UNIT-V

Expert system and applications: Introduction phases in building expert systems, expert system versus traditional systems, rule-based expert systems blackboard systems truth maintenance systems, application of expert systems, list of shells and tools

---

5.1 Introduction

One of the goals of AI is to understand the concept of intelligence and develop intelligent computer programs. An example of a computer program that exhibits intelligent behaviour is an expert system (ES). Expert systems are meant to solve real-world problems which require specialized human expertise and provide expert quality advice, diagnoses, and recommendations. An ES is basically a software program or system that tries to perform tasks similar to human experts in a specific domain of the problem. It incorporates the concepts and methods of symbolic inference, reasoning, and the use of knowledge for making these inferences. Expert systems may also be referred to as knowledge-based expert systems. Expert systems represent their knowledge and expertise as data and rules within the computer system; these rules and data can be called upon whenever needed to solve problems. It should be noted that the term expert systems is often reserved for programs whose knowledge base contains the knowledge provided by human experts in contrast to knowledge gathered from textbooks or non-experts. The earliest examples of ES are briefly mentioned below:

- The system called MYCIN was developed using the expertise of best diagnosticians of bacterial infections whose performance was found to be better than the average clinician,
- In another real-world case, at a chemical refinery, a knowledge engineer was assigned to produce an ES to reproduce the expertise of an experienced retired employee to save the company from incurring the loss of the valued knowledge asset that the employee possessed.

An important characteristic of an ES is that the sequence of steps taken to reach a conclusion is dynamically synthesized with each new case. The sequence is not explicitly programmed during the development of the system. Expert systems can process multiple values for any problem parameter. This causes them to pursue more than one line of reasoning hence leading to results of incomplete (not fully determined) reasoning being presented. Problem solving by these systems is accomplished by applying specific knowledge rather than specific technique. This is the key idea in ES technology. It reflects the belief that human experts do not process their knowledge differently from others, but they do possess different knowledge. Therefore, the power of an ES lies in its store of knowledge regarding the problem domain; the more knowledge a system is provided, the more competent it becomes. Expert systems may or may not possess learning components; however, once they are fully developed, their performance is evaluated by subjecting them to real-world problem-solving situation (Luger & Stubblefield, 1993).

5.2 Phases in Building Expert Systems

Building an ES initially requires extracting the relevant knowledge from a human domain expert; this knowledge is often based on useful thumb rules and experience rather than absolute certainties. Usually experts find it difficult to express concretely the knowledge and rules used by them while solving a problem. This is because their knowledge is almost subconscious or appears so obvious that they do not bother mentioning it. After extracting knowledge from domain experts, the next step is to represent this knowledge in the system. Representation of knowledge in a

computer is not straight forward and requires special expertise. A knowledge engineer handles the responsibility of extracting this knowledge and building the ES's knowledge base. This process of gathering knowledge from a domain expert and codifying it according to the formalism is called knowledge engineering. This phase is known as knowledge acquisition, which is a big area of research. A wide variety of techniques have been developed for this purpose. Generally, an initial prototype based on the information extracted by interviewing the expert is developed. This prototype is then iteratively refined on the basis of the feedback received from the experts and potential users of the ES. Refinement of a system is possible only if the system is scalable and modifiable and does not require rewriting of major code. The developed system should be able to explain its reasoning to its users and answer questions about the solution process. Moreover, updating the system should just involve adding or deleting localized regions of knowledge [Waterman 1956).

A simple ES primarily consists of a knowledge base and an inference engine, while features such as reasoning with uncertainty and explanation of the line of reasoning enhance the capabilities of ES. Since an ES uses uncertain or heuristic knowledge just like humans, its credibility is often questionable. In real-life situations whenever we obtain an answer to a problem which seems questionable, we want to know the rationale behind it, and we believe the answer only if the rationale seems plausible to us. Similar is the case with expert systems; most expert systems han the ability to answer questions of the form 'Why the answer X? explanations can be tracing the line of reasoning used by the inference engine.

To be more precise, the different interdependent and overlapping phases involved in building ES may be categorized as follows:
generated by
Identification Phase In this phase, the knowledge engineer determines important features of the problem with the help of the human domain expert. The parameters that are determined in this phase include the type and scope of the problem, the kind of resources required, and the goal and objective of the ES.
Conceptualization Phase In this phase, knowledge engineer and domain expert decide the concepts, relations, and control mechanism needed to describe the problem-solving method. At this stage, the issue of granularity is also addressed, which refers to the level of details required in the knowledge
Formalization Phase This phase involves expressing the key concepts and relations in some framework supported by ES building tools. Formalized knowledge consists of data structures, inference rules, control strategies, and languages required for implementation.
Implementation Phase During this phase, formalized knowledge is converted to a working computer program, initially called prototype of the whole system.
Testing Phase This phase involves evaluating the performance and utility of prototype system and revising the system, if required. The domain expert evaluates the prototype system and provides feedback, which helps the knowledge engineer to revise it.

5.2.1 Knowledge Engineering

The whole process of building an ES is often referred to as knowledge engineering. It typically involves a special form of interaction between ES builder, or the knowledge engineer, one or more domain experts, and potential users. Although there are different ways and methods of knowledge engineering, the basic approach remains the same. The tasks and responsibilities of a knowledge engineer involve the following:
• Ensuring that the computer has all the knowledge needed to solve a problem.
• Choosing one or more forms to represent the required knowledge.
• Ensuring that the computer can use the knowledge efficiently by selecting some of the reasoning methods.

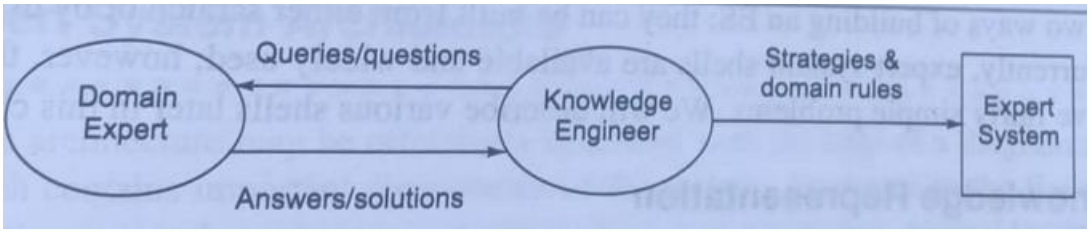

Figure 5.1 shows the interaction between the knowledge engineer and the domain expert to produce an ES.

The main role of the knowledge engineer begins only once the problem of some domain for an ES is decided. The job of the knowledge engineer involves close collaboration with the domain expert(s) and the end user(s). The knowledge engineer may or may not have any knowledge of the application domain initially; however, he/she must become familiar with the problem domain by reading introductory texts or literature and talking to the domain expert(s). The next step of the process involves a more systematic interviewing of the expert.
The knowledge engineer will then extract general rules from the discussion and interview held with expert(s) and get them checked by the expert(s) for correctness. The engineer then translates the knowledge into a computer-usable language and designs an inference engine, which is a reasoning structure that uses the knowledge appropriately. He/she also determines the mechanism to integrate the use of uncertain knowledge in the reasoning process, and should know the kinds of explanation that may be useful to the end user.

The domain knowledge, consisting of both formal, textbook knowledge and experiential knowl-edge (obtained by the expert' experiences), is entered into the program piece by piece. In the initial stages, the knowledge engineer may encounter a number of problems such as the inference engine may not be right, the form of knowledge representation may not be appropriate for the kind of knowledge needed for the task, or the expert may find the pieces of knowledge incorrect. As all these are discovered and modified, the ES gradually gains competence.

The development of ES would remain incomplete if it did not involve close collaboration with end users. The basic development cycle should include the development of an initial prototype and iterative testing and modification of that prototype by both experts (for checking the validity of the rules) and users (for checking the performance of the system and explanations for the answers). In order to develop the initial prototype, the knowledge engineer will have to take provisional decisions regarding appropriate knowledge representation (e.g., rules, semantic net or frames, etc.) and inference methods (e.g., forward chaining or backward chaining or both). To test these basic design decisions, the first prototype may be so designed that it only solves a small part of the overall problem. If the methods used seem to work well for that small part, then that would imply that it is worth investing effort in representing the rest of the knowledge in the same form.
During the initial years of ES development era, there were unrealistic expectations about the potential benefits of these systems. But now it has been realized that building expert systems for very complicated problems is not successful and may not fulfil expectations.
There are two ways of building an ES: they can be built from either scratch or by using ES shells or tools. Currently, expert system shells are available and widely used; however, they are often used to solve fairly simple problems. We will describe various shells later in this chapter.

5.2.2 Knowledge Representation

It must have already become clear by now that the most important ingredient in any ES is know). edge. The power of expert systems resides in the specific, high-quality knowledge they contain is accumulated during the system-building phase; accumulation and codification of knowledge is one of the most important aspects of ES. Expert knowledge of the problem domain is organized in

such a way that this knowledge is separated from other knowledge possessed by the system such as knowledge about user's interaction, general knowledge about how to solve a problem, etc. The collection of domain knowledge is called knowledge base, while the general problem-solving knowledge may be called inference engine, user interface, etc. The most common knowledge representation scheme for expert systems consists of production rules, or simply rules; they are of the form if-then, where the if part contains a set of conditions in some logical combination. The piece of knowledge represented by the production rule is relevant to the line of reasoning being developed when if part of the rule is satisfied; consequently, the then part can be concluded. Expert systems in which knowledge is represented in the form of rules are called rule-based systems. The rules may have certain conclusions or may have some degree of uncertainty; statistical techniques (such as probability) are used to handle such rules. The concept of certainty has been discussed separately in Chapter 9. Rule-based systems are easily modifiable and helpful in giving traces of the system's reasoning. These traces can be used in providing explanations of how the system solved the problem.

