A Real-time Expert System Shell For Process Control

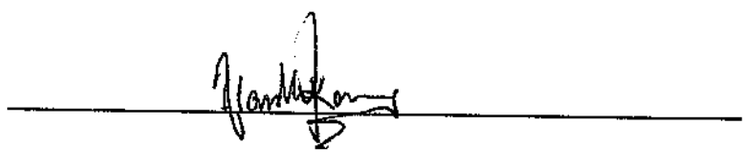
Alan Montzy Kang

A dissertation submitted to the Faculty of Engineering, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science in Engineering

Johannesburg 1990

Declaration

I declare that this dissertation is my own, unaided work. It is being submitted for the Degree of Master of Science in Engineering in the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination in any other university.



8th day of January 199 1

Abstract

A multi-layered expert system shell that specifically addresses real-time issues is designed and implemented. The architecture of this expert system shell supports the concepts of parallelism, concurrent computation and competitive reasoning in that it allows several alternatives to be explored simultaneously. An inference engine driven by a hybrid of forward and backward chanining methods is used to achieve real-time response, and certainty factors are used for uncertainty management. Real-time responsiveness is improved by allowing the coexistence of procedural and declarative knowledge within the same system.

A test bed that was set up in order to investigate the performance of the implemented shell is described. It was found in the performance analysis that the proposed system meets the real-time requirements as specified in this research.

To my parents

Acknowledgements

I want to give special thanks to my supervisor, Professor I.M.Macleod, who has provided me with invaluable assistance throughout the course of this research.

Special thanks also to Mr. V.Lun, who has willingly provided much assistance, and kindly provided his executive system shell for the use of this research.

To Carolyn I want to express my heart-felt gratitude for her generous support and encouragement in the latter part of this work.

Contents

Chapter 1

Introduction

1.1 Background

The use of computers to solve problems is a relatively young science. Ever since the invention of the digital computer there have been increasing expectations that computers will one day exhibit human-like intelligence, or even surpass it. As the use of computer technology becomes more widespread, great strides are being made to bring this expectation closer to reality. Efforts in this endeavour are particularly vigorous in the field of *artificial intelligence* (AI).

Winston [39] has defined artificial intelligence to be that field concerned with the *computations* that connect situations to *complex, human-like* actions. Research interests in this field range from robotics, expert systems, natural language processing and computer vision to general cognitive sciences. But from the commercial perspective, expert systems have long been considered the most immediately practical application, and are central to most activities.

Early artificial intelligence research has revealed that intelligence requires knowledge. The conceptual break-through lies in the realization that the problem-solving power of a computer (program) comes not just from the formalisms and inference mechanisms it employs, but mainly from the knowledge it possesses [17]. Expert systems are based on this concept.

Expert systems are a subset of knowledge-based systems; they represent a new approach to computer programming. Instead of the algorithmic approach, where computers are programmed to follow step-by-step procedures, expert systems are

programs that instruct the computer to use the facts about the problem, plus the domain knowledge encoded in a knowledge base, and some general problem-solving strategies to find and apply a specific solution [13] — the heuristic approach.

In simple terms, the intelligence of an expert system comes from the domain knowledge embedded in its knowledge base, and its ability to apply this knowledge to problem-solving. Conversely, an expert system is only as good as the knowledge it contains. In the quest to endow computers with human-like intelligence, expert systems are built on a base of knowledge patterned after a human being. In particular, they are fashioned after human *experts*.

Expert systems have found commercial success in wide-ranging disciplines such as medicine, banking, and engineering. Notable examples are MYCIN [33], a medical diagnosis program developed in the 1970s, which has demonstrated an ability to diagnose infectious diseases with nearly the accuracy of a human expert; and PROSPECTOR [5], which has been used successfully to locate deposits of several minerals.

1.2 Expert systems in computer control

In a process control system, information about the state of the outside world is received via sensors. The process control computer exerts its control by signalling an alarm condition, or by activating actuators, for example, by closing a switch, opening a valve. The control algorithms are embedded in the computer.

Three classes of reasoning and computation in analyzing software required for computer control systems are identified in [18]. These are heuristic reasoning, qualitative analysis and quantitative computation. Quantitative computation represents the numerical aspect of problem-solving, and is familiar to all engineers. In addition to this, expert systems provide the heuristic approach in that, problem-solving is based on knowledge obtained empirically. Recent developments in knowledge based systems have begun to look for avenues of incorporating all three aspects of problem-solving.

The general need for the three classes of computation has made process control an increasingly important application area for expert systems. They are used for smart control systems, alarm interpretation, operator decision support, process diagnostics,

instrument and control system diagnostics, and modelling and simulation.

Like their counterparts in other disciplines, most of these expert systems work almost exclusively with *static data*, in a non-real-time fashion, where the expert system would solicit data from the user when it needs them. They tend to operate in the form of advisers adjunct to humans. For example, the knowledge-based simulation system presented in [29], the use of an expert system for qualitative modelling in [35], and the process diagnosis expert system in [11].

Increasingly commonplace are expert systems directly connected to test and measurement systems which gather data in an on-line fashion. Such systems have the potential to automate tasks such as machine health monitoring, system performance monitoring, alarm monitoring, data reduction, processing and to participate in *real-time* decision-making.

1.3 Expert systems for real-time control

Real-time expert systems are being applied to industrial process control, air traffic control, autonomous vehicle control and various military applications. The technology provides a means of extending automation, either as an intelligent co-operative assistant, or by replacing the human operator altogether [3].

In real-time applications, the expert system performs tasks in a dynamic environment under time stress. The operation of expert systems in real-time control environment involves reading from sensors, performing a stream of tasks based on these readings, and controlling actuators in the case where the expert system is used to influence the plant operations directly.

An important characteristic of process control systems is the short response time required. Typically the required response time ranges from one millisecond to one second or more. Furthermore, information flow is dynamic and carries a *validity time limit*, that is, data values are valid only for a limited period — until new values are created to update/replace them. In order to be useful, the expert system must process information before the validity of data expires. An expert system for real-time control must guarantee this response under both peak load and normal load. In short, the response of such a system must be *predictable*, *correct* and *timely*.

Two approaches are possible to meet this requirement. One is to construct

computers capable of running expert systems at an adequate speed. Even though developments in computer technology may produce machines which run at faster and faster speeds, the scope of problems that can be handled by this 'brute force' approach is nonetheless, intrinsically limited by hardware.

Alternatively, we could attack the problem from the software perspective. Software issues deal mainly with the way the expert system processes information. The problem is one of architectural and algorithmic design.

1.4 Overview of the dissertation

The aim of this investigation is to realize a means of effectively applying existing expert system techniques to real-time problems. We focus on software solutions to the problem. In this regard, the aim is on developing an *expert system shell* that meets real-time constraints, where an expert system shell contains the reasoning mechanisms of the expert system.

With this in mind, the first area to look at is the implication of real-time. Section 2 of this dissertation examines this issue and defines the way we model real-time problems.

A survey of existing expert system techniques is reported in Section 3. Recognizing the fundamental need to re-define some of the conventional rules to designing an expert system, we focus on the pros and cons of different techniques in an effort to come up with a suitable design for the purpose of this research. The design of AKShell, the experimental expert system shell developed in this research, is discussed in Section 4.

Implementation issues such as the programming language and operating system used, and the design of various algorithms, are covered in Section 5. A discussion on the performance of AKShell is then given in Section 6. Finally, Section 7 concludes this dissertation and gives recommendations for further research.

Chapter 2

Real-time issues

2.1 Real-time response

A clear distinction between real-time systems and traditional systems is that, real-time systems must act in response to events in the outside world. Most conventional computer programs are non-real-time. These programs run naturally to completion as quickly as the computer allows. The computer is in charge and receives no interruptions from the outside world. A real-time system on the other hand, is slave to the environment [2], and its acceptable response time is dictated by the environment.

In dealing with real-time problems, timing of the system response becomes an important issue. A system response time is the time within which the system detects an event and responds with an action. This is usually a direct function of the frequency of interrupts from an external device, or the frequency with which the status of some device must be polled [42]. Thus time is both a resource and a constraint [32], and it needs to be addressed as a critical parameter in designing real-time systems.

An enemy missile detecting system is an example of a critical real-time system. Timing is critical in missile detection, and the system must detect and identify the enemy missile before it strikes the target, in order that appropriate counter-measures may be executed.

