fewer actual rules.

The second major feature of expert system programming is that the program need never be considered "finished." A large heuristic knowledge base will always have limitations. The modularity and ease of modification available in the production system model make it natural to add new rules or make up for the shortcomings of the present rule base at any time. For example, DEC has continued to add rules to the XCON program to extend its capabilities to the rest of their product line. In 1981, XCON had about 500 rules and could configure the Vax 780; it was progressively refined until, today with over 10,000 rules, it configured most of the DEC product line.

Furthermore, as DEC changed the specifications for its computers, previously correct rules needed updating. One report noted that up to 50 percent of the XCON rules were changed each year just to keep up with changes in the product line.

Figure 9.2 presents a schematic overview of the development cycle of expert-system software.

Even though expert systems of commercial quality have been available since 1980, research in the development of expert systems has gone on since the mid-1960s. In fact, the emergence of commercially viable expert systems in the present world market has been a direct result of this earlier research. Figure 9.3 shows the history of the programs now considered classic in this area.

The major development activity for these early programs took place at three universities: Stanford, Massachusetts Institute of Technology, and Carnegie Mellon. The main research and development thrust for the rule-based expert system was at Stanford, under

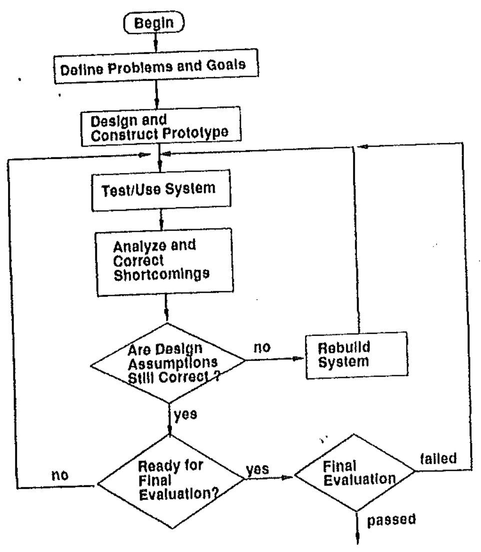
Begin
Define Problems and Goals
Design and Construct Prototype
Test/Use System
Analyze and Correct Shortcomings
Are Design Assumptions Still Correct?
no
Rebuild System
yes
Ready for Final Evaluation?
no
yes
Final Evaluation
failed
passed

FIGURE 9.2 Exploratory development cycle for rule-based expert systems.

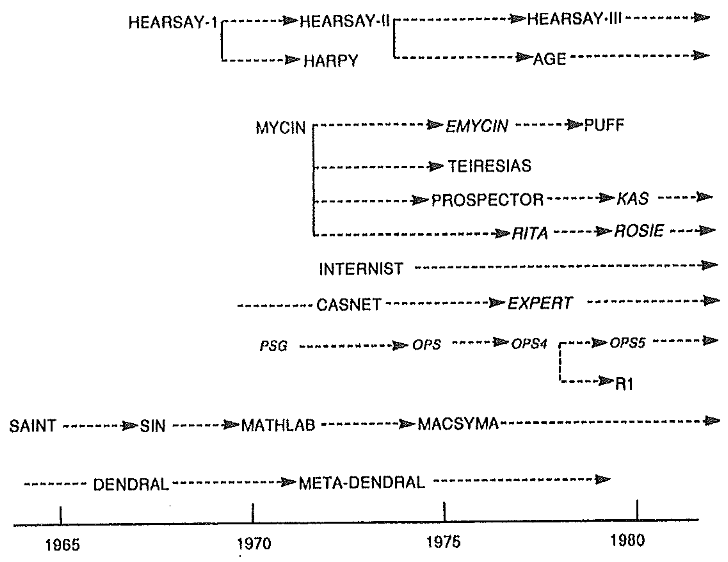

FIGURE 9.3 Development history for some classic expert systems. *(From P. H. Winston and K. A. Prendergast, eds., The AI Business. Cambridge, Mass.: MIT Press, 1984. Reprinted by permission.)*

the direction of Edward Feigenbaum, although other work in the area certainly took place at, for instance, Rutgers and the University of Pittsburgh. The early development of the production system model for problem solving, including the development of the OPS languages, took place primarily at Carnegie Mellon University.

Figure 9.4 presents another important aspect of the evolution of expert-system programs: The average development time for expert systems has been drastically reduced across the decades of evolution. The emergence of expert-system shells was instrumental in reducing the design time. This is perhaps the most important reason for current commercial successes.

9.2 A FRAMEWORK FOR ORGANIZING HUMAN KNOWLEDGE

9.2.1 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 [15]. 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 Fig. 9.1 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

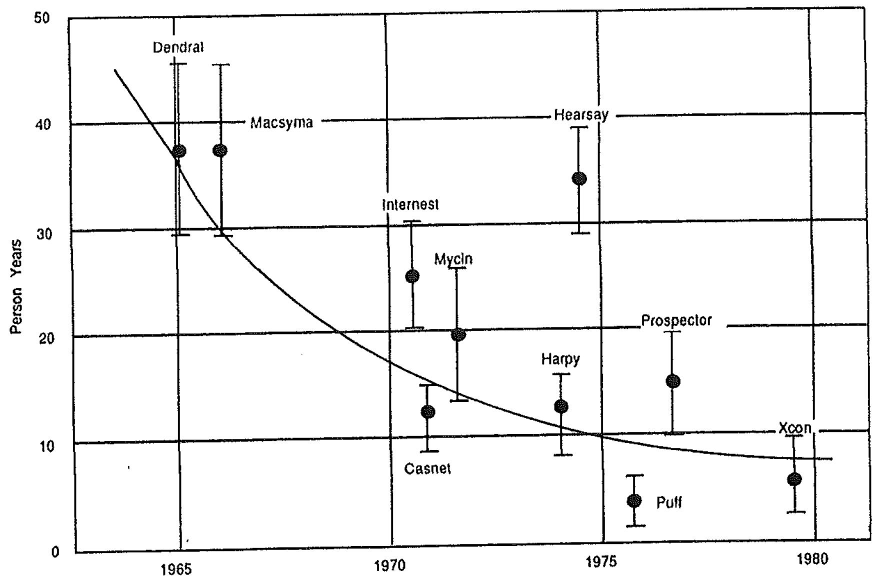

FIGURE 9.4 Development time for selected expert systems between 1965 and 1985. *(From F. Hayes-Roth, D. Waterman, and D. Lenat, Building Expert Systems. Reading, Mass.: Addison-Wesley, 1984. Reprinted by permission.)*

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, e.g., most of the data for the problem are initially given, and it is often difficult to formulate a hypothesis (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 the fuel tank and
there is gas in the 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 Fig. 9.5. Here *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 Fig. 9.6. 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 (Fig. 9.8).

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 decompo-

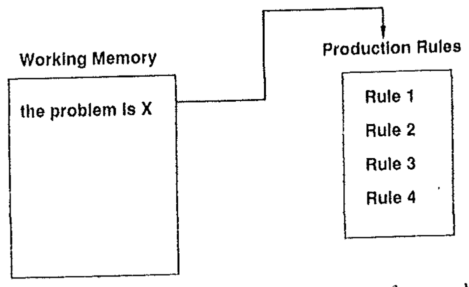

FIGURE 9.5 Working memory at the start of a consultation in the car diagnostic example.

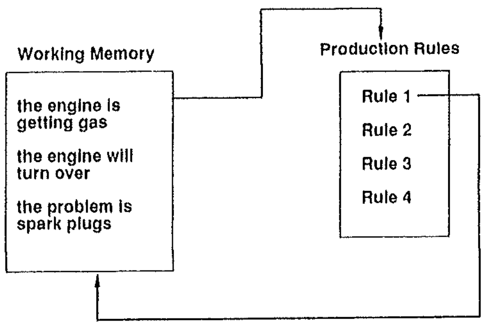
Working Memory
the engine is getting gas
the engine will turn over
the problem is spark plugs
Production Rules
Rule 1
Rule 2
Rule 3
Rule 4

FIGURE 9.6 Working memory after rule 1 has fired.

sition of the problem (finding if the spark plugs are bad) into two subproblems—finding 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 Fig. 9.7.

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 Fig. 9.8.

This is, of course, a very simple example. Not only is its automotive knowledge limited at best, but a number of important aspects of real implementations are 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 Fig. 9.7 and in the graph of Fig. 9.8.

