8 STRONG METHOD PROBLEM SOLVING

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

Nam et ipsa scientia potestas est (the knowledge is power).

—FRANCIS BACON (1620)

8.0 Introduction

We continue studying issues of representation and intelligence by considering an important component of AI: *knowledge-intensive* or *strong method* problem solving.

Human experts are able to perform at a successful level because they know a lot about their areas of expertise. This simple observation is the underlying rationale for the design of strong method or knowledge-based problem solvers (Introduction, Part III). An *expert system*, for example, uses knowledge specific to a problem domain to provide "expert quality" performance in that application area. Generally, expert system designers acquire this knowledge with the help of human domain experts, and the system emulates the human expert's 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: the human experts that provide the system's knowledge have generally augmented their own theoretical understanding of the problem domain with tricks, shortcuts, and heuristics for *using* the knowledge they have gained through problem-solving experience.

Because of their heuristic, knowledge-intensive nature, expert systems generally:

1. Support inspection of their reasoning processes, both in presenting intermediate steps and in answering questions about the solution process.
2. Allow easy modification in adding and deleting skills from the knowledge base.
3. Reason heuristically, using (often imperfect) knowledge to get useful solutions.

The reasoning of an expert system should be open to inspection, providing information about the state of its problem solving and explanations of the choices and decisions that the program is making. Explanations are important for a human expert, such as a doctor or an engineer, if he or she is to accept the recommendations from a computer. Indeed, few human experts will accept advice from another human, let alone a machine, without understanding the justifications for it.

The exploratory nature of AI and expert system programming requires that programs be easily prototyped, tested, and changed. AI programming languages and environments are designed to support this iterative development methodology. In a pure production system, for example, the modification of a single rule has no global syntactic side effects. Rules may be added or removed without requiring further changes to the larger program. Expert system designers often comment that easy modification of the knowledge base is a major factor in producing a successful program.

A further feature of expert systems is their use of heuristic problem-solving methods. As expert system designers have discovered, informal "tricks of the trade" and "rules of thumb" are an essential complement to the standard theory presented in textbooks and classes. Sometimes these rules augment theoretical knowledge in understandable ways; often they are simply shortcuts that have, empirically, been shown to work.

Expert systems are built to solve a wide range of problems in domains such as medicine, mathematics, engineering, chemistry, geology, computer science, business, law, defense, and education. These programs address a variety of problems; the following list, from Waterman (1986), is a useful summary of general expert system problem categories.

Interpretation—forming high-level conclusions from collections of raw data.

Prediction—projecting probable consequences of given situations.

Diagnosis—determining the cause of malfunctions in complex situations based on observable symptoms.

Design—finding a configuration of system components that meets performance goals while satisfying a set of design constraints.

Planning—devising a sequence of actions that will achieve a set of goals given certain starting conditions and run-time constraints.

Monitoring—comparing a system's observed behavior to its expected behavior.

Instruction—assisting in the education process in technical domains.

Control—governing the behavior of a complex environment.

In this chapter we first examine the technology that makes knowledge-based problem solving possible. Successful *knowledge engineering* must address a range of problems, from the choice of an appropriate application domain to the acquisition and formalization of problem-solving knowledge. In Section 8.2 we introduce rule-based systems and present the production system as a software architecture for solution and explanation processes. Section 8.3 examines techniques for model-based and case-based reasoning. Section 8.4 presents *planning*, a process of organizing pieces of knowledge into a consistent sequence of actions that will accomplish a goal. Chapter 9 presents techniques for reasoning in uncertain situations, an important component of strong method problem solvers.

8.1 Overview of Expert System Technology

8.1.1 The Design of Rule-Based Expert Systems

Figure 8.1 shows the modules that make up a typical expert system. The user interacts with the system through a *user interface* that simplifies communication and hides much of the complexity, such as the internal structure of the rule base. Expert system interfaces employ a variety of user styles, including question-and-answer, menu-driven, or graphical interfaces. The final decision on the interface type is a compromise between user needs and the requirements of the knowledge base and inferencing system.

The heart of the expert system is the *knowledge base*, which contains the knowledge of a particular application domain. In a rule-based expert system this knowledge is most often represented in the form of *if... then...* rules, as in our examples of Section 8.2. The knowledge base contains both *general knowledge* as well as *case-specific* information.

The *inference engine* applies the knowledge to the solution of actual problems. It is essentially an interpreter for the knowledge base. In the production system, the inference engine performs the recognize-act control cycle. The procedures that implement the

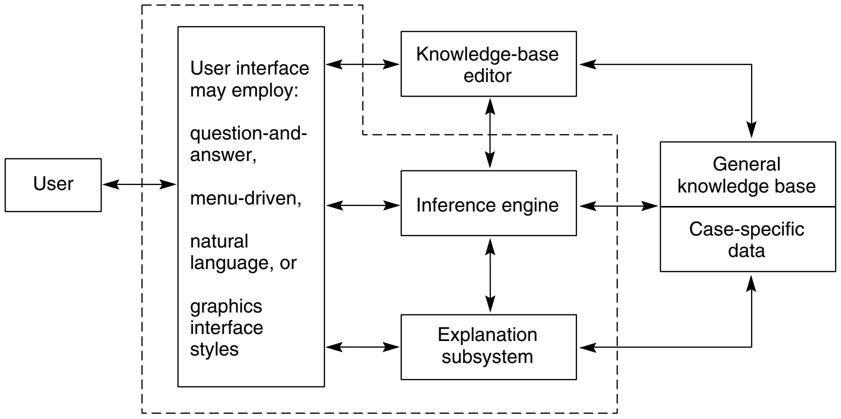


Figure 8.1 Architecture of a typical expert system for a particular problem domain.

control cycle are separate from the production rules themselves. It is important to maintain this separation of the knowledge base and inference engine for several reasons:

1. This separation makes it possible to represent knowledge in a more natural fashion. *If ... then...* rules, for example, are closer to the way in which humans describe their problem-solving skills than is lower-level computer code.
2. Because the knowledge base is separated from the program's lower-level control structures, expert system builders can focus on capturing and organizing problem-solving knowledge rather than on the details of its computer implementation.
3. Ideally, the separation of knowledge and control allows changes to be made in one part of the knowledge base without creating side effects in others.
4. The separation of the knowledge and control elements of the program allows the same control and interface software to be used in a variety of systems. The *expert system shell* has all the components of Figure 8.1 except that the knowledge base and case-specific data are empty and can be added for a new application. The broken lines of Figure 8.1 indicate the shell modules.

The expert system must keep track of *case-specific data*: the facts, conclusions, and other information relevant to 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, in response to *how queries* (Section 8.2), explanations of why the system needs a particular piece of data, *why queries* (Section 8.2), and, where useful, tutorial explanations or deeper theoretical justifications of the program's actions.

Many systems also include a *knowledge-base editor*. Knowledge-base editors help the programmer locate and correct bugs in the program's performance, often accessing the information provided by the explanation subsystem. They also may assist in the addition of new knowledge, help maintain correct rule syntax, and perform consistency checks on any updated knowledge base.

An important reason for the decrease in design and deployment times for current expert systems is the availability of *expert system shells*. NASA has created CLIPS, JESS is available from Sandia National Laboratories, and we offer shells in Lisp and Prolog in our supplementary materials. Unfortunately, shell programs do not solve all of the problems involved in building expert systems. Although 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 formalization of domain knowledge still remain difficult tasks.

8.1.2 Selecting a Problem and the Knowledge Engineering Process

Expert systems involve a considerable investment of money and human effort. Attempts to solve a problem that is too complex, too poorly understood, or otherwise unsuited to the available technology can lead to costly and embarrassing failures. Researchers have developed guidelines to determine 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**. Many expert systems have been built in domains such as mineral exploration, business, defense, and medicine where a large potential exists for savings in terms of money, time, and human life.

2. **Human expertise is not available in all situations where it is needed**. In geology, for example, there is a need for expertise at remote mining and drilling sites. Often, geologists and other 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.

3. **The problem may be solved using symbolic reasoning.** Problem solutions should not require physical dexterity or perceptual skill. Robots and vision systems currently lack the sophistication and flexibility of humans.

4. **The problem domain is well structured and does not require common sense reasoning**. Highly technical fields have the advantage of being well studied and formalized: terms are well defined and domains have clear and specific conceptual models. In contrast, common sense reasoning is difficult to automate.

5. **The problem may not be solved using traditional computing methods**. Expert system technology should not be used where unnecessary. If a problem can be solved satisfactorily using more traditional techniques, then it is not a candidate.

6. **Cooperative and articulate experts exist.** The knowledge used by expert systems 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 knowledge.

7. **The problem is of proper size and scope.** For example, a program that attempted to capture all of the expertise of a medical doctor would not be feasible; a program that advised MDs on the use of a particular piece of diagnostic equipment or a particular set of diagnoses would be more appropriate.

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 the domain expert articulate the necessary knowledge, and implement that knowledge in a correct and efficient knowledge base. Often, the knowledge engineer is initially 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, such as shortcuts, handling imprecise data, evaluating partial solutions, and all the other skills that mark a person as an expert problem solver. 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 by and large wasted. The skills and needs of the user must be considered throughout the design cycle: Will the program make the user's work easier, quicker, more comfortable? What level of explanation does the user need? Can the user provide correct information to the system? Is the interface appropriate? Does the user's work environment place restrictions on the program's use? An interface that required typing, for example, would not be appropriate for use in the cockpit of a fighter.

Like most AI programming, building expert systems requires a nontraditional development cycle based on early prototyping and incremental revision of the code. Generally, work on the system begins with the knowledge engineer 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 and by observing experts during the performance of their job. 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 him or her explain the techniques used in their solution. Video and/or audio tapes are often essential for capturing this process.

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 their field. Knowledge engineers, by virtue of their relative naiveté in the domain, can spot these conceptual jumps and ask for help.

Once the knowledge engineer has obtained a general overview of the problem domain and gone through several problem-solving sessions with the 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 search strategy, forward, backward, depth-first, best-first etc., 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 provide 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, the prototype can be incrementally extended until it becomes a final 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. Figure 8.2 presents a flow chart describing the exploratory programming development cycle. This approach to programming was investigated by Seymour Papert with his LOGO language (Papert 1980) as well as Alan Kay's work with Smalltalk at Xerox PARC. 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. This process of trying and correcting candidate designs is common to expert systems development, and contrasts with such neatly hierarchical processes as top-down design.

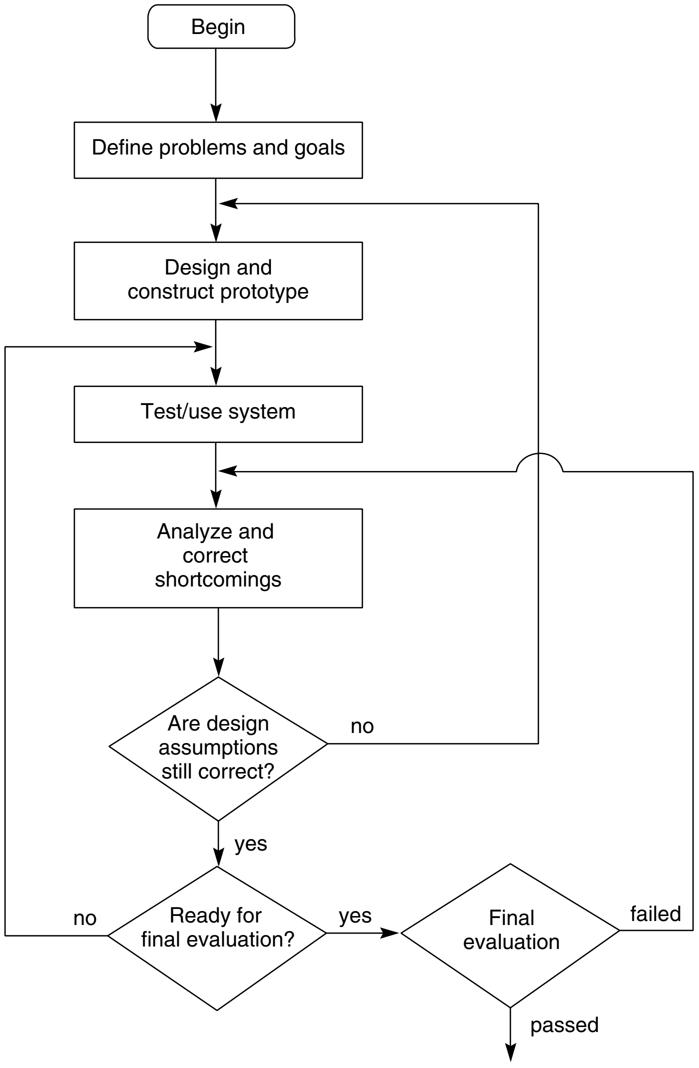

Begin
Define problems and goals
Design and construct prototype
Test/use system
Analyze and correct shortcomings
Are design assumptions still correct?
no
yes
Ready for final evaluation?
no
yes
Final evaluation
failed
passed

Figure 8.2 Exploratory development cycle.

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 then often write a cleaner version, usually with fewer 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 of 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.

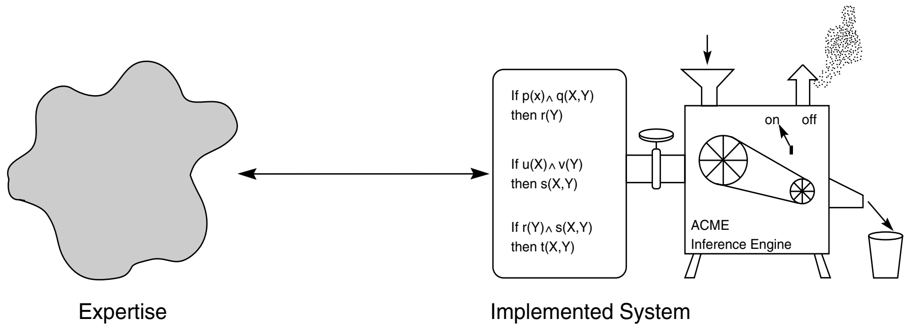


Figure 8.3 The standard view of building an expert system.

8.1.3 Conceptual Models and Their Role in Knowledge Acquisition

Figure 8.3 presents a simplified model of the knowledge acquisition process that will serve as a useful "first approximation" for understanding the problems involved in acquiring and formalizing human expert performance. The human expert, working in an application area, operates in a domain of knowledge, skill, and practice. This knowledge is often vague, imprecise, and only partially verbalized. The knowledge engineer must translate this informal expertise into a formal language suited to a computational system. A number of important issues arises in the process of formalizing human skilled performance:

1. Human skill is often inaccessible to the conscious mind. As Aristotle points out in his *Ethics*, "what we have to learn to do, we learn by doing." Skills such as those possessed by medical doctors are learned as much in years of internship and residency, with their constant focus on patients, as they are in physiology lectures, where emphasis is on experiment and theory. Delivery of medical care is to a great extent practice-driven. After years of performance these skills become highly integrated and function at a largely unconscious level. It may be difficult for experts to describe exactly what they are doing in problem solving.
2. Human expertise often takes the form of knowing *how* to cope in a situation rather than knowing *what* a rational characterization of the situation might be, of developing skilled performance mechanisms rather than a fundamental understanding of what these mechanisms are. An obvious example of this is riding a unicycle: the successful rider is not, in real time, consciously solving multiple sets of simultaneous differential equations to preserve balance; rather she is using an intuitive combination of feelings of "gravity," "momentum," and "inertia" to form a usable control procedure.

3. We often think of knowledge acquisition as gaining factual knowledge of an objective reality, the so-called "real world". As both theory and practice have shown, human expertise represents an individual's or a community's *model* of the world. Such models are as influenced by convention, social processes, and hidden agendas as they are by empirical methodologies.
4. Expertise changes. Not only do human experts gain new knowledge, but also existing knowledge may be subject to radical reformulation, as evidenced by ongoing controversies in both scientific and social fields.

Consequently, knowledge engineering is difficult and should be viewed as spanning the life cycle of any expert system. To simplify this task, it is useful to consider, as in Figure 8.4, a *conceptual model* that lies between human expertise and the implemented program. By a conceptual model, we mean the knowledge engineer's evolving conception of the domain knowledge. Although this is undoubtedly different from the domain expert's, it is this model that actually determines the construction of the formal knowledge base.