An automatic bank teller-machine is an on-line rather than a 'hard' real-time system. There is no strict timing constraint. Money or statements issued now or

after some delay are equally good. The teller-machine is in charge. If the customer becomes impatient, there is always an option of coming back later, or going to another teller-machine. But an enemy missile detected too late may result in catastrophe — the approaching missile is in charge.

We see that in a non-real-time application, the response time can be of arbitrary length, and the only real damage stemming from a delayed response is the patience of the user. In a process control system, however, a delayed response may result in disaster. For instance, the fate of an entire city may be at stake.

The most important property of a real-time system then is the *timeliness* of its response. A real-time computer system may be defined as one which is able to receive raw data from the environment, and to process the information sufficiently quickly so that timely action can be taken. [42]. In other words, it must be able to respond quickly enough to any pertinent change in the environment in order to successfully track or modify the behaviour of the environment.

As real-time processes are continuous, the measured values of real-time data change all the time and at different rates. The validity of real-time data therefore, must be time-limited, that is, any observation of the state of the environment is current only for an application-dependent time interval [20].

Real-time response therefore is the ability of the system to produce results while they are still applicable; it is the ability of the system to synchronize with real events as they happen [34].

2.2 Modelling the real-time environment

The main difficulty in designing a real-time system lies in coping with the dynamism of the real-time environment. Often, it is difficult to predict when and where the system needs to adapt (to changes). Facts and data must be continually added, retracted, updated, and their consistency enforced. In designing an expert system, complex issues like currency-time limit of information [20] and non-monotonic reasoning [30] will need to be addressed. Computational overhead can escalate to an unmanageable proportion. Therefore, before we begin to design the system, it is necessary that we define a suitable way in which to represent the environment.

We describe the environment by its *states*. The state of the environment is a

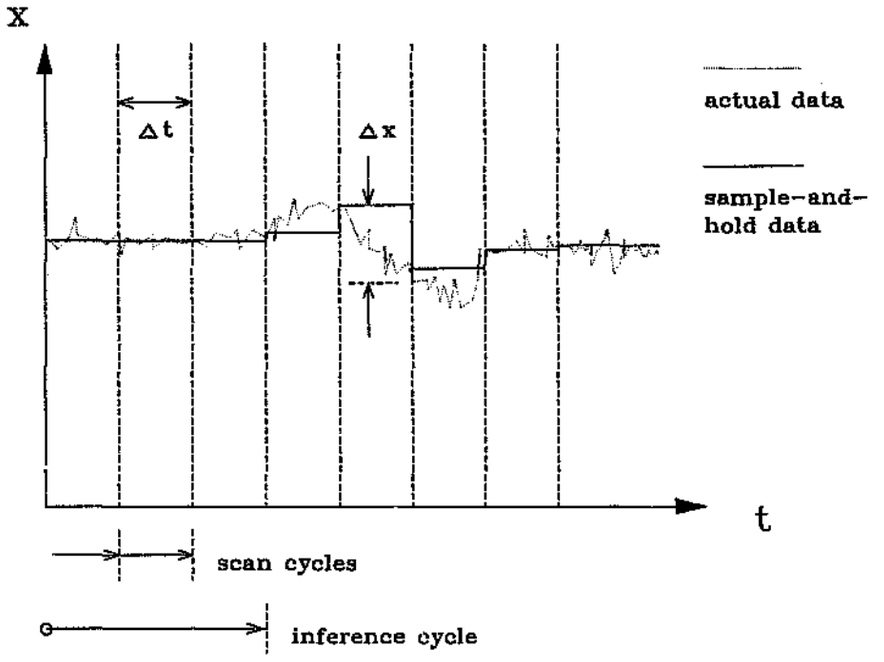


Figure 2.1: Sample-and-hold variable

snapshot of all its variables. Thus, at any given point in time, the state is represented by the on-line parameters, i.e., the actual measurements taken at that instant. Representing the environment by its states is therefore, based on sampling of the variables. The rate at which each state changes determines the rate at which the system must sample the variable.

If we assume that the environment will not exhibit significant change within a certain time interval, Δt, then by *sample-and-hold* scanning or polling, we 'freeze' the environment for Δt and assume that all information has a life-span (validity) of at least Δt. This is illustrated in Figure 2.1.

Δt represents the worst-case timing constraint where, at the end of this period, the system must produce results. This is a *cyclical* system, where an equi-distant time signal initiates all system activities at pre-defined points in time. The time period between the sequences of actions is called a *scan-cycle* [14].

The determination of Δt is application dependent and reflects the shortest time during which the most volatile variable will show *significant* change. That is, a variable x will not change by more than Δx within Δt, where Δx is a *tolerance threshold* defined for x. Thus, for the discrete function $x(t)$, where $x(t_i)$ are the

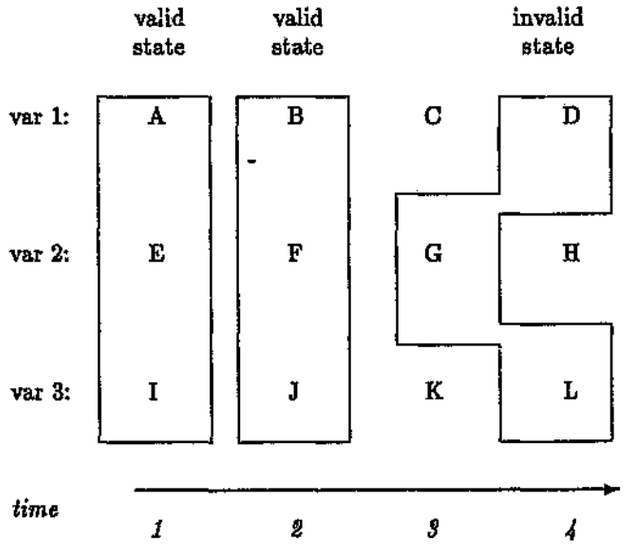


Figure 2.2: Invalid states caused by sampling variable values at different moments

sampled values at times t_i, for $i = 0$ to ∞,

$$x(t_{i+1}) - x(t_i) \leq \Delta x$$

$$t_{i+1} - t_i = \Delta t$$

The tolerance threshold is determined by the reasoning process behind the expert system. For example, the variable x may not change by more than Δx before Rule n is triggered by the expert system. The tolerance threshold is thus a reflection of the resolution, or accuracy with which the expert system rules are applied. This resolution and the maximum rate of change of the environment together determine the length of the scan-cycle.

To ensure that the decisions are made on the basis of states that actually occurred, it is important that all system variables are frozen at the same moment [34]. Otherwise, if the variable values were sampled at different points in time, their combination may result in an invalid state. An example invalid state is illustrated in Figure 2.2 [1].

An approximate model of the real-world can thus be built by asserting that snap-shots (sample-and-hold data) of the world are adequate descriptions. By this

[1] This example is taken from [34].

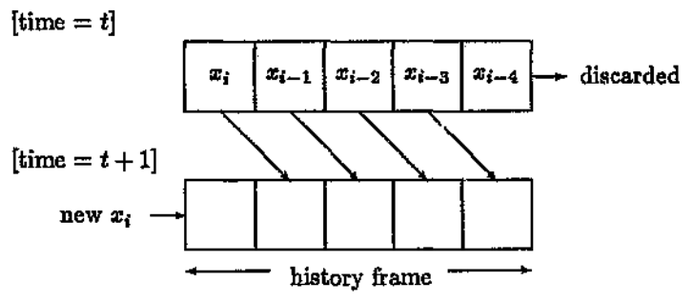


Figure 2.3: Updating data history

assumption, the task of data consistency and validity checking is simplified because old data are automatically updated at every scan-cycle. The change of states becomes a predictable event — it happens at most at every scan-cycle. By this token, we can assume only monotonic reasoning strategies in the expert system design.

2.3 History of events

The notion of states allows us to reason in the temporal sense, for example, "Did event X happen at time T?", "When did event X happen?", or, "What happened at time T?". We need the temporal reference in order to establish trends in the environment states, or make predictions. These temporal references can be represented by a record of past events, that is, a history of data values.

As is the case with current events, the validity of past events is also time-limited, except that the record of history is not updated by re-sampling the variables. Instead, at every scan-cycle, the newly-sampled data 'pushes' the old data back one time frame. This is shown in Figure 2.3, where we keep four past records of data x and its most recent value, x_i, in a circular list, or *history frame*.