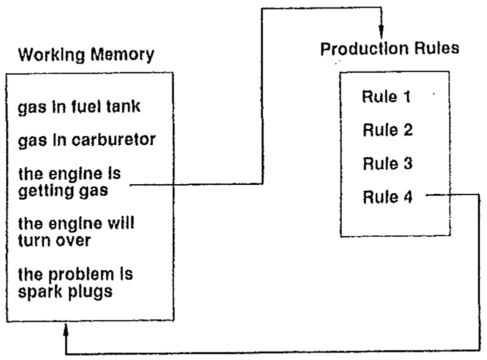

FIGURE 9.7 Working memory after rule 4 has fired.

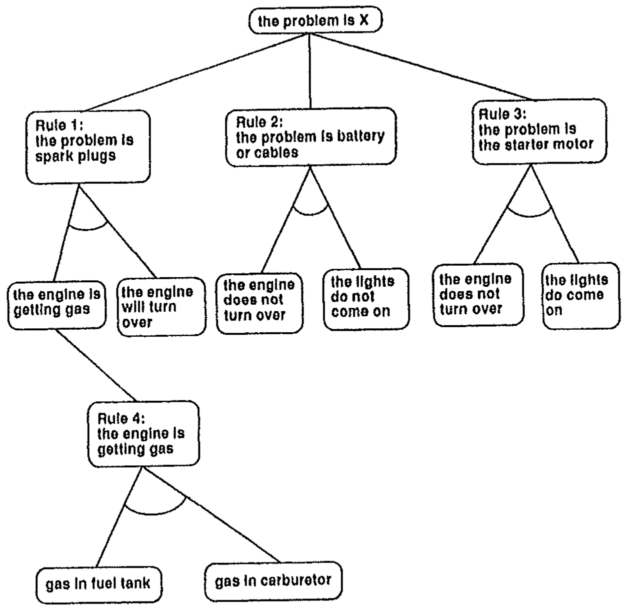
the problem is X
Rule 1: the problem is spark plugs
Rule 2: the problem is battery or cables
Rule 3: the problem is the starter motor
the engine is getting gas
the engine will turn over
the engine does not turn over
the lights do not come on
the engine does not turn over
the lights do come on
Rule 4: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

FIGURE 9.8 The AND/OR graph searched in the car diagnostic example.

This example of expert systems underscores their foundation in the production system and AND/OR graph search models of problem solving.

At the beginning of this chapter, we emphasized that an expert system needed to be open to inspection, easily modified, and heuristic. The production system architecture is an important factor in each of these requirements. Ease of modification, e.g., 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 Sec. 9.3, from use of confidence measures. The next section discusses the way in which this architecture supports explanations and inspection of the reasoning process.

9.2.2 Explanation and Transparency

The production system implements a graph search; the explanation routines of Fig. 9.1 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

Fig. 9.7. The following dialogue begins with the computer asking the user about the goals present in working memory. User responses are underlined.

```
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 was 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 to 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.

9.2.3 Heuristics and Control in Expert Systems

Because of the separation of the knowledge base and the inference engine and the fixed control regimes provided by the inference engine, the only way that a programmer can control the search is through the structure of the rules in the knowledge base. This is an advantage, since the control strategies required for expert-level problem solving tend to be domain-specific and knowledge-intensive. Although a rule of the form if *p*, *q*, and *r*, then *s* resembles a logical expression, it may also be interpreted as a series of procedures for solving a problem: to do *s*, first do *p*, then do *q*, then do *r*.

This procedural interpretation allows us to control search through the structure of the rules. For example, we may order the premises of a rule so that the premise that is most likely to fail or is easiest to confirm will be tried first. This gives the opportunity of eliminating a rule (and hence a portion of the search space) as early in the search as possible. Rule 1 in the automotive example tries to determine if the engine is getting gas before it asks if the engine turns over. This is inefficient, in that trying to determine if the engine is getting gas invokes another rule and eventually asks the user two questions. By reversing the order of the premises, a negative response to the query "Engine will turn over?" eliminates this rule from consideration before the more involved condition is examined.

Also, from a semantic point of view, it makes more sense to determine whether the engine is turning over before checking to see if it is getting gas; if the engine will not turn over it does not matter if it is getting gas. Rule 4 asks the user to check for gas in the fuel tank before asking that the user open up the carburetor and look there. It is performing the easier check first.

In addition to the ordering of a rule's premises, the content of a rule itself may be fundamentally heuristic. In the automotive example, all the rules are heuristic; consequently, the system may obtain erroneous results. For example, if the engine is getting gas and turning over, the problem may be a bad distributor rather than bad spark plugs. In the next section, we examine this problem and some ways of dealing with it.

9.3 USING UNCERTAINTY MEASURES IN EXPERT SYSTEMS

9.3.1 Introduction

As mathematicians or scientists, we often want our inference procedures to follow the model presented with the predicate calculus: From correct premises sound inference rules produce new, guaranteed-correct, conclusions. In expert systems, however, we must often draw correct conclusions from poorly formed and uncertain evidence, using unsound inference rules.

This is not an impossible task; we do it successfully in almost every aspect of our daily survival. We deliver correct medical treatment for ambiguous symptoms; we mine natural resources with little or no guarantee of success before we start; we comprehend language statements that are often ambiguous or incomplete, and so on.

The reasons for this ambiguity may be better understood by referring once again to our automotive expert system example. Consider rule 2:

```
if
     the engine does not turn over and
     the lights do not come on,
then
     the problem is battery or cables.
```

This rule is heuristic in nature; it is possible, although less likely, that the battery and cables are fine but that the car simply has a bad starter motor and burned-out headlights. This rule seems to resemble a logical implication, but it is not: Failure of the engine to turn over and the lights to come on does not necessarily imply that the battery and cables are bad. What is interesting to note, however, is that the *converse* of the rule is a true implication:

```
if
     the problem is battery or cables,
then
     the engine does not turn over and
     the lights do not come on.
```

Barring the supernatural, a car with a dead battery will not light its headlamps or turn the starter motor.

This is an example of abductive reasoning. Formally, abduction states that from $P \rightarrow Q$ and Q it is possible to infer P.

Abduction is an *unsound* rule of inference, meaning that the conclusion is not necessarily true for every interpretation in which the premises are true. For example, if someone says, "If it rains, then I will not go running at 3:00 p.m." and you do not see that person on the track at 3:00 p.m., does it necessarily follow that it is raining? It is possible that the individual decided not to go running because of an injury or needing to work late.

Although abduction is unsound, it is often essential to solving problems. The "correct" version of the battery rule is not particularly useful in diagnosing car troubles since its premise (bad battery) is our goal and its conclusions are the observable symptoms that we must work with. Modus ponens cannot be applied, and the rule must be used in an abductive fashion. This is generally true of diagnostic (and other) expert systems. Faults or diseases cause (imply) symptoms, not the other way around, but diagnosis must work from the symptoms back to the cause.

Uncertainty results from the use of abductive inference as well as from attempts to reason with missing or unreliable data. To get around this problem, we can attach some measure of confidence to the conclusions. For example, although battery failure does not always accompany the failure of a car's lights and starter, it almost always does, and confidence in this rule is justifiably high.

Note that there are problems that do not require certainty measures. When one is configuring a computer, for instance, either the components go together, or they do not. The idea that "A particular disk drive and bus go together with certainty 0.75" does not even make sense. Similarly, if MACSYMA is attempting to find the integral of a function, a confidence of "0.6" that a result is correct is not useful. These programs may be either data-driven (Digital's XCON) or goal-driven (Massachusetts Institute of Technology's MACSYMA), but because they do not require abductive rules of inference or do not deal with unreliable data, they do not require the use of confidence measures.

In this section we discuss several ways of managing the uncertainty that results from heuristic rules: first, the bayesian approach and, second, the Stanford certainty theory. Finally, we briefly consider Zadeh's *fuzzy set theory,* the Dempster/Shafer theory of evidential reasoning, and nonmonotonic reasoning.

9.3.2 Bayesian Probability Theory

The bayesian approach to uncertainty is based in formal probability theory and has shown up in several areas of AI research, including pattern recognition and classification problems. The PROSPECTOR expert system, built at Stanford and SRI International and employed in mineral exploration (copper, molybdenum, and others), also uses a form of the bayesian statistical model.

Assuming random distribution of events, probability theory allows the calculation of more complex probabilities from previously known results. In simple probability calculations we are able to conclude, e.g., how cards might be distributed to a number of players.

Suppose that I am one person of a four-person card game where all the cards are equally distributed. If I do not have the queen of spades, I can conclude that each of the other players has it with probability $1/3$. Similarly, I can conclude that each player has the ace of hearts with probability $1/3$ and that any one player has both cards at $1/3 \times 1/3$, or $1/9$.

In the mathematical theory of probability, individual probability instances are worked out by sampling, and combinations of probabilities are worked out as above, using a rule such as

$$\text{probability}(A \text{ and } B) = \text{probability}(A) \times \text{probability}(B)$$

given that A and B are independent events.

One of the most important results of probability theory is Bayes' theorem. Bayes' results provide a way of computing the probability of a hypothesis following from a particular piece of evidence, given only the probabilities with which the evidence follows from actual causes (hypotheses).

Bayes' theorem states:

$$P(H_i|E) = \frac{P(E|H_i) \times P(H_i)}{\sum_{k=1}^{n}[P(E|H_k) \times P(H_k)]}$$

where $P(H_i|E)$ = probability that H_i is true given evidence E

$P(H_i)$ = probability that H_i is true overall

$P(E|H_i)$ = probability of observing evidence E when H_i is true

n = number of possible hypotheses

Suppose we desire to examine the geological evidence at some location to see if the location is suited to finding copper. We must know in advance the probability of finding each of a set of minerals and the probability of certain evidence being present when each particular mineral is found. Then we can use Bayes' theorem to determine the likelihood that copper will be present, using the evidence we collect at the location. This is the approach taken in the PROSPECTOR program, which has found commercially significant mineral deposits [9] at several sites.

There are two major assumptions for the use of Bayes' theorem: First, all the statistical data on the relationships of the evidence with the various hypotheses are known; second, and more difficult to establish, all relationships between evidence and hypotheses, or $P(E|H_k)$, are independent. Actually, this assumption of independence can be quite a tricky matter, especially when many assumptions of independence are needed to establish the validity of this approach across many rule applications. This represents the entire collected probabilities on the evidence given various hypothesis relationships in the denominator of Bayes' theorem. In general, and especially in areas like medicine, this assumption of independence cannot be justified.

A final problem, which makes keeping the statistics of the "evidence given hypotheses" relationships virtually intractable, is the need to rebuild all probability relationships when any new relationship of hypothesis to evidence is discovered. In many active research areas, such as medicine, this is happening continuously. Bayesian reasoning requires complete and up-to-date probabilities if its conclusions are to be correct. In many domains, such extensive data collection and verification are not possible.

Where these assumptions are met, bayesian approaches offer the benefit of a well-founded and statistically correct handling of uncertainty. Most expert-system domains do not meet these requirements and must rely on more heuristic approaches. It is also felt that the human expert does not use the bayesian model in successful problem solving. In the next section we describe certainty theory, a heuristic approach to the management of uncertainty.

9.3.3 A Theory for Certainty

Several early expert-system projects (besides PROSPECTOR) attempted to adapt Bayes' theorem to their problem-solving needs. The independence assumptions, continuous updates of statistical data, and other problems mentioned in Sec. 9.3.2 gradually led these researchers to search for other measures of "confidence." Probably the most important alternative approach was used at Stanford in developing the MYCIN program [3].

Certainty theory is based on a number of observations. The first is that in traditional probability theory the sum of confidence for a relationship and confidence against the same relationship must sum to 1. However, often an expert might have confidence 0.7, say, that some relationship is true and have no feeling about its being not true.

Another assumption that underpins certainty theory is that the knowledge content of the rules is much more important than the algebra of confidences which holds the system together. Confidence measures correspond to the informal evaluations that human experts attach to their conclusions, e.g., "It is probably true" or "It is highly unlikely."

Certainty theory makes some simple assumptions for creating confidence measures and has some equally simple rules for combining these confidences as the program moves toward its conclusion. The first assumption is to split "confidence for" from "confidence against" in a relationship:

Call $\mathrm{MB}(H|E)$ the measure of belief of a hypothesis H given evidence E.

Call $\mathrm{MD}(H|E)$ the measure of disbelief of a hypothesis H given evidence E.

Now either

$$1 > \mathrm{MB}(H|E) > 0 \qquad \text{while} \qquad \mathrm{MD}(H|E) = 0$$

or

$$1 > \mathrm{MD}(H|E) > 0 \qquad \text{while} \qquad \mathrm{MB}(H|E) = 0$$

The two measures constrain each other in that a given piece of evidence is either for or against a particular hypothesis. This is an important difference between certainty theory and probability theory. Once the link between measures of belief and disbelief has been established, they may be tied together again with the *certainty-factor* (CF) calculation:

$$\mathrm{CF}(H|E) = \mathrm{MB}(H|E) - \mathrm{MD}(H|E)$$

As the certainty factor (CF) approaches 1, the evidence is stronger for a hypothesis; as CF approaches -1, the confidence against the hypothesis gets stronger; and a CF around 0 indicates that there is little evidence either for or against the hypothesis.

When experts put together the rule base, they must agree on a CF to go with each rule. This CF reflects their confidence in the rule's reliability. Certainty measures may be adjusted to tune the system's performance, although slight variations in this confidence measure tend to have little effect on the overall running of the system (again, the knowledge gives the power).

The premises for each rule are formed of the AND and OR of a number of facts. When a production rule is used, the certainty factors that are associated with each condition of the premise are combined to produce a certainty measure for the overall premise in the following manner.

For P_1 and P_2, premises of the rule,

$$\mathrm{CF}(P_1 \text{ and } P_2) = \min(\mathrm{CF}(P_1), \mathrm{CF}(P_2))$$

and

$$\mathrm{CF}(P_1 \text{ or } P_2) = \max(\mathrm{CF}(P_1), \mathrm{CF}(P_2))$$

The combined CF of the premises, by using the above combining rules, is then multiplied by the CF of the rule to get the CF for the conclusions of the rule.

For example, consider the rule in a knowledge base

$$(P_1 \text{ and } P_2) \text{ or } P_3 \rightarrow R_1(0.7) \text{ and } R_2(0.3)$$

where P_1, P_2, and P_3 are premises and R_1 and R_2 are the conclusions of the rule having CF values of 0.7 and 0.3, respectively. These numbers are added to the rule when it is designed and represent the expert's confidence in the conclusion if all the premises are known with complete certainty. If the running program has produced P_1, P_2, and P_3 with CF values of 0.6, 0.4, and 0.2, respectively, then R_1 and R_2 may be added to the collected case-specific results with CF values of 0.28 and 0.12, respectively.

Here are the calculations for this example:

$$\mathrm{CF}(P_1(0.6) \text{ and } P_2(0.4)) = \min(0.6, 0.4) = 0.4$$

$$\mathrm{CF}((0.4) \text{ or } P_3(0.2)) = \max(0.4, 0.2) = 0.4$$

The CF for R_1 is 7 in the rule, so R_1 is added to the set of true facts with the associated CF of $7 * 0.4 = 0.28$. The CF for R_2 is 0.3 in the rule, so R_2 is added to the set of true facts with the associated CF of $0.3 * 0.4 = 0.12$.

One further measure is required: how to combine multiple CFs when two or more rules support the same result R. This is the certainty theory analog of the probability theory procedure of multiplying the probability measures to combine independent evidence. By using this rule repeatedly, one can combine the results of any number of rules that are used for determining result R. Suppose $\mathrm{CF}(R_1)$ is the present certainty factor associated with result R and a previously unused rule produces result R (again) with $\mathrm{CF}(R_2)$; then the new CF of R is calculated by