Because of the complexity of most interesting problems, we should not take this intermediate stage for granted. Knowledge engineers should document and make public their assumptions about the domain through common software engineering methodologies. An expert system should include a requirements document; however, because of the constraints of exploratory programming, expert system requirements should be treated as co-evolving with the prototype. Data dictionaries, graphic representations of state spaces, and

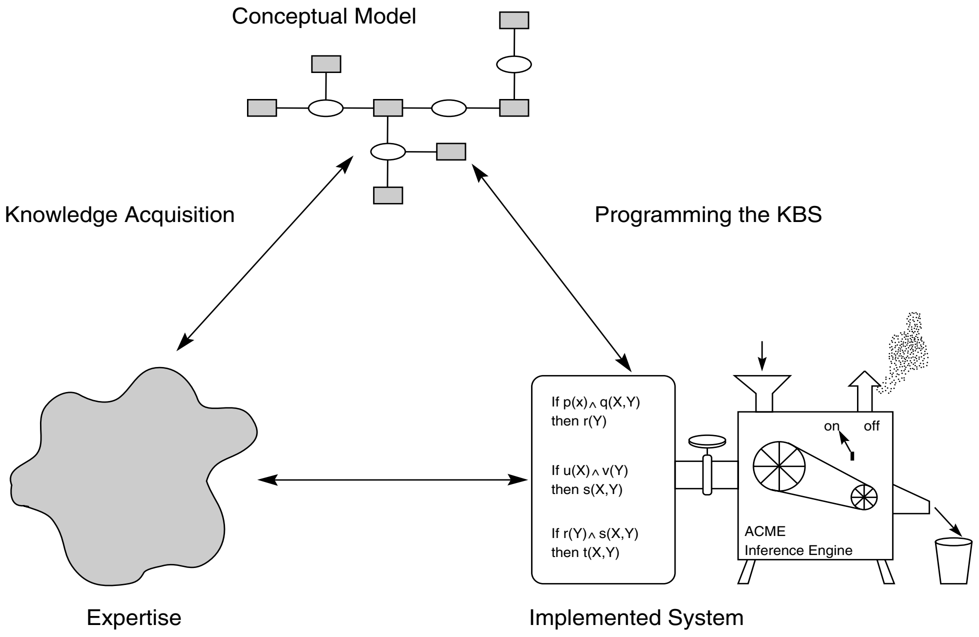


Figure 8.4 The role of mental or conceptual models in problem solving.

comments in the code itself are all part of this model. By publicizing these design decisions, we reduce errors in both the implementation and maintenance of the program.

Knowledge engineers should save recordings of interviews with domain experts. Often, as knowledge engineers' understanding of the domain grows, they form new interpretations or discover new information about the domain. The recordings, along with documentation of the interpretation given them, play a valuable role in reviewing design decisions and testing prototypes. Finally, this model serves an intermediate role in the formalization of knowledge. The choice of a representation language exerts a strong influence on a knowledge engineer's model of the domain.

The conceptual model is not formal or directly executable on a computer. It is an intermediate design construct, a template to begin to constrain and codify human skill. It can, if the knowledge engineer uses a predicate calculus model, begin as a number of simple networks representing states of reasoning through typical problem-solving situations. Only after further refinement does this network become explicit *if... then...* rules.

Questions often asked in the context of a conceptual model include: Is the problem solution deterministic or search-based? Is the reasoning data-driven, perhaps with a generate and test flavor, or goal-driven, based on a small set of likely hypotheses about situations? Are there stages of reasoning? Is the domain well understood and capable of providing deep predictive models, or is all problem-solving knowledge essentially heuristic? Can we use examples of past problems and their solutions to solve future problems directly, or must we first convert these examples into general rules? Is the knowledge exact or is it "fuzzy" and approximate, lending itself to numeric ratings of certainty (Chapter 9)? Will reasoning strategies allow us to infer stable facts about the domain, or do change and uncertainty within the system require nonmonotonic reasoning, that is, the ability to make assertions about the domain that may later be modified or retracted (Section 9.1)? Finally, does the structure of the domain knowledge require us to abandon rule-based inference for alternative schemes such as neural networks or genetic algorithms (Part IV)?

The eventual users' needs should also be addressed in the context of the conceptual model: What are their expectations of the eventual program? Where is their level of expertise: novice, intermediate, or expert? What levels of explanation are appropriate? What interface best serves their needs?

Based on the answers to these and relatd questions, the knowledge obtained from domain experts, and the resulting conceptual model, we begin development of the expert system. Because the production system, first presented in Chapter 6, offers a number of inherent strengths for organizing and applying knowledge, it is often used as the basis for knowledge representation in rule-based expert systems.

8.2 Rule-Based Expert Systems

Rule-based expert systems represent problem-solving knowledge as *if... then...* rules. This approach lends itself to the architecture of Figure 8.1, and is one of the oldest techniques for representing domain knowledge in an expert system. It is also one of the most natural, and remains widely used in practical and experimental expert systems.

8.2.1 The Production System and Goal-Driven Problem Solving

The architecture of rule-based expert systems may be best understood in terms of the production system model for problem solving presented in Part II. The parallel between the two is more than an analogy: the production system was the intellectual precursor of modern expert system architectures, where application of production rules leads to refinements of understanding of a particular problem situation. When Newell and Simon developed the production system, their goal was to model human performance in problem solving.

If we regard the expert system architecture in Figure 8.1 as a production system, the domain-specific knowledge base is the set of production rules. In a rule-based system, these condition action pairs are represented as *if... then...* rules, with the premises of the rules, the *if* portion, corresponding to the condition, and the conclusion, the *then* portion, corresponding to the action: when the condition is satisfied, the expert system takes the action of asserting the conclusion as true. Case-specific data can be kept in the working memory. The inference engine implements the recognize-act cycle of the production system; this control may be either data-driven or goal-driven.

Many problem domains seem to lend themselves more naturally to forward search. In an interpretation problem, for example, most of the data for the problem are initially given and it is often difficult to formulate an hypotheses or goal. This suggests a forward reasoning process in which the facts are placed in working memory and the system searches for an interpretation, as first presented in Section 3.2.

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's goal into simpler subgoals. The process continues in the next iteration of the production system, 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, as also first presented in Section 3.2.

In an expert system, subgoals can be solved by asking the user for information. Some expert systems allow the system designer to specify which subgoals may be solved by asking the user. Others simply ask the user about any subgoals that fail to match rules in the knowledge base; i.e., if the program cannot infer the truth of a subgoal, it asks the user.

As an example of goal-driven problem solving with user queries, we next offer a small expert system for analysis of automotive problems. This is not a full diagnostic system, as it contains only four very simple rules. It is intended as an example to demonstrate goal-driven rule chaining, the integration of new data, and the use of explanation facilities:

```
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 shown in Figure 8.5. X is a variable that can match with any phrase, as an example, the problem is battery or cables; it will become bound to the solution when the problem is solved.

Three rules match with this expression in working memory: rule 1, rule 2, and rule 3. If we resolve conflicts in favor of the lowest-numbered rule, then rule 1 will fire. This causes X to be bound to the value spark plugs and the premises of rule 1 to be placed in the working memory as in Figure 8.6. The system has thus chosen to explore the possible

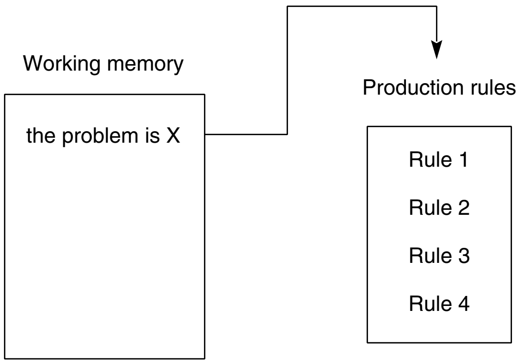


Figure 8.5 The production system at the start of a consultation in the car diagnostic example.

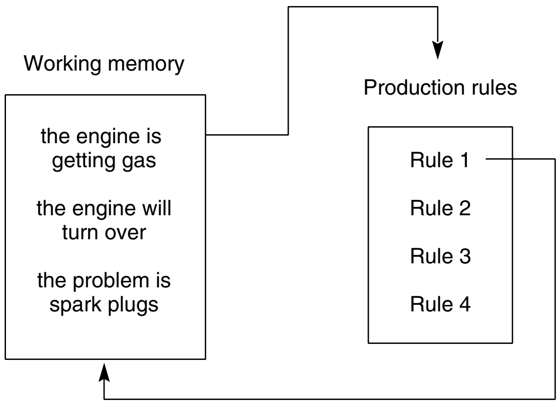

Working memory
the engine is getting gas
the engine will turn over
the problem is spark plugs
Production rules
Rule 1
Rule 2
Rule 3
Rule 4

Figure 8.6 The production system after Rule 1 has fired.

hypothesis that the spark plugs are bad. Another way to look at this is that the system has selected an or branch in an and/or graph (Chapter 3).

Note that there are two premises to rule 1, both of which must be satisfied to prove the conclusion true. These are and branches of the search graph representing a decomposition of the problem (finding whether the problem is spark plugs) into two subproblems (finding whether the engine is getting gas and whether the engine will turn over). We may then fire rule 4, whose conclusion matches with the engine is getting gas, causing its premises to be placed in working memory as in Figure 8.7.

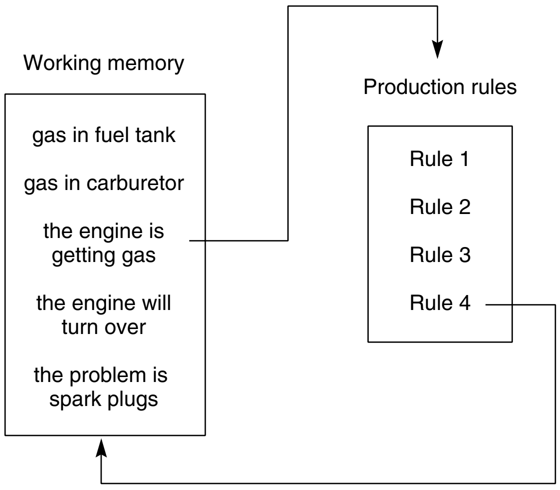


Figure 8.7 The system after Rule 4 has fired. Note the stack-based approach to goal reduction.

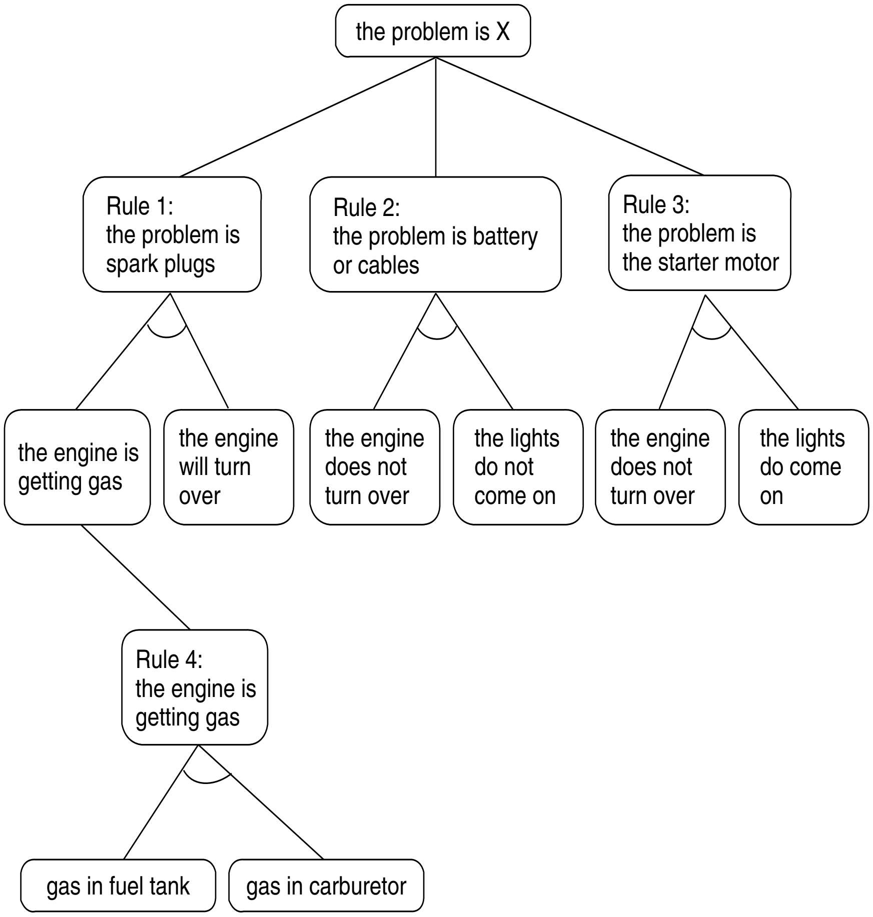

the problem is X
Rule 1: the problem is spark plugs
Rule 2: the problem is battery or cables
Rule 3: the problem is the starter motor
the engine is getting gas
the engine will turn over
the engine does not turn over
the lights do not come on
the engine does not turn over
the lights do come on
Rule 4: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

Figure 8.8 . The and/or graph searched in the car diagnosis example, with the conclusion of Rule 4 matching the first premise of Rule 1.

At this point, there are three entries in working memory that do not match with any rule conclusions. Our expert system will, in this situation, 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 leftmost branch of the and/or graph presented in Figure 8.8.

This is, of course, a very simple example. Not only is its automotive knowledge limited at best, but it also ignores a number of important aspects of real implementations. The rules are phrased in English, rather than a formal language. On finding a solution, a real expert system will tell the user its diagnosis (our model simply stops). Also, we should maintain enough of a trace of the reasoning to allow backtracking if necessary. In our example, had we failed to determine that the spark plugs were bad, we would have needed to back up to the top level and try rule 2 instead. Notice that this information is implicit in the ordering of subgoals in working memory of Figure 8.7 and in the graph of Figure 8.8. In spite of its simplicity, however, this example underscores the importance of production system based search and its representation by the and/or graph as a foundation for rule-based expert systems.

Earlier we emphasized that an expert system needed to be open to inspection, easily modified, and heuristic in nature. The production system architecture is an important factor in each of these requirements. Ease of modification, for example, is supported by the syntactic independence of production rules: each rule is a chunk of knowledge that can be independently modified. There are semantic constraints, however, in that the meanings of the individual rules are related. These must be reconciled in any editing/modification process. We next discuss the generation of explanations and the monitoring of inference.

8.2.2 Explanation and Transparency in Goal-Driven Reasoning

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

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

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

gas in fuel tank?
yes
gas in carburetor?
yes
engine will turn over?
why

Here the user enters a why query, i.e., why do you want to know whether the engine will turn over? The system responds with a literal presentation of the rule under question:

It has been established that:
1. the engine is getting gas,
therefore if
2. the engine will turn over,
then the problem is spark plugs.

Because 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 for the system's behavior is just a restatement of the current rule under consideration. However, this provides an adequate explanation for why the program wants to know whether the engine will turn over. In the response to the why query, the system states that it has concluded that the engine is getting gas. Suppose the user then asks:

how the engine is getting gas

The system will respond to how it concluded that the engine is getting gas. The response is a trace of the reasoning that led to this conclusion, working back from the goal along the rules that support it to the user responses:

```
This follows from rule 4:
if
gas in fuel tank, and
gas in carburetor
then
engine is getting gas.
gas in fuel tank was given by the user
gas in carburetor was given by the user
```

Once again, 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. Because each rule represents a complete chunk of problem-solving knowledge, the current rule provides a context for the explanation. Contrast this production system approach with more traditional program architectures: if we stop a C, C++, or Java program in mid execution, it is doubtful that the current statement will have much meaning.

In summary, the knowledge-based system answers why queries by showing the current rule that it is attempting to fire; it answers how queries by giving a trace of the reasoning that led to a goal or subgoal. Although the mechanisms are conceptually simple, they can exhibit remarkable explanatory power if the knowledge base is organized in a logical fashion. The supplementary materials in Java, Lisp and Prolog, demonstrate the use of rule stacks and proof trees to implement these explanations.

If explanations are to behave logically, it is important not only that the knowledge base gets the correct answer but also that each rule corresponds 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 vague, arbitrary, or illogical in responding to how and why queries. This can not only undermine the user's faith in the system but can also make the program much more difficult for its builders to understand and modify.

8.2.3 Using the Production System for Data-Driven Reasoning

The automobile diagnosis demonstration of Section 8.2.1 illustrated the use of a production system to implement goal-driven search. Search was also depth-first in that it searched each subgoal found in the rule base exhaustively before moving onto any sibling goals. As we saw in Section 6.3, however, the production system is also an ideal architecture for data-driven reasoning. Example 6.3.1 demonstrated this process with the 8-puzzle and Examples 6.3.2 and 6.3.3 with the Knight's Tour. In each of these problems we did conflict resolution by taking the first rule found in the knowledge base and then followed the results of that rule. This gave the search a depth-first flavor, although there was no mechanism, such as backtracking, to handle the problem of "dead ends" in the search space.

Breadth-first search is even more common in data-driven reasoning. The algorithm for this is simple: compare the contents of working memory with the conditions of each rule in the rule base according to the order of the rules in the rule base. If the data in working memory supports a rule's firing the result is placed in working memory and then control moves on to the next rule. Once all rules have been considered, search starts again at the beginning of the rule set.

Consider, for example, the automobile diagnosis problem and the rules of Section 8.2.1. If a piece of information that makes up (part of) the premise of a rule is not the conclusion of some other rule then that fact will be deemed "askable" when control comes to the situation (rule) where that information is needed. For example, the engine is getting gas is not askable in the premise of rule 1, because that fact is a conclusion of another rule, namely rule 4.

The breadth-first, data-driven example begins as in Figure 8.5, with no information in working memory, as in Figure 8.9. We first examine premises of the four rules in order to see what information is "askable". The premise, the engine is getting gas, is not askable, so rule 1 fails and control moves to rule 2. The engine does not turn over is askable. Suppose the answer to this query is false, so the engine will turn over is placed in working memory, as in Figure 8.10.

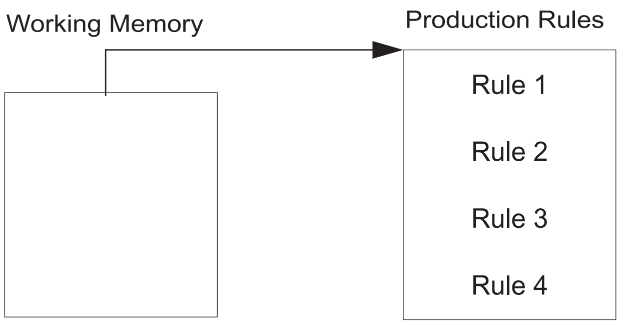


Figure 8.9 The production system at the start of a consultation for data-driven reasoning.

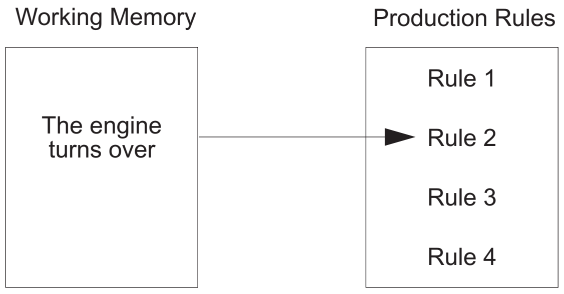


Figure 8.10 The production system after evaluating the first premise of Rule 2, which then fails.

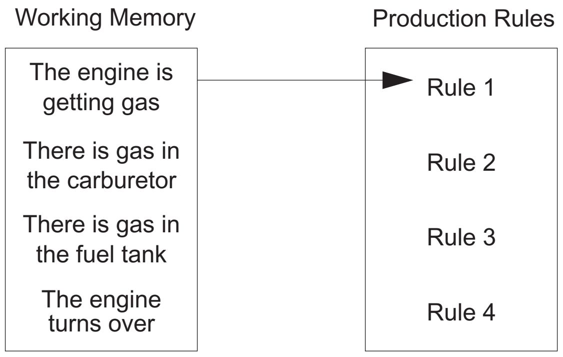

Working Memory
Production Rules
The engine is getting gas
There is gas in the carburetor
There is gas in the fuel tank
The engine turns over
Rule 1
Rule 2
Rule 3
Rule 4

Figure 8.11 The data-driven production system after considering Rule 4, beginning its second pass through the rules.

But rule 2 fails, since the first of two and premises is false, and consideration moves to rule 3, where again, the first premise fails. At rule 4, both premises are askable. Suppose the answer to both questions is true. Then there is gas in the fuel tank and there is gas in the carburetor are placed in working memory, as is the conclusion of the rule, the engine is getting gas.

At this point all the rules have been considered so search now returns, with the new contents of working memory, to consider the rules in order a second time. As is seen in Figure 8.11, when the working memory is matched to rule 1, its conclusion, the problem is spark plugs, is placed in working memory. In this example no more rules will match and fire, and the problem-solving session is completed. A graph of the search process, with the information content of working memory (WM) as the nodes of the graph, is presented as Figure 8.12.

An important refinement on the breadth-first search strategy used in the previous example is what is called *opportunistic search*. This search strategy is simple: whenever a rule fires to conclude new information, control moves to consider those rules which have that new information as a premise. This makes any new *concluded* information (search does not change as the result of “askable” premises) the controlling force for finding the next rules to fire. This is termed *opportunistic* because each conclusion of new information drives the search. By the accident of rule ordering, the very simple example just presented was also opportunistic.

We conclude this section on data-driven reasoning with several comments on explanation and transparency in forward chaining systems. First, in comparison with goal-driven systems, Sections 8.2.1–2, data-driven reasoning is much less “focused” in its search. The reason for this is obvious: in a goal-driven system, reasoning is in pursuit of a particular goal; that goal is broken into subgoals that support the top-level goal and these subgoals may be even further broken down. As a result, search is always directed through this goal and subgoal hierarchy. In data-driven systems this goal orientation does not exist. Rather, the search moves about the tree depending only on rule order and the discovery of new information. As a result, the progress of search can often seem to be very diffuse and unfocused.

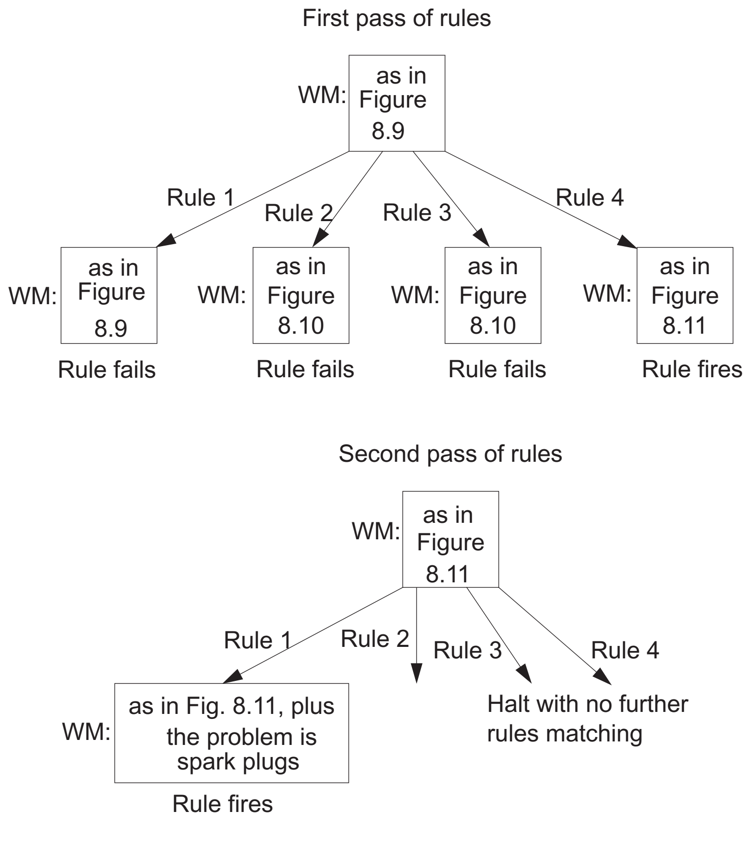


Figure 8.12 The search graph as described by the contents of working memory (WM) for the data-driven breadth-first search of the rule set of Section 8.2.1.

Second, and as a direct result of the first point, the explanation available to the user at any time in the search is quite limited. There is a rule-level accountability in that, when the user asks why some information is required, the why query in Section 8.2.2, the rule under consideration is presented. The explanation cannot go much further, however, unless explicit rule tracking is added into the system, say with opportunistic search. The diffuse nature of the data-driven search makes this difficult to do. Finally, when a goal is achieved, getting a full how explanation for that goal is also difficult. About the only thing that can be used as a partial and very limited explanation is a presentation of the contents of the working memory or a list of rules fired. But again, these will not offer the consistent focused accountability we saw with goal-driven reasoning.

8.2.4 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, an important method for the programmer to control search is through the structuring and ordering of the rules in the knowledge base. This micro-managing of the rule set offers an important opportunity, especially as 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 or steps for solving a problem: to do s, first do p, then do q, then do r. The role of rule and premise ordering was implicit in the examples of Section 8.2 just presented.

This procedural method of rule interpretation is an essential component of practical knowledge use and often reflects the human expert's solution strategy. For example, we can order the premises of a rule so that what is most likely to fail or is easiest to confirm is tried first. This gives the opportunity of eliminating a rule (and hence a portion of the search space) as early as possible. Rule 1 in the automotive example tries to determine whether the engine is getting gas before it asks if the engine turns over. This is inefficient, in that trying to determine whether 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, thus making the system more efficient.

It also makes more sense to determine whether the engine is turning over before checking to see whether it is getting gas; if the engine won't turn over it doesn't matter whether or not it is getting gas! In Rule 4, the user is asked to check the fuel tank before checking the carburetor for gas. In this situation, it is performing the easier check first. There is an important point here, if the overall system is to be more efficient all aspects must be considered: rule order, organization of premises, cost of tests, amount of search eliminated through the answers to tests, the way the majority of cases occur, and so on.

Thus the planning of rule order, the organization of a rule's premises, and the costs of different tests are all fundamentally heuristic in nature. In most rule-based expert systems these heuristic choices reflect the approaches taken by the human expert, and indeed can have erroneous results. In our example, if the engine is getting gas and turning over, the problem may be a bad distributor rather than bad spark plugs.

Data-driven reasoning provides additional problems and opportunities for control of reasoning. Some of these include the high-level heuristics, such as refraction, recency (opportunistic search), and specificity presented in Section 6.3.3. A more domain-specific approach groups sets of rules according to stages of the solution process. For example, in diagnosing car problems we might create the four distinct stages of 1) organize situation, 2) collect the data, 3) do the analysis (there may be more than one problem with the car), and finally, 4) report to the user the conclusions and recommended fixes.

This staged problem solving can be accomplished by creating descriptors for each stage of the solution and placing that description as the first premise in all rules that belong to that stage. For example, we might begin by placing the assertion organize situation in working memory. If no other stage descriptions occur in working memory, then only rules that have organize situation in their set of premises will fire. Of course, this should be the first premise for each of these rules. We would move to the next stage by having the last rule to fire in the organizational stage remove (retract) the fact that stage is organize solution and assert the new fact that the stage is data collection. All the rules in the data collection stage would then have their first premise IF stage is data collection and . . .. When the data collection stage is finished, the last rule would retract this fact, and assert the data analysis fact into working memory to match only those rules whose premises begin with the fact IF stage is data analysis . . ..

In our discussion so far, we have described the behavior of the production system in terms of exhaustive considerations of the rule base. Although this is expensive, it captures the intended semantics of the production system. There are, however, a number of algorithms such as RETE (Forgy 1982) that can be used to optimize search for all potentially usable rules. Essentially, the RETE algorithm compiles rules into a network structure that allows the system to match rules with data by directly following a pointer to the rule. This algorithm greatly speeds execution, especially for larger rule sets, while retaining the semantic behavior we have described in this section.

To summarize, rules are the oldest approach to knowledge representation in expert systems, and remain an important technique for building knowledge-intensive problem solvers. An expert system's rules capture human expert knowledge as it is used in practice; consequently, they are often a blend of theoretical knowledge, heuristics derived from experience, and special-purpose rules for handling odd cases and other exceptions to normal practice. In many situations, this approach has proven effective. Nonetheless, strongly heuristic systems may fail, either on encountering a problem that does not fit any available rules, or by misapplying a heuristic rule to an inappropriate situation. Human experts do not suffer from these problems, because they have a deeper, theoretical understanding of the problem domain that allows them to apply the heuristic rules intelligently, or resort to reasoning from "first principles" in novel situations. *Model-based* approaches, described next in Section 8.3.1 attempt to give an expert system this power and flexibility.

The ability to learn from examples is another human capability that knowledge-intensive problem solvers emulate. *Case-based* reasoners, Section 8.3.3, maintain a knowledge base of example problem solutions, or *cases*. When confronted with a new problem, the reasoner selects from this stored set a case that resembles the present problem, and then attempts to apply a form of its solution strategy to this problem. In legal reasoning, the argument through precedent is a common example of case-based reasoning.

8.3 Model-Based, Case-Based, and Hybrid Systems

8.3.1 Introduction to Model-Based Reasoning

Human expertise is an extremely complex amalgamation of theoretical knowledge, experience-based problem-solving heuristics, examples of past problems and their solutions, perceptual and interpretive skills and other abilities that are so poorly understood that we can only describe them as intuitive. Through years of experience, human experts develop very powerful rules for dealing with commonly encountered situations. These rules are often highly "compiled", taking the form of direct associations between observable symptoms and final diagnoses, and hiding their more deeply explanatory foundations.

For example, the MYCIN expert system would propose a diagnosis based on such observable symptoms as "headaches", "nausea", or "high fever". Although these parameters can be indicative of an illness, rules that link them directly to a diagnosis do not reflect any deeper, causal understanding of human physiology. MYCIN's rules indicate the results of an infection, but do not explain its causes. A more deeply explanatory approach would detect the presence of infecting agents, note the resulting inflammation of cell linings, the presence of inter-cranial pressures, and infer the causal connection to the observed symptoms of headache, elevated temperatures, and nausea.

In a rule-based expert system example for semiconductor failure analysis, a descriptive approach might base a diagnosis of circuit failure on such symptoms as the discoloration of components (possibly indicating a burned-out component), the history of faults in similar devices, or even observations of component interiors using an electron microscope. However, approaches that use rules to link observations and diagnoses do not offer the benefits of a deeper analysis of the device's structure and function. A more robust, deeply explanatory approach would begin with a detailed model of the physical structure of the circuit and equations describing the expected behavior of each component and their interactions. It would base its diagnosis on signal readings from various locations in the device, using this data and its model of the circuit to determine the exact points of failure.

Because first-generation expert systems relied upon heuristic rules gained from the human expert's description of problem-solving techniques, they exhibited a number of fundamental limitations (Clancy 1985). If a problem instance did not match their heuristics, they simply failed, even though a more theoretical analysis would have found a solution. Often, expert systems applied heuristics in inappropriate situations, such as where a deeper understanding of the problem would have indicated a different course. These are the limitations that *model-based* approaches attempt to address. A knowledge-based reasoner whose analysis is founded directly on the specification and functionality of a physical system is called a *model-based system*. In its design and use, a model-based reasoner creates a software simulation, often referred to as "qualitative", of the function of that which is to be understood or fixed. (There are other types of model-based systems, of course, in particular, the logic-based and stochastic, which we present in Chapter 9.)

The earliest model-based reasoners appeared in the mid-1970s and continued to mature through the 1980s (Davis and Hamscher 1992). It is interesting to note that some of the earliest work was intended to create software models of various physical devices, such as electronic circuits, for instructional purposes (deKleer 1976, Brown et al. 1982).