The length of this frame is fixed and is determined by the application. When a new x is created, the new value replaces the last x_i. The old records are then moved one place to the right in the history frame. Since we want to keep only four past records, the last record (old x_{i-4}) is discarded when this update takes place.

With reference to the history of x, we can then refer to its past value n scan-cycles ago, which is kept as x_{i-n}, at time $t = i$.

2.4 Managing limited resources

In a real-time control environment, the control object (the physical process) dictates the acceptable response time of the system. This in turn influences the way the system allocates its resources.

Two major scarce resources are identified as,

- time,
- computing hardware.

Within these constraints, real-time response must be guaranteed for both normal loads and for peak loads, i.e., the system must perform satisfactorily in worst-case conditions.

We have defined in our model of the real-world that the scan-cycles reflect the duration of the shortest significant state. This represents the worst timing constraint on the system performance. If the system can perform satisfactorily within a scan-cycle, it would have met the peak load condition. Given fixed hardware and fixed time allowance, we want to devise a method that guarantees *reliable* performance. The problem of resource allocation is viewed as balancing the processor load within a fixed time frame. In this section, we focus on the way the reasoning process takes place in the expert system.

2.4.1 Inference-cycle

A traditional expert systems pursues a single line of reasoning when it is executed. The system stops when a decision is reached, and each run of the expert system consists of a single *inference-cycle.*

By definition, a real-time expert system never stops, because the environment does not stop. The reasoning process is continuously synchronized with the events of the real-world.

In the state-based interpretation of the real-world, the reasoning process consists of sequences of actions synchronized to the changes in the state of the environment. The sequences of actions are state-dependent and a problem-solving strategy is active while the state of the environment remains applicable. Provided that the state remains constant, a single line of reasoning, or inference cycle can be maintained and

extended; and a decision previously made can be further substantiated or modified for improvement. A change in the state of environment then prompts the system to formulate new plans, or modify the existing strategy.

Each change in state is called an *event*. It is thus the occurrence of events rather than the equi-distant scan-cycles that initiates new inference-cycles.

This sort of behaviour suggests an *opportunistic scheduler* which is intelligent enough to exploit the situation and allocate more time to a line of reasoning whenever it is possible. Thus, if a given event remains valid over several scan-cycles, the reasoning process initiated by this event will be allowed to continue until the event changes, rather than starting a new reasoning process at every scan-cycle on the same event.

An example inference-cycle that extends over several scan-cycles is shown in Figure 2.1. In this example, the system sees the state of the environment remaining unchanged over three t cycles, and allows the inference-cycle to last for this period.

2.4.2 Parallel processing

> "Parallelism is fundamental to the design and implementation of expert systems in many domains." [15]

One needs only to look at the finest of all computing devices, the human brain, which spends most of its time performing a large array of concurrent tasks, to realize that there is both an opportunity and a need for parallel execution of tasks in real-time problems. Parallel processing is a necessary aspect of real-time expert system designs [32] [37].

The most important advantage of parallel processing stems from its inherent provisions for *error reduction*. We reduce the risk of error by simultaneously considering more than one alternative. This can be applied in an expert system by forcing it to follow several lines of reasoning concurrently.

Essentially, an expert system receives a set of input data and computes toward a decision. There are two ways the expert system can arrive at the answer: One is to accord all computing resources to one possible solution at a time — the serial approach; the other is to spawn many possible solution paths simultaneously, and

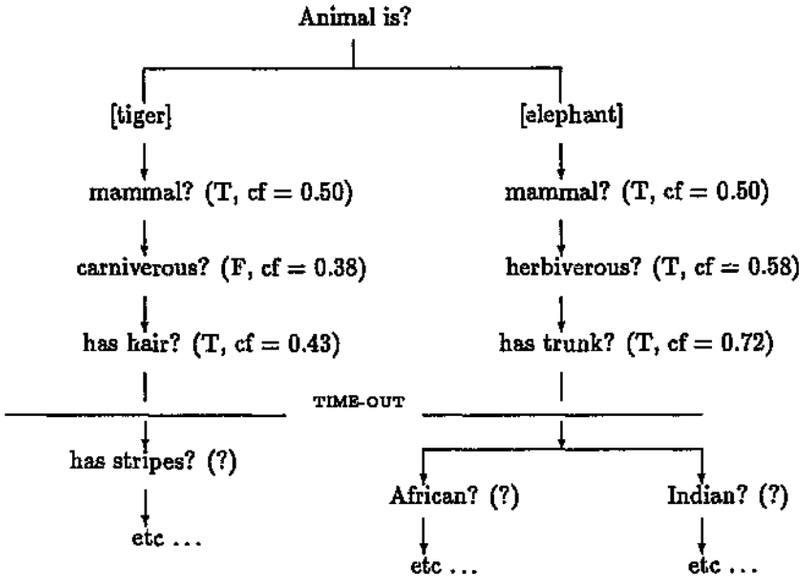


Figure 2.4: Parallel approach to the animal problem

share the system resources among these routes — the parallel approach.

A parallel approach considers all the likely-to-succeed solutions concurrently. Even if none of these were completely analyzed by the end of the inference-cycle, a reasonable conclusion can still be drawn from the evidence so far. By appropriate assessment of the quality of the partial solutions, we can select a decision from the partial solutions; or give a firm statement that no satisfactory solution can be given.

By the serial approach, the expert system must sequentially analyze each draft solution before finally arriving at a conclusion. On time-out, the serial system has evaluated (fully or partially) only one possible solution, in the absence of competition, this solution automatically becomes the final answer. In this light, the serial approach is inherently unreliable against real-time demands.

The superiority of parallel design becomes apparent if we consider the example of identifying an animal, given its properties. A parallel approach is presented in Figure 2.4:

Two possible animals are selected: tiger and elephant. At each node (e.g., [mammal?]), an accumulative measure of faith, the certainty factor (CF) is calculated. At time-out, the inference process has reached nodes [has hair?] and [has trunk?] under

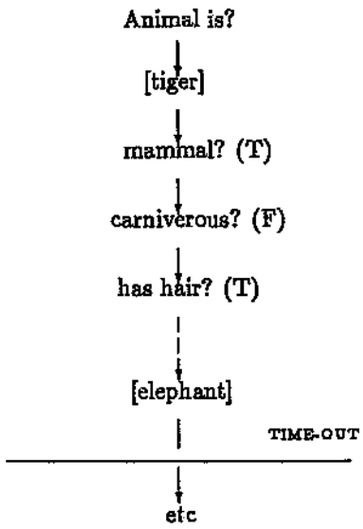


Figure 2.5: Serial approach to the animal problem

partial solutions, [tiger] and [elephant], respectively. At these nodes, the aggregate CF for the proposition that the animal is a tiger is 0.43, and that it is an elephant, 0.72. Therefore the expert system concludes that the animal is an elephant.

Given the same time allowance, a serial approach may present an outcome as shown in Figure 2.5: The expert system has at time-out, completed an analysis of the tiger-assumption, and just started assessing the elephant-assumption. Since the elephant-assumption was not given an equal opportunity to compete with the tiger-assumption, the expert system concludes erroneously that the animal is a tiger.

We see in this example that, although neither approach had time to complete the inference process, the risk of error is reduced by the parallel approach.

2.5 Conclusion

In this chapter we have covered several issues concerning real-time that are important to the design of a real-time expert system. In summary, the following key points are listed:

- Real-time data are time-limited.

- Real-time response is the ability of the system to synchronize with real events as they happen.
- The real-world is modelled by the state-based sample-and-hold approach.
- Real events are frozen at every scan-cycle.
- The history of events are kept in fixed-length history frames.
- Inference-cycles are synchronized to the change of events.
- An opportunistic scheduler is required to synchronize inference-cycles.
- Parallel processing of tasks is required to reduce the risk of errors under time-stress.

Chapter 3

Overview of conventional expert system techniques

3.1 Fundamentals of expert systems

Expert systems are a class of computer programs that contain domain knowledge about a specific field of interest, and when interrogated, respond like an expert. Unlike traditional programs that follow a *Von Neumann* architecture, where data and control algorithms reside in the same program as one inseparable entity, the structure of expert systems is organized in such a way that more or less rigidly separates the standard computational components of data and control.