$\mathrm{CF}(R_1) + \mathrm{CF}(R_2) - (\mathrm{CF}(R_1) * \mathrm{CF}(R_2))$ when $\mathrm{CF}(R_1)$ and $\mathrm{CF}(R_2)$ are positive

$\mathrm{CF}(R_1) + \mathrm{CF}(R_2) + (\mathrm{CF}(R_1) * \mathrm{CF}(R_2))$ when $\mathrm{CF}(R_1)$ and $\mathrm{CF}(R_2)$ are negative

$\mathrm{CF}(R_1) + \mathrm{CF}(R_2)$ over $1 - \min (|\mathrm{CF}(R_1)|, |\mathrm{CF}(R_2)|)$ otherwise

where $|X|$ is the absolute value of X.

Besides being easy to compute, these equations have other desirable properties. First, the CFs that result from applying this rule are always between 1 and −1, as are the other CFs. Second, the result of combining contradictory CFs is that they cancel, as would be desired. Finally, the combined CF measure is a monotonically increasing (decreasing) function in the manner one would expect for combining evidence.

Certainty theory has been criticized as being excessively ad hoc. Although it is defined in a formal algebra, the meaning of the certainty measures is not as rigorously founded as in formal probability theory. However, certainty theory does not attempt to produce an algebra for "correct" reasoning. Rather it is the "lubrication" that lets the expert system combine confidences as it moves along through the problem at hand. Its measures are ad hoc in the same sense that a human expert's confidence in her or his results is approximate, heuristic, and informal. In Sec. 9.4, when MYCIN is considered, the CFs will be used in the heuristic search to give a priority for goals to be attempted and a cutoff point when a goal need not be considered further. But even though the CF is used to keep the program running and collecting information, the power of the program lies in the content of the rules themselves. This is the justification for the weakness of the certainty algebra.

9.3.4 Other Approaches to Uncertainty

Because of the importance of uncertain reasoning to expert-level problem solving and the limitations of certainty theory, work continues in this important area. In concluding this subsection, we mention briefly three other approaches to modeling uncertainty: Zadeh's *fuzzy set theory*, the Dempster/Shafer *theory of evidence*, and *nonmonotonic reasoning*.

Zadeh's main contention [25] is that although probability theory is appropriate for measuring randomness of information, it is inappropriate for measuring the *meaning* of information. Indeed, much of the confusion surrounding the use of English words and phrases is related to lack of clarity (vagueness) rather than randomness. This is a crucial point for analyzing language structures and can also be important in creating a measure of confidence in production rules. Zadeh proposes *possibility theory* as a measure of vagueness, just as probability theory measures randomness.

Zadeh's theory expresses lack of precision in a quantitative fashion by introducing a set membership function that can take on real values between 0 and 1. This is the notion of a *fuzzy set* and can be described as follows: Let S be a set and s a member of that set. A fuzzy subset F of S is defined by a membership function $mF(s)$ that measures the "degree" to which s belongs to F.

A standard example of the fuzzy set is for S to be the set of positive integers and F to be the fuzzy subset of S called "small integers." Now various integer values can have a "possibility" distribution defining their "fuzzy membership" in the set of small integers: $mF(1) = 1$, $mF(2) = 1$, $mF(3) = 0.9$, $mF(4) = 0.8$, . . . , $mF(50) = 0.001$, etc. For the statement that positive integer X is a "small integer," mF creates a possibility distribution across all the positive integers (S).

Fuzzy set theory is not concerned with how these possibility distributions are created, but rather with the rules for computing the combined possibilities over expressions that each contain fuzzy variables. Thus it includes rules for combining possibility measures for expressions containing fuzzy variables. The laws for the OR, AND, and $\neg$ of these expressions are similar to those just presented for certainty factors. In fact, the approach at Stanford was modeled on some of the combination rules described by Zadeh [3].

Dempster and Shafer approach the problem of measuring certainty by asking us to make a fundamental distinction between uncertainty and ignorance. In probability theory we are *forced* to express the extent of our knowledge about a belief X in a single number $P(X)$. The problem with this, say Dempster and Shafer, is that we simply cannot always know the values of prior probabilities, and thus any particular choice of $P(X)$ may not be justified.

The Dempster-Shafer approach recognizes the distinction between uncertainty and ignorance by creating "belief functions." These belief functions satisfy axioms that are weaker than those of probability theory. Thus probability theory is seen as a subclass of belief functions, and the theory of evidence can reduce to probability theory when all the probabilities are obtainable. Belief functions therefore allow us to use our knowledge to bound the assignment of probabilities to events without having to come up with exact probabilities, when these may be unavailable.

Even though the Dempster-Shafer approach gives us methods of computing these various belief parameters, their greater complexity adds to the computational cost as well. Besides, any theory that has probability theory as a special case is plagued by the assumptions that have made probability theory already quite difficult to use. Conclusions using beliefs, even though they avoid commitment to a stronger (and often unjustified) assignment of probability, produce conclusions that are necessarily weaker. But as the Dempster-Shafer model points out quite correctly, the stronger conclusion may not be justified.

All the methods we have examined can be criticized for using numeric approaches to the handling of uncertain reasoning. It is unlikely that humans use any of these techniques for reasoning with uncertainty, and many applications seem to require a more qualitative approach to the problem. For example, numeric approaches do not support adequate explanations of the causes of uncertainty. If we ask human experts why their conclusions are uncertain, they can answer in terms of the qualitative relationships between features of the problem instance. In a numeric model of uncertainty, this information is replaced by numeric measures. Similarly, numeric approaches do not address the problem of changing data. What should the system do if a piece of uncertain information is later found to be true or false?

Nonmonotonic reasoning addresses these problems directly. A nonmonotonic reasoning system handles uncertainty by making the most reasonable assumptions in light of uncertain information. It proceeds with its reasoning as if these assumptions were true. At a later time, it may find out that an assumption was erroneous, either through a change in problem data or by discovering that the assumption led to an impossible conclusion. When this occurs, the system must change both the assumption and all the conclusions that depend on it.

Nonmonotonicity is an important feature of human problem solving and commonsense reasoning. When we drive to work, e.g., we make numerous assumptions about the roads and traffic. If we find that one of these assumptions is violated, perhaps by construction or an accident on our usual route, we change our plans and devise an alternative route to work.

Nonmonotonic reasoning contrasts sharply with the inference strategies we have discussed so far in the text. These strategies assume that axioms do not change and that the conclusions drawn from them remain true. They are called *monotonic* reasoning systems because knowledge can only be added through the reasoning process. Since information may also be retracted in a nonmonotonic reasoning system when something that was previously thought to be true is later shown to be false, it is important to record the justification for all new knowledge. When an assumed fact is withdrawn, all conclusions that depend on that fact must be reexamined and possibly withdrawn as well. This record keeping is the task of *truth maintenance systems* [7, 8].

9.4 MYCIN: A CASE STUDY

9.4.1 Introduction

Although our small automotive diagnostic example is useful in presenting some of the concepts of expert-system design, it is a toy system and ignores much of the complexity encountered in building large knowledge-based programs. For this reason, we end this chapter with a case study of an actual working expert system.

The MYCIN project was a cooperative venture by the Department of Computer Science and the Medical School at Stanford University. The main work on the project was done during the middle and late 1970s, and about 50 person-years were expended in the effort. MYCIN was written in INTERLISP, a dialect of the LISP programming language. One of the earliest expert systems to be proposed and designed, it has become an important classic in the field. Indeed, it is often presented as the archetype for the rule-based program, with many commercial systems essentially duplicating the MYCIN approach. One of the main reasons for the influence of the MYCIN program is the extensive documentation of the project by the Stanford research teams [3].

MYCIN was designed to solve the problem of diagnosing and recommending treatment for meningitis and bacteremia (blood infections). This particular domain was chosen largely because the program architects wanted to explore the way in which human experts reason with missing and incomplete information. Although there are diagnostic tools that can unambiguously determine the identity of the infecting organisms in a case of meningitis, these tools require on the order of 48 h (chiefly to grow a culture of the infecting organisms) to return a diagnosis. Meningitis patients, however, are very sick, and some treatment must begin immediately. Because of this need, doctors have developed considerable expertise, based on initial symptoms and test results, for forming a diagnosis that *covers* (i.e., includes as a subset) the actual infecting organisms. Treatment begins with this diagnosis and is refined when more conclusive information becomes available. Thus, the domain of meningitis diagnosis provided a natural focus for studying how humans solve problems by using incomplete or unreliable information.

Another goal of the MYCIN design team was to pattern the behavior of the program after the way in which human physicians interact in actual consultations. This was seen as an important factor in system acceptability, particularly since medical consultations tend to follow a standard set of protocols.

In the next subsections we examine the composition of the MYCIN rule and fact descriptions, including the syntax of MYCIN rules. Next, we present a trace of a MYCIN consultation, along with explanatory comments and a discussion of the design and structure of the dialogue. We then discuss the problem of evaluating expert systems and, finally, demonstrate the use of an experimental knowledge-base editor *Teiresias*. Editors such as Teiresias assist the domain experts (in this case the doctors) in correcting the MYCIN knowledge base without the need for a computer language expert. Although such tools are still mainly experimental, this is an important area of research and a potential key to increasing the range of applicability of expert-system technology.

9.4.2 Representation of Rules and Facts

Facts in MYCIN are represented as *attribute-object-value triples*. The first element of this structure is an attribute of an object in the problem domain. For example, we may wish to describe the identity of a disease organism or its sensitivity to certain drugs. The name of the object and the value of the attribute follow. Since information in this domain may be uncertain, a certainty factor is associated with MYCIN facts. Recall from Sec. 9.3.3 that this certainty factor will be between 1 and −1, with 1 being certain, −1 indicating certainty that the attribute is not true, and 0 indicating that little is known.

For example, MYCIN facts may be represented in English by:

```
There is evidence (0.25) that the identity of the organism is
Klebsiella.
```

and

```
It is known that the organism is not sensitive to penicillin.
```

These may be translated into the LISP s expressions:

```
(ident organism_1 klebsiella 0.25)
```

and

```
(sensitive organism_1 penicillin -1.0)
```

We next present an example MYCIN rule both in English and in its LISP equivalent. This sample rule is a condition-action pair in which the premise (condition) is the conjunction of three facts (attribute-object-value triples) and the action adds a new fact to the set of things known about the patient, the identity of a particular organism that is added to the "cover set" of possible infecting organisms. Because this is an abductive rule of inference (inferring the cause from the evidence), it has an attached certainty factor.

In English: IF (a) the infection is primary bacteremia, and (b) the site of the culture is one of the sterile sites, and (c) the suspected portal of entry is the gastrointestinal tract, then there is suggestive evidence (0.7) that infection is bacteroid.

This is rendered for MYCIN thus:

```
IF:    (AND (same_context infection primary_bacteremia)
       (membf_context site sterilesite)
       (same_context portal GI))

THEN:  (conclude context_ident bacteroid tally 0.7).
```

Syntactically, attribute-objective-value (A-O-V) triples are essentially a restriction of predicate calculus expressions to binary predicates. This restriction is not particularly limiting, since algorithms exist for translating predicates of any arity into a set of binary predicates.

There is a deeper difference between predicate calculus and the A-O-V triples used in MYCIN: Predicates may be either true or false. Predicate calculus uses sound rules of inference to determine the truth value of conclusions. In MYCIN, a fact has an attached confidence rather than a truth value. When the system is deriving new facts, it fails to fire any rule whose premise has a certainty value of less than 0.2. This contrasts with logic, which causes a rule to fail if its premise is found to be false.

This comparison of attribute-object-value triples and predicate calculus expressions is included here to emphasize both the similarities and the tradeoffs involved in different knowledge representations. At the highest conceptual level, the characteristics that define a rule-based expert system are independent of any commitment to a particular representational format. These characteristics have been discussed in the comparison of rule-based expert systems and the production system model of problem solving. At the level of an implementation, however, the selection of a particular representational formalism does involve a number of important tradeoffs. These issues include the clarity of the representation, expressiveness (what knowledge can the formalism effectively capture?), naturalness of expression, and ease of implementing and verifying the system.

The action of a MYCIN rule can perform a number of tasks once the rule is satisfied. It may add new information about a particular patient to the working memory. It may also write information to the terminal (or other output device), change the value of an attribute or its certainty measure, look up information in a table (where this representation is more efficient), or execute any piece of LISP code.

Although the ability to execute arbitrary LISP code gives unlimited added power, it should be used judiciously, since excessive use of escapes to the underlying language can lose many of the benefits of the production system formalism. As an absurd example of this, imagine an expert system with only one rule, whose action is to call a large and poorly structured LISP program that attempts to solve the problem!

MYCIN rule and fact descriptions, like all knowledge representation, are a formal language and have a formal syntactic definition. This formal syntax is essential to the definition of well-behaved inference procedures. Furthermore, the formal syntax of rules can help a knowledge-base editor, such as Teiresias, determine when the domain expert has produced an incorrectly formed rule and can prevent that rule from entering the knowledge base. Teiresias can detect syntactic errors in a rule and, to an impressive degree, automatically remedy the situation (see Sec. 9.4.5). This is possible only when the entire program is built on a set of formal specifications.

The syntax of MYCIN rules is given below. It is stated in *Backus-Naur form* (BNF) [20]. BNF is a form of context-free grammar that is widely used to define programming languages. In the notation, key words in the language are written in uppercase letters. These are terminals in the syntax. Nonterminals are enclosed in angle brackets (<>). The BNF grammar is a collection of rewrite rules; the nonterminal figure to the left of the ::= can be replaced with the expression on the right. If we begin with the nonterminal, anything that can be produced through a series of legal substitutions is a legal MYCIN rule.

```
<rule> ::= (IF <antecedent> THEN <action> [ELSE <action>])
<antecedent> ::= (AND {<condition>}+)
<condition> ::= (OR {<condition>}+ | <predicate><associative-triple
<associative-triple> ::= (<attribute><object><value>)
<action> ::= {<consequent>}+ | {<procedure>}+
<consequent> ::= (<associative-triple><certainty-factor>)
<certainty-factor> ::= ( ranges from -1 (false) to +1 (true))
<predicate> is any LISP predicate.
<procedure> is any piece of LISP code.
```

9.4.3 MYCIN Diagnosing an Illness

MYCIN is a goal-driven expert system. Its main action is to try to determine whether a particular organism may be present in the meningitis infection in the patient. A possible infecting organism forms a goal that is either confirmed or eliminated. It is interesting to note that the decision to make MYCIN a goal-driven system was not based exclusively on the effectiveness of the strategy in pruning the search space. Early in the design of the system, the MYCIN architects considered the use of forward search. This was rejected because the questions the system asked the user seemed random and unconnected. Goal-driven search, because it is attempting to either confirm or eliminate a particular hypothesis, causes the reasoning to seem more focused and logical. This builds the user's understanding and confidence in the system's actions. An expert system needs to do more than obtain a correct answer; it must also do so in an intelligent and understandable fashion. This criterion influences the choice of search strategies as well as the structure and order of rules.

In keeping with this goal of making the program behave as a doctor, MYCIN's designers noted that doctors ask routine questions at the beginning of a consultation (e.g., "How old are you?" and "Have you had any childhood diseases?") and ask more specific questions when needed. Collecting the general information at the beginning makes the session seem more focused in that it is not continually interrupted by trivial questions such as the name, age, sex, and race of the patient. Certain other questions are asked at other well-defined stages of a consultation, such as at the beginning of considering a new hypothesis.

Eventually the questions get more specific (questions 16 and 17 in the trace below) and related to possible meningitis. When positive responses are given to these questions, MYCIN determines that the infection is meningitis, goes into full goal-driven mode, and tries to determine the actual infecting organisms. It does this by considering each infecting organism that it knows about and by attempting to eliminate or confirm each hypothesis in turn. Since a patient may have more than one infecting organism, MYCIN searches exhaustively, continuing until all possible hypotheses have been considered.

MYCIN controls its search in a number of ways that have not been discussed yet. The knowledge base includes rules that restrict the hypotheses to be tested. For example, MYCIN concludes that it should test for meningitis when the patient has had headaches and other abnormal neurological signs.

Another feature of MYCIN's inference engine is the order in which it tries the backward chaining rules. After the general category of the infection (meningitis, bacteremia, etc.) is determined, each candidate diagnosis is examined exhaustively in a depth-first fashion.

To make the search behave more intelligently, MYCIN first examines all the premises of a rule to determine if any are already known to be false. This prevents the program from trying to solve several of the premises, only to discover that the rule could have been eliminated immediately by one of the later premises.

MYCIN also attempts to find a *unity path*. In trying to prove a goal, it looks for rules that can conclude that goal with a certainty of 1. If one of these rules succeeds, the system does not have to try other rules; the goal is known to be true. This increases both the efficiency and the rationality of the search procedure.

When the confidence measures for a rule premise get below a certain value, the depth-first search is terminated. As mentioned above, the certainty measure for each patient's data is combined with the measures resident in the rules to give the certainty of the new conclusions for the patient. When these get below 0.2, MYCIN abandons the search and goes back up the graph to try another branch of the search space.

When the user wants MYCIN to justify its request for some information, he or she can ask why. Initially, this is answered by the phrase "because I need the information as a necessary attribute of each patient." Once MYCIN begins its backward search for the causes of the patient's problems, however, the why question is answered by printing the rule that MYCIN is currently considering.

As mentioned earlier in this chapter, the "English-like" interaction that the computer offers the user is not a full English language understanding program. Appropriate forms for English language comments and questions are attached to each rule when it is created so that the proper query can be presented to the user when information is needed for use of the rule or so that a rule can be printed in response to a user query.

The following dialogue is a simulated run of the MYCIN program adapted from transcripts of actual MYCIN consultations. It is not necessary to understand either the medical terminology or the medical reasoning to appreciate MYCIN's problem solving.

Our comments on the MYCIN trace are reproduced in italics throughout the interaction. The MYCIN questions and answers are indented, with the user's responses underlined.