In these early intelligent tutoring situations, the specifications for a device or circuit were reflected in sets of rules, e.g., Kirchoff's and Ohm's laws. The tutoring system both tested the student's knowledge of the device or circuit as well as conveyed to the student knowledge he or she might be missing. Rules were both the representation of the functionality of the hardware as well as the medium for conveying this knowledge to the student.

From these early tutoring systems, where the task was to both model and teach the functionality of a system, qualitative model-based reasoners moved to trouble-shooting systems. In trouble-shooting faults in a physical system the model leads to sets of predicted behaviors. A fault is reflected in the discrepancy between predicted and observed behavior. The model-based system tells its user what to expect, and when observations differ from these expectations, how these discrepancies lead to identification of faults.

Qualitative model-based reasoning includes:

1. A description of each component in the device. These descriptions can simulate the behavior of the component.
2. A description of the device's internal structure. This is typically a representation of its components and their interconnections, along with the ability to simulate component interactions. The extent of knowledge of internal structure required depends on the levels of abstraction applied and diagnosis desired.
3. Diagnosis of a particular problem requires observations of the device's actual performance, typically measurements of its inputs and outputs. I/O measurements are easiest to obtain, but in fact, any measure could be used.

The task is then to determine which of the components could have failed in a way that accounts for the observed behaviors. This requires additional rules that describe known failure modes for the different components and their interconnections. The reasoner must find the most probable failures that can explain the observed system behavior.

A number of data structures can be used for representing the causal and structural information in models. Many model-based program designers use rules to reflect the causality and functionality of a device. Rules can also be used to capture the relationships between components. An object-oriented system also offers an excellent representational tool for reflecting device and component structure within a model, with the slots of an object representing a device or component's state, and the object or class methods defining its functionality.

To be more concrete in the design and evaluation of a model, we now consider several examples of device and circuit analysis from Davis and Hamscher (1992). Device behavior is represented by a set of expressions that capture the relationships between values on the terminals of the device. For the adder of Figure 8.13, there will be three expressions:

If we know the values at A and B, the value of C is A + B (the solid line).

If we know C and A the value at B is C – A (the dashed line).

If we know C and B, the value at A is C – B (the dotted line).

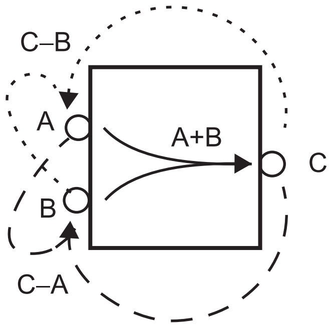


Figure 8.13 The behavior description of an adder, after Davis and Hamscher (1992).

We need not have used an algebraic form to represent these relationships. We could equally well have used relational tuples or represented the constraints with Lisp functions. The goal in model-based reasoning is to represent the knowledge that captures the functionality of the adder.

In a second example, consider the circuit of three multipliers and two adders linked as in Figure 8.14. In this example the input values are given A to E and the output values at F and G. The expected output values are given in ( ) and the actual outputs in [ ]. The task is to determine where the fault lies that will explain this discrepancy. At F we have a conflict, expecting a 12 and getting a 10. We check the dependencies at this point and determine that the value at F is a function of ADD-1 which in turn depends on the outputs of MULT-1 and MULT-2. One of these three devices must have a fault, and so we have three hypotheses to consider: either the adder behavior is bad or one of its two inputs was incorrect, and the problem lies further back in the circuit.

Reasoning from the result (10) at F and assuming correct behavior of ADD-1 and one of its inputs X (6), input Y to ADD-1 must be a 4. But that conflicts with the expectation of 6, which is the correct behavior of MULT-2 and inputs B and D. We have observed these inputs and know they are correct, so MULT-2 must be faulty. In a parallel argument, our second hypothesis is that ADD-1 is correct and MULT-1 is faulty.

Continuing this reasoning, if the first input X to ADD-1 is correct and ADD-1 itself is correct, then the second input Y must be a 4. If it were a 4, G would be 10 rather than 12, so the output of MULT-2 must be a 6 and correct. We are left with the hypotheses that the fault lies in either MULT-1 or ADD-1 and we can continue to constrain these devices with further testing.

In our reasoning about the situation of Figure 8.14 we had three tasks:

1. Hypothesis generation, in which, given a discrepancy, we hypothesized which components of the device could have caused it.

2. Hypothesis testing, in which, given a collection of potential faulty components, we determined which of them could have explained the observed behavior.

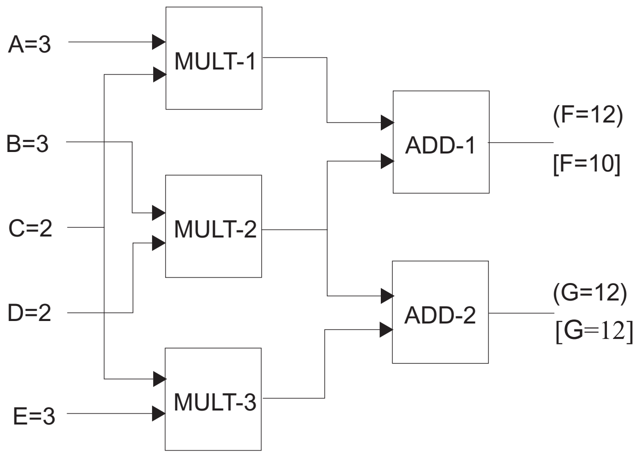


Figure 8.14 Taking advantage of direction of information flow, after Davis and Hamscher (1988).

3. Hypothesis discrimination, in which when more than one hypothesis survives the testing phase, as happened in the case of Figure 8.14, we must determine what additional information can be gathered to continue the search for the fault.

Finally, we should note that in the example of Figure 8.14 there was assumed to be a single faulty device. The world is not always this simple, although a single fault assumption is a useful, and often correct, heuristic.

Because they are based on a theoretical understanding of the devices in question, qualitative model-based techniques remedy many of the limitations of more heuristic approaches. Rather than reasoning directly from observed phenomena to causal explanations, model-based approaches attempt to represent devices and configurations of devices on a causal or functional level. The program code reflects both the function of devices and the dependencies within a system of devices. Such models are often more robust than heuristic approaches. However, the down side of this explicit modeling of function is that the knowledge acquisition stage can be quite demanding and the resulting program large, cumbersome, and slow. Because heuristic approaches "compile" typical cases into a single rule, they are often more efficient, so long as other system constraints are appropriate.

There are deeper problems, however, with this approach. As with rule-based reasoning, the model of a system is just that, a model. It will of necessity be an abstraction of the system, and at some level of detail, be incorrect. For example, consider the input wires of Figure 8.14. In our discussion we considered these values as given and correct. We did not examine the state of the wire itself, and in particular, the other end where it joined the multipliers. What if the wire were broken, or had a faulty connection to the multiplier? If the user failed to detect this faulty connection, the model would not match the actual device.

Any model attempts to describe the ideal situation, what the system is supposed to do, and not necessarily what the system does do. A "bridging" fault is a contact point in the system where two wires or devices are inadvertently linked, as when a bad solder joint bridges between two wires that should not be in contact. Most model-based reasoning has difficulty hypothesizing a bridging fault because of the a priori assumptions underlying the model and the search methods for determining anomalies. Bridging faults are simply "new" wires that aren't part of the original design. There is an implicit "closed world assumption" (Section 9.1) that the structure description of the model is assumed to be complete and anything not in the model simply doesn't exist.

In spite of these shortcomings, model-based reasoning is an important addition to the knowledge engineer's tool kit. Researchers continue to expand our understanding of diagnosis, both how human experts do it so efficiently, as well as how better algorithms can be implemented on machines (Stern and Luger 1997, Pless et al. 2006).

8.3.2 Model-Based Reasoning: a NASA Example (Williams and Nayak)

NASA has supported its presence in space by developing a fleet of intelligent space probes that autonomously explore the solar system (Williams and Nayak 1996a, Bernard et al. 1998). This effort was begun with software for the first probe in 1997 and the launch of Deep Space 1 in 1998. To achieve success through years in the harsh conditions of space travel, a craft needs to be able to radically reconfigure its control regime in response to failures and then plan around these failures during its remaining flight. To achieve acceptable cost and fast reconfiguration, one-of-a-kind modules will have to be put together quickly to automatically generate flight software. Finally, NASA expects that the set of potential failure scenarios and possible responses will be much too large to use software that supports preflight enumeration of all contingencies. Instead, the spacecraft will have to reactively think through all the consequences of its reconfiguration options.

Livingstone (Williams and Nayak 1996b) is an implemented kernel for a model-based reactive self-configuring autonomous system. The model-based reasoning representation language for Livingstone is propositional calculus, a shift from the first-order predicate calculus (Chapter 2), the traditional representation language for model-based diagnosis. Williams and Nayak felt, based on their past research on building fast propositional logic conflict-based algorithms for diagnosis (de Kleer and Williams 1989), that a fast reactive system, performing significant deduction in the sense/response loop, was possible.

A long-held vision of model-based reasoning has been to use a single centralized model to support a variety of engineering tasks. For model-based autonomous systems this means using a single model to support a diversity of execution tasks. These include keeping track of developing plans (Section 8.4), confirming hardware modes, reconfiguring hardware, detecting anomalies, diagnosis, and fault recovery. Livingstone automates all these tasks using a single model and a single core algorithm, thus making significant progress towards achieving the vision for model-based problem solvers.

Figure 8.15 shows an idealized schematic of the main engine subassembly of Cassini, the most complex spacecraft built to that time. It consists of a helium tank, a fuel tank, an oxidizer tank, a pair of main engines, regulators, latch valves, pyro valves, and pipes. The helium tank pressurizes the two propellant tanks, with the regulators acting to reduce the high helium pressure to a lower working pressure. When propellant paths to a main engine

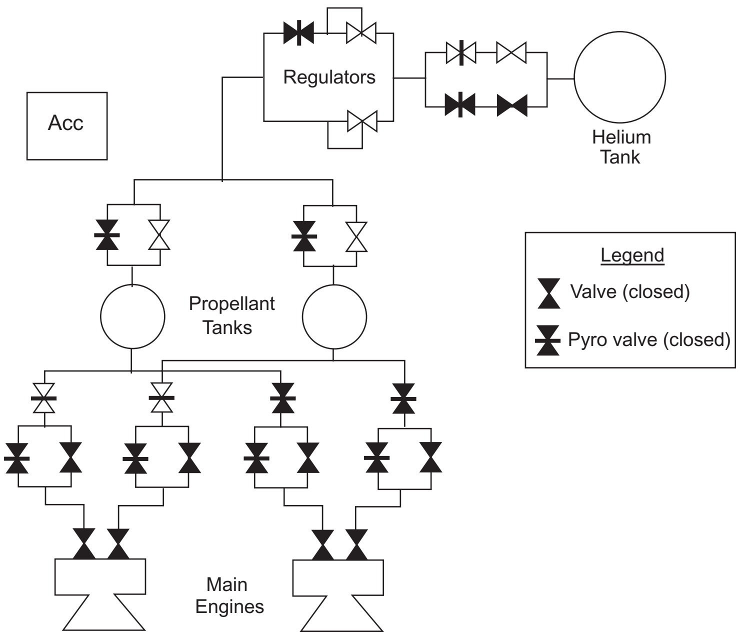


Figure 8.15 A schematic of the simplified Livingstone propulsion system, from Williams and Nayak (1996b).

are open (the valve icon is unfilled), the pressurized tank forces fuel and oxidizer into the main engine, where they combine, spontaneously ignite, and produce thrust. The pyro valves can be fired exactly once, that is, they can change state only once, either going from open to closed or vice versa. Their function is to isolate parts of the main engine subsystem until they are needed, or to permanently isolate failed components. The latch valves are controlled using valve drivers (not shown in Figure 8.15) and the Acc (accelerometer) senses the thrust generated by the main engines.

Starting from the configuration shown in Figure 8.15, the high-level goal of producing thrust can be achieved using a variety of different configurations: thrust can be provided by either of the main engines and there are a number of ways of opening propellant paths to either main engine. For example, thrust can be provided by opening the latch valves leading to the engine on the left, or by firing a pair of pyros and opening a set of latch valves leading to the engine on the right. Other configurations correspond to various combinations of pyro firings. The different configurations have different characteristics since pyro firings are irreversible actions and since firing pyro valves requires significantly more power than opening or closing latch valves.

Suppose the main engine subsystem has been configured to provide thrust from the left main engine by opening the latch valves leading to it. Suppose that this engine fails, for example, by overheating, so that it fails to provide the required thrust. To ensure that the desired thrust is provided even in this situation, the spacecraft must be transitioned to a new configuration in which thrust is now provided by the main engine on the right side.

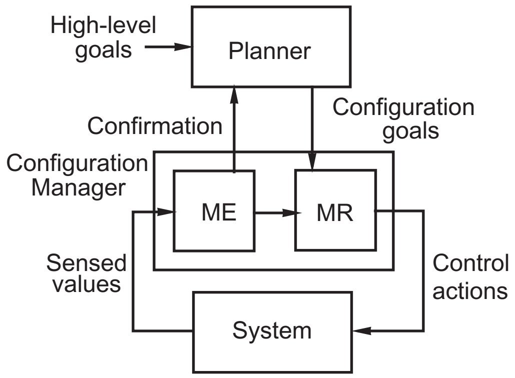


Figure 8.16 A model-based configuration management system, from Williams and Nayak (1996b).

Ideally, this is achieved by firing the two pyro valves leading to the right side and opening the remaining latch valves, rather than firing additional pyro valves.

A *configuration manager* constantly attempts to move the spacecraft into lowest cost configurations that achieve a set of high-level dynamically changing goals. When the spacecraft strays from the chosen configuration due to failures, the manager analyzes sensor data to identify the current configuration of the spacecraft, and then moves the space craft to a new configuration which, once again, achieves the desired configuration goals. A configuration manager is a discrete control system that ensures that the spacecraft's configuration always achieves the set point defined by the configuration goals. The planning algorithms that support the configuration manager are presented in Section 8.4.4.

Reasoning about the configurations (and autonomous reconfigurations) of a system requires the concepts of operating and failure modes, reparable failures, and configuration changes. NASA expresses these concepts in a state space diagram: reparable failures are transitions from a failure state to a nominal state; configuration changes are between nominal states; and failures are transitions from a nominal to a failure state.

Williams and Nayak (1997) view an autonomous system as a combination of a reactive configuration manager and a high level planner. The planner, for details see Section 8.4, generates a sequence of hardware configuration goals. The configuration manager evolves the transition system of the application domain, here the propulsion system of the spacecraft, along the desired trajectory. Thus configuration management is achieved by sensing and controlling the state of a transition system.

A model-based configuration manager is a configuration manager that uses a specification of the transition system to compute the desired sequence of control values. In the example of Figure 8.15, each hardware component is modeled by a component transition system. Component communication, denoted by wires (links) in the figure is modeled by shared propositional terms between the corresponding component transition systems.

The configuration manager makes extensive use of the model to infer the system's current state and to select optimal control actions to meet configuration goals. This is essential in situations where mistakes can lead to disaster, ruling out simple trial and error methods. The *model-based* configuration manager uses a model to determine the desired control sequence in two stages: mode estimation and mode reconfiguration (ME and MR in Figure 8.16). ME incrementally generates the set of all system trajectories consistent with the plant transition model and the sequence of system control and sensed values. MR uses a plant transition model and the partial trajectories generated by ME up to the current state to determine a set of control values such that all predicted trajectories achieve the configuration goal of the next state. Both ME and MR are reactive. ME infers the current state from knowledge of the previous state and observations within the current state. MR only considers actions that achieve the configuration goal within the next state.

In the next section we consider case-based reasoning, a knowledge-intensive technique that supports the reuse of past experience in a problem domain to address new situations. In Section 8.4 we present planning, and return to the NASA control example.

8.3.3 Introduction to Case-Based Reasoning

Heuristic rules and theoretical models are two types of information human experts use to solve problems. Another powerful strategy experts use is reasoning from cases, examples of past problems and their solutions. *Case-based reasoning* (CBR) uses an explicit database of problem solutions to address new problem-solving situations. These solutions may be collected from human experts through the knowledge engineering process or may reflect the results of previous search-based successes or failures. For example, medical education does not rely solely on theoretical models of anatomy, physiology, and disease; it also depends heavily on case histories and the intern's experience with other patients and their treatment. CASEY (Koton 1988*a*, *b*) and PROTOS (Bareiss et al. 1988) are examples of case-based reasoning applied to medicine.