The bulk of the database contains *domain knowledge*. By domain knowledge, we refer to knowledge specific to a field of application. In the expert system, this database is called the *knowledge base*. The general control mechanism, or problem-solving knowledge, is contained in the *inference engine*. The concept of expert systems is thus broken down into two areas: that of the knowledge base, and that of the inference engine. These two major components of the expert system are shown in Figure 3.1.

The knowledge base is unique to a particular domain. For example, the knowledge base in MYCIN contains medical knowledge; and in PROSPECTOR, the knowledge base contains knowledge specific to mining. Knowledge can usually be represented in terms of facts that describe the world, relationships between the facts, procedures or rules for manipulating facts, and information that specify when or how to apply

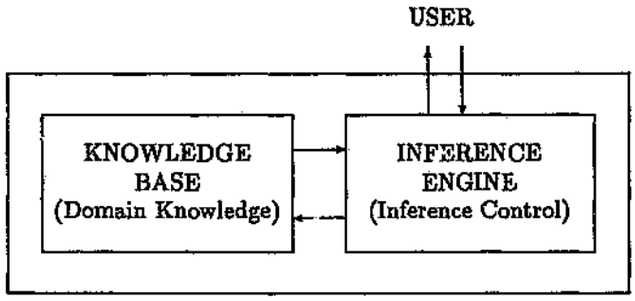


Figure 3.1: Basic structure of an expert system

the rules or procedures. This is typically implemented by one of many well-known schemes, for example, if-then production rules, frames, and semantic nets.

Knowledge representation is a crucial issue in the design of an expert system, since it defines the way and the ease in which facts are retrieved and used. An efficient representation scheme is necessary in order to facilitate real-time performance.

An inference engine on the other hand, contains general problem-solving strategies that may be common to a number of domains which have similar characteristics. For instance, commonly available *expert system shells* contain general-purpose inference engines plus some essential user-interface and an empty knowledge base. These expert system shells can be customized for a specific application by inserting the necessary domain-knowledge.

By inference, we mean that features of the task environment are pattern-matched against the stored modules of premises, or facts about the environment [22]. In order for a system to reason, it must be able to infer new facts and draw conclusions from what it has already been told. It is the task of the inference engine to draw conclusions about a given environment state by referencing the knowledge base. Furthermore, this must be done sufficiently quickly in order to meet real-time requirements.

In this chapter, we look at various forms of knowledge representation and inference strategies. The aim is to draw from this survey a suitable combination of conventional techniques, modified where necessary, to meet real-time requirements.

3.2 Knowledge representation

A knowledge representation scheme is a set of syntactic and semantic conventions used to describe the knowledge required for a given domain. This is a way of codifying human expertise in the computer. Whatever form of representation, a computer must be able to store and process the represented knowledge in order to apply it to problem-solving. Rich states four main criteria for assessing a representation scheme [30], these are:

- Representational adequacy — whether the formalism is capable of expressing all kinds of knowledge required in a given domain.
- Inferential adequacy — whether new knowledge can be inferred from the knowledge represented.
- Inferential efficiency — how easy it is for the inference engine to focus on the relevant information.
- Acquisitional efficiency — how easy it is to import new knowledge and to edit the existing knowledge base.

In designing an expert system, we must carefully consider the trade-offs between these criteria and strike a suitable balance that meets performance requirements. The key to efficient problem-solving lies in efficient representation. For real-time applications, the ease and the efficiency at which relevant knowledge may be extracted are prerequisite since available time is strictly limited.

Knowledge can be represented *declaratively* or *procedurally*. Declarative knowledge is represented, usually in the form of predicate logic, as a static collection of facts accompanied by a small set of general-purpose procedures for manipulating the information. Procedural knowledge is represented as explicitly detailed, algorithmic instructions. This is characterized by its rigid structure, and fixed flow of control and data utilization [30].

The advantage of declarative methods lies in storage efficiency: each fact needs to be stored only once, regardless of the number of different ways in which it can be used; and the ease of modification: new facts can be added to the system without affecting existing facts or procedures. In an expert system, we want the flexibility of a

human expert in decision-making that copes with changing demands and situations, declarative methods seem to be the natural choice. Unfortunately, the processing overhead required for declarative representation is large.

Most domains need both kinds of information. In practice, a combination of both is often used [30], and the role played by either is adjusted according to the application. To overcome the drawback in the processing overhead, the role of procedures needs to be expanded, and the role played by static facts contracted.

Three main classes of knowledge representation are favoured by expert system designers [12]:

- production rules — These are characterized by the classic *if-then* paradigm. Production rules are the most basic and common form of representation. Systems based on production rules are called *rule-based systems.*

- predicate logic — a formal language that represents real-world facts as logical propositions written as *well-formed formulae* in propositional logic [30]. For example, the statements "John is an engineer" and "John lives in 123 Anylane" can be represented by

  ```
  occupation(John, engineer)
  address(John, '123 Anylane')
  ```

  PROLOG (PROgramming in LOGic) is an important programming language for artificial intelligence that bases its reasoning processes on the principles of predicate logic [38].

- structured objects — units or structures that aggregate several related predicate logic expressions and form fundamental building blocks for representing knowledge [26]. These building blocks are analogous to the nodes and arcs of graph theory, or the slots and fillers of record structures [12]. Two popular structured representation schemes are derived directly from the concept of structured objects, namely, semantic nets and frames.

The methods of using production rules, semantic net and frames are investigated in this section, because they provide the most direct ways of representing knowledge.

There are many other schemes of knowledge representation, e.g., *Conceptual Dependency* and *Scripts* [30], but these are mostly specialized variations of those already mentioned.

In general, knowledge has varying degrees of 'certainty'. Whatever representation scheme we choose to adopt, we need a way of representing *uncertainty* for knowledge and situations that are imprecise, unreliable or incomplete. This is discussed under the subsection entitled "Uncertainty management".

3.2.1 Production rules

Production rules consists of premise-conclusion pairs. These rules simply state that given a situation, then certain consequents are valid. Both the premise and the conclusion may be linked by *AND* or *OR* connectives. For example,

if P_1 and ...and P_n,
then Q_1 and ...and Q_m.

That is, if the premises P_i, $i = 1$ to n, are true, then we conclude that Q_j, $j = 1$ to m. A premise is a set of conditions, for example, 'valve opening is 100%', 'flow volume is high'. A conclusion may be an action, a causal link, or a definitional link [17]. Examples of these are given below:

- action: sound the alarm; close the valve.
- causal link: the flow is blocked by valve A.
- definitional link: the valve is blocked.

Premises and conclusions are typically object-attribute-value triplets as in the MYCIN expert systems, or attribute-value pairs. For example,

if (valve opening 100%)
and (flow volume high)
then (valve close 50%).

The rule states that "if the valve opening is 100% and the volume of the flow is high, then close the valve by 50%." We can interpret 'valve' and 'flow' as objects, then the attributes are 'opening', 'volume' and 'close', and the values are '100%',

'high', and '50%'. Alternatively, we can make 'valve opening', 'flow volume' and 'valve close' as attributes with values '100%', 'high', and '50%', respectively.

Rule-based systems are popular because many experts and many programmers find it relatively easy to express and encode knowledge in this form. Rule-based systems are flexible, they do not enforce any strict order or overall structure: Rules can be included in any order, and rules can be added to, or removed from, the knowledge base without disturbing the existing set of rules.

The disadvantages against rule-based systems are summarized in [12]:

1. It is difficult to predict the outcome of competition between rules that have equal chances of being fired during inference. Rule-based systems need to pay special attention to *conflict resolution.*

2. Representing knowledge as an unordered and unstructured set of rules

   - imposes strict discipline on the knowledge engineer to ensure a satisfactory representation;
   - fails to take advantage of whatever explicit structure the domain possesses in terms of taxonomic, part-whole or cause-effect relations that hold between objects and between classes of objects.

3. Although production rules are suited to expressing empirical associations between situations and actions that have the general form 'if *condition* then *action*', it is less effective at expressing other more subtle forms of knowledge.

3.2.2 Semantic nets

Information is represented graphically in a semantic net by a set of connected nodes. The nodes of a semantic net consists of objects which may be physical objects, conceptual entities or descriptors [17]. They are linked by labelled, *directed* arcs which represent the relationship between the nodes. Thus, to assert that "A chair is a piece of furniture", we would link the objects 'chair' and 'furniture' by a link labelled 'is_a'. Figure 3.2 [1] shows a fragment of a typical semantic net.

Semantic nets are a general form of knowledge representation. They are useful for descriptive purposes because they give a simple, structured picture of a body

[1] This example is taken from [30].