Another widely used representation in ES is called the unit (also known as frame, semantic net, etc.), which is based upon a more passive view of knowledge. The unit is an assemblage of

associated symbolic knowledge about an entity to be represented. Typically, a unit consists of a list of properties of an entity and associated values for those properties. Since every task domain consists of many entities that occur in various relations, properties can also be used to specify relations. The values of these properties represent the names of other units that are linked according to relations. One unit can also represent knowledge that is a special case of another unit, or some units can be parts of another unit.

Al researchers continue to explore and add to the current list of knowledge representation techniques and reasoning methods. We now understand that knowledge is an integral part of ES. However, the current knowledge acquisition methods are slow and tedious. Thus, the future of ES depends on developing better methods of knowledge acquisition and in codifying and representing a large knowledge infrastructure.

5.3 Expert System Architecture

Expert system architecture may be effectively described with the help of a diagram as given in Fig. 5.2, which contains important components of the system. As shown in the figure, the user interacts with the system through a user interface which may use menus, natural language, or any other style of interaction. Then, an inference engine is used to reason with the expert knowledge as well as the data specific to the problem being solved. Case-specific data includes both data provided by the user and partial conclusions along with certainty measures based on this data. In a simple forward-chaining rule-based system, case-specific data will be included in working memory. Generally, all expert systems possess an explanation subsystem, which allows the program to explain its reasoning to the user. Some systems also have a knowledge acquisition module that helps the expert or knowledge engineer to easily update and check the knowledge base. Each of these components is briefly described in the following subsection.

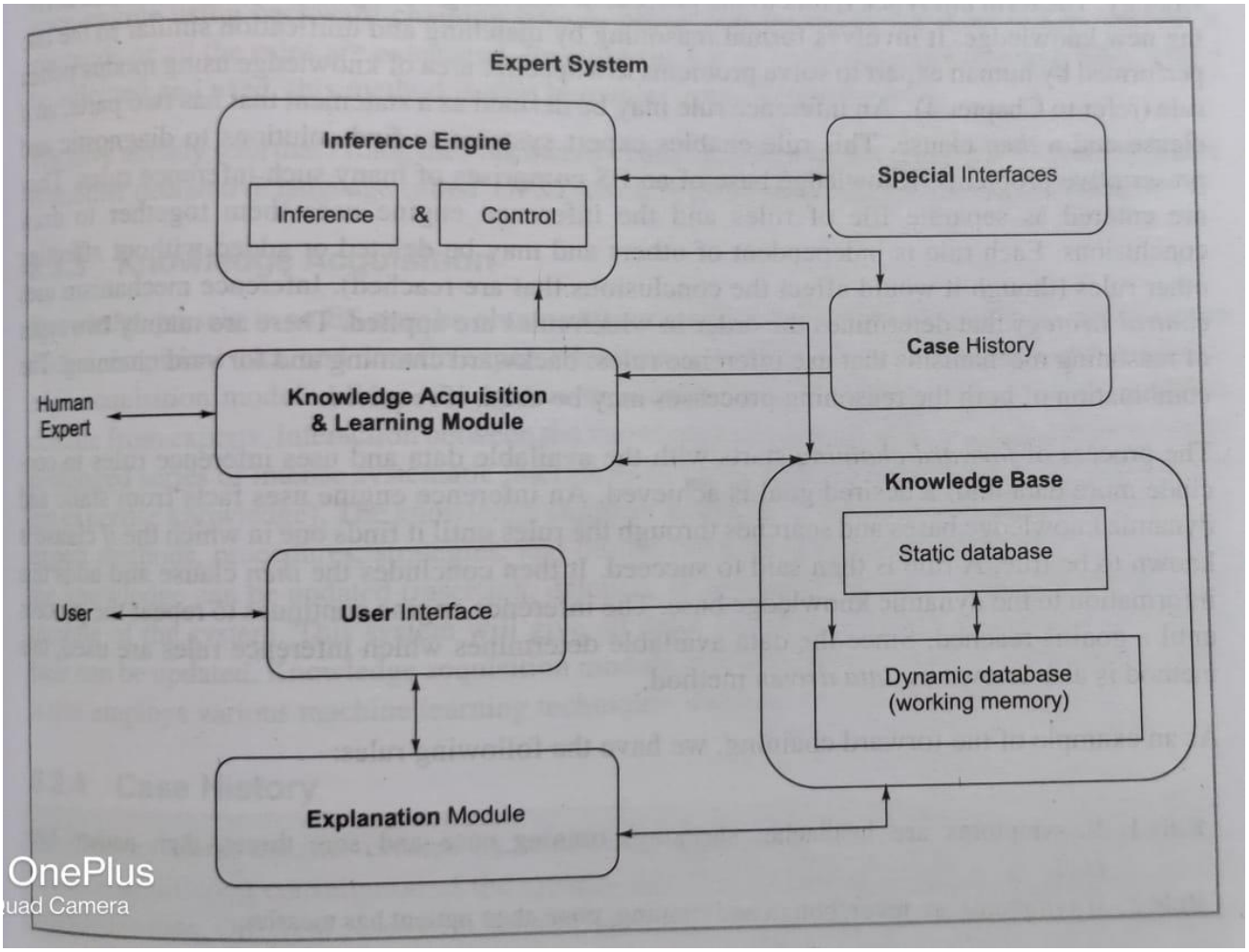

Figure 5.2 Architecture of an Expert System

5.3.1 Knowledge Base

Knowledge base of an ES consists of knowledge regarding problem domain in the form of static and dynamic databases. Static knowledge consists of rules and facts, or any other form of knowledge representation which may be compiled as a part of the system and does not change during the execution of the system. On the other hand, dynamic knowledge consists of facts related to a particular consultation of the system collected by asking various questions to the user who is consulting the ES. At the beginning of the consultation, the dynamic knowledge base (often called working memory) is empty. As the consultation progresses, dynamic knowledge base (in the form of facts only) grows and is used in decision making along with static knowledge. Working memory is deleted at the end of consultation of the system.

5.3.2 Inference Engine

An inference engine developed for an ES consists of inference mechanism as well as control ing new knowledge. It involves formal reasoning by matching and unification similar to the on: strategy. The term inference refers to the process of searching through knowledge base and deriv performed by human expert to solve problems in a specific area of knowledge using modus rule (refer to Chapter 4). An inference rule may be defined as a statement that has two parts, an clause and a then clause. This rule enables expert systems to find solutions to diagnostic an prescriptive problems. Knowledge base of an ES comprises of many such inference rules. The are entered as separate file of rules and the inference engine uses them together to draw conclusions. Each rule is independent of others and may be deleted or added without affecting other rules (though it would affect the conclusions that are reached). Inference mechanism uses control strategy that determines the order in which rules are applied. There are mainly two types of reasoning mechanisms that use inference rules: backward chaining and forward chaining. The combination of both the reasoning processes may be used, if required.

The process of forward chaining starts with the available data and uses inference rules to conclude more data until a desired goal is achieved. An inference engine uses facts from static and dynamic knowledge bases and searches through the rules until it finds one in which the if clause is

known to be true. A rule is then said to succeed. It then concludes the then clause and adds this information to the dynamic knowledge base. The inference engine continues to repeat the process until a goal is reached. Since the data available determines which inference rules are used, this method is also known as data driven method.

As an example of the forward chaining, we have the following rules:

Rule 1 If symptoms are headache, sneezing, running nose and sore_throat, then patient has cold.

Rule 2 If symptoms are fever, cough and running nose, then patient has measles.

Facts are generated in working memory by asking questions to the user whether he has fever, running nose, cough, etc.

Thus, in forward chaining, we start with the facts given by the user and try to find an appropriate rule whose if part is satisfied and subsequently the then part is concluded.

The other important inference mechanism that we are familiar with is backward chaining. Backward chaining starts with a list of goals and works backwards to see if there is data which will allow it to conclude any of these goals. An inference engine using backward chaining would search the inference rules until it finds one whose then part matches a desired goal. If the if part of that inference rule is not known to be true, then it is added to the list of goals.

Consider the same example discussed above. In order to satisfy a goal called cold, the inference engine will select a rule with conclusion as cold and will try to find the facts in the if part of the role whether the user has headache, sneezing, running nose, and sore throat. If yes, then cold is established otherwise it tries other rule for goal, if it exists. If we are not able to satify all the rules with the goal cold then other goals such as measles will be tried. Using rule 2, if the symptoms of the user are fever, running nose, and cough, then measles is concluded. The inferonce engine using backward chaining tries to prove conclusion of the rules one by one till it succeeds or all the rules are exhausted. Because the list of goals determines which rules would be selected and used, this method is also known as goal-driven method.

5.3.3 Knowledge Acquisition

Knowledge present in an ES may be obtained from many sources such as textbooks, reports, case studies, empirical data, and domain expert which are a prominent source of knowledge. A knowledge acquisition module allows the system to acquire more knowledge regarding the problem domain from experts. Interaction between the knowledge engineer and the domain expert involves prolonged series of intense systematic interviews or using a questionnaire (carefully designed to get expertise knowledge). The knowledge engineer working on a system should be able to extract expert methods, procedures, strategies, and thumb rules for solving the problem at hand. Later, the knowledge can be updated (insertion, deletion, or updation) by using knowledge acquisition module of the system. This system will give all these facilities to expert so that the knowledge base can be updated. Knowledge acquisition module may also have a learning module attached to it that employs various machine learning techniques discussed in Chapter 11.