Lawyers select past law cases that are similar to their client's and that suggest a favorable decision, and try to convince the court that these similarities merit similar findings. Although general laws are made by democratic processes, their interpretation is usually based on legal precedents. How a law was interpreted in some earlier situation is critical for its current interpretation. Thus, an important component of legal reasoning is identifying from case law precedents for decisions in a particular case. Rissland (1983) and Rissland and Ashley (1987) have designed case-based reasoners to support legal arguments.

Computer programmers often reuse their code, adapting an old program to fit a new situation with similar structure. Architects draw on their knowledge of esthetically pleasing and useful buildings of the past to design new buildings that people find pleasing and comfortable. Historians use stories from the past to help statesmen, bureaucrats, and citizens understand past events and plan for the future. The ability to reason from cases is fundamental to human intelligence.

Other obvious areas for reasoning from cases include design, where aspects of a successfully executed artifact may be appropriate for a new situation, and diagnosis, where the failures of the past often recur. Hardware diagnosis is a good example of this. An expert in this area, besides using extensive theoretical knowledge of electronic and mechanical systems, brings past successful and failed experiences in diagnosis to bear on the current problem. CBR has been an important component of many hardware diagnostic systems, including work on the maintenance of signal sources and batteries in earth orbiting satellites (Skinner and Luger 1992) and the failure analysis of discrete component semiconductors (Stern and Luger 1997).

CBR offers a number of advantages for the construction of expert systems. Knowledge acquisition can be simplified if we record a human expert's solutions to a number of problems and let a case-based reasoner select and reason from the appropriate case. This would save the knowledge engineer the trouble of building general rules from the expert's examples; instead, the reasoner would generalize the rules automatically, through the process of applying them to new situations.

Case-based approaches can also enable an expert system to learn from its experience. After reaching a search-based solution to a problem, a system can save that solution, so that next time a similar situation occurs, search would not be necessary. It can also be important to retain in the case base information about the success or failure of previous solution attempts; thus, CBR offers a powerful model of learning. An early example of this is Samuel's (1959, Section 4.1.1) checker-playing program, where board positions that were found through search or experience to be important are retained in the chance that this position might occur again in a later game.

Case-based reasoners share a common structure. For each new problem they:

1. **Retrieve appropriate cases from memory**. A case is appropriate if its solution may be successfully applied to the new situation. Since reasoners cannot know this in advance, they typically use the heuristic of choosing cases that are similar to the problem instance. Both humans and artificial reasoners determine similarity on the basis of common features: for example, if two patients share a number of common features in their symptoms and medical histories, there is a good probability that they have the same disease and will respond to the same treatment. Retrieving cases efficiently also requires that the case memory be organized to aid such retrieval. Typically, cases are indexed by their significant features, enabling efficient retrieval of cases that have the most features in common with the current problem. The identification of salient features is highly situation dependent.

2. **Modify a retrieved case so that it will apply to the current situation**. Typically, a case recommends a sequence of operations that transform a starting state into a goal state. The reasoner must transform the stored solution into operations suitable for the current problem. Analytic methods, such as curve fitting the parameters common to stored cases and new situations can be useful; for example, to determine appropriate temperatures or materials for welding. When analytic relations between cases are not available more heuristic methods may be appropriate; for example, in help desks for hardware diagnosis.

3. **Apply the transformed case**. Step 2 modifies a stored case, which, when applied, may not guarantee a satisfactory problem solution. This may require modifications in the solution case, with further iterations of these first three steps.

4. **Save the solution, with a record of success or failure, for future use**. Storage of the new case requires updating of the index structure. There are methods that can be used to maintain indices, including clustering algorithms (Fisher 1987) and other techniques from machine learning (Stubblefield and Luger 1996).

The data structures for case-based reasoning can be quite varied. In the simplest situation, cases are recorded as relational tuples where a subset of the arguments record the features to be matched and other arguments point to solution steps. Cases can also be represented as more complex structures, such as proof trees. A fairly common mechanism for storing cases is to represent cases as a set of large situation–action rules. The facts that describe the situation of the rule are the salient features of the recorded case and the operators that make up the action of the rule are the transformations to be used in the new situation. When this type of rule representation is used algorithms such as RETE (Forgy 1982) can be used to organize and optimize the search for appropriate cases.

The most difficult issue in case-based problem solving, regardless of the data structure chosen for case representation, is the selection of salient features for the indexing and retrieval of cases. Kolodner (1993) and others actively involved in case-based problem solving set as a cardinal rule that cases be organized by the goals and needs of the problem solver. That is, that a careful analysis be made of case descriptors in the context of how these cases will be used in the solution process.

For example, suppose a *weak communication signal* problem occurs in a satellite at 10:24:35 GMT. Analysis is made, and it is also determined that the power system is low. The low power can occur because the solar panels are not properly oriented towards the sun. The ground controllers make adjustments in the satellite's orientation, the power improves, and the communication signal is again strong. There are a number of salient features that might be used to record this case, the most obvious being that there is a weak communication signal or that the power supply is low. Another feature to describe this case is that the time of the problem was 10:24:35 GMT. The goals and needs of the problem solver in this case suggest that the salient features are weak communication signal and/or low power supply; the time this all happens may well be irrelevant, unless, of course, the fault occurs just after the sun disappears over the horizon (Skinner and Luger 1995).

Another essential problem to be addressed in CBR is the representation of such notions as weak signal or low power. Since the precise situation will probably never again be matched, e.g., some exact real number describing signal strength, the reasoner will probably represent the values as a range of real numbers indicating, for example, good, borderline-good, weak, and danger-alert levels.

Kolodner (1993) offers a set of possible preference heuristics to help organize the storage and retrieval of cases. These include:

1. *Goal-directed preference*. Organize cases, at least in part, by goal descriptions. Retrieve cases that have the same goal as the current situation.

2. *Salient-feature preference*. Prefer cases that match the most important features or those matching the largest number of important features.

3. *Specify preference*. Look for as exact as possible matches of features before considering more general matches.

4. *Frequency preference*. Check first the most frequently matched cases.

5. *Recency preference*. Prefer cases used most recently.

6. *Ease of adaptation preference*. Use first cases most easily adapted to the current situation.

Case-based reasoning has a number of advantages for the design of expert systems. Once knowledge engineers have arrived at the proper case representation, continued knowledge acquisition is straightforward: simply gather and store more cases. Often, case acquisition can be done from historical records or by monitoring current operations, minimizing demands on the human expert's time.

In addition, CBR raises a number of important theoretical questions relating to human learning and reasoning. One of the most subtle and critical issues raised by CBR is the question of defining *similarity*. Although the notion that similarity is a function of the number of features that two cases have in common is quite reasonable, it masks a number of profound subtleties. For example, most objects and situations have a large number of potential descriptive properties; case-based reasoners typically select cases on the basis of a tiny retrieval vocabulary. Typically, case-based reasoners require that the knowledge engineer define an appropriate vocabulary of highly relevant features. Although there has been work on enabling a reasoner to determine relevant features from its own experience (Stubblefield 1995), determining relevance remains a difficult problem.

Another important problem in case-based reasoning deals with store/compute tradeoffs. As a case-based reasoner acquires more cases, it becomes more intelligent and better able to solve a variety of target problems. Indeed, as we add cases to a reasoner, its performance will improve – up to a point. The problem is that as the case base continues to grow, the time needed to retrieve and process an appropriate case also grows. A decline in efficiency for large case bases can also be due to overlapping concepts, noise, and the distribution of problem types. One solution to this problem is to only save the "best" or "prototype" cases, deleting those that are redundant or used infrequently; i.e., forgetting those cases that fail to prove useful. See Samuel's (1959) retention algorithm for saving checker board positions for an important early example of this approach. In general, however, it is not clear how we can automate such decisions; this remains an active research area (Kolodner 1993).

An automated explanation for why a solution is recommended is also difficult for a case-based reasoner. When asked why a solution was selected to remedy a current situation, the only explanation the system can provide is to say that this particular fix worked at a previous time period. There may also be weak explanations based on similarities of top-level goal descriptions between the current situation and a set of stored cases. In the example of the satellite communication problem, the relevant case was chosen on the basis of a weak communication signal. This mode of reasoning has no deeper explanation than that it worked before in a similar situation. But as noted previously, this may be sufficient explanation for many situations.

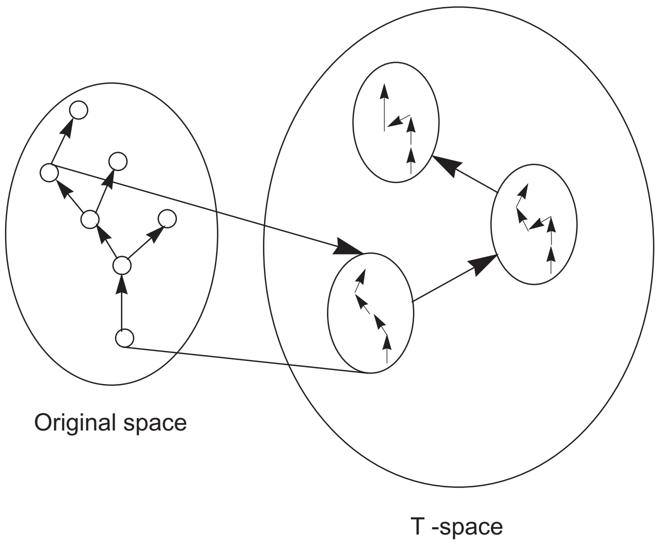


Figure 8.17 Transformational analogy, adapted from Carbonell (1983).

Many researchers, however, feel that the simple repetition of top-level goals and the case to be applied offers insufficient explanation (Leake 1992, Stern and Luger 1992), especially when we need an explanation of why some fix doesn't work. Take the satellite situation again. Suppose the solar panel reorientation works but three hours later the signal again becomes weak. Using both the frequency and recency heuristics, we again reorient the solar panel. In three hours the weak signal recurs. And again three hours later; we always apply the same fix. This example is based on an actual satellite situation in which it was found that a more complex problem existed, namely a gyroscope was overheating and giving a distorted reading that disoriented the satellite. The system that finally solved the problem used a model-based reasoner to simulate the behavior of the satellite to determine the root causes of the weak communication signal (Skinner and Luger 1995).

Case-based reasoning is related to the problem of learning through analogy. To reuse past experiences we must both recognize the salient features of the past as well as build a mapping of how that experience may be used in the present situation. *Transformational analogy* (Carbonell 1983) is an example of a case-based approach to problem solving. It solves new problems by modifying existing solutions until they may be applied to the new instance. Operators that modify complete problem solutions define a higher-level of abstraction or T-space in which states are problem solutions and operators transform these solutions, as in Figure 8.17. The goal is to transform a source solution into a possible solution for the target problem. Operators modify solutions in ways such as inserting or deleting steps in a solution path, reordering steps in a solution, splicing new solutions into a portion of an old solution, or changing the bindings of parameters in the current solution.

Transformational analogy typifies the approach used by case-based problem solving. Later work has refined the approach, considering such issues as the representation of cases, strategies for organizing a memory of prior cases, retrieval of relevant prior cases, and the use of cases in solving new problems. For further information on case-based reasoning, see Hammond (1989) and Kolodner (1988*a*, *b*). Reasoning through analogies is discussed further in the context of symbol-based machine learning (Section 10.5.4).

8.3.4 Hybrid Design: Strengths/Weaknesses of Strong Method Systems

Successes in building expert systems that solve hard, practical problems have demonstrated the truth of the central idea behind knowledge-based systems: that the power of a reasoner is in its domain knowledge rather than the sophistication of its reasoning methods. This observation, however, raises one of the central issues in artificial intelligence: that of knowledge representation. At a practical level, every knowledge engineer must make choices about how to represent domain knowledge in a way that will be most appropriate to the given domain. Knowledge representation also raises a number of theoretically important, intellectually difficult issues, such as the handling of missing and uncertain information, the measurement of a representation's expressiveness, the relationship between a representation language and such issues as learning, knowledge acquisition, and the efficiency of a reasoner.

In this chapter, we considered a number of basic approaches to knowledge representation: rule-based expert systems, model-based reasoners, and case-based reasoners. As an aid to practical knowledge engineering, we summarize the strengths and weaknesses of each knowledge-intensive approach to problem solving.

Rule-based Reasoning

The advantages of a rule-based approach include:

1. The ability to use, in a very direct fashion, experiential knowledge acquired from human experts. This is particularly important in domains that rely heavily on heuristics to manage complexity and/or missing information.
2. Rules map into state space search. Explanation facilities support debugging.
3. The separation of knowledge from control simplifies development of expert systems by enabling an iterative development process where the knowledge engineer acquires, implements, and tests individual rules.
4. Good performance is possible in limited domains. Because of the large amounts of knowledge required for intelligent problem solving, expert systems are limited to narrow domains. However, there are many domains where design of an appropriate system has proven extremely useful.
5. Good explanation facilities. Although the basic rule-based framework supports flexible, problem-specific explanations, it must be mentioned that the ultimate quality of these explanations depends upon the structure and content of the rules. Explanation facilities differ widely between data- and goal-driven systems.

Disadvantages of rule-based reasoning include:

1. Often the rules obtained from human experts are highly heuristic in nature, and do not capture functional or model-based knowledge of the domain.
2. Heuristic rules tend to be “brittle” and can have difficulty handling missing information or unexpected data values.
3. Another aspect of the brittleness of rules is a tendency to degrade rapidly near the “edges” of the domain knowledge. Unlike humans, rule-based systems are usually unable to fall back on first principles of reasoning when confronted with novel problems.