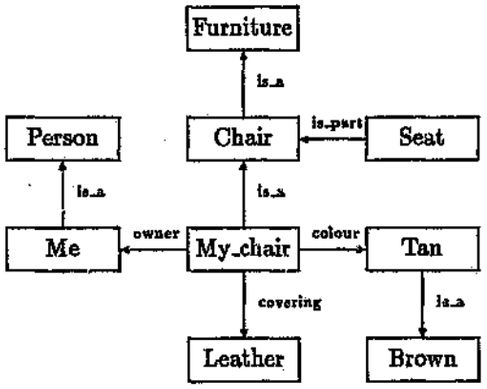


Figure 3.2: Example of a semantic net

of facts [26]. Hierarchies of concepts can be built using semantic nets. Inheritance hierarchies, e.g., *trees* and *lattices*, provide an efficient way of simplifying representation and reducing the amount of information needed at each node [1]. This implies considerable increase in the speed of information processing.

However, the very simplicity of the network formalism leads to shortcomings in two areas: The first lies in logical adequacy — the meanings assigned to nodes are often unclear. For example, faced with the node labelled 'chair', do we interpret it as the concept of a chair, or the class of all chairs, or a typical chair? The second aspect has to do with heuristic adequacy — because searches for knowledge are not themselves knowledge-based, that is, the directions for finding information within the system are not embedded within the networks, it is often difficult to say when is it safe to terminate a inconclusive search [12].

3.2.3 Frames

Frames are used to describe a collection of attributes pertaining to a given object or situation, the descriptions are often from different points of view [30]. Essentially, the properties of some object or event are grouped together to form a prototype. Each instance of a frame consists of a name, attributes, and the associated values of these attributes. Frames may be linked, where the links represent the relationship

CHAIR	
SLOT	*FILLER*
c ur	Tan
covering	Leather
owner	Me

Figure 3.3: Example of a frame

between the views that the frames represent. Thus, a collection of frames is simply a *relational database*. Figure 3.2.3 shows an example of a frame that describes information of a chair.

The attribute-value pairs are represented by the slot-and-filler structure. The slots describe aspects of the object and may be filled by other frames describing other objects. The fillers are filled by a set of conditions that must be met, or by the attachment of *procedural knowledge*.

The procedural attachment contains the plan or some kind of control information about the manipulation of the slot contents. These are known as *demons* and are activated by *triggers* which watch for some condition to come true. Triggering conditions fall under three general categories [17]:

- if-added: The if-added routines are invoked when slots are filled with new information.
- if-removed: when fillers are removed or retracted from the slots.
- if-needed: when an attribute is queried but is not present in the slots.

The frame structure is often used to represent stereotyped situations [23]. It offers mechanisms for dealing with exceptions and defaults which are not easily handled by standard logic [12]. The combination of both declarative and procedural representations in the frame structure can be used to great advantage in optimizing knowledge representation for real-time purposes.

3.2.4 Uncertainty management

There is a need for some form of uncertainty management in expert systems because knowledge and information can be imprecise, unreliable or incomplete. A real-time expert system must be able to continue functioning in the presence of varying amounts of uncertainty. Therefore we must be able to represent and quantify uncertainty, and reason with it.

Uncertainty management is used to handle the variable degree of confidence in a given factual expression or assertion, and to resolve conflicts between competing conclusions. In essence, we want to be able to say (in terms of a rule construct) that,

if A(with certainty x) then B(with certainty y)

In expert system research, several strategies have been formulated to handle uncertainty such as the *Bayesian* models of uncertainty management, the *Dempster-Shafer theory of evidence*, and Shortliffe's *certainty factor*.

Shortliffe's certainty factor as used in MYCIN is a numerical expression designed to measure confidence. It could be placed in any given conclusion as a result of the evidence so far. Evidence both for and against hypotheses are collected during inference. A measure of belief in a hypothesis h, based on evidence e, denoted MB(h,e), is associated with evidence for a hypothesis. Likewise a measure of disbelief denoted MD(h,e) is associated with evidence against the hypothesis. Both MB and MD are in the range of 0 to 1. The actual CF is a simple balance of the measures of belief and disbelief, calculated by taking the difference between them such that,

$$CF(h,e) = MB(h,e) - (h,e)$$

$CF(h,e)$ lies in the range $[-1,+1]$. A positive CF indicates a degree of belief, a negative CF a degree of disbelief, and a CF value of 0 indicates ignorance [12].

The Bayesian model has been applied to expert systems such as PROSPECTOR with impressive results. The model is based on the theory of probability. Uncertainty is represented as a probability between 0 and 1 according to *Bayes' theorem* as,

$$P(H_i \mid E) = \frac{P(E \mid H_i)P(H_i)}{P(E)}$$

$$P(E) = \sum_{j}^{k} P(E \mid H_j)(P(H_j))$$

where $P(H_i)$ represents the *a priori* probability that hypothesis i is true in the absence of any evidence, $P(H \mid E)$ the probability of the hypothesis after finding evidence E, $P(E \mid H_i)$ the conditional probability that evidence E will be observed given the hypothesis is true, and k is the number of possible hypotheses [17].

The Dempster-Shafer theory of evidence improves on the Bayesian method in being able to distinguish between uncertainty, lack of knowledge, and equal certainty. Essentially this theory allows a degree of unknown to be added to the summation of probabilities [17]. If P and P' are degrees of belief and disbelief, respectively, and U is a degree of unknown then,

$$P + P' + U = 1$$

Many other numerical approaches have been proposed, e.g. the *Plausible Inference* [9], Quinlan's *Inferno* [28]. Symbolic approaches include *Linguistic Reasoning*, *Truth Maintenance systems*, and *Theory of Endorsement* [28].

3.3 Inference

The inference process *reasons* about knowledge and infers new knowledge from old knowledge; it draws from the knowledge base information relevant to the problem at hand, and formulates decisions, or conclusions, on the problem. In most cases, inference is *pattern-directed* [10] in that patterns from the input data are identified and matched against those stored in the knowledge base, and appropriate modules of the represented knowledge are then 'fired'.

Inference involves controlling the information flow. Commonly used rules of inference are *modus ponens*, which states that "*if A then B* and *A*, we can deduce *B*", and *modus tolen*, "*if A then B* and *not B*, we can duduce *not A*" [17]. Chief areas to look at regarding inference are:

- search strategies — how the inference engine locates pertinent information in the knowledge base;
- chaining methods — how the pieces of information are linked together during problem-solving;
- the use of meta-reasoning;

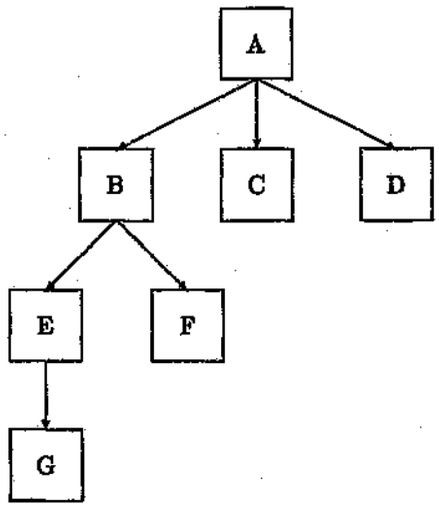


Figure 3.4: Example of a logic tree

- whether the system needs non-monotonic reasoning.

3.3.1 Search strategies

Searching has to do with finding a (best) path from the origin (problem statement) to the goal (solution). There are basically two ways by which the inference engine searches for solution: breadth-first and depth-first search. Consider the logic tree given in Figure 3.4:

In breadth-first search, all the nodes on the same level of the tree will first be examined before advancing to the next level. To explore all the nodes in this tree, the inference engine will follow the sequence

A–B–C–D–E–F–G.

Breadth-first search is guaranteed to find a *shortest length* path to a goal node, providing a path exists at all [26].

Depth-first search delves as deep (far) as possible into the tree on a chosen path (context). It involves backtracking when a solution is not found in the chosen path.

By depth-first search and probing the tree from left to right, the inference engine will traverse the tree in the order of

A–B–E–G–(E–B)–F–(B–A)–C–(A)–D,

The bracketed nodes are nodes covered by backtracking. This method is efficient if it is known beforehand which path should be expanded to provide the solution [17]. Performance of depth-first search can be quite poor if a particularly long branch of the tree needs to be explored only to find that there is no solution at its end.