5.3.4 Case History

Case history *stores* the files created by inference engine using the dynamic database and is used by the learning module

Different cases with solutions are stored in Case Base System and these cases are used for solving the problem using Case Base Reasoning (CBR)

5.3.5 User Interfaces

User interface of an ES allows user to communicate with the system in an interactive manner and helps the system in creating working knowledge for the problem that has to be solved. Knowledge may be entered using some editor or specialized designed user interface. The function of the user interface is to present questions and information to the user and supply the responses of user to the inference engine. Any values provided by the user need to be received and interpreted by the user interface. Responses of the user may or may not be restricted to a set of possible legal answers. The user interface checks all responses to ensure that they are of the correct data type. Any responses that are restricted to a legal set of answers are compared against the legal answers. So, whenever the user enters an illegal answer, the user interface informs the user that the answer is invalid and prompts the user to provide the correct answer.

Therefore, the end-user usually sees an ES through an interactive dialogue module. We observe from the dialogue module given in Table 5.1 how the system leads the user through a set of questions, whose purpose is to determine a set of symptoms. Note that the dialogues are not pre-planned in expert systems and there is no fixed control structure. They are generated from the current information present in the system and the knowledge contained in the knowledge base. The answers to questions shown in Table 5.1 are binary in yes or no form for simplification. An advanced interface may accept answers in natural language and then it employs natural language processing techniques to extract out relevant information from the response. Responses of user may be fuzzy also. Lot of research is in progress in this area.

Table 5.1 Dialogue Module of a User Interface

System	Do you have fever?
User	Yes
System	Do you have sore throat?
User	No
System	Do you have cough?
User	Yes
System	Are you suffering from running nose?
User	Yes
System	Are you suffering from headache?
User	No

5.3.6 Explanation Module

Most of the ES has explanation facilities that enable users to query the system about why it asked some questions and how it reached some conclusion; these modules are called How and Why, The sub-module How tells users regarding the process through which the system has reached a particular solution, while the Why sub-module explains to the user, the reasoning behind arriving at a solution to any particular problem. These questions are answered by referring to the system goals, the rules being used, and any existing problem data. An ES can generate an explanation by retracing the steps of its reasoning. The response of the ES to the question why may provide an insight into the underlying knowledge structure. The knowledge structure of an ES consists of a rule, a set of antecedent conditions, which, if true, allows the assertion of a consequent. To illustrate the use of explanation facilities, the Table 5.2 shows explanation for why and how questions.

Table 5.2 Explanation Module

Invoke	Why/How modules
System	Do you have running nose?
User	Yes
System	I am trying to prove that you have cold. Do you have headache?
User	Yes
System	Do you have sore throat?
User	No
System	Then you do not have cold. Do you have fever?
User	Yes
System	Now I am looking at the possibility of measles. Do you have cough?
User	Yes
System	I can infer measles using the rule "If symptoms are fever, cough, running_nose, then patient has measles". Measles is concluded.
User	**How?**
System	Since you have fever, running_nose, and cough and there is a rule "If symptoms are fever, cough, running_nose, then patient has measles". So, measles is concluded for you.

Providing explanation facility in a system involves, at least, recording the rules that are used in

drawing conclusions; these records are then used to compose the explanations. Providing simple explanations as those given in the above table is not very difficult (because the answers always have a simple set form); these explanations prove to be quite useful at times. However, it is very difficult to implement a general explanation system in a traditional computer program.

5.3.7 Special Interfaces

Special interfaces may be used in ES for performing specialized activities, such as handling uncertainty in knowledge. These interfaces form a major area of expert system research which involves methods for reasoning with uncertain data and uncertain knowledge. A point to be kept in mind regarding knowledge is that it is generally incomplete and uncertain. To deal with uncertain a confidence factor or a weight may be associated with a rule. The set of methods for using uncertain knowledge in combination with uncertain data in the reasoning process is called reasoning with uncertainty. Probability theory is the oldest method used to determine these certainties. Another important subclass of methods for reasoning with uncertainty is called fuzzy logic and the systems that use them are known as fuzzy systems. Uncertainty and fuzzy logic have been discussed in Chapters 9 and 10, respectively.

5.4 Expert Systems versus Traditional Systems

The basic difference between an ES and a traditional system is that an ES manipulates knowledge, whereas a traditional system manipulates data. The distinction between these systems lies in the manner in which the problem-related expertise is coded into them. In traditional applications, problem expertise is encoded in program as well as in the form of data structures. On the other hand, in the ES approach, all problem-related expertise is encoded in data structures only and not in the programs. However, in expert systems, small fragments of human experience are collected into a knowledge base which are used to reason through a problem. A different problem, within the domain of the knowledge base, can be solved using the same program without having to reprogram the system.

Another advantage of expert systems over traditional systems is that they allow the use of confidences or certainty factors. This is similar to human reasoning where one cannot always conclude things with 100% confidence. For example, consider the statement If weather is humid, then it might probably rain. The use of words such as if, then, might, probably, etc., indicate that there is some uncertainty involved in the statement. This type of reasoning can be imitated by using numeric values called confidences in ES. For example, if it is known that the weather is humid, it might be concluded with 0.9 confidences that it rains. Although, these numbers are similar to probabilities, they are not the same. They are meant to imitate the confidences humans use in reasoning rather than to follow the mathematical definitions used in calculating probabilities. Therefore, the ability of ES to explain the reasoning process through back-traces and to handle levels of confidence and uncertainty provides an additional feature as compared to conventional or traditional programs.

Further, conventional programs are designed to always produce correct answers, whereas expert systems are designed to behave like human experts and may sometimes produce incorrect results. In the following subsections, we will discuss some important characteristics of expert systems, their evaluations, advantages and disadvantages, and the languages that can be used for their development.

5.4.1 Characteristics of Expert Systems

Some key characteristics that every ES must possess are described as follows:

- **Expertise** An ES should exhibit expert performance, have high level of skill, and possess adequate robustness. The high-level expertise and skill of an ES aids in problem solving and makes the system cost effective.
- **Symbolic reasoning** Knowledge in an ES is represented symbolically which can be easily reformulated and reasoned.
- **Self knowledge** A system should be able to explain and examine its own reasoning.

Learning capability A system should learn from its mistakes and mature as it grows. Flexibility provided by the ES helps it grow incrementally.

- **Ability to provide training** Every ES should be capable of providing training by explaining the reasoning process behind solving a particular problem using relevant knowledge.
- **Predictive modelling power** This is one of the important features of ES. The system can act as an information processing model of problem solving. It can explain how new situation led to the change, which helps users to evaluate the effect of new facts and understand their relationship to the solution.

5.4.2 Evaluation of Expert Systems

Evaluation of an ES consists of performance and utility evaluation. Performance evaluation consists of answering various questions such as the ones given below.

• Does the system make decisions that experts generally agree to?
• Are the inference rules correct and complete?
• Does the control strategy allow the system to consider items in the natural order that the expert prefers?
. Are relevant questions asked to the user in proper order (otherwise it will be an irritating process)?
• Are the explanation given by the ES adequate for describing how and why conclusions?

Utility evaluation consists of answering the following questions:

• Does the system help user in some significant way?
• Are the conclusions of the system organized and ordered in a meaningful way?
• Is the system fast enough to satisfy the user?
• Is the user interface friendly enough?

5.4.3 Advantages and Disadvantages of Expert Systems

One has to justify the need for developing an ES as it involves a lot of time and money. In order to avoid costly and embarrassing failures, one should evaluate whether a problem is suitable for an ES solution using the following guidelines:

Specialized knowledge problems If there is a rare specialized knowledge involved in some special domain (say, oil exploration and medicine), then it is worth developing an ES to solve problems pertaining to the domain as human experts are scarce and unavailable. This implies that we typically need to develop ES for problems that require highly specialized expertise, which is likely to be lost due to personnel changes or retirement.

High payoff An ES may be developed if the task to be performed has a very high payoff. The company may need similar expertise at a large number of different physical locations.

Existence of cooperative experts For an ES to be successfully developed, it is essential that the experts are willing to help and provide their expertise. We also need potential users to be involved who will be using the system once it is fully developed.

Justification of cost involved in developing ES There must be a realistic assessment of the costs and benefits involved.

The type of problem The problem for which an ES is to be developed must be structured and it does not require common sense knowledge as common sense knowledge is hard to capture and represent. It is easier to deal with ES developed for highly technical fields.

If there exists a good algorithmic solution to a problem, there is no need to build an ES. The problems that cannot be easily solved using more traditional computing methods are potential problems for ES. It should be clear that only a small range of problems are appropriate for ES technology. However, given a suitable problem, expert systems can bring enormous benefits. The advantages and disadvantages of ES are listed as follows:

Advantages

• Helps in preservating scarce expertise.
• Provides consistent answers for repetitive decisions, processes, and tasks.
• Fastens the pace of human professional or semi-professional work.
• Holds and maintains significant levels of information.
• Provides improved quality of decision making.
• Domain experts are not always able to explain their logic and reasoning unlike ES.
• Encourages organizations to clarify the logic of their decision-making.

• Leads to major internal cost savings within companies.
• Causes introduction of new products.
• Never forgets to ask a question, unlike a human.

Disadvantages

• Unable to make creative responses as human experts would in unusual circumstances.
• Lacks common sense needed in some decision making.
. May cause errors in the knowledge base, and lead to wrong decisions.
• Cannot adapt to changing environments, unless knowledge base is changed.