4. Explanations function at the descriptive level only, omitting theoretical explanations. This follows from the fact that heuristic rules gain much of their power by directly associating problem symptoms with solutions, without requiring (or suporting) deeper reasoning.
5. The knowledge tends to be very task dependent. Formalized domain knowledge tends to be very specific in its applicability. Currently, knowledge representation languages do not approach the flexibility of human reasoning.

Case-based Reasoning

The advantages of case-based reasoning include:

1. The ability to encode historical knowledge directly. In many domains, cases can be obtained from existing case histories, repair logs, or other sources, eliminating the need for intensive knowledge acquisition with a human expert.
2. Allows shortcuts in reasoning. If an appropriate case can be found, new problems can often be solved in much less time than it would take to generate a solution from rules or models and search.
3. It allows a system to avoid past errors and exploit past successes. CBR provides a model of learning that is both theoretically interesting and practical enough to apply to complex problems.
4. Extensive analysis of domain knowledge is not required. Unlike a rule-based system, where the knowledge engineer must anticipate rule interactions, CBR allows a simple additive model for knowledge acquisition. This requires an appropriate representation for cases, a useful retrieval index, and a case adaptation strategy.
5. Appropriate indexing strategies add insight and problem-solving power. The ability to distinguish differences in target problems and select an appropriate case is an important source of a case-based reasoner’s power; often, indexing algorithms can provide this functionality automatically.

The disadvantages of case-based reasoning include:

1. Cases do not often include deeper knowledge of the domain. This handicaps explanation facilities, and in many situations it allows the possibility that cases may be misapplied, leading to poor quality or wrong advice.
2. A large case base can suffer problems from store/compute trade-offs.
3. It is difficult to determine good criteria for indexing and matching cases. Currently, retrieval vocabularies and similarity matching algorithms must be carefully hand crafted; this can offset many of the advantages CBR offers for knowledge acquisition.

Model-based Reasoning

The advantages of model-based reasoning include:

1. The ability to use functional/structural knowledge of the domain in problem-solving. This increases the reasoner's ability to handle a variety of problems, including those that may not have been anticipated by the system's designers.
2. Model-based reasoners tend to be very robust. For the same reasons that humans often retreat to first principles when confronted with a novel problem, model-based reasoners tend to be thorough and flexible problem solvers.
3. Some knowledge is transferable between tasks. Model-based reasoners are often built using scientific, theoretical knowledge. Because science strives for generally applicable theories, this generality often extends to model-based reasoners.
4. Often, model-based reasoners can provide causal explanations. These can convey a deeper understanding of the fault to human users, and can also play an important tutorial role.

The disadvantages of model-based reasoning include:

1. A lack of experiential (descriptive) knowledge of the domain. The heuristic methods used by rule-based approaches reflect a valuable class of expertise.
2. It requires an explicit domain model. Many domains, such as the diagnosis of failures in electronic circuits, have a strong scientific basis that supports model-based approaches. However, many domains, such as some medical specialties, most design problems, or many financial applications, lack a well-defined scientific theory. Model-based approaches cannot be used in such cases.
3. High complexity. Model-based reasoning generally operates at a level of detail that leads to significant complexity; this is, after all, one of the main reasons human experts have developed heuristics in the first place.

4. Exceptional situations. Unusual circumstances, for example, bridging faults or the interaction of multiple failures in electronic components, can alter the functionality of a system in ways difficult to predict using an a priori model.

Hybrid Design

An important area of research and application is the combination of different reasoning models. With a hybrid architecture two or more paradigms are integrated to get a cooperative effect where the strengths of one system can compensate for the weakness of another. In combination, we can address the disadvantages noted in the previous discussion.

For example, the combination of rule-based and case-based systems can:

1. Offer a natural first check against known cases before undertaking rule-based reasoning and the associated search costs.
2. Provide a record of examples and exceptions to solutions through retention in the case base.
3. Record search-based results as cases for future use. By saving appropriate cases, a reasoner can avoid duplicating costly search.

The combination of rule-based and model-based systems can:

1. Enhance explanations with functional knowledge. This can be particularly useful in tutorial applications.
2. Improve robustness when rules fail. If there are no heuristic rules that apply to a given problem instance, the reasoner can resort to reasoning from first principles.
3. Add heuristic search to model-based search. This can help manage the complexity of model-based reasoning and allow the reasoner to choose intelligently between possible alternatives.

The combination of model-based and case-based systems can:

1. Give more mature explanations to the situations recorded in cases.
2. Offer a natural first check against stored cases before beginning the more extensive search required by model-based reasoning.
3. Provide a record of examples and exceptions in a case base that can be used to guide model-based inference.
4. Record results of model-based inference for future use.

Hybrid methods deserve the attention of researchers and application developers alike. However, building such systems is not a simple matter, requiring the resolution of such problems as determining which reasoning method to apply in a given situation, deciding when to change reasoning methods, resolving differences between reasoning methods, and designing representations that allow knowledge to be shared.

We next consider planning, or the organization of procedures into potential solutions.

8.4 Planning

The task of a planner is to find a sequence of actions that allow a problem solver, such as a a control system, to accomplish some specific task. Traditional planning is very much knowledge-intensive, since plan creation requires the organization of pieces of knowledge and partial plans into a solution procedure. Planning plays a role in expert systems in reasoning about events occurring over time. Planning has many applications in manufacturing, such as process control. It is also important for designing a robot contrller.

8.4.1 Introduction to Planning: Robotics

To begin, we consider examples from traditional robotics. (Much of modern robotics uses reactive control rather than deliberative planning, Sections 7.3.1 and 8.4.3.) The steps of a traditional robot plan are composed of the robot's *atomic actions*. In our example, we do not describe atomic actions at the micro-level: "turn the sixth stepper motor one revolution." Instead, we specify actions at a higher level, such as by their effects on a world. For example, a blocks world robot might include such actions as "pick up object a" or "go to location x." The micro-control is built into these higher-level actions.

Thus, a sequence of actions to "go get block a from room b" might be:

1. put down whatever is now held
2. go to room b
3. go over to block a
4. pick up block a
5. leave room b
6. return to original location

Plans are created by searching through a space of possible actions until the sequence necessary to accomplish the task is discovered. This space represents states of the world that are changed by applying each of the actions. The search terminates when the goal state (the appropriate description of the world) is produced. Thus, many of the issues of heuristic search, including finding A* algorithms, are also appropriate in planning.

The act of planning does not depend on the existence of an actual robot to carry out the plans, however. In the early years of computer planning (1960s), entire plans were formulated before the robot performed its first act. Thus plans were devised without the pres-

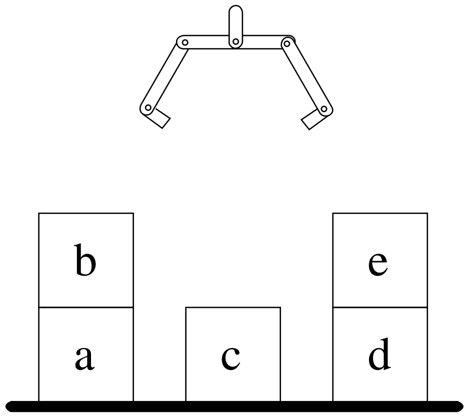


Figure 8.18 The blocks world.

ence of a robot at all! More recently, with the implementation of sophisticated sensing and reactive devices, research has focused on more integrated plan/action sequencing.

Traditional planning relies on search techniques, and raises a number of interesting issues. For one, the description of the states of the world may be considerably more complex than in previous examples of search. Consider the number of predicates that may be necessary to describe rooms, corridors, and objects in a robot's environment. Not only must we represent the robot's world; we must also represent the effect of atomic actions on that world. The full description of each state of the problem space can be large.

Another difference in planning is the need to characterize what is *not* changed by a particular action. Picking up an object does change (a) the location of the object and (b) the fact that the robot hand is now grasping the object. It does not change (a) the locations of the doors and the rooms or (b) the locations of other objects. The specification of what is true in one state of the world and *exactly* what is changed by performing some action in the world has become known as the *frame problem* (McCarthy 1980, McCarthy and Hayes 1969). As the complexity of the problem space increases, the issue of keeping track of the changes that occur with each action and the features of a state description that remain unchanged becomes more important. We present two solutions for coping with the frame problem, but, as will be seen, neither of these is totally satisfactory.

Other important issues include generating plans, saving and generalizing good plans, recovering from unexpected plan failures (part of the world might not be as expected, perhaps by being accidentally moved from its anticipated location), and maintaining consistency between the world and a program's internal model of the world.

In the examples of this section, we limit our robot's world to a set of blocks on a tabletop and the robot's actions to an arm that can stack, unstack, and otherwise move the blocks about the table. In Figure 8.18 we have five blocks, labeled a, b, c, d, e, sitting on the top of a table. The blocks are all cubes of the same size, and stacks of blocks, as in the figure, have blocks directly on top of each other. The robot arm has a gripper that can grasp any clear block (one with no block on top of it) and move it to any location on the tabletop or place it on top of any other block (with no block on top of it).

The robot arm can perform the following tasks (U, V, W, X, Y, and Z are variables):

goto(X,Y,Z)	Goto location described by coordinates X, Y, and Z. This location might be implicit in the command pickup (W) where block W has location X, Y, Z.
pickup(W)	Pick up block W from its current location. It is assumed that block W is clear on top, the gripper is empty at the time, and the computer knows the location of block W.
putdown(W)	Place block W down at some location on the table and record the new location for W. W must be held by the gripper at the time.
stack(U,V)	Place block U on top of block V. The gripper must be holding U and the top of V must be clear of other blocks.
unstack(U,V)	Remove block U from the top of V. U must be clear of other blocks, V must have block U on top of it, and the hand must be empty before this command can be executed.

The state of the world is described by a set of predicates and predicate relationships:

location(W,X,Y,Z)	Block W is at coordinates X, Y, Z.
on(X,Y)	Block X is immediately on top of block Y.
clear(X)	Block X has nothing on top of it.
gripping(X)	The robot arm is holding block X.
gripping( )	The robot gripper is empty.
ontable(W)	Block W is on the table.

ontable(W) is a short form for the predicate location(W,X,Y,Z), where Z is the table level. Similarly, on(X,Y) indicates that block X is located with its bottom coincident with the top of block Y. We can greatly simplify the world descriptions by having the computer record the present location(X,Y,Z) of each block and keep track of its movements to new locations. With this location assumption, the goto command becomes unnecessary; a command such as pickup(X) or stack(X) implicitly contains the location of X.

The blocks world of Figure 8.18 may now be represented by the following set of predicates. We call this collection of predicates STATE 1 for our continuing example. Because the predicates describing the state of the world for Figure 8.18 are all true at the same time, the full state description is the conjunction ($\wedge$) of all these predicates.

	STATE 1	
ontable(a).	on(b, a).	clear(b).
ontable(c).	on(e, d).	clear(c).
ontable(d).	gripping( ).	clear(e).

Next, a number of truth relations (in the declarative sense) or rules for performance (in the procedural sense) are created for clear(X), ontable(X), and gripping( ):

1. $(\forall X)$ (clear(X) $\leftarrow \neg$ $(\exists Y)$ (on(Y,X)))
2. $(\forall Y)$ $(\forall X)$ $\neg$ (on(Y,X) $\leftarrow$ ontable(Y))
3. $(\forall Y)$ gripping( ) $\leftrightarrow \neg$ (gripping(Y))

The first statement says that if block X is clear, there does not exist any block Y such that Y is on top of X. Interpreted procedurally, this says "to clear block X go and remove any block Y that might be on top of X."

We now design rules to operate on states and produce new states. In doing so, we again impute a procedural semantics to a predicate logic-like representation. The operators (pickup, putdown, stack, unstack) are:

4. $(\forall X)$ (pickup(X) $\rightarrow$ (gripping(X) $\leftarrow$ (gripping( ) $\wedge$ clear(X) $\wedge$ ontable(X)))).
5. $(\forall X)$ (putdown(X) $\rightarrow$ ((gripping( ) $\wedge$ ontable(X) $\wedge$ clear(X)) $\leftarrow$ gripping(X))).
6. $(\forall X)$ $(\forall Y)$ (stack(X,Y) $\rightarrow$ ((on(X,Y) $\wedge$ gripping( ) $\wedge$ clear(X)) $\leftarrow$ (clear(Y) $\wedge$ gripping(X)))).
7. $(\forall X)(\forall Y)$ (unstack(X,Y) $\rightarrow$ ((clear(Y) $\wedge$ gripping(X)) $\leftarrow$ (on(X,Y) $\wedge$ clear(X) $\wedge$ gripping( ))).

Consider the fourth rule: for all blocks X, pickup(X) means gripping(X) if the hand is empty and X is clear. Note the form of these rules: A $\rightarrow$ (B $\leftarrow$ C). This says that operator A produces new predicate(s) B when condition(s) C is(are) true. We use these rules to generate new states in a space. That is, if predicates C are true in a state, then B is true in its child state. Thus, operator A can be used to create a new state described by predicates B when predicates C are true. Alternative approaches to creating these operators include STRIPS (Nilsson 1980 and Section 8.4.2) and Rich and Knight's (1991) do function.

We must first address the *frame problem* before we can use these rule relationships to generate new states of the blocks world. *Frame axioms* are rules to tell what predicates describing a state are *not* changed by rule applications and are thus carried over intact to help describe the new state of the world. For example, if we apply the operator pickup block b in Figure 8.18, then all predicates related to the rest of the blocks remain true in the child state. For our world of blocks we can specify several such frame rules:

8. $(\forall X)$ $(\forall Y)$ $(\forall Z)$ (unstack(Y,Z) $\rightarrow$ (ontable(X) $\leftarrow$ ontable(X))).
9. $(\forall X)$ $(\forall Y)$ $(\forall Z)$ (stack(Y,Z) $\rightarrow$ (ontable(X) $\leftarrow$ ontable(X))).

These two rules say that ontable is not affected by the stack and unstack operators. This is true even when X and Z are identical; if Y = Z either 6 or 7 above won't be true.

Other frame axioms say that on and clear are affected by stack and unstack operators only when that particular on relation is unstacked or when a clear relation is stacked. Thus, in our example, on(b,a) is not affected by unstack(c,d).

Similarly, frame axioms say that clear(X) relations are unaffected by gripping(Y) even when X = Y or gripping( ) is true. More axioms say that gripping does not affect on(X,Y) relations but affects only the ontable(X) relation where X is gripped. A number of further frame axioms need to be specified for our example.

Together, these operators and frame axioms define a state space, as illustrated by the operator unstack. unstack(X,Y) requires three conditions to be true simultaneously, namely on(X,Y), gripping( ), and clear(X). When these conditions are met the new predicates gripping(X) and clear(Y) are produced by applying the unstack operator. A number of other predicates also true for STATE 1 will remain true in STATE 2. These states are preserved for STATE 2 by the frame axioms. We now produce the nine predicates describing STATE 2 by applying the unstack operator and the frame axioms to the nine predicates of STATE 1, where the net result is to unstack block e:

	STATE 2	
ontable(a).	on(b,a).	clear(b).
ontable(c).	clear(c).	clear(d).
ontable(d).	gripping(e).	clear(e).

To summarize:

1. Planning may be seen as a state space search.
2. New states are produced by general operators such as stack and unstack plus frame rules.
3. The techniques of graph search may be applied to find a path from the start state to the goal state. The operations on this path constitute a plan.

Figure 8.19 shows an example of a state space searched by applying the operators as described above. If a goal description is added to this problem-solving process, then a plan may be seen as a set of operators that produces a path that leads from the present state of this graph to the goal. (See Section 3.1.2.)

This characterization of the planning problem defines its theoretical roots in state space search and predicate calculus representation and inferencing. However, it is important to note how complex this manner of solution can be. In particular, using the frame rules to calculate what remains unchanged between states can add exponentially to the search, as can be seen from the complexity of our very simple blocks example. In fact, when any new predicate descriptor is introduced, for color, shape, or size, new frame rules must be defined to relate it to all other appropriate actions!

This discussion also assumes that the subproblems that make up a task are independent and may thus be solved in an arbitrary order. This is very seldom the case in interesting and/or complex problem domains, where the preconditions and actions required to

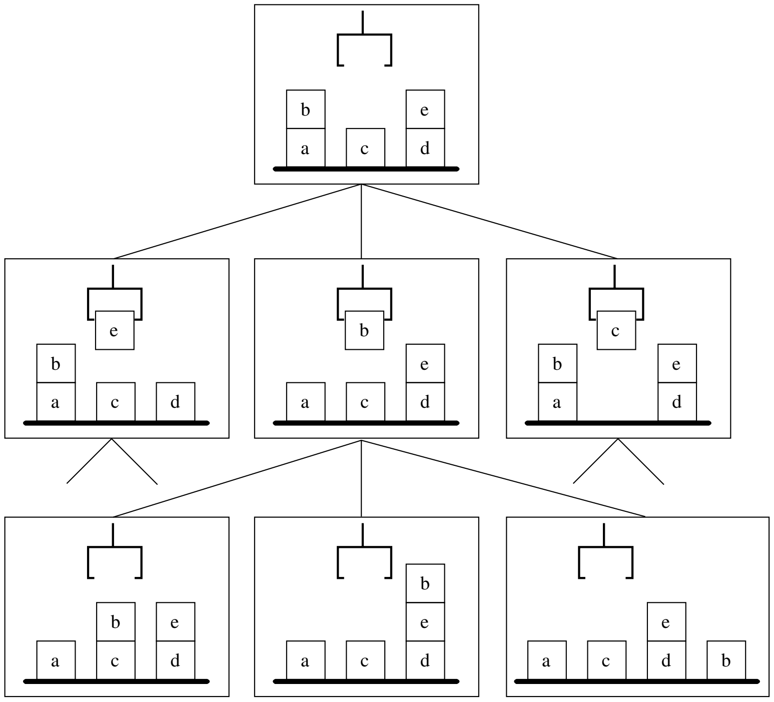


Figure 8.19 Portion of the state space for a portion of the blocks world.

achieve one subgoal can often conflict with the preconditions and actions required to achieve another. Next we illustrate these problems and discuss an approach to planning that greatly assists in handling this complexity.

8.4.2 Using Planning Macros: STRIPS

STRIPS, developed at what is now SRI International, stands for STanford Research Institute Planning System (Fikes and Nilsson 1971, Fikes et al. 1972). This controller was used to drive the *SHAKEY* robot of the early 1970s. STRIPS addressed the problem of efficiently representing and implementing the operations of a planner. It addressed the problem of conflicting subgoals and provided an early model of learning; successful plans were saved and generalized as *macro operators*, which could be used in similar future situations. In the remainder of this section, we present a version of STRIPS-style planning and *triangle tables*, the data structure used to organize and store macro operations.

Using our blocks example, the four operators pickup, putdown, stack, and unstack are represented as triples of descriptions. The first element of the triple is the set of *preconditions* (P), or conditions the world must meet for an operator to be applied. The second element of the triple is the *add* list (A), or the additions to the state description that are a result of applying the operator. Finally, there is the *delete* list (D), or the items that are removed from a state description to create the new state when the operator is applied. These lists are intended to eliminate the need for separate frame axioms. We can represent the four operators in this fashion:

	P: gripping( ) ∧ clear(X) ∧ ontable(X)
pickup(X)	A: gripping(X)
	D: ontable(X) ∧ gripping( )
	P: gripping(X)
putdown(X)	A: ontable(X) ∧ gripping( ) ∧ clear(X)
	D: gripping(X)
	P: clear(Y) ∧ gripping(X)
stack(X,Y)	A: on(X,Y) ∧ gripping( ) ∧ clear(X)
	D: clear(Y) ∧ gripping(X)
	P: clear(X) ∧ gripping( ) ∧ on(X,Y)
unstack(X,Y)	A: gripping(X) ∧ clear(Y)
	D: gripping( ) ∧ on(X,Y)

The important thing about the add and delete lists is that they specify *everything* that is necessary to satisfy the frame axioms! Some redundancy exists in the add and delete list approach. For example, in unstack the *add* of gripping(X) could imply the *delete* of gripping( ). But the gain of this redundancy is that every descriptor of a state that is not mentioned by the *add* or *delete* remains the same in the new state description.

A related weakness of the precondition-add-delete list approach is that we are no longer using a theorem-proving process to produce (by inference) the new states. This is not a serious problem, however, as proofs of the equivalence of the two approaches can guarantee the correctness of the precondition-add-delete method. The precondition-add-delete list approach may be used to produce the same results we produced with the inference rules and frame axioms in our earlier example. The state space search, as in Figure 8.19, would be identical for both approaches.

A number of other problems inherent in planning are not solved by either of the two approaches presented so far. In solving a goal we often divide it into subproblems, for instance, unstack(e,d) and unstack(b,a). Attempting to solve these subgoals independently can cause problems if the actions needed to achieve one goal actually undo the other. Incompatible subgoals may result from a false assumption of *linearity* (independence) of subgoals. The non-linearity of a plan/action space can make solution searches unnecessarily difficult or even impossible.

We now show a very simple example of an incompatible subgoal using the start state STATE 1 of Figure 8.18. Suppose the goal of the plan is STATE G as in Figure 8.20, with on(b,a) $\wedge$ on(a,c) and blocks d and e remaining as in STATE 1. It may be noted that one of the parts of the conjunctive goal on(b,a) $\wedge$ on(a,c) is true in STATE 1, namely on(b,a). This already satisfied part of the goal must be undone before the second subgoal, on(a,c), can be accomplished.

The *triangle table* representation (Fikes and Nilsson 1971, Nilsson 1980) is aimed at alleviating some of these anomalies. A triangle table is a data structure for organizing sequences of actions, including potentially incompatible subgoals, within a plan. It addresses the problem of conflicting subgoals within macro actions by representing the global interaction of sequences of operations. A triangle table relates the preconditions of one action to the postconditions, the combined add and delete lists, of actions preceding it.

Triangle tables are used to determine when that macro operator could be used in building a plan. By saving these macro operators and reusing them, STRIPS increases the efficiency of its planning search. Indeed, we can generalize a macro operator, using variable names to replace the block names in a particular example. Then we can call the new generalized macro to prune search. In Chapter 10, with our presentation of learning in symbol based environments, we discuss techniques for generalizing macro operations.

The reuse of macro operators also helps to solve the problem of conflicting subgoals. As the following example illustrates, once the planner has developed a plan for goals of the form stack(X,Y) $\wedge$ stack(Y,Z), it may store and reuse that plan. This eliminates the need to break the goal into subgoals and avoids the complications that may follow.

Figure 8.21 presents a triangle table for the macro action stack(X,Y) $\wedge$ stack(Y,Z). This macro action can be applied to states where on(X,Y) $\wedge$ clear(X) $\wedge$ clear(Z) is true. This triangle table is appropriate for starting STATE 1 with X = b, Y = a, and Z = c.

The atomic actions of the plan are recorded along the diagonal. These are the four actions, pickup, putdown, stack, and unstack, discussed earlier in this section. The set of preconditions of each of these actions are in the row preceding that action, and the

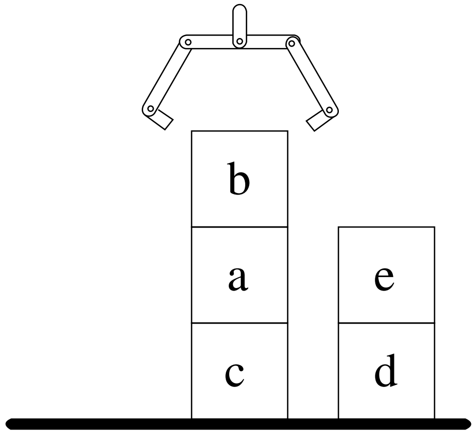
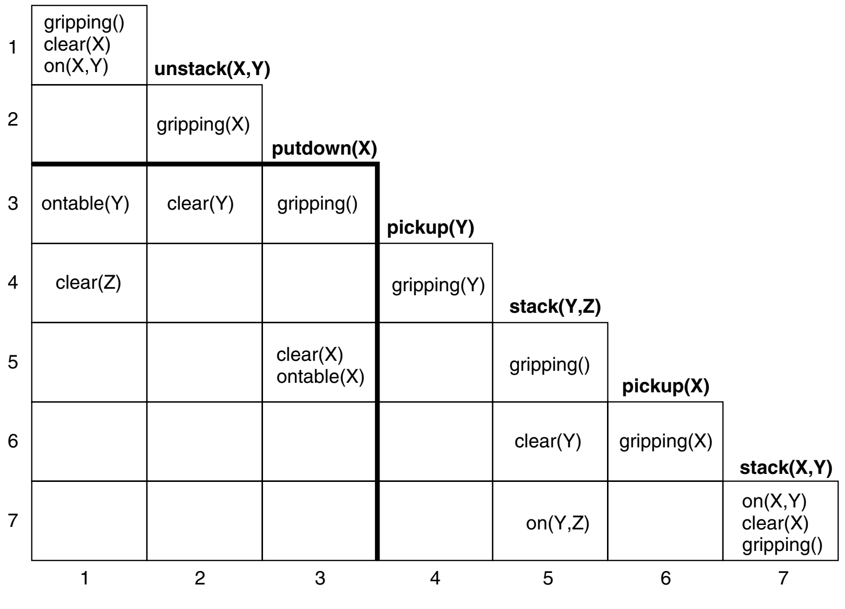


Figure 8.20 Goal state for the blocks world.

postconditions of each action are in the column below it. For example, row 5 lists the preconditions for pickup(X) and column 6 lists the postconditions (the add and delete lists) of pickup(X). These postconditions are placed in the row of the action that uses them as preconditions, organizing them in a manner relevant to further actions. The triangle table's purpose is to properly interleave the preconditions and postconditions of each of the smaller actions that make up the larger goal. Thus, triangle tables address non-linearity issues in planning on the macro operator level; Partial-Order Planners (Russell and Norvig 1995) and other approaches have further addressed these issues.

One advantage of triangle tables is the assistance they can offer in attempting to recover from unexpected happenings, such as a block being slightly out of place, or accidents, such as dropping a block. Often an accident can require backing up several steps before the plan can be resumed. When something goes wrong with a solution the planner can go back into the rows and columns of the triangle table to check what is true. Once the planner has figured out what is still true within the rows and columns, it then knows what the next step must be if the larger solution is to be restarted. This is formalized with the notion of a *kernel*.

The *nth kernel* is the intersection of all rows below and including the nth row and all columns to the left of and including the nth column. In Figure 8.21 we have outlined the third kernel in bold. In carrying out a plan represented in a triangle table, the ith operation (that is, the operation in row i) may be performed only if all predicates contained in the ith kernel are true. This offers a straightforward way of verifying that a step can be taken and also supports systematic recovery from any disruption of the plan. Given a triangle table, we find and execute the highest-numbered action whose kernel is enabled. This not only



1	gripping()
	clear(X)
	on(X,Y)
		unstack(X,Y)					

2		gripping(X)	**putdown(X)**				
3	ontable(Y)	clear(Y)	gripping()	**pickup(Y)**			
4	clear(Z)			gripping(Y)	**stack(Y,Z)**		

5			clear(X)
			ontable(X)
					gripping()	**pickup(X)**	

6					clear(Y)	gripping(X)	**stack(X,Y)**

7					on(Y,Z)		on(X,Y)
							clear(X)
							gripping()

	1	2	3	4	5	6	7

Figure 8.21 A triangle table, adapted from Nilsson (1971).

lets us back up in a plan but also allows for the possibility that an unexpected event might let us jump forward in a plan.

The conditions in the leftmost column are the preconditions for the macro action as a whole. The conditions in the bottom row are the conditions added to the world by the macro operator. Thus a triangle table may be saved as a *macro operator* with its own set of pre and post conditions or add and delete lists.

Of course, the triangle table approach does lose some of the semantics of the previous planning models. Notice, for example, that only those postconditions of an act are retained that are also preconditions of later acts. Thus, if guaranteed correctness is a desired result, further verification of the triangle tables, perhaps with additional information that might allow sequences of triangle tables to be composed, might be desirable.

Other problems arise with the use of macro operators in planning. As the number of macro operators increases, the planner has more powerful operations to use, decreasing the size of the state space that must be searched. Unfortunately, at each step of the search, all of these operators must be examined. The pattern matching needed to determine whether an operator may be applied can add considerable overhead to the search process, often counteracting the gains made by saving macro operations. The problems of determining when a macro operation should be saved and how to determine the next operator to use remain the subject of much research. In the next section we describe an algorithm under which many subgoals may be simultaneously satisfied, teleo-reactive planning.

8.4.3 Teleo-Reactive Planning (Nilsson 1994, Benson 1995)

Since the early work in planning described in the previous section (Fikes and Nilsson 1971), there have been a number of significant advances. Much of this work has been done on domain independent planning, but more recently more domain specific planning, where distributed sense/react mechanisms are employed, has become important. In the next sections we describe a domain independent system, teleo-reactive planning, as well as a more domain dependent planner, Burton, from NASA. For a summary of the first two decades of planning research we recommend (Allen et al. 1990).

Teleo-reactive (TR) planning was first proposed by Nilsson (1994) and Benson (1995) at Stanford University. TR planning offers a general purpose architecture, with application to many domains where control of complex subsystems must be coordinated to accomplish a higher level goal. Thus it combines the top-down approach of hierarchical control architectures with an "agent-based" (Section 7.4) or bottom-up assist. The result is a system that can accomplish complex problem solving through the coordination of simple task-specific agents. The justification for this approach is that simple agents have the advantage of working in smaller and more constrained problem spaces. The higher level controller, on the other hand, can make more global decisions about the entire system, for example, how the current result of a purely local decision can affect the result of the general problem-solving task.

Teleo-reactive control combines aspects of feedback-based control and discrete action planning. Teleo-reactive programs sequence the execution of actions that have been assembled into a goal-oriented plan. Unlike more traditional AI planning environments

(Weld 1994), no assumptions are made that actions are discrete and uninterruptible and that every action's effects are completely predictable. To the contrary, teleo-actions can be sustained over extended periods of time, that is, teleo-actions are executed as long as the actions' preconditions are met and the associated goal has not yet been achieved. Nilsson (1994) refers to this type of action as *durative*. Durative actions can be interrupted when some other action closer to the top-level goal is activated. A short sense-react cycle ensures that when the environment changes the control actions also quickly change to reflect the new state of the problem solution.

Teleo-reactive action sequences are represented by a data structure called a TR tree as in Figure 8.22. A TR tree is described by a set of condition → action pairs (or production rules, Section 6.3). For example:

$$\begin{array}{l} C_0 \rightarrow A_0 \\ C_1 \rightarrow A_1 \\ C_2 \rightarrow A_2 \\ \ldots \\ C_n \rightarrow A_n \end{array}$$

where the C_i are the conditions and the A_i are the associated actions. We refer to C_0 as the top-level goal of the tree and A_0 as the null action, indicating that nothing further needs to be done once the top-level goal is achieved. At each execution cycle of the teleo-reactive system, each C_i is evaluated from the top of the rules to the bottom (C_0, C_1, ... C_n) until the first true condition is found. The action associated with this true condition is then performed. The evaluation cycle is then repeated at a frequency that approximates the reactivity of circuit-based control.

The $C_i \rightarrow A_i$ productions are organized in such a way that each action A_i, if continually executed under normal conditions, will eventually make some condition higher in the TR rule tree, Figure 8.22, true. TR tree execution may be seen as adaptive in that if some unanticipated event in the control environment reverses the effect of previous actions, TR execution will fall back to the lower level rule condition that reflects that condition. From that point it will restart its work towards satisfying all higher level goals. Similarly, if something good inadvertently happens, TR execution is opportunistic in that control will automatically shift to the action of that true condition.

TR trees can be constructed with a planning algorithm that employs common AI goal reduction methods. Starting from the top-level goal, the planner searches over actions whose effects include achievement of that goal. The preconditions of those actions generate a new set of subgoals, and this procedure recurses. Termination is achieved when the preconditions of the leaf nodes are satisfied by the current state of the environment. Thus the planning algorithm regresses from the top-level goal through goal reduction to the current state. Actions, of course, often have side effects and the planner must be careful to verify that an action at any level does not alter conditions that are required as preconditions of actions at a higher level in the TR tree. Goal reduction is thus coupled with constraint satisfaction, where a variety of action reordering strategies are used to eliminate possible constraint violations.

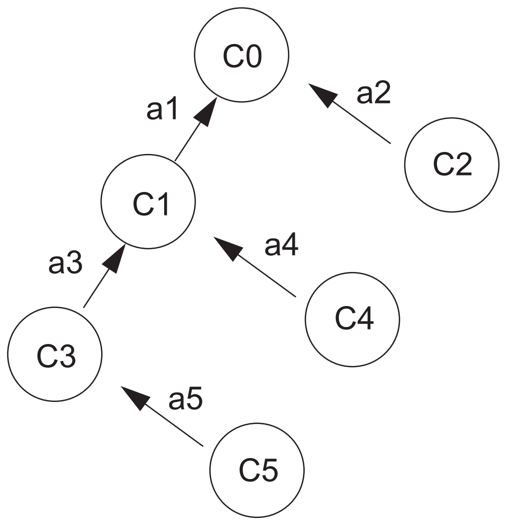


Figure 8.22 A simple TR tree showing condition action rules supporting a top-level goal, from Klein et al. (2000).