Both breadth-first and depth-first search guarantee a solution, since both methods eventually degenerate into *exhaustive* searches, at the cost of efficiency.

Efficiency is improved in both cases if heuristic is employed. Paradigms like *hill-climbing* and *beam searching*, which explore only the most promising nodes; *best-first*, which starts the search from the most promising open node; and *branch and bound*, all have merits in different applications [40].

3.3.2 Chaining methods

There are two basic reasoning strategies used in expert systems: the forward chaining method that works forward from the evidence to the conclusions (data-driven); and the backward chaining method that works from hypothesis to evidence (goal-driven).

In forward chaining, no hypothesis is at first assumed, but current information is examined for certain patterns that infer further information, and eventually a conclusion is deduced. Forward chaining sometimes lacks a definite goal, therefore the reasoning process is often not coherent. Also, the reasoning process will never terminate if the initial evidence does not lead to a conclusion, i.e., the forward chaining process will continue for ever due to the lack of a definite goal.

In backward-chaining, inference starts from a tentative (yet unrefined) hypothesis and progressively refines the hypothesis until time-out, by matching specific findings or performing tests. In this case the hypothesis is a goal to be proven. Reasoning follows a defined path of logic because there is a definite goal.

For performance optimization, a hybrid of the two methods are often used. Commonly, data-driven inferencing is used to suggest a set of hypotheses based on initial data, and backward chaining is used to prove these hypotheses [13].

3.3.3 Meta-reasoning

Meta-reasoning, or reasoning about reasoning, is a technique for planning the reasoning process using meta-knowledge. In meta-reasoning, the inference engine is made an expert of its own operations. Knowledge about the contents of the knowledge base and about reasoning strategies are used to guide the system in its selection of an appropriate approach to solving the current problem. In practice, we try to prune the search space over conventional search methods [12], thereby optimizing system performance.

3.3.4 Monotonic reasoning

A point of increasing importance is whether the expert system needs *non-monotonic* reasoning. By monotonic reasoning, we mean that the number of statements known to be true is strictly increasing over time [30], that is, once the facts, data and conclusions are asserted, they remain unchanged and will not be retracted for the duration of the session. Whereas in non-monotonic reasoning, the addition of one piece of information may force the deletion of another. Facts are allowed to be retracted and consistency is enforced by resolving conflicting conclusions or decisions. Doyle's [30] Truth Maintenance System is an example of non-monotonic system.

Although non-monotonic reasoning is required in order to propagate changes in the system and to check proofs for current validity, it implies a complex and difficult process of updating conclusions and intermediate decisions. Computing overhead can be staggering. For real-time applications, this is a luxury we can ill afford. To get around the need for non-monotonic reasoning, we must be able to assume that no changes will occur during the reasoning process. In Section 2.2, we discussed the method of approximating the real world by 'freezing' the environment for the duration of an inference cycle. Within this period, we assume an unchanging environment and can therefore, adopt only monotonic reasoning.

3.4 Conclusion

In this chapter, we have looked at various expert system techniques commonly employed in existing expert system. These include rules, semantic nets, frames for rep-

resenting knowledge; uncertainty management for representing and reasoning with uncertainty; inference strategies regarding searching and chaining of knowledge represented in the knowledge base; and a brief introduction to meta-reasoning. From these we can now draw suitable knowledge representation and inference schemes that can be used in our design of an expert system shell.

Chapter 4

Design of an experimental real-time expert system shell

4.1 Introduction

This chapter deals with the design of AKShell. It is presented in three parts: the knowledge base, the inference engine, and the overall architecture. Then we define the communication protocol by which AKShell tasks communicate with each other. This is followed by a section dealing with the explanation function of an expert system. Finally, we present a qualitative analysis of the design's ability to meet real-time constraints.

4.2 Knowledge base

The knowledge representation scheme of AKShell was designed to allow *progressive reasoning*. The concept of progressive reasoning is based on the argument that many problems can be reasoned at several different levels of abstraction [41]. If different levels of knowledge representation correspond to different levels of specialization (in terms of precision) then at the highest level, a problem is viewed at a microscopic scale, in terms of individual quantities and data values; and qualitatively at a lower level, where only an overall insight of the problem is required. Progressive reasoning basically involves solving problems level by level, by increasing orders of specialization. This is illustrated in Figure 4.1.

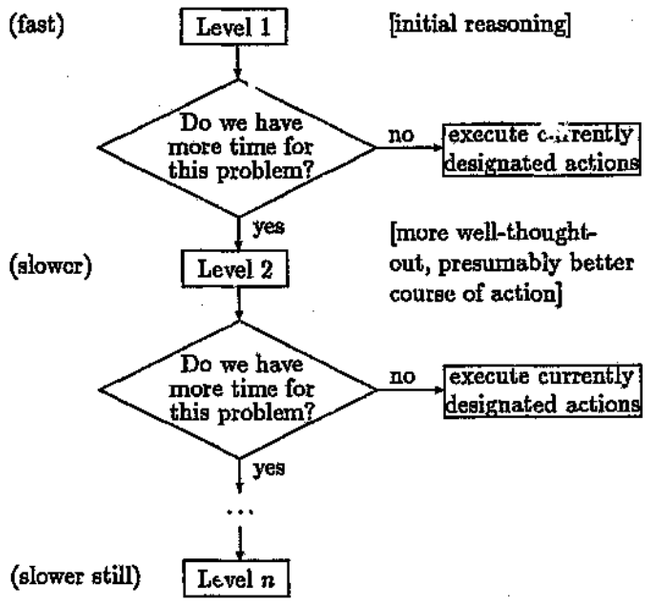


Figure 4.1: Progressive reasoning algorithm

As the complexity of problem-solving algorithm grows with increasing levels of specialization, the required computing time also increases as is indicated in the figure. Real-time reasoning requires the inference engine to zoom in step by step on more specialized diagnoses, progressively improving the solution until time runs out.

We chose to implement a two-level design in AKShell, where level-one contains meta-knowledge, and level-two contains specialized knowledge. Meta-knowledge helps the inference process to decide which parts of the specialized knowledge base to use. Specialized knowledge is then used to generate the final decision. The following subsections present the two levels separately.

4.2.1 Level-one knowledge base

This level contains knowledge about the applicability of knowledge contained in the next level. Based on the input data, we want to be able to quickly focus the expert system's attention on relevant subsets of the domain knowledge best suited to solving the problem at hand. Essentially, we want a *pattern-directed* classifier.

For speed considerations, decisions made through the level-one knowledge will be based on pattern recognition rather than on exhaustive reasoning. This is modelled after the 'quick reflex', or intuitive 'guesses' often followed by human experts.

Thus, level-one knowledge base will store patterns, or templates that are used to compare against the input data. Each template in turn leads to a particular subset of the knowledge contained in level-two. In our prototype, we have adopted a goal-driven system where each template represents a *partial solution* to be verified at the second level. But the templates themselves are data-driven, that is, a particular data pattern will result in the firing of a particular partial solution.

The templates are stored in a sequential list. Each template is represented by an If-Then production rule as follows:

If *data pattern* Then *hypothesis X*

Where hypothesis X is a partial solution that will trigger a corresponding subset of the level-two knowledge base.

4.2.2 Level-two knowledge base

The knowledge representation scheme at level-two is derived from semantic nets and frames. It is a hybrid scheme designed to allow a combination of expert system and conventional algorithmic techniques in that, procedural knowledge can be incorporated into the AKShell along side with the declaration knowledge.

The knowledge base is partitioned into *knowledge sources*, where each knowledge source represents the focused knowledge contributing to a decision [25]. A tree structure is used to build a knowledge source. We call this tree a *semantic tree* — it forms an *island of expertise* relevant to a partial solution. It follows from Section 4.2.1 that there must be as many semantic trees as there are partial solutions in the level-one knowledge base. In inference, the partial solutions become 'goals' at the root of the semantic trees. The hierarchy of the AKShell knowledge base is shown in Figure 4.2.

A semantic tree consists of linked nodes. The nodes contain predicate conditions of arriving at a decision, goal or subgoal. An example semantic tree is shown in Figure 4.3.