5.4.4 Languages for ES Development

The basic hypothesis of AI is that intelligent behaviour can be described as symbol manipulation and can be modelled with the symbol processing capabilities of the computer. Special programming languages were invented in the late 1950s and they facilitate symbol manipulation. The most prominent of them is called LISP (LISt Processing), which is based on lambda calculus. It is a simple, elegant, and flexible language; most AI research programs are written in LISP. However, commercial applications have moved away from LISP towards newer improved languages.

Another AI programming language, known as Prolog (PROgramming in LOGic), was invented in the early 1970s. Prolog is based on first-order predicate calculus (refer Chapter 5). Prolog programs behave in a manner similar to rule-based systems. A variety of logic-based programming languages have been developed since the development of Prolog causing the term prolog to oheormel generic.

5.5 Rule-based Expert Systems

In this section, we will demonstrate how expert systems based on if-then rules work. We will also develop a very simple ES shell in Prolog. Rule-based systems can be either goal driven (using backward chaining to test whether a given hypothesis is true) or data driven (using forward chaining to draw new conclusions from existing data). Expert systems may use either one or both strategies, but the most common is probably the goal-driven/backward-chaining strategy. One reason for this is that normally an ES will have to collect information about the problem from the user by asking questions. However, in case of a goal-driven strategy, we can just ask questions that are relevant to a hypothesized solution.

5.5.1 Expert System Shell in Prolog

In this section, we will develop simple expert system shell which uses backward chaining control strategy. Even though we can write ES shell that uses forward chaining, since Prolog provides backward chaining, it is simple for us to write shell in Prolog using this mechanism. We define a special syntax for the rules using operator declarations (:-op). Using op, any standard system operator declaration can be changed or new operator can be defined by the user. A goal op with three arguments takes the following form:

:- op(Prec, Type, Atom)

where Prec is an unsigned integer in the range [1,1200], Type e {fx, fy, xfx, xfy, yfx, yfy} is a specification representing associativity, and Atom is a symbol or a name declared as an operator with precedence Prec and Type.

For declaring infix operators, possible values of specification Type are xfx, xfy, yfx, and yfy where f represents an operator and x and y represent arguments. The choice of x and y in the positions mentioned above convey associativity information. The character x means the operator in the argument must have strictly lower precedence value of operator f and y means the argument can contain operator with the same or lower precedence value than the operator f. Therefore, an operator declared with specification as yfx is left associative and declared with specification as xfy is right associative. If an operator is non associative, then the specification used is xfx. For example,

:- op(500, yfx, +).

op(500, yfx,-).
op(300, xfx, mod).
op(700, xfx, >).

The Type fx indicates non-associative (prefix) operators, fy denotes right-associative (prefix) operators, xf implies non-associative (postfix) operators, and yfis used for left-associative (postfix) operators. For example, logical negation - is defined as
- op(900. fy.--)
Similarly, op allows us to define operators if and then as non-associative prefix operator and right associative infix operator, respectively. We can handle expressions of the form if (symptom (SI) symptom (S2) then disease (D). However, these are really just Prolog facts, which can be pattern matched as normal. The following program is a simple ES shell in Prolog that uses backward chaining inference mechanism

Table 5.3 Expert System Shell

Operator Declaration		
:- op(975, fx,if)		
:- op(950, xfy, then).		
Consult('knowledge_base') *%load knowledge base in the working expert*		
Knowledge extraction module from Expert		
% Get the name of diseases([flu, cold, measles, chicken_pox,mumps]) from the expert		
expert_input	:-	writeln('Input the list of diseases as [d1, d2, d3...]'); readIn(D), asserta(diseases(D), 'knowledge_base'), !, input_sym(L), input_disease(C), asserta(rule(if L then C), 'knowledge_base').
input_sym(L)	:-	writeIn('Input the list of symptoms as [s1, s2, s3, ...]'), readIn(L).
input_diseases(L)	:-	writIn('Input the corresponding Disease'), readIn(L).

(Contd.)

Table 8.3 (Contd.)

Knowledge Base generated by knowledge extraction module		
diseases([measles, flu, cold, mumps, cough, chicken_pox]).		
rule(if [fever, cough, running_nose, conjunctivitis, rash)] then measles).		
rule(if [fever, headache, body_ache, sore_throat, cough, chills, running_nose, conjunctivitis] then flu).		
rule(if [headache, sneezing, chills, sore_throat, running_nose] then cold).		
rule(if [fever, swollen glands] then mumpas).		
rule(if {cough, sneezing, running_nose] then cough).		
rule(if [fever, chills, body_ache, rash] then chicken_pox).		
User Query Module		
get_symptom(X)	:-	user_response(['Do you have', X, '? Type(y/n)'], X).
% if the symptom has already been entered earlier then no need to enter again		
user_response(_, X)	:-	yes(X), !.
% if symptom is the new one then assert it in working memory appropriately		
user_response(Q, X)	:-	ask_query(Q, X, R), R = 'y'.
ask_query(Q, X, R)	:-	writeln(Q), readIn(R), store(X, R).
store(X, 'y')	:-	asserta(yes(x)).
store(X, 'n')	:-	asserta (no(X)).
Main ES Shell		
consultation	:-	writeln ('Welcome to diagnostic System'),
		writeln('Input your name), readIn(Name), hypothesis(D).
hypothesis(D)	:-	diseases(D), member(G, D), check(G), !,
		writeln(Name, 'probably has', G), clear_consult_facts.
hypothesis(D)	:-	writeln('Sorry, not able to diagnose'), clear_consult_facts.
check(G)	:-	rule(if L then G), check_symptom(L).
check_symptom([]).		
check_sumptom([G|Gs])	:-	get_symptom(G),!, check_symptom(G),!check_symptom(Gs).
check_symptom(G)	:-	no(G), fail.
check_symptom(G)	:-	yes(G)
clear_consult_facts	:-	retractall(yes(_)).
clear_consult_facts	:-	retractall(no(_)).

The main top level Prolog predicate is consultation; this predicate collects information about the user and calls the main predicate hypothesis, which uses backtracking to go through all the members of a list of diseases to see whether each can be proved true by backward chaining. Once it satisfies all the symptoms of a disease in hand, then it crosses cut and displays an appropriate message probably has that disease to the user. If it fails to satisfy a particular disease, it tries another disease till all the diseases are exhausted. If it fails for all diseases, then the system expresses regret by displaying Sorry, not able to diagnose. Finally, all facts yes(.) and no() of symptoms collected by user and stored in database are removed.

5.5.2 Problem-Independent Forward Chaining

As already discussed, forward chaining inference mechanism is a reasoning process that begins with known facts and proceeds forward to find a successful goal. Although Prolog uses backward chaining as a control strategy, it is possible to write a program in Prolog which uses forward chaining concept (Rowe, 1955). The facts from static and dynamic knowledge bases are used to test the rules through the process of unification. When a rule succeeds, its conclusion is added to the dynamic knowledge. For doing this, we code Prolog rules as facts with two arguments: the first argument is the head (left side) of the rule and the second argument is the body (list of sub goals) at the right side of the rule. Let us represent a rule as a fact using a predicate named rulelif

[subgoals) then G) and simple facts by a predicate named fact. Consider the rules and facts along with their corresponding new fact representations as given in Table 5.4.

Prolog Program	Corresponding Facts
a :- b, c.	rule(if [b, c] then a).
c :- b, e, f.	rule (if [b, e, f] then c).
b.	fact(b).
e.	fact(e).
f.	fact(f).

5.5.3 ES Shells and Tools

There are only a few Al methods, such as knowledge representation, inferences strategies, etc., essentially required in building expert systems, but there is a wide variation in domain knowledge .Thus, we can develop systems containing these useful methods without any domain-specific knowledge. Such systems are known as skeletal systems, shells, or simply AI tools. Most expert systems are developed using these specialized software tools. A shell is a complete development environment that may be used for building and maintaining knowledge-based applications. It provides knowledge engineer with a step-by-step methodology, which allows domain experts to be directly involved in structuring and encoding the knowledge. These shells have in-built inference mechanism (backward chaining, forward chaining, or both) and require knowledge to be entered according to a specified format while developing ES. The shells typically come with a number of other features, such as tools for constructing friendly user interfaces; manipulating lists, strings, and objects; interfacing with external programs and databases; and writing hypertext, etc. For a detailed coverage of ES shells, interested readers can refer to the 'ES Shells at Work? series by Schmuller (1991, 1992).

Using shells for building expert systems has certain significant advantages such as reduction in development time and costs. Moreover, the necessary knowledge regarding an application domain can be entered to perform a unique task using a shell thus generating an ES for that particular application in the domain. If the program is not very complicated and if an expert has had some training in the use of a shell, the expert can enter the knowledge himself or herself as well.
A large number of commercial shells are available for all types of systems such as PCs, workstations and large mainframe computers. These shells may range in complexity from simple, forward-chained, rule-based systems, which require a few days of training, to complex systems that can be used by highly trained knowledge engineers only. They also range from general-purpose shells to custom-tailored shells for a class of applications, such as real-time process control or financial planning.

5.5.4 MYCIN Expert System and Various Shells

MYCIN is one of the oldest expert systems. It was developed at Stanford in the 1970s. The design of most of the commercial expert systems and shells has been influenced by that of MYCIN. It was developed as an ES that could diagnose and recommend treatment for certain blood infem tions. MYCIN was developed for exploring the ways in which human experts make guesses on the basis of partial information. In MYCIN, the knowledge is represented as a set of if-then rules with certainty factors. The concept of certainty factors has already been mentioned earlier and will be discussed in detail in Chapter 9. The MYCIN rule may be written in English for the sake of clarity in the following manner (Shortliffe, 1976):

• IF the infection is primary-bacteremia AND the site of the culture is one of the sterile sites AND the suspected portal of entry is the gastrointestinal tract. THEN there is suggestive evidence (0.7) that infection is bacteroid.