```
@mycin
  MYCIN 26-March....
  Instructions? (Y or N)
  N
  26-March 10:42
  -----PATIENT-1-----
```

Mycin has created an object (PATIENT-1) in its dynamic knowledge base or working memory. Certain attributes are immediately requested and are necessary for every patient. These make up the first four questions. . . .

```
  (1) Patient's name:
  Alan Apt
  (2) Age?
  34
  (3) Sex
  Why
```

MYCIN restates the question before answering. . . .

```
  [i.e., Why is it important to determine the sex of PATIENT-1?]
  [1.0] The sex of the patient is one of the four standard parameters
  that identify a patient. This may be useful later....
  [back to question 3...]
  Male
  (4) Race?
  Caucasian
  (5) Are there any cultures for Alan Apt that may be related to the
  present illness and from which organisms have been grown
  successfully in the laboratory?
  No
```

MYCIN is a program to analyze and treat infectious diseases, so before it gets into close questioning of the patient's present condition, it asks about any present or pending data (cultures) that are available. . . .

```
  (6) Are there any pending cultures for Alan Apt?
  Yes
  -----CULTURE-1-----
```

Another object (CULTURE-1) is created and linked to PATIENT-1. . . .

```
  (7) From what site was the specimen for CULTURE-1 taken?
  CSF
```

```
(8) Please give the date and time when the pending CSF culture was
obtained?
26-March-7:00
(9) Was a smear examined at the time the specimen was obtained?
Yes
(10) Were any cultures seen on the smear of the specimen?
No
```

Indention indicates that MYCIN's attention has returned from consideration of the new object (CULTURE-1) back to the original object (PATIENT-1).

```
(11) Are there any other pending cultures?
No
 .
 .
 .
(16) Has Alan Apt recently had symptoms of persistent headache or
other abnormal neurologic symptoms (dizziness, lethargy, etc.)?
Yes
(17) Has Alan Apt recently had objective evidence of abnormal
neurologic signs (nuchal rigidity, coma, seizures, etc.) documented.
by a physician?
Yes
```

With the answers of these last two questions in the affirmative, MYCIN decides to consider meningitis infections and goes into a full backchaining search to try to identify the organisms that might be causing the meningitis.

```
(18) Please give the date on which clinical evidence (symptoms,
signs, or laboratory tests) of the meningitis first appeared.
25 March 10:00
 .
 .
 .
(36) Has Alan Apt had neurosurgery?
Why
[i.e., Why is it important to determine whether PATIENT-1 has had
neurosurgery?]
[6.0] This will aid in determining the organisms (other than those
seen on cultures or smears) that might be causing the infection. It
has already been established that
[6.1] the culture that requires therapy is meningitis, and
[6.2] organisms were not seen on the stain of the pending CSF
culture (CULTURE-1), and
[6.3] the type of infection is bacterial
Therefore, if
[6.4] Alan Apt has had neurosurgery, and
[6.5] the time since the neurosurgical operation is less than 2
months, and
[6.5] the type of neurosurgery is not ventricular urethral shunt,
then
there is evidence that the organisms that might be causing the
infection are
    staphylococcus-coag-pos (0.75), staphylococcus-coag-neg (0.5),
    E. coli (0.4), klebsiella....
Author: Yu
    [RULE511]
[back to question 36...]
How [6.3]
```

The physician is concerned how MYCIN arrived at the conclusion that the infection is bacterial. MYCIN responds, after restating its interpretation of the question the doctor is asking, with the rules it used to determine that the infection was indeed bacterial. The answer to how is given as the rules used, as well as their (positive and negative) certainty factors, for determining this result. . . .

```
[i.e., How was it established that the type of infection is
bacterial?]
The following rules concluded about the likelihood that the type of
infection is bacterial

     Rule    Yes   No
     [6.7]   500  (0.4)
     [6.8]   501  (0.17)
     [6.9]   502  (0.1)
     [6.10]  524  (0.42)
     [6.11]  504  (0.28)

Thus it has been established that the type of infection might be
bacterial.
[back to question 36...]
No
 .
 .
 .
(42) Is Alan Apt a burn patient?
Yes
I have found evidence (RULE 507) that treatment should cover for
        Diplococcus pneumoniae causing meningitis.
I have found evidence (RULE 545) that treatment should cover for E.
        coli causing meningitis
I have found evidence (RULE 545 RULE 578) that the treatment should
        cover for Pseudomonas aeruginosa causing meningitis.
```

MYCIN now attempts to prescribe appropriate therapies for these infections. First, it must be determined whether the patient has any allergies to antibody therapies. The height and weight of the patient are also requested to determine proper dosages . . .

```
(43) Does Alan Apt have focal neurological signs?
No
[Determining which dosages are desirable for use against the E.
coli ...]
[Determining which drugs are desirable for use ...]
INFECTION-1 is MENINGITIS
+ DIPLOCOCCUS-PNEUMONIAE (from clinical evidence only)
+ E. COLI (from clinical evidence only)
+ PSEUD...
 .
 .
 .
[REC-1] My preferred therapy recommendation is as follows:
In order to cover for items:
Give the following in combination:
(1) AMPICILLIN
Dose: 3.85 g (30.8 ml) q4h IV
(2) GENTAMICIN
Dose: 131 mg (3.2 ml) q8h IV
```