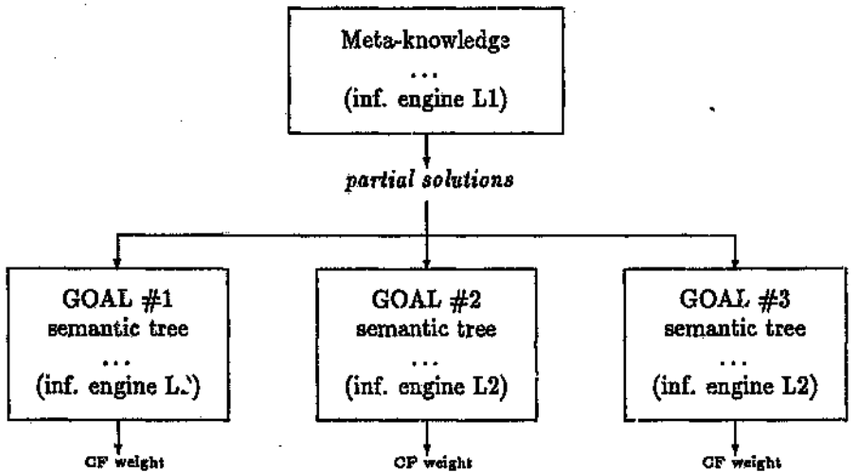


Figure 4.2: The knowledge base hierarchy

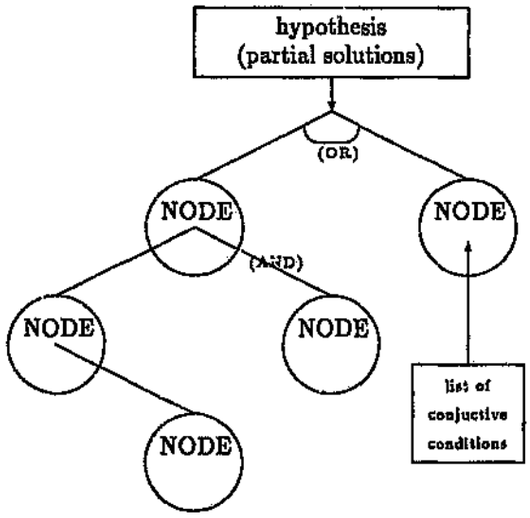


Figure 4.3: A semantic tree

```
typedef struct predicate {
   float *param;            /* reference */
   float *fact_value;       /* fact values */

   /* demon slot */
   BOOLEAN (*extract_value)(float *, float *);

   struct predicate *next;
} PREDICATE;

typedef struct node {
   PREDICATE *assumption;   /* predicate list */

   /*---------------------*
      confidence factors:
      cf[0] = success cf;
      cf[1] = failure cf
    *---------------------*/
   float cf[2];

   /* user-definable CF function */
   float (*cf_func)(float *, float *);

   struct node *sibling;
   struct node *offspring;
} NODE;
```

Figure 4.4: The structure of a node

A node is similar to a frame. Each node embodies a subset of the conditions required to satisfy a goal. The node structure (expressed in C code) is shown in Figure 4.4. This structure was designed to couple procedural and declarative knowledge.

A slot in the node structure accepts declarative definitions, e.g.,

```
likes(John, any_book)
```

Where the terms inside brackets are *fillers*, similar to the attribute-value pair; and the operative, 'likes', can be regarded as a pointer to the function that receives the fillers as parameters.

Instead of defining operatives on top of existing operatives (as in PROLOG), user-defined operatives are written in a procedural language such as C and linked to the expert system when the knowledge is compiled. This follows the general idea of demons [30], and the triggers to activate a required process are embedded in the slots of a node. The general format of a demon function can be expressed as a C

function prototype as

BOOLEAN *function name*(TYPE *node fillers*).

Demon functions will return either *true* or *false* after they are invoked by the inference engine. The declaration of the demon slot is shown in the PREDICATE structure in Figure 4.4.

Where there are conjunctive associations of facts within the same node, the facts are chained in a linked list inside a slot. An arrangement of conjunctive and disjunctive nodes form a semantic tree where the peer nodes depict disjunctive, or *OR*ed, conditions, and the sequential nodes embody the global conjunctive conditions. The 'AND' and the 'OR' relationships on the semantic tree are shown in Figure 4.3.

In allowing the coexistence of procedural and declarative knowledge, numerically intensive subroutines, low-level control algorithms, and complicated mathematical calculations are hidden from the inference engine. The inference engine needs only to interpret in declarative form, the knowledge represented by the hidden procedures. Since inference engines rely on an *interpreted* mode of operation, as opposed to demons, which are *compiled* codes, this division of labour will further optimize system performance.

Quantification of confidence is managed by assigning two confidence weights (certainty factors) to each node: weight of success and weight of failure, or measures of belief and disbelief, respectively. The use of independent weights is useful where the absence of some precondition does not reduce our confidence in a solution by the same amount its presence will boost the confidence.

If the conditions of a node are satisfied, the weight of success is used to modify, by user-defined formulae, the existing CF; otherwise, the weight of failure is used. The new CF value is then inherited by the next node interrogated by the inference engine.

User-defined formulae to modify CF may be simple addition and subtraction, or some statistical method like the *Bayes Theorem*. Again, the quantification methods are triggered by means of demon-activation. Thus, within the same system, a knowledge engineer has the freedom to employ different strategies to aggregate confidence.

A CF-demon function takes the existing CF value and the chosen CF weight from

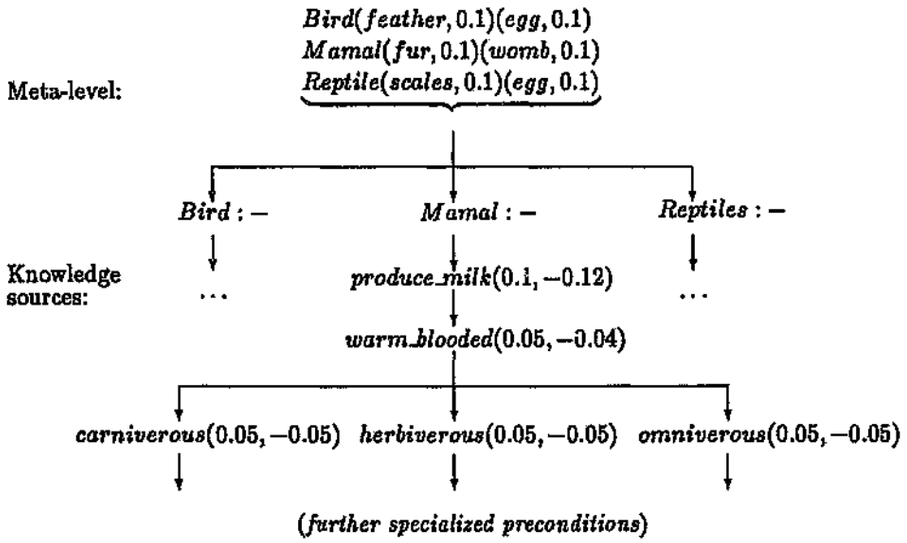


Figure 4.5: Knowledge base on animals

the current node as parameters. It computes and returns a new CF value.

As an example, Figure 4.5 depicts a typical layout of a knowledge base designed for identifying animals. The syntax of the meta-level rules depicted in the figure is,

assumption(*precondition*,*confidence weight*)<(*precond.*,*conf. wgt.*) ...>;

and those in the knowledge sources,

precondition(*weight of success*,*weight of failure*).

At the meta-level, three broad patterns are available: that a bird is feathered and lays eggs; a mammal has fur and a womb; and a reptile is scaled and also lays eggs. If the input data matches the second pattern best, then we generate a partial solution reckoning that the animal is a mammal. Its corresponding subset of the knowledge base (a semantic tree) will then be used to prove this hypothesis.

The inference process on this semantic tree verifies this partial solution by progressive building up the confidence factor. The confidence factor is initialized at level-one. For example, if the preconditions "It has fur" and "It has a womb" are satisfied, then an initial confidence factor of 0.2 is assigned to the mammals-hypothesis.

```
BOOLEAN
is_warm_blooded(int body_temperature)
{
   if (body_temperature > 10)
      return true;
   else
      return false;
}
```

Figure 4.6: An example demon function

Thereafter, each step in the search on the semantic tree either increments or decrements the confidence factor as the preconditions at each node are satisfied or disaffirmed. The final aggregated confidence factor represents the confidence in the solution. In our example, if all the listed preconditions are satisfied, we will have a total CF of 0.5 (50% probability of being a mammal).