Interpretation of the rule described above is that the conclusion bacteroid will be true with a certainty factor 0.7 given all the evidences are present. In case of evidence being uncertain, the

certainties of the bits of evidences will be combined with the certainty of the rule to determine the certainty of the conclusion. MYCIN was written in LISP language and its rules were formally represented as LISP expressions. It was a goal-directed system that used backward-chaining strategy.

MYCIN used various heuristics to control the search for a solution while proving some of the hypotheses. These were needed to make the reasoning efficient as well as to prevent the user from being asked too many unnecessary questions. One of the strategies was to begin the diagnosis by asking the user specific questions that were absolutely essential, then the system would focus on more specific blood disorders it uses the backward-chaining strategy to try and prove each one of the hypotheses. There were other strategies that were related to the way rules were invoked. According to the first strategy, if a possible rule to use was given, MYCIN first checked all the premises of the rule to determine if any of them were known to be false. If any of the premises were found to be false, there was not much point using the rule. The other strategies were related more to the certainty factors. MYCIN first looks at rules that have more certain conclusions and abandons a search once the certainties involved become lower than some threshold, such as 0.2 in this case.
A user conversation with MYCIN had three stages. In the first stage, initial data regarding the case was gathered to enable the system to come up with a very broad diagnosis. In the second stage, the questions that were asked to test specific hypotheses were more direct in nature. In the final stage ,a diagnosis was proposed. More questions were asked to determine an appropriate treatment and given the diagnosis and facts about the patient, a treatment was recommended. At any stage, the user could put up queries regarding why a particular question was asked or how a conclusion was reached. After a treatment was recommended, the user could also ask for alternative treatments if the first did not appear to be satisfactory.
EMYCIN was the first expert shell developed from MYCIN. A new ES called PUFF was developed using EMYCIN in the new domain of heart disorders. To train doctors, the system called NEOMYCIN was developed, which takes them through various sample cases, checks their diagnoses, determines whether their conclusions are right, and explains where they went wrong.

5.6 Blackboard Systems

A traditional way of combining diverse software modules is to connect them according to their data-flow requirements. Although the modules can appear many times whenever appropriate , the connections are predetermined and direct. This approach works well when the modules and the appropriate communications among modules are static. However, in dynamic environment, the specific modules and their ordering are subject to change and cannot be determined until specific data values are known at the time of execution, direct interaction induces inflexibility and hence becomes unusable (Corkill, 1991 and Carver & Lesser, 1994).
In such *situations,* indirect and anonymous communication approach among modules with the help of an intermediary, such as a blackboard data repository (Fig. 5.3), proves to be extremely useful. In this approach, all processing paths are possible, and a separate moderator mechanism dynamically selects a path among the possible paths. The information placed on the blackboard is public and is made available to all modules. Blackboard systems were first developed in the 1970s to solve complex, difficult, and ill-structured problems in a wide range of application areas. Black board architecture is a way of representing and controlling the knowledge bases; using independent groups of rules called knowledge sources (KSs) that communicate through at data control database called a blackboard.

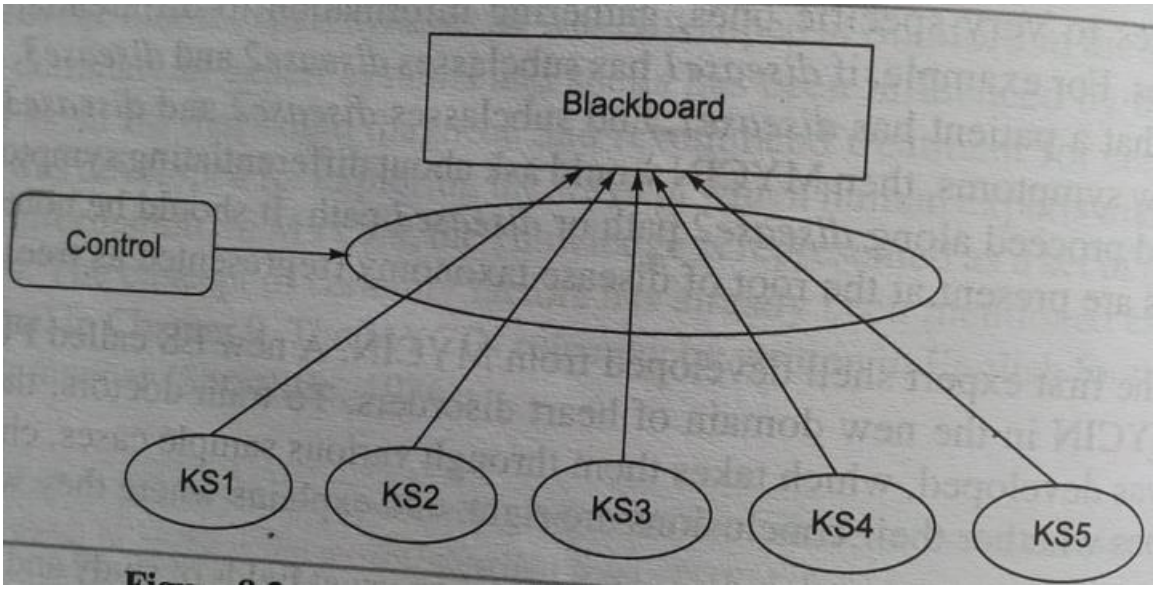

Figure 5.3 Indirect and Anonymous Communications

Because of the significance of blackboard system, it is important to look into their details and the basis of their utility. Figure 5.4 shows the main modules of a blackboard system. It consists of the following three main components:

i. Knowledge sources,
ii. Blackboard, and
iii. Control shell

The three components are discussed in the following subsections:

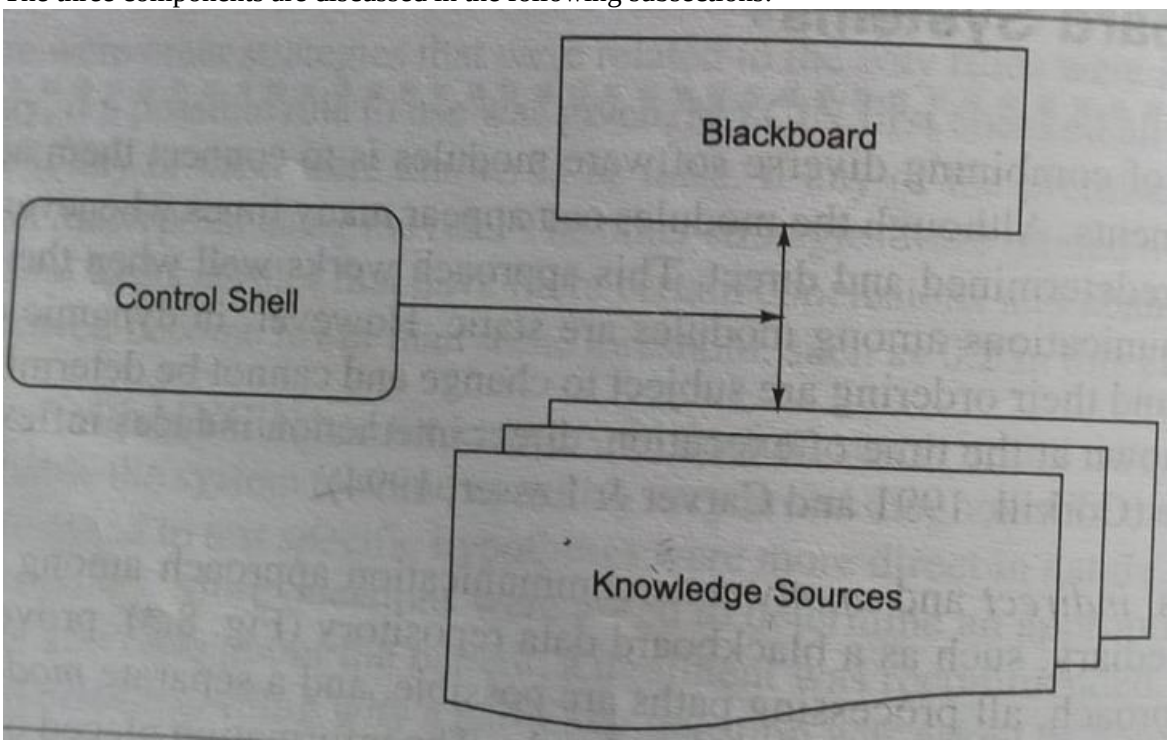

Figure 5.4 Blackboard System

5.6.1 Knowledge Sources

Blackboard systems use a functional modularization of expertise knowledge in the form of knowledge sources (KS). These are independent computational modules that contain the expert knowledge needed to solve a given problem. It is important to note that these modules may be widely diverse in their internal representation and computational techniques and do not interact directly with each other. A KS is regarded to be a specialist at solving certain aspects of the overall application and is separate and independent of all other KSs in the blackboard system. Once it obtains the information required by it on the blackboard, it can proceed further without any assistance from other KSs. It is possible to add additional KSs to the blackboard system and upgrade or even remove existing KSs. Each KS is aware of the conditions under which it can contribute toward solving a particular problem. This knowledge in problem-solving process is

known as a triggering condition.