Thus, TR planning algorithms are used to build plans whose leaf nodes are satisfied by the current state of the problem environment. They usually do not build complete plans, that is plans that can start from any world state, because such plans are generally too large to store or to execute efficiently. This final point is important because sometimes an unexpected environmental event can shift the world to a state in which no action's preconditions in the TR tree are satisfied and some form of replanning is necessary. This is usually done by reactivating the TR planner.

Benson (1995) and Nilsson (1994) have used teleo-reactive planning in a number of application domains including the control of distributed robot agents and building a flight simulator. Klein et al. (1999, 2000) have used a teleo-reactive planner to build and test a portable control architecture for the acceleration of a charged particle beam. These latter researchers gave a number of reasons for the use of a teleo-reactive controller in the domain of particle beam controllers:

1. Accelerator beams and their associated diagnostics are typically dynamic and noisy.

2. The achievement of accelerator tuning goals is often affected by stochastic processes, RF breakdown, or oscillations in the beam source.

3. Many of the actions required for tuning are durative. This is especially true of tweaking and optimization operations that need to be continued until specific criteria are met.

4. TR trees offer an intuitive framework for encoding tuning plans acquired from accelerator physicists. In fact with very little help, the physicists themselves were able to develop their own TR trees.

Further details of these applications can be found by consulting the references. We next revisit the NASA model-based reasoning example of Section 8.3, to describe control/planning algorithms for the propulsion system of space vehicles.

8.4.4 Planning: a NASA Example (Williams and Nayak)

In this section we describe how a planner can be implemented in the context of a model-based reasoner. We continue the NASA example of Williams and Nayak (1996b) introduced in Section 8.3.2. Livingstone is a reactive configuration manager that uses a compositional, component-based model of the space craft propulsion system to determine configuration actions, as is seen in Figure 8.23.

Each propulsion component is modeled as a transition system that specifies the behaviors of the operating and failure modes of the component, the nominal and failure transitions between modes, and the costs and likelihoods of transitions, as in Figure 8.24. In Figure 8.24, open and closed are normal operation modes, but stuck open and stuck closed are failure modes. The open command has unit cost and causes a mode transition from closed to open, similarly for the close command. Failure transitions move the valve from normal operating modes to one of the failure modes with probability 0.01.

Mode behaviors are specified using formulae in propositional logic, but transitions between modes are specified using formulae in a restricted temporal, propositional logic. The restricted temporal, propositional logic is adequate for modeling digital hardware,

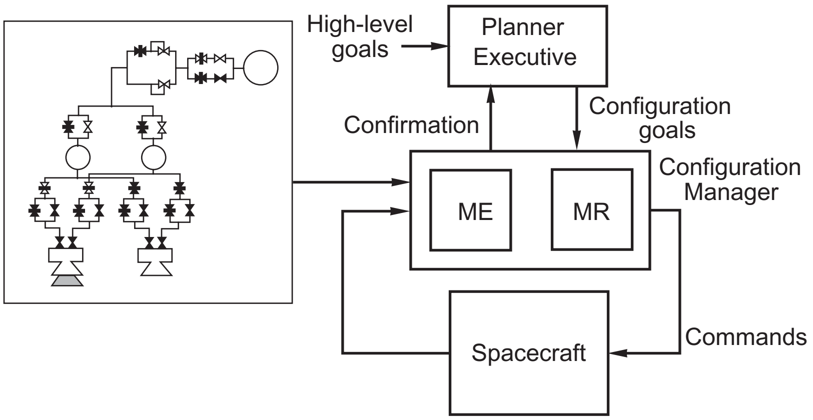


Figure 8.23 Model-based reactive configuration management, from Williams and Nayak (1996b).

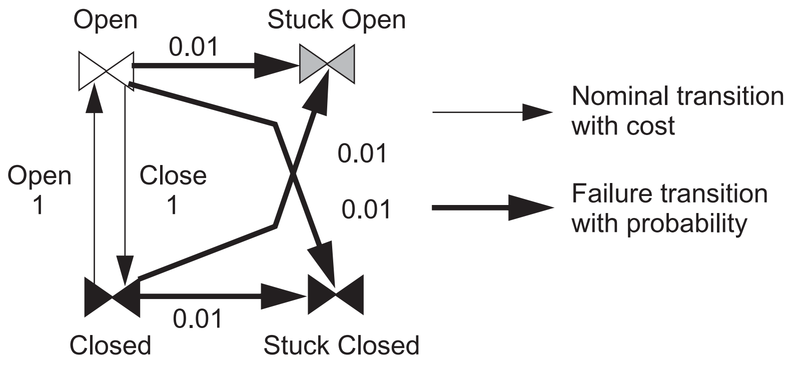

Open
Stuck Open
0.01
Nominal transition with cost
0.01
Open 1
Close 1
0.01
Failure transition with probability
0.01
Closed
Stuck Closed

Figure 8.24 The transition system model of a valve, from Williams and Nayak (1996a).

analog hardware using qualitative abstractions (de Kleer and Williams 1989, Weld and de Kleer 1990), and real time software using the models of concurrent reactive systems (Manna and Pnueli 1992). The spacecraft transition system model is a composition of its component transition systems in which the set of configurations of the spacecraft is the cross product of the sets of component modes. We assume that the component transition systems operate synchronously; that is, for each spacecraft transition every component performs a transition.

A model-based configuration manager uses its transition-system model to both identify the current configuration of the spacecraft, called *mode estimation* ME, and move the spacecraft into a new configuration that achieves the desired configuration goals, called *mode reconfiguration*, MR. ME incrementally generates all spacecraft transitions from the previous configuration such that the models of the resulting configurations are consistent with the current observations. Thus, in Figure 8.25, a situation is shown where the left engine is firing normally in the previous state, but no thrust is observed in the current state. ME's must identify the configurations into which the spacecraft has transitioned that accounts for this observation. The figure shows two possible transitions, corresponding to one of the main engine valves failing. The failed or stuck closed valves are circled. Many other transitions, including unlikely double faults, can also account for the observations.

MR determines the commands to be sent to the spacecraft such that the resulting transitions put the spacecraft into a configuration that achieves the configuration goal in the next state, as in Figure 8.26. This figure shows a situation in which mode identification has identified a failed main engine valve leading to the left engine. MR reasons that normal thrust can be restored in the next state if an appropriate set of valves leading to the right engine is opened. The figure shows two of the many configurations that can achieve the desired goal, when the circled valves are commanded to change state. Transitioning to the configuration at the top has lower cost because only necessary pyro valves (see the discussion in Section 8.3.2) are fired. The valves leading to the left engine are turned off to

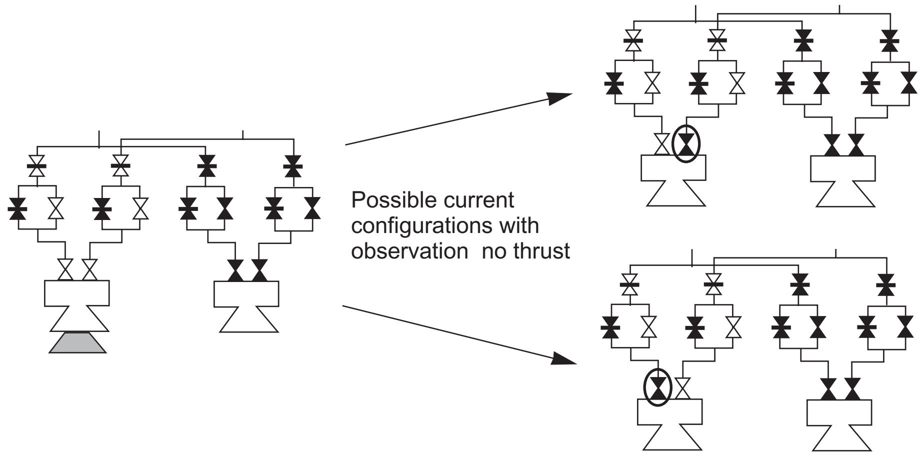


Figure 8.25 Mode estimation (ME), from Williams and Nayak (1996a).

satisfy a constraint that, at most, one engine can fire at a time. The use of a spacecraft model in both ME and MR ensures that configuration goals are achieved correctly.

Both ME and MR are reactive (see Section 6.4). ME infers the current configuration from knowledge of the previous configuration and current observations. MR only considers commands that achieve the configuration goal in the next state. Given these commitments, the decision to model component transitions as synchronous is key. An alternative is to model multiple component transitions through interleaving. However, interleaving can place an arbitrary distance between the current configuration and a goal configuration, defeating the desire to limit inference to a small fixed number of states. Hence model component transitions are synchronous. If component transitions in the underlying hardware-software are not synchronous, Livingstone's modeling assumption is that all interleavings of transitions are correct and support achieving the desired configuration. This assumption is removed in *Burton*, a follow-on of Livingstone, whose planner determines a sequence of control actions that produce all desired transitions (Williams and Nayak 1997). NASA is using the Burton architecture on a mission called Tech Sat 21.

For Livingstone, ME and MR need not generate all transitions and control commands, respectively. Rather all that is required is just the most likely transitions and an optimal control command. These are efficiently regenerated by recasting ME and MR as combinatorial optimization problems. In this reformulation, ME incrementally tracks the likely spacecraft trajectories by always extending the trajectories leading to the current configurations by the most likely transitions. MR then identifies the command with the lowest expected cost that transitions from the likely current configurations to a configuration that achieves the desired goal. These combinatorial optimization problems are efficiently solved using a conflict-directed best-first search algorithm. See Williams and Nayak (1996a, 1996b, 1997) for a more formal characterization of ME and MR as well as a more detailed description of the search and planning algorithms.

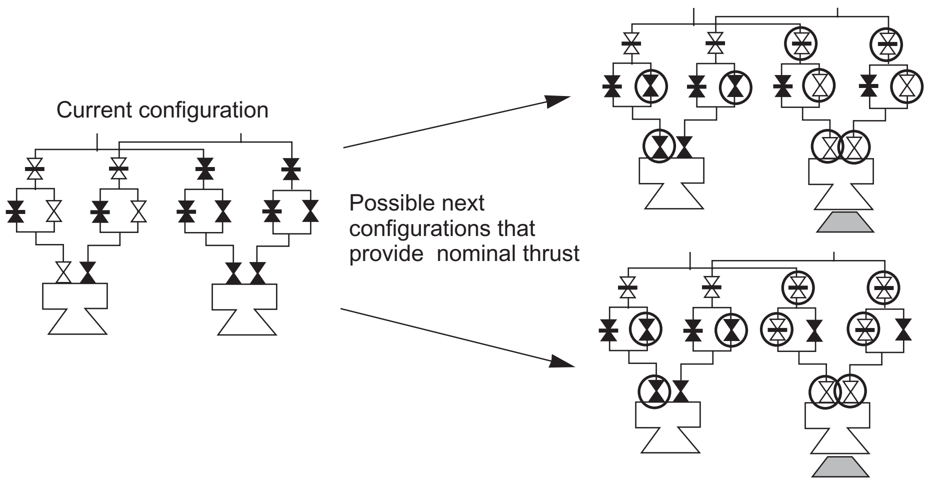


Figure 8.26 Mode reconfiguration (MR), from Williams and Nayak (1996a).

8.5 Epilogue and References

The architecture for the rule-based expert system is the production system. Whether the final product is data-driven or goal-driven, the model for the software is production system-generated graph search, as presented in Chapter 6. We implement production system-based expert system shells in Java, Prolog, and Lisp in the auxilliary materials supplied with this book. The rules of these systems can include certainty measures in a limited form for design of a heuristic-based search.

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 (1984). Other early expert systems work is described in Waterman (1968), Michie (1979), Patil et al. (1981), Clancy (1983), and Clancy and Shortliffe (1984a, 1984b). For a robust implementation of data-driven rule-based search, we recommend the CLIPS software, distributed by NASA. An excellent reference for CLIPS is Giarratano and Riley (1989, 2005).

Other important books on general knowledge engineering include *Building Expert Systems* by Hayes-Roth et al. (1984), *A Guide to Expert Systems* by Waterman (1986), *Expert Systems: Concepts and Examples* by Alty and Coombs (1984), *Expert Systems Technology: A Guide* by Johnson and Keravnou (1985), and *Expert Systems and Fuzzy Systems* by Negoita (1985), *Introduction to Expert Systems* by Jackson (1986), and *Expert Systems: Tools and Applications* by Harmon et al. (1988). See also *Introduction to Expert Systems* by Ignizio (1991), *An Introduction to Expert Systems* by Mockler and Dologite (1992), *Expert Systems: Design and Development* by John Durkin (1994), and conference proceedings in AI applications and expert systems, IEA/AIE (2004 - 2007).

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 (1986), *Competent Expert Systems: A Case Study in Fault Diagnosis* by Keravnou and Johnson (1986), *The CRI Directory of Expert Systems* by Smart and Langeland-Knudsen (1986), *Developments in Expert Systems* by Coombs (1984), and *Developing and Managing Expert Systems* by Prerau (1990). We especially recommend *Expert Systems: Design and Development* by Durkin (1994) for its vast number of practical suggestions on building systems.

A number of techniques for knowledge acquisition have been developed. For more information on specific methodologies see *Knowledge Acquisition: Principles and Guidelines* by McGraw and Harbison-Briggs (1989) and *Knowledge Engineering* by Chorafas (1990), as well as *An Introduction to Expert Systems* by Mockler and Dologite (1992), and books and conference proceedings on expert systems IEA/AIE (2004 - 2007).

Case-based reasoning is an offshoot of earlier research by Schank's group at Yale and their work on scripts (see Sections 7.1.3 - 4). *Case-Based Reasoning* by Kolodner (1993) is an excellent introduction to the CBR technology. Other CBR-related work from Kolodner and her research group include (Kolodner 1987, 1991). Leake (1992, 1996) offers important comment on explanations in case-based systems. CBR software is now available in commercial software products that support the case-based technology.

Model-based reasoning had its origins in the explicit representation of medicine, logic circuits, and other mathematical domains, often used for teaching purposes (deKleer 1975, Weiss et al. 1977, Brown and Burton 1978, Brown and VanLehn 1980, Genesereth 1982, Brown et al. 1982). More modern research issues in MBR are presented in *Readings in Model-Based Diagnosis* (edited by Hamscher et al. 1992), *Diagnosis Based on Description of Structure and Function* (Davis et al. 1982), and diagnosis in the context of multiple faults (deKleer and Williams 1987, 1989). Skinner and Luger (1995) and other writers on agent architectures (Russell and Norvig 1995, 2003) describe hybrid expert systems, where the interactions of multiple approaches to problem solving can create a synergistic effect with the strengths of one system compensating for the weaknesses of others.

The planning section demonstrates some of the data structures and search techniques used for general-purpose planners. Further references include ABSTRIPS or ABstract specification for STRIPS generator relationships (Sacerdotti 1974), Warren's work (1976), and NOAH for nonlinear or hierarchical planning (Sacerdotti 1975, 1977). For a more modern planner see *teleo-reactive* planning, Section 15.3 (Benson and Nilsson 1995).

Meta-planning is a technique for reasoning not just about the plan but also about the process of planning. This can be important in expert systems solutions. References include Meta-DENDRAL for DENDRAL solutions (Lindsay et al. 1980) and Teiresias for MYCIN solutions (Davis 1982). For plans that interact continuously with the world, that is, model a changing environment, see McDermott (1978).

Further research includes *opportunistic planning* using blackboards and planning based on an object-oriented specification (Smoliar 1985). There are several surveys of planning research in the *Handbook of Artificial Intelligence* (Barr and Feigenbaum 1989, Cohen and Feigenbaum 1982) and the *Encyclopedia of Artificial Intelligence* (Shapiro 1992). Non-linearity issues and partial-order planning are presented in Russell and Norvig (1995). *Readings in Planning* (Allen et al. 1990) is relevant to this chapter.

Our description and analysis of the NASA model-based reasoning and planning algorithms was taken from Williams and Nayak (1996a, 1996b, 1997). We thank Williams and Nayak, as well as AAAI Press, for allowing us to cite their research.

8.6 Exercises

1. In Section 8.2 we introduced a set of rules for diagnosing automobile problems. Identify possible knowledge engineers, domain experts, and potential end users for such an application. Discuss the expectations, abilities, and needs of each of these groups.
2. Take Exercise 1 above. Create in English or pseudocode 15 if-then rules (other than those prescribed in Section 8.2) to describe relations within this domain. Create a graph to represent the relationships among these 15 rules.
3. Consider the graph of Exercise 2. Do you recommend data-driven or goal-driven search? breadth-first or depth-first search? In what ways could heuristics assist the search? Justify your answers to these questions.
4. Pick another area of interest for designing an expert system. Answer Exercises 1–3 for this application.
5. Implement an expert system using a commercial shell program. These are widely available for personal computers as well as larger machines. We especially recommend CLIPS from NASA (Giarratano and Riley 1989) or JESS from Sandia National Laboratories.
6. Critique the shell you used for Exercise 5. What are its strengths and weaknesses? What would you do to improve it? Was it appropriate to your problem? What problems are best suited to that tool?
7. Create a model-based reasoning system for a simple electronic device. Combine several small devices to make a larger system. You can use *if... then...* rules to characterize the system functionality.
8. Read and comment on the paper *Diagnosis based on description of structure and function* (Davis et al., 1982).
9. Read one of the early papers using model-based reasoning to teach children arithmetic (Brown and Burton 1978) or electronic skills (Brown and VanLehn 1980). Comment on this approach.
10. Build a case-based reasoner for an application of your choice. One area might be for selecting computer science and engineering courses to complete an undergraduate major or a MS degree.
11. Use commercial software (check the WWW) for building the case-based reasoning system of Exercise 10. If no software is available, consider building such a system in Prolog, Lisp, or Java.
12. Read and comment on the survey paper *Improving Human Decision Making through Case-Based Decision Aiding* by Janet Kolodner (1991).
13. Create the remaining *frame axioms* necessary for the four operators pickup, putdown, stack, and unstack described in rules 4 through 7 of Section 8.4.

14. Use the operators and frame axioms of the previous question to generate the search space of Figure 8.19.

15. Show how the *add* and *delete* lists can be used to replace the frame axioms in the generation of STATE 2 from STATE 1 in Section 8.4.

16. Use *add* and *delete* lists to generate the search space of Figure 8.19.

17. Design an automated controller that could use *add* and *delete* lists to generate a graph search similar to that of Figure 8.19.

18. Show two more incompatible (precondition) subgoals in the blocks world operators of Figure 8.19.

19. Read the ABSTRIPS research (Sacerdotti 1974) and show how it handles the linearity (or incompatible subgoal) problem in planning.

20. In Section 8.4.3 we presented a planner created by Nilsson and his students at Stanford (Benson and Nilsson 1995). *Teleo-reactive* planning allows actions described as *durative*, i.e., that must continue to be true across time periods. Why might teleo-reactive planning be preferred over a STRIPS-like planner? Build a teleo-reactive planner in Prolog, Lisp or Java.

21. Read Williams and Nayak (1996, 1996a) for a more complete discussion of their model-based planning system.

22. Expand the propositional calculus representation scheme introduced by Williams and Nayak (1996a, p. 973) for describing state transitions for their propulsion system.