If a node is used to represent each precondition in the animal example, the semantic tree on mammals will have nodes that inquire whether the animal produces milk, whether it's warm-blooded, etc. The preconditions are represented by demon functions embedded in the nodes. Take the 'warm-blooded' precondition as example, we could represent it in C code as shown in Figure 4.6.

4.3 Inference engine

As the knowledge base has two-levels, the inference process is also two-tiered. The level-one, or meta-level inference engine must decide from input data which subset of the knowledge base is most applicable for solving the problem effectively. Inference at this level is therefore *data-driven*, and backward-chains to produce tentative results, or *partial solutions*. These partial solutions are then passed on to, and refined by, the inference engine at level-two. This way, the level-one inference engine spawns or, fires the level-two inference process. The inference processes at the two levels are discussed in the following subsections.

4.3.1 Level-one inference engine

Essentially, we want to implement a classifier at level-one. Input data are pattern-matched against templates stored in the level-one knowledge base. If the partial

solutions are the *then-clauses* of a production rule, then the templates constitute the *if-clauses.*

The templates and the corresponding partial solutions are stored in sequential order. One by one, the inference engine takes each partial solution, initializes its CF to 0 (no confidence), then tries to match the template against given data. The closer the match is, the higher is the CF finally assigned to the partial solution. This process continues until all the available partial solutions are tested.

Based on the CF's, the inference engine selects a number of most promising partial solutions. These are then passed on to the level-two inference process with their current CF values logged in the blackboard. To the level-two inference process, the CF values reflect the degree of confidence in the initial acceptance of the partial solutions.

There are several methods to implement the level-one inference process. The most elementary method involves constructing a straight-forward rule-based system as implied in this section. Frey [8] proposed a 'bit-mapped classifier', which treats information about the set of conditions as a string of bits. System performance will be improved because computing overhead is vastly reduced by avoiding extensive symbolic processing. However, for clarity of illustration, we have opted for a simple rule-based meta-level inference engine for the prototype.

The inference algorithm is described in pseudo-code and shown in Figure 4.7.

4.3.2 Level-two inference engine

Inference at level-two is *goal-driven*, the goal being the hypotheses, or partial solutions generated at level-one. Each partial solution defines a search area of the knowledge base, i.e., a semantic tree. The conjunctive conditions embedded in each node on the semantic tree are examined according to the *modus ponens* rule of inference.

The confidence factor in a partial solution is initialized to the value evaluated at level-one. The result of inference at a node determines how the CF is updated. In examining a node, if all its conditions are satisfied, then the weight of success of this node is used to increment the CF; otherwise, the weight of failure is used to decrement the CF.

```
Process level-one inference (input, output);

Data Structure:
   knowledge structure = Record of
      partial solution;
      linked list of preconditions;
   End Record;

Constant:
   n = total number of partial solutions;

Variables:
   input variable : Array of input data structure;
   level-one record : Array of knowledge structure;
   current solution : knowledge structure;
   current rule : precondition;
   i, j : integer;

Output:
   partial solutions;

Begin
   Get: input variables;

   For i := 1 to n Do
      current solution := level-one record [index];
      j := 1;
      While (more rules in the record) Do
         current rule := rule [j];
         test rule;
         Update CF;
         j := j + 1;
      End While;
   End For loop;

   Compare CFs and select partial solutions for level two;
End.
```

Figure 4.7: Level-one inference algorithm

```
Process level-two inference (input, output);

Constant:
   semantic tree : tree of knowledge nodes;

Input:
   partial solutions;
   initial CF value;

Output:
   inferred CF value;

Variable:
   CF value, node value : confidence factors;

Begin
   Get: initial CF value from level one;
   While (still more child nodes to test) and
      (not time-out) Do
      Repeat Until (no more peer nodes to test)
         test nodal conditions;
         obtain and store node value;
         select next peer node;
      End Repeat;
      current node := node with highest node value;
      update CF value;
      chain current node;
   End While;
End.
```

Figure 4.8: Level-two inference algorithm

In order to prove the hypothesis, the inference engine traverses through the semantic tree by a *hill-climbing* process [30] and finds a solution path on the tree that generates the highest CF. This implies that the inference engine must employ *breadth-first search* to explore all possibilities at each level of the tree, in order to find the most promising path. The algorithm describing this inference process is shown in Figure 4.8.

In traversing the semantic tree, the inference engine interrogates the nodes along the inferred path. The conjunctive conditions of a node are examined by triggering the corresponding demons. The demon functions take a subset of the input data as parameter, process the data, and return a boolean value informing the inference engine whether the condition has been met.

The inference process continues until all the nodes on the semantic tree are examined, or until time-out. This is indicated by the 'while-loop' in Figure 4.8.

Since decision making of the expert system is based on the aggregate value of the confidence factor, it can draw information from the inference engine even if it were prematurely stopped — as soon as the level-two inference engine is started, a decision already exists in the form of a partial solution. The passage of time serves only to reinforce or refute that decision by means of CF assessments. A solution is therefore *guaranteed*, whose quality improves with the passage of time.

Take again the animal example: If meta-level inference suggests that the animal could be a mammal based on a rudimentary, cursory analysis, this partial solution then becomes the starting point of the second level inference. The inference process then reviews this decision using the more specialized, level-two knowledge base. From this point on, the expert system can state at any time should it be stopped, that it believes the animal to be a mamm: with $x\%$ certainty.

For reasons discussed in Section 2.4.2, parallel programming is applied at level-two: the expert system will invoke a separate inference engine for each partial solution generated at level-one. Inference is based on a system of *competitive reasoning* where the inference processes run in parallel, each toward its own goal (to prove the solution). CF's serve to resolve conflicts between these parallel inference processes (PIP.'s), that is, the final decision made by the expert system will be the solution with the highest aggregated CF.

Since CF's reflect the degree of belief in the partial solutions, a task-scheduler can use the CF values as a basis to prioritize the PIPs dynamically, and allocate more computing resources to the more promising ones.

A complete inference-cycle thus embodies meta-reasoning at level-one, and parallel pattern-directed invocation of inference processes at level-two. Figure 4.9 illustrates the algorithm behind such an inference-cycle.

4.4 Architectural design

The chief components of AKShell are

- a blackboard;
- an event-indicator;
- a level-one inference engine;

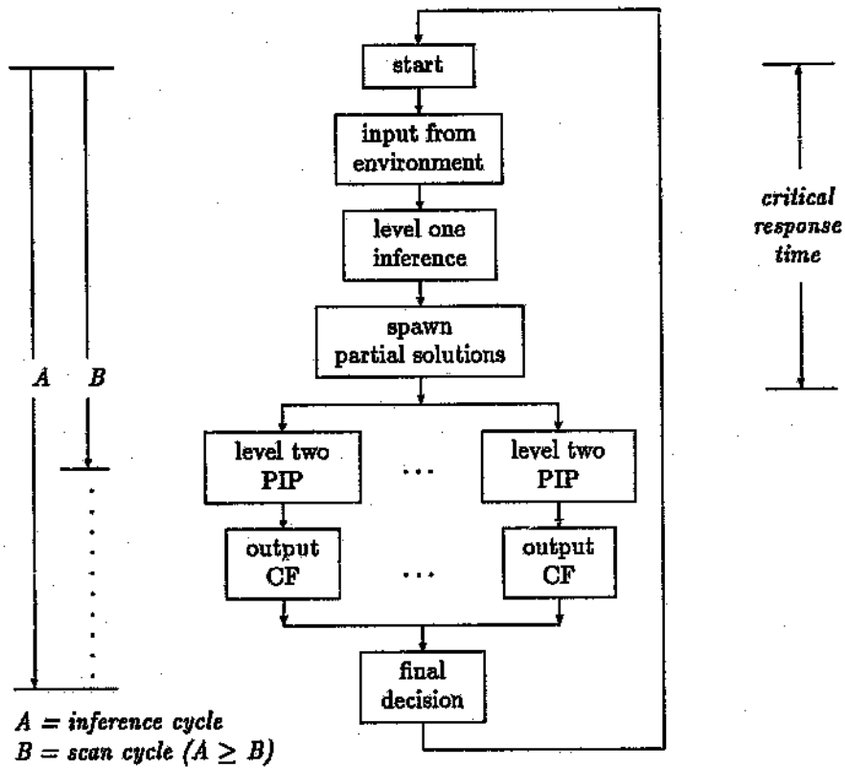


Figure 4.9: An inference cycle