5.6.2 Blackboard

Blackboard represents a global data repository and shared data structure available to all KSs. It contains several important constituents such as raw input data, partial solutions, alternatives and final solutions, control information, communication medium and buffer, and a KS trigger mechanism used in various phases in problem-solving. An important aspect of such a system is structuring of information on blackboard. In case of the number of contributions placed on the blackboard becoming large, we start facing problems in locating the required information. This issue can be resolved by subdividing the blackboard into regions, with each region corresponding to a particular kind of information. This approach is mostly used in blackboard systems. Related objects may be grouped with the help of different levels, planes, or multiple blackboards, while ordering metrics may be used in each region to organize information numerically, alphabetically, or by relevance. An important characteristic of the blackboard approach is its ability to integrate contributions dynamically for which it would be difficult for the KS writer to specify relationships in advance.

An advantage of the blackboard system is that the system can retain the results of problems that have been solved earlier, thus avoiding the task of re-computing them later. However, many contributions placed on the blackboard may never prove useful and maintaining the state of numerous, partially completed patterns are expensive. Therefore, an important characteristic of blackboard systems is that they allow a KS to efficiently inspect the blackboard to confirm if relevant information is present.

5.6.3 Control Component

The control component of a blackboard system helps in making runtime decisions regarding the course of problem solving and the expenditure of problem-solving resources. Control component is separate from the individual KSs. In a blackboard system, a separate control mechanism, sometimes called the control shell , directs the problem-solving process by allowing KS respond opportunistically to changes made to the blackboard. A blackboard system uses the pro ess of incremental reasoning, that is, the solution to the given problem is built one step at a tin In a classic blackboard-system control approach, the currently executing KS activation (KSA generates events as it makes changes on the blackboard. These events are ranked and maintaine until the executing KSA is completed. The control shell uses these events to trigger and activat KSs. Out of all the ranked KSAs, the best KSA is selected for execution. This KS-execution cycle continues indefinitely or until the problem is solved. In the basic blackboard-system control cycle only a single KSA is executed at any given point of time. This KS execution runs to completion or termination by the control shell before another KS execution begins. To further simplify the architecture in this basic control cycle, only the executing KS is allowed to make changes to the blackboard during the process of execution.

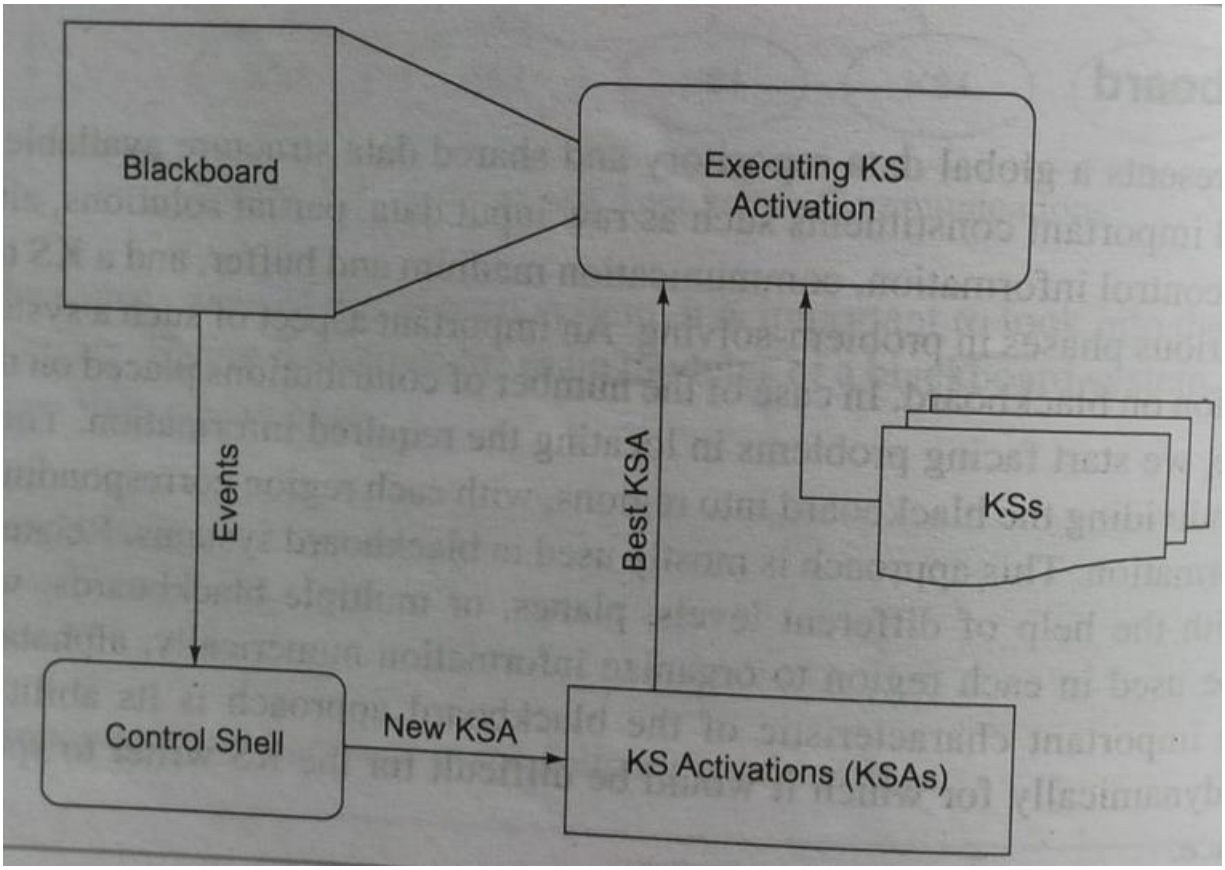

Figure 5.5 Blackboard-System Control Cycle

This requirement prevents the incorporation of complex blackboard locking or transaction mechanisms, which reduce the pace of blackboard operations. Figure 5.5 represents the control cycle used in blackboard system.

5.6.4 Knowledge Source Execution Method

It should be noted that KSs are triggered in response to specific types of blackboard events in a blackboard application. The control shell reports about the kind of events that each KS is concerned with. It maintains this triggering information and directly considers the KS for activation whenever an event occurs. The steps involved in most of the KS executions in a blackboard system are listed as follows:

• The control shell is notified of an event of interest to the KS.

This triggering information is used by the control shell to activate the KS in case such events occur.

. KS uses the triggering context to determine the ranges of attribute values and searches the blackboard to find blackboard objects possessing attributes within those ranges.

. The retrieved objects triggering context information are used by the KS to perform its computations.

The results of this computation are written onto the blackboard.

5.6.5 Issues in Blackboard Systems for Problem Solving

The core of the blackboard system approach is the structure of information on it. The blackboard representation should not be based on any specific set of KSs but its design should directly be based on the characteristics of the application and the goal of allowing any potential KSs to make contributions towards a solution. We can state by experience that choices made in the blackboard representation majorly effect the performance and complexity of the system. Information can be represented on blackboard in two ways:

• Specialized representation

• Fully general representation

In the specialized representation, KSs may only operate on a few classes of blackboard objects. In other words, some portions of the blackboard may be relevant to only a few KSs. Sharing data by limited number of KSs limits the extendibility and scalability of the system. Applying other KSS that information in the future cannot be achieved straightaway. However, in fully general

representations, all aspects of blackboard data are understood by all KSs. There is a trade-off between these two representations. Determining the proper balance between specialized and general representations is an important aspect of blackboard-application engineering.
Further, sharing of contributions among KSs involves the issue to which extent the details in the contributions to be placed on the blackboard. This issue may be handled in two different ways: In the first case, KSs may put the minimal amount of information on the blackboard, which is required to convey the results of their work to other KSs and all specific details remain unshared. In the second case, KSs may put a detailed representation of the relationship between inputs, partial solution, the knowledge procedures, etc., on the blackboard along with the basic KS results. However, placing on the blackboard, detailed information, which is never used, limits the efficiency of the blackboard. Any architectural mechanisms that increase the cost of placing information onto the blackboard must be carefully evaluated in the context of the corresponding decrease in the performance. Therefore, deciding on the right level of sharing is an important engineering decision.
It should be noted that blackboard systems do not allow direct interaction among modules as all communication is done via blackboard through control shell. Traditional blackboard system had only a single control thread that executed one KS activation at a time. This implies that all executions. Thus, it may encounter a problem of unbounded latency, which can be a significant On limitation in the environment of the collaborating software. However, this severe restriction interaction greatly simplifies the development of blackboard applications. If there exists a parallel and distributed blackboard system that is an extension of a single-threaded blackboard architecture, then it allows true concurrent KS executions. This raises another important interaction issue. If the KSs are to remain anonymous and indirect in their interaction, then all interactions must still occur via changes to the blackboard. Executing KSs must be able to detect and respond to altera tions made to the blackboard during their execution for supporting such indirect interaction.
We could also extend the KS model to enable direct communication among co-executing KSS activations. However, this is a major departure from the blackboard-system model, and it may lead to various problems because of the uncertainty regarding the KS activations that will be executing concurrently at any given point of time.
In blackboard representation, integration and representation are closely linked. The choices of representations not only affect the ability of KSs to use the results of others, but also affects the ways the KSs results are combined. In a blackboard application, integration of results involves three major activities: relationship management, attribute merging, and value propagation. The need for relationship management occurs when a KS execution wishes to create a new object on the blackboard and the semantics of the blackboard representation requires that the relationship between the new object and some existing objects be represented.