```
Comments:
   Monitor serum concentrations
   Since high concentrations of penicillins can inactivate
   aminoglycosides,
   do not mix these antibiotics in the same IV bottle.
Do you wish to see the next choice therapy?
No
```

If the physician did not like the recommended treatment, she or he could have asked for another, eliminating one of the drugs (ampicillin, for instance). MYCIN would then have attempted to find another treatment. . . .

9.4.4 Evaluation of Expert Systems

One of the difficult issues faced by expert-system design teams is to determine when the program is ready for actual use. This problem is all the more difficult when the program, like MYCIN, deals with life-threatening situations; here mistakes, such as overlooking an infecting agent, can be catastrophic. Many expert systems, including MYCIN [3], have been evaluated by using a form of the Turing test.

Ten randomly selected case histories of meningitis were rediagnosed by MYCIN and eight practitioners at the Stanford Medical School. These included five faculty members, one research fellow in infectious diseases, one resident physician, and one medical student. The actual therapy given by the original doctors on the case was also included, for a total of 10 diagnoses.

These 10 diagnoses were evaluated by eight other infectious-disease experts. The diagnoses were "blind" in that they were uniformly coded so that the experts did not know whether they were looking at the computer's diagnosis or that of the humans. The evaluators rated each diagnosis as acceptable or unacceptable, with a practitioner being given one point for each acceptable rating. Thus, a perfect score would be 80 points. The results of this evaluation are given in Table 9.1.

Table 9.2 presents the results of asking another important question in this "life-and-death" analysis: In how many cases did the recommended therapy fail to cover for a treatable infection? The results in both tables indicate that MYCIN performed at least as well as the Stanford experts. This is an exciting result, but it should not surprise us since

TABLE 9.1 Experts Evaluate MYCIN and Nine Other Prescribers

Prescriber	Score	Percent
MYCIN	55	69
Faculty-5	54	68
Fellow	53	66
Faculty-3	51	64
Faculty-2	49	61
Faculty-4	47	59
Actual RX	47	59
Faculty-1	45	56
Resident	39	49
Student	28	35

TABLE 9.2 Number of Cases in Which Therapy Missed a Treatable Pathogen

Prescriber	Number
MYCIN	0
Faculty-5	1
Fellow	1
Faculty-3	1
Faculty-2	0
Faculty-4	0
Actual RX	0
Faculty-1	0
Resident	1
Student	3

the MYCIN knowledge base represents the combined expertise of some of the best medical minds available. Finally, MYCIN gave fewer drugs than the human experts and thus did not overprescribe for the infections. Note: On the average MYCIN gave fewer drugs than the human experts.

Another noteworthy aspect of this evaluation is how little agreement there was among human experts concerning the correctness of diagnoses. Even the best of the diagnosticians failed to receive a unanimous endorsement from the evaluators. This observation underscores the extent to which human expertise is still largely heuristic.

These positive evaluation results do not mean that MYCIN is now ready to set up a medical practice and take on patients. In fact, MYCIN is not used for delivering medical care. There are several important reasons. First and most important, the rules (approximately 600 of them) did not give a speedy response in the doctor-computer interaction Each session with the computer lasted about 0.5 h and required, as can be seen from the trace presented above, a good amount of typing.

Second, the program was "locked into" the particular part of its graph search for response to questions. It was not able to extrapolate to other situations or previous patients, but remained strictly within the nodes and links of the graph it was evaluating.

Third, MYCIN's explanations were limited: When the doctor expected a deep medical justification for some particular result—a physiological or antibacterial justification, say—MYCIN simply returned the "condition action plus certainty factor" relationship contained in its rule base. It is difficult for an expert system to relate its heuristic knowledge to any deeper understanding of the problem domain. For example, MYCIN does not really understand human physiology; it simply applies rules to case-specific data. Folklore has it that an early version of MYCIN, before recommending a particular drug, asked if the patient was pregnant, in spite of the fact that MYCIN had been told that the patient was male! Whether this actually happened or not, it does indicate the limitations of current expert-system technology. Attempting to give expert systems the flexibility and deeper understanding demonstrated by human beings is an important and open area of research.

Finally, unlike humans, who are extremely flexible in the way they apply knowledge, expert systems do not "degrade gracefully" when confronted with situations that do not fit the knowledge in their rule bases. That is, while a human can shift to reasoning from first principles or to intelligent guesses when confronted with a new situation, expert systems simply fail to get any answer at all.

All these issues have meant that MYCIN does not enjoy common use in the medical field. Nonetheless, it does serve as an archetype of this class of expert system, and many of its descendants, such as PUFF, are being used in clinical situations.

9.4.5 Knowledge Acquisition and the Teiresias Knowledge-Base Editor

Another one of the long-term goals of expert-systems research is to design software that will allow the human expert to interact directly with the knowledge base to correct and improve it. This is seen as a potential remedy for what has been called the *knowledge engineering bottleneck*. This bottleneck is caused by the fact that building an expert system generally requires a substantial commitment of time by both the domain expert and the knowledge engineer. This contributes to the expense and complexity of building expert systems. Both individuals tend to be highly paid professionals, and the loss in productivity caused by taking the domain expert away from other work during system development can add to the cost of the project. Additional cost and complexity come from the effort needed to communicate the domain expert's knowledge to the knowledge engineer, the logistics of getting these two people together, and the complexity of the programming required for an expert system.

The obvious solution to this problem is to automate as much of the process as possible. This has been done to a great extent through the development of expert-system shells and environments that reduce the amount of programming required by the knowledge engineer. However, these shells still require that the programmer understand the methodologies of knowledge representation and search. A more ambitious goal is to eliminate the need for a knowledge engineer entirely. One way to accomplish this is to provide knowledge-base editors that allow the domain expert to develop the program, interacting with the knowledge base in terms that come from the problem domain and letting the software handle the details of representation and knowledge organization. This approach has been explored in an experimental program called *Teiresias* [5, 6], a knowledge-base editor developed at Stanford University as part of the MYCIN project. We discuss the Teiresias project in this section.

Another approach to the problem is to develop programs that can learn on their own by refining their problem-solving efforts in response to feedback from the outside. These approaches have been explored with varying degrees of success by the machine learning community [16].

With the assistance of a knowledge-base editor, the domain expert can analyze the performance of a knowledge base, find missing or erroneous rules, and correct the problem in a language appropriate to the expert's way of thinking about the domain. The knowledge-base editor designed for MYCIN is called Teiresias, after the blind seer of the Greek tragedian Euripides. The name is altogether appropriate, as Teiresias was able to see and describe the relations of the world without the (literal) gift of sight; similarly, this software is able to understand relations in a knowledge base without (literally) understanding the referents of these relations.

Teiresias allows a doctor to step through MYCIN's treatment of a patient and locate problems with its performance. Teiresias also translates the doctor's corrections into the appropriate internal representations for the knowledge base. Thus Teiresias is able to maintain the syntactic consistency of the knowledge base by updating all appropriate program structures when new information is to be integrated into the knowledge base.

For example, MYCIN stores considerable knowledge in the form of tables. Certain rules include LISP code that retrieves information from these tables, such as characteristics of disease organisms. This improves the efficiency of the knowledge base, since a single table can capture information that would require numerous rules to encode. However, if a new object, such as a disease organism, is added to the knowledge base, then all tables that contain information about disease organisms must be updated to reflect this.

Semantic consistency is a more difficult problem, requiring knowledge of how well the knowledge base actually reflects the problem domain. Although Teiresias cannot be said to truly maintain semantic consistency, it does make an effort to ensure that rules appear consistent with one another. It does this by maintaining *rule models*, or statistical profiles of all the rules. If a new rule is added to the knowledge base, Teiresias compares this new rule with the rule model for all existing rules that draw the same conclusion. The assumption is that rules that draw similar conclusions should examine similar conditions in their premises. This enables Teiresias to detect potential inconsistencies in a new rule and to prompt the user for corrections.

To illustrate Teiresias, we analyze its interaction with the doctor after MYCIN has given its recommendations. The doctor disagrees with MYCIN's result and with Teiresias' help analyzes MYCIN's solution. The trace is a simulation taken from a session by Stanford researchers, where the analysis is in italics, Teiresias' questions are in regular type, and the doctor's responses are in boldface type.

When a conclusion is given with which the doctor disagrees, the rule that drew this conclusion is not necessarily wrong. A number of things could have happened, as Teiresias correctly shows: A fact of the rule could have been improperly concluded; the rule might, in fact, be wrong; or another rule might be missing from the knowledge base. Finding the cause of an error is a difficult problem that potentially requires examination of the entire line of reasoning. Teiresias assists the physician in determining the problem by maintaining an AND/OR graph of the solution process and working systematically through this trace.

All rules are translated into an English language syntax that assists the physician in understanding the trace. Once a problem is located, Teiresias lets the user add the missing rule in the same simplified syntax, translating the rule into an internal representation and checking the new rule for consistency with similar rules in the knowledge base.

MYCIN has reached a set of infections that it feels should be treated (as in the dialogue of Sec. 9.4.3). These are presented to the physician, who disagrees with MYCIN's analysis:

```
My therapy recommendation will be based on the following probable
infections and potential causative organisms:
        INFECTION-1 is PRIMARY BACTEREMIA
        <item-1>E. COLI
        <item-2>PSEUDOMONAS-AERUGINOSA
```

Now Teiresias steps in:

```
Are all these identities correct?
No
Are there any identities missing?
Yes
Which ones?
KLEBSIELLA PNEUMONIAE, PROTEUS-NON-MIRABILIS
Are there any identities which shouldn't have appeared?
Yes
Which ones?
PSEUDOMONAS-AERUGINOSA
Do you want to
(1) track down the problem yourself, or
(2) allow me to guide the debugging, or
(3) ignore it for now and fix it another time
      Type 1, 2, or 3
2
```

If the doctor had typed 1, he or she could have tried to sort out the problems with the traditional why and how queries on the preceding solution. Note that Teiresias has already begun to categorize the errors that appeared in the conclusions. Also note that the three problems could well be related (perhaps to one missing rule). Teiresias starts by trying to find out why the wrong organism appeared for treatment. . . .

```
The following rules were used in deducing that the identity of
ORGANISM-1 is PSEUDOMONAS-AERUGINOSA.
      RULE 184
      Since
            [1.1] The category of ORGANISM-1 is not known
            [1.2] The gram stain of ORGANISM-1 is gram-negative
            [1.3] The morphology of ORGANISM-1 is rod
            [1.4] The aerobicity of ORGANISM-1 is facultative
      There is weakly suggestive evidence (0.3) that the identity
      of ORGANISM-1 is PSEUDOMONAS-AERUGINOSA.
```

```
Is this rule correct?
Yes
Should its premise have failed in this case?
Yes
Which clause of the premise should have failed?
Clause 1
```

The expert feels the program should have been able to establish a category and that rule 184 should not have fired.