5.6.6 Blackboard System versus Rule-based System

A major point of difference between a blackboard system and a rule-based system lies in the size and scope of rules when compared to the size and complexity of KSs. The KSs are substantially larger and more complex than each isomorphic rule in an ES. As we have already discussed expert systems work by firing a rule in response to a query, while a blackboard system works by executing an entire KS in response to an event. Each KS can be arbitrarily complex and internally different from one another. In particular, a single KS in a blackboard system may be implemented as a complete rule-based system. The instances of KS are activated as computational resources. A KS activation occurs as a result of a specific triggering context and the KS knowledge. The main difference between a KS and KS activation is that there may exist a number of different events in an application that may trigger the same KS. KSs are static deposits of knowledge, whereas KS activations may be considered to be active computational resources that are created in response to each triggering context. Hence, KSs are not active in a blackboard system. The KS activations remain alive only until execution.

5.7 Truth Maintenance Systems

Truth maintenance system is a structure which helps in revising sets of beliefs and maintaining thr truth every time a new information contradicts information already present in the system.

5.7.1 Monotonic System and Logic

5.7.2 Non-monotonic System and Logic

In non-monotonic systems, truths that are present in the system can be retracted whenever contradictions arise. Hence, number of axioms can increase as well as decrease. The system is continually updated depending upon the changes in knowledge base. In non-monotonic systems, the addition of an assertion of a belief to a theory can violate conclusions which have already been made. In non-monotonic logic, if a formula is a theorem for a formal theory, then it need not be theorem for an augmented theory. Common sense reasoning is an example of non-monotonic rea. soning. Non-monotonic reasoning is based on inferences made by applying non-monotonic logic .

Both monotonic and non-monotonic reasoning can be best implemented using TMS described earlier. The term truth maintenance is synonymous with the term knowledge base maintenance and is defined as keeping track of the interrelations between assertions in a knowledge base. It may also be regarded as a knowledge-based management system that is activated each time the reasoning system or inference system generates a new truth value. TMSs are companion components to inference systems. Main job of TMS is to maintain consistency of the knowledge being used by the problem solvers. The inference engine solves domain problems based on its current belief set while TMS maintains the currently active belief set. Figure 5.6 shows the problem solver that uses interaction between inference engine, TMS, and knowledge base.

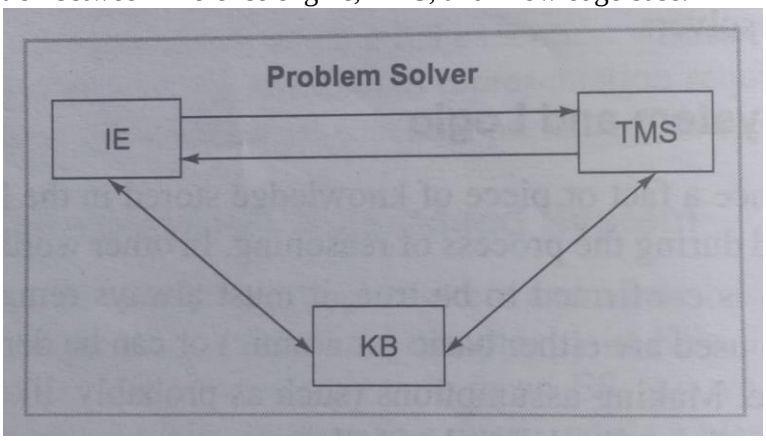

Figure 5.6 Interaction between Different Components I a Problem Solver

5.7.3 Monotonic TMS

The most practical applications of monotonic systems using TMS are qualitative simulation, fault diagnosis, and search applications. Furthermore, monotonic TMSs provides a solid foundation upon which other kinds of systems could be built. Algorithms for monotonic systems can usually be used in non-monotonic systems as well but the converse might not hold true.
(A monotonic TMS is a general facility for manipulating Boolean constraints on proposition symbols. The constraint has the form P Q where P and Q are proposition symbols that an outside observer can interpret as representations of the statements. Given a set of propositions regarding some problem and a set of constraints on those propositions, such as implication and a set of observations, a TMS can be used to ask questions about the consequences of the observations.

Functionality of a Monotonic TMS

A TMS stores a set of Boolean constraints, Boolean formulas (premises) and assigns truth values to literals that satisfy this stored set of constraints. These constraints are known as internal constraints and so do not appear explicitly as arguments in most of the interface functions. A monotonic TMS generally consists of the following generic interface functions:

• Add_constraint
• Follow from

• Interface Functions

Each of these generic functions is bridfly described as given below.

Add constraint: This interface functions adds a constraint to the internal constraint set. Once a constraint has been added, it can never be removed.

Follows from: This function takes two arguments, namely, a literal (L) and a premise set (2) and returns the values yes, no, or unknown. If yes is returned, then the TMS guarantees that L follows from the Land the internal constraints. If no is returned, then the TMS guarantees that L does not follow from . that is, there exists an interpretation satisfying both the internal constraints and in which L is false. If the TMS is unable to determine, then it returns unknown.

Interface Functions. The interface functions compute justifications. If the TMS can determine that L follows from the internal constraints and a premise set E, then one can ask the TMS to justify this fact by producing a proof of L. There are two interface functions used to generate such proofs namely justifying literals and justifying constraints.

let us consider an example to explain the concepts discussed above. Suppose, we have a premise set S= {P, W} and an internal constraint set {P – Q. (P AW) – R (Q A R) —$}. Most truth maintenance systems are able to derive S from these constraints and the premise set E. TMS should provide the justifications of deriving S from constraints and premises. Therefore, for any given set of internal constraints and premise set , if a formula S can be derived from these, then the justification functions generate a justification tree for S (Fig. 5.7). The justification is given in Table 5.5.

Table 5.5 Justification Functions

Justifying literals	Derived literals	Justifying constraints
{P, W}	R	$(P \wedge W) \rightarrow R$
{P}	Q	$P \rightarrow Q$
{Q,R}	S	$(Q \wedge R) \rightarrow S$

Alternatively, justification functions can produce a tree with literal S at the root as shown in Fig 5.7. at each node of the tree, the function justifying_literals can be use to get children nodes until one reaches members of the premise set. The justifications are required to be non-circular,i.e, if Q appears in the justification tree rooted at P, then P must not appear in the justification tree rooted at Q.

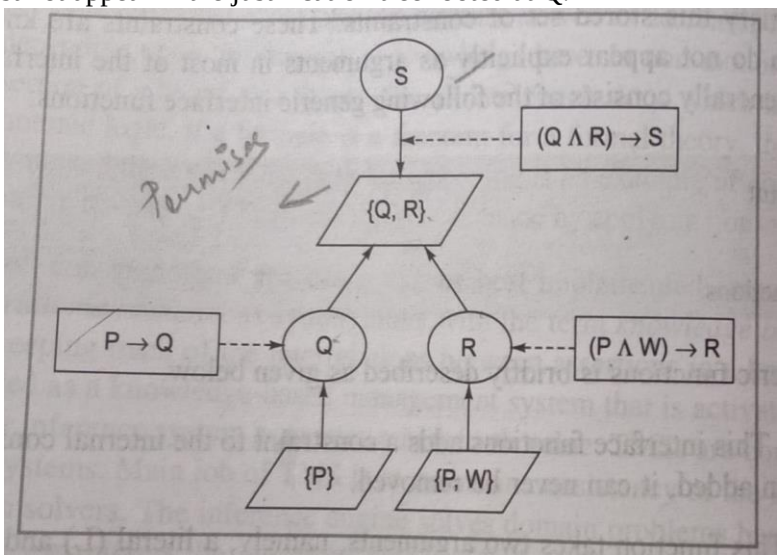

Figure 5.7 Justification tree

Contradiction Handling

The TMS interface function enables contradictory information to be given to the TMS. Most TMSs possess a technique of informing the user that a given premise set is inconsistent with the internal constraints. This can be done by adding a special proposition symbol called contradic tion. If the TMS is able to determine that a given premise set Σ contradicts the internal justifying_literals(contradiction, Σ) and justifying_constraints(contradiction, Σ) return a set of constraints, then the function follows_from(contradiction, Σ) returns yes. In this case, literals and a set of constraints, respectively, that underlie the contradiction. The justification functions can be used to construct a justification tree whose leaves are literals in Σ.

5.7.4 Non-monotonic TMS

The basic operation of a TMS is to attach a justification to a fact. A fact can be linked with any component of program knowledge which is to be connected with other components of program information. Typically, a fact might be connected with each assertion and rule in a database, or might be attached, with different meanings, to various subsystem structures. On the basis of the justifications attached to the facts, TMS decides which beliefs in the truth of facts are suppoted by the recorded justifications.

Truth Maintenance Table

TMS is regarded as a knowledge-based management system that is activated every time the reasoning system or inference system generates a new truth value. TMS interacts with inference systems and after each inference or updation, information is exchanged between two components. TMS maintains a consistent set of beliefs for the inference engine. The inference engine solves domain problems based on its current belief set, while TMS maintains a record of thecurrently active belief set.

TMS basically operates with two kinds of objects, namely, propositions, which declares different beliefs, and justifications, which is related to individual proposition for backing up the belief or disbelief expressed by the proposition. For every TMS, there are two kinds of justifications required, namely, support list and conditional proof. These have been described as given below.

Support List

(Support list (SL) is defined as SL(IN-node)(OUT-node), where IN-node represents a list of all IN-nodes (propositions) that support the considered node as true. Here, IN means that the belief is true. OUT-node is a list of all OUT-nodes that do not support the considered node as true. OUT means that belief is not true for considered node. For example, suppose, we are given the premises as shown in Table 5.9 which may be either beliefs or derived alongwith the justification of all the facts.

Table 5.9 Justification of Facts

Node number	Facts/assertions	Justification(Justified belief)
1	It is sunny	SL(3)(2,4)
2	It rains	SL()()
3	It is warm	SL(1)(2)
4	It is night time	SL()(1)

In this case, an empty list indicates that its justification does not depend on the current belief or disbelief. Node 1 assumes that it is sunny, provided that it is warm and it does not rain, and it is not night Time. Node 3 assumes that it is warm given that it is sunny and it does not rain. Nodes 1 and 3 are valid for the facts whose numbers are explicitly stated in (IN-nodes) and (OUT-nodes) list. Node 4 does not depend on the list in its (IN-node) part. Node 2 has both empty lists indicating that its justification does not depend on the current beliefs or disbeliefs.

Conditional Proof

A belief may be justified on the basis of several other beliefs, that is, a belief may be justified by the conditional proof on one belief relative to other beliefs or by the lack of belief in some fact.) The latter form of justification allows consistent representation and maintenance of hypothetical assumptions. When new justifications change previously existing beliefs, then the truth maintenance processing is required. In such cases, the status of all beliefs that depends on the changed beliefs have to be determined again. From the justifications used in this judgement of beliefs, a number of dependencies between beliefs are determined, such as the set of beliefs depending on each particular belief or the beliefs upon which a particular belief depends. Several useful processes are supported by the above functions and representations.

5.7.5 Applications of TMS to Search

TMS is also useful in controlling searches. It more specifically controls those searches that are

Generally needed in constraint satisfaction problems. Constraint satisfaction and search problems fall into the class of problems for which a direct algorithmic solution does not exist. The solution of these problems requires the examination of state spaces. It is not possible for a problem solver (or inference engine) alone to organize and maintain the state space consistently on its own. For this purpose, a TMS is required; it organizes data within a data abstraction called a context, which corresponds to a single problem state and contains currently believed data. The TMS provides believed data retrieval, belief revision, contradiction handling, and non-monotonicity handling: these features help a problem solver to examine state spaces. In order to use TMS for solving arbitrary constraint satisfaction problems, their constraints need to be specified as a set of boolean constraints. There are two simple ways of translating the constraints of a constraint satisfaction problem into constraints on these proposition symbols. A simple program that uses backtracking can be written to search the assignments of values to the variables of the problem where a partial assignment is encoded as a TMS premise sets.

5.8 Applications of Expert Systems

Expert systems have been widely developed and used to solve problems in different types of domains. The appropriate problem-solving technique depends generally on the type of problem and the domain. Applications may be categorized into the following major classes:

Diagnosis The expert systems belonging to this class perform the task of inferring malfunctioning of system from observations. Such expert systems use situation descriptions, behaviour characteristics, or knowledge about component design to determine the probable cause of system malfunction. These systems may also be used for diagnosis of faulty modules in large signal switching networks and for finding faults in computer hardware system. Diagnosis can refer to inferring a possible disease from a given set of symptoms in the field of medicine.

Planning and scheduling The expert systems of this class help in designing actions and plans before actually solving a given problem. They analyze a set of one or more potentially complex and interacting goals in order to determine a set of actions that are needed to achieve these goals. They provide a detailed temporal ordering of these actions in the form of plans, taking into account personnel and other constraints. Examples of this class may be

• airline scheduling of flights,
• manufacturing job-shop scheduling,
• creating plan for applying series of chemical reactions,
• manufacturing process planning,
• creating air strike plan projected over several days, etc.

Design and manufacturing This is one of the most important areas for ES applications. Here, a solution to a problem is configured by a given set of objects under a set of constraints. Configuration applications were pioneered by computer companies to facilitate the manufacturing of semi-custom minicomputers. Examples of the applications of this class are

• gene cloning,
integrated circuits layouts designing,
• creation of complex organic molecules,
• modular home building,
• manufacturing, etc.

Prediction The expert systems of this class perform the task of inferring the likely consequences of a given situation. For example,

• weather prediction for rains, storms, and tornado
• prediction of crops
• share market, etc.

Interpretation The expert systems of this class perform the task of interpreting and inferring situation description data of any domain such as geological data, census data, medical data, etc. For example,

• interpreting data from ICU test equipment for further investigation or monitoring,
• interpreting intelligent sensors reports to perform situation assessment,
• interpreting radar images to classify ships, etc.

Financial decision making The financial services industry has been a prominent user of F techniques. Such systems assist insurance companies to assess the risk presented by the custom and to determine a price for the insurance. Bankers use expert systems for assistance in deteri ing whether to grant loans to businesses and individuals. A typical applications are

• foreign exchange trading

formulating financial strategies

giving advices, etc.

Process monitoring and control Expert systems belonging to this class analyze real-time dan from physical devices with the goal by comparing observations to expected outcomes, predictir! trends, and controlling for both optimality and failure correction. Examples of real-time system that actively monitor processes can be found in

• steel-making and oil-refining industries,

• nuclear reactor (to detect accident conditions),

• patient monitoring system in hospitals (for controlling the treatment of the patient in ICUS, monitoring components to track the behaviour of the system, etc.), and so on.

Instruction An ES can offer tutorials and instructions to students by incorporating a students behaviour and learning capability models and can also evaluate a student's acquired skills

Debugging The systems of this class prescribe remedies for malfunctioning devices. They may suggest the type of maintenance needed to correct faulty equipments,

• help in debugging large computer programs to improve the performance, etc.

Knowledge Publishing This is a relatively new but a potentially important area now-a-day. The primary function of the ES in this field is to deliver knowledge to user that is in context of the user's problem. For example, the expert systems of this class may act as

advisors, which counsel a user on appropriate grammatical usage in a text,

a tax advisor that accompanies a tax preparation program, which advises the user regarding tax strategy, tactics, and individual tax policy.

Other Miscellaneous Applications There exist a number of other applications where experim systems can be built. For example,

. Fraud detection, object identification and information retrieval system.

• Handling certain tedious and dangerous situations such as coal mining.

• Judicial systems to solve new cases intelligently using past case history, advising on legal issues, etc.

• In defence, performing situation assessment from intelligence reports, analyzing radar signals and images, predicting when and where armed conflict will occur next, simulation of war, etc.

• Household activities such as giving advice on cooking, shopping, and so on are also potential applications.

5.9 List of Shells and Tools

Following are the list of shells and tools that understands and provides knowledge of expert systems and their applications:

ACQUIRE It is primarily a knowledge-acquisition system and an ES shell, which provides a complete development environment for the building and maintenance of knowledge-based applications. ACQUIRE-SDK is a software development kit, which provides callable libraries for systems such MS-DOS, and SCO Unix, Windows, Windows NT, Windows 95 and Win 32. It is also used in custom controls for Visual Basic.

Arity Expert Development Package is an ES that was developed by Arity Corporation, MA to perform the task of integrating rule-based and frame-based representations of knowledge with various types of certainty factors.?

ART It is called Automated reasoning tool that is an ES shell, which is based on LISP. It supports rule-based reasoning, hypothetical reasoning, and case-based reasoning.

CLIPS It is used to represent C Language Integrated Production System. It is a public domain software tool that is used for building expert systems. It provides a cohesive tool for handling a wide variety of knowledge and provides support to three different programming disciplines, namely, rule-based, object-oriented, and procedural. Rule-based programming allows knowledge

to be represented as heuristics, or 'thumb rules', which specify a set of actions to be performed for a given situation. Object-oriented programming allows complex systems to be modeled as modular components (which can be easily reused to model other systems or to create new components). Procedural programming capabilities provided by CLIPS are similar to those found in languages such as C, Java, Ada, and LISP. CLIPS is probably the most widely used ES tool because it is fa efficient, and free.
on a PC. It is available from most of the major vendors under license. It is basically a toolkit
FLEX Flex is a hybrid ES which is implemented in Prolog. It was originally developed by LP: of different hardware platforms and offers frames, procedures, and
rules integrated within a logic programming environment. It supports interleaved forward as backward chaining, multiple inheritance, and procedural attachment, and also possesses a automatic question and answer system. Rules, frames, and questions are described in an easi understandable knowledge specification language (KSL) which enables the development of eas to-read and easy-to-maintain knowledge bases. FLEX has been used in numerous commercia expert systems.
BOOT
one on
Gensym's G2 It offers a graphical, object-oriented environment for the creation of intelligen applications that are able to monitor, diagnose, and control dynamic events in various environments. G2 features a structured natural language that is utilized for creating rules, models, and procedures. All Gensym application products and end-user applications, also include the G2 diagnostic assistant (GDA)-a visual programming environment, which is used for creating intelligent process management applications.
GURU It is an ES development environment and offers a wide variety of information proces ing tools combined with knowledge-based capabilities such as forward chaining, back chaining, mixed chaining, multi-value variables, and fuzzy reasoning.
HUGIN SYSTEM It is a software package for the construction of model-based expert syne in domains characterized by inherent uncertainty. The Hugin System contains an easy-tone probability-based deduction system, applicable to complex networks with cause-effect came relations subject to uncertainty.
Knowledge Craft It is an expert-system development toolkit for scheduling, design, ani configuration applications.
K-Vision It is a knowledge acquisition and visualization tool. It runs on Windows, DOS, n. UNIX workstations.
MailBot It is a personal e-mail agent that reads an e-mail message on standard input and creates an e-mail reply to be sent to the sender of the original message. It provides filtering, forwarding notification, and automatic question-answering capabilities.