```
Should it have been possible to conclude a category?
Yes
Which one?
ENTEROBACTERIACEAE
There were no rules successfully called in the attempt to deduce
this category. The following (potentially applicable) rules were NOT
executed because the marked [++] clause failed:
     RULE 116
     If
        (1) the identity of ORGANISM-1 is not known
      ++(2) the gram stain of ORGANISM-1 is not known
        (3) the morphology of ORGANISM-1 is not known
        (4) the site of CULTURE-1 is CSF
        (5) the infection is meningitis
        (6) the age of the patient is less than or equal to 2 months
       Then
             there is weakly suggestive evidence (0.33) that the
             category of ORGANISM-1 is ENTEROBACTERIACEAE
     RULE 050
     If
        (1) the morphology of ORGANISM-1 is rod
        (2) the gram stain of ORGANISM-1 is gram-negative
        (3) the aerobicity of ORGANISM-1 is facultative
      ++(4) infection ORGANISM-1 was acquired while the
                   patient was hospitalized
     Then
             there is suggestive evidence (0.7) that the category of
             ORGANISM-1 is ENTEROBACTERIACEAE
Are all these rules correct?
Yes
Should any of them have been successfully invoked?
No
Is there a rule missing?
Yes
Would you like to give me such a rule now?
Yes
Good ...
       If
             1 ** THE PATIENT'S INFECTION IS PRIMARY-
                  BACTEREMIA
             2 ** THE SITE IS ONE OF THE STERILE SITES
       Then:
             ** THERE IS SUGGESTIVE EVIDENCE (0.8) THAT THE
             CATEGORY IS ENTEROBACTERIACEAE
```

Teiresias parses this rule into its internal representation and then translates it back into English so the expert can check it. After the expert agrees on the parse, Teiresias attempts to integrate the new rule into the knowledge base. . . .

```
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?
Yes
How about:
  the portal of entry is gastrointestinal. OK?
Yes
```

Teiresias is itself a knowledge-based program with rules for reasoning about rules and objects in a MYCIN knowledge base. As the above trace illustrates, the program systematically searches back along the trace of the reasoning to find the source of the problem. Once this is done, it helps the user add to or correct the knowledge base. Using the rule models discussed at the beginning of this section, Teiresias is able to check the rule for consistency and to correct the missing premise.

To help the user add new information, Teiresias keeps a model of each class of object that appears in the system. The model for a MYCIN object is called a *schema*. Each schema describes how to create new instances of a class of objects such as a new disease organism, drug, or test. The schema for a class of objects also describes how information is to be obtained for rules of that class, e.g., to compute it from existing data, look it up, or ask the user.

Schemata record the interrelationships of all classes of objects in the MYCIN rule base. For example, if the addition of a new disease requires that a table of diseases and their sensitivities to various drugs also be updated, this is recorded in the disease schema. Schemata also include pointers to all current instances of each schema. This is very important if it is decided to change the form of the schema throughout the program.

Teiresias organizes its schemata into a hierarchy: A schema for describing bacterial infections may be a specification of a general class schema, which in turn is a specification of a general schema for MYCIN facts. Each schema includes pointers to its parents and children in this hierarchy.

Schemata also contain bookkeeping information. This includes the author and date of creation and addition of each instance to the program. It includes a description of the schema used to create it. Documentation of who created rules when is critical for discussion and analysis of the knowledge base; when rules are primarily condition-action pairs representing important aspects of an application domain and not full explications of these relationships, it is important to be able to trace the rules back to their authors for more complete discussion of the factors underlying their creation.

The top-level structure in the hierarchy of Teiresias' knowledge is the *schema-schema*. This structure is a schema for creating new schemata, when, e.g., the domain expert might wish to create a new category of objects. The schema-schema allows Teiresias to reason about its own knowledge structures, including creating new ones when appropriate. A schema-schema has the same basic structure as a schema itself and provides all the bookkeeping information to reconstitute the entire knowledge base.

Although the development of commercial programs to help with knowledge acquisition is still in the future, Teiresias has shown how AND/OR graph search and knowledge representation techniques provide a basis for automating knowledge-base refinement.

9.5 EPILOGUE

A number of references complement the material presented in this chapter; especially recommended is a collection of the original MYCIN publications from Stanford entitled *Rule-Based Expert Systems* by Buchanan and Shortliffe [2].

Other important books on general knowledge engineering include *Building Expert Systems* by Hayes-Roth et al. [12], *A Guide to Expert Systems* by Waterman [23], *Expert Systems: Concepts and Examples* by Alty and Coombs [1], *Expert Systems Technology: A Guide* by Johnson and Keravnou [13], *Expert Systems: Tools and Applications* by Harmon et al. [11], and *Expert Systems and Fuzzy Systems* by Negoita [18].

Because of the domain specificity of expert-system solutions, case studies are an important source of knowledge in the area. Books in this category include *Expert Systems: Techniques, Tools, and Applications* by Klahr and Waterman [15], *Competent Expert Systems: A Case Study in Fault Diagnosis* by Keravnou and Johnson [14], *The CRI Directory of Expert Systems* by Smart and Langeland-Knudsen [21], and *Developments in Expert Systems* by Coombs [4].

A pair of more general books that give an overview of commercial uses of AI are *The AI Business*, edited by Winston and Prendergast [24], and *Expert Systems: Artificial Intelligence in Business* by Harmon and King [10].

REFERENCES

1. J. L. Alty and M. J. Coombs, *Expert Systems: Concepts and Examples*. Manchester, Eng.: NCC Publications, 1984.
2. J. Bachant and J. McDermott, "R1 Revisited: Four Years in the Trenches," *AI Magazine*, 5(3):21–32, 1984.
3. B. G. Buchanan and E. H. Shortliffe (eds.), *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Reading, Mass.: Addison-Wesley, 1984.
4. M. J. Coombs (ed.), *Developments in Expert Systems*. New York: Academic Press, 1984.
5. R. Davis, "Applications of Meta Level Knowledge to the Construction, Maintenance, and Use of Large Knowledge Bases," in R. Davis and D. B. Lenat, *Knowledge-Based Systems in Artificial Intelligence*. New York: McGraw-Hill, 1982.
6. R. Davis and D. B. Lenat, *Knowledge-Based Systems in Artificial Intelligence*. New York: McGraw-Hill, 1982.
7. J. de Kleer, "An Assumption Based Truth Maintenance System," *Artificial Intelligence*, 28:127–162, 1986.
8. J. Doyle, "AAAA Truth Maintenance System," *Artificial Intelligence*, vol. 12, 1979.
9. R. O. Duda, J. Gaschnig, and P. E. Hart, "Model Design in the PROSPECTOR Consultant System for Mineral Exploration," in D. Michie (ed.), *Expert Systems in the Micro-Electronic Age*. Edinburgh, Scotland: Edinburgh University Press, 1979.
10. P. Harmon and D. King, *Expert Systems: Artificial Intelligence in Business*. New York: Wiley, 1985.
11. P. Harmon, R. Maus, and W. Morrissey, *Expert Systems: Tools and Applications*. New York: Wiley, 1988.
12. F. Hayes-Roth, D. Waterman, and D. Lenat, *Building Expert Systems*. Reading, Mass.: Addison-Wesley, 1984.
13. L. Johnson and E. T. Keravnou, *Expert Systems Technology: A Guide*. Cambridge, Mass.: Abacus Press, 1985.

14. E. T. Keravnou and L. Johnson, *Competent Expert Systems: A Case Study in Fault Diagnosis.* London: Kegan Paul, 1985.

15. P. Klahr and D. A. Waterman (eds.), *Expert Systems: Techniques, Tools, and Applications.* Reading, Mass.: Addison-Wesley, 1986.

16. G. F. Luger and W. A. Stubblefield, *Artificial Intelligence and the Design of Expert Systems.* Palo Alto, Calif.: Benjamin/Cummings, 1989.

17. J. McDermott, "R1, the Formative Years," *AI Magazine,* 2(2):21–29, 1981.

18. C. V. Negoita, *Expert Systems and Fuzzy Systems.* Menlo Park, Calif.: Benjamin/Cummings, 1984.

19. S. Papert, *Mindstorms.* New York: Basic Books, 1980.

20. T. W. Pratt, *Programming Languages: Design and Duplementation.* Englewood Cliffs, N.J.: Prentice-Hall, 1984.

21. G. Smart and J. Langeland-Knudsen, *The CRI Directory of Expert Systems.* Oxford, Eng.: Learned Information (Europe) Ltd., 1986.

22. E. Soloway, J. Bachant, and K. Jensen, "Assessing the Maintainability of XCON-in-RIME: Coping with the Problems of a Very Large Rule Base," *Proceedings of AAAI-87.* Los Altos, Calif.: Bowman Kaufmann, 1987.

23. D. A. Waterman, *A Guide to Expert Systems.* Reading, Mass.: Addison-Wesley, 1986.

24. P. H. Winston and K. A. Prendergast (eds.), *The AI Business.* Cambridge, Mass.: MIT Press, 1984.

25. L. A. Zadeh, "Commonsense Knowledge Representation Based on Fuzzy Logic," *Computer,* 16:61–65, 1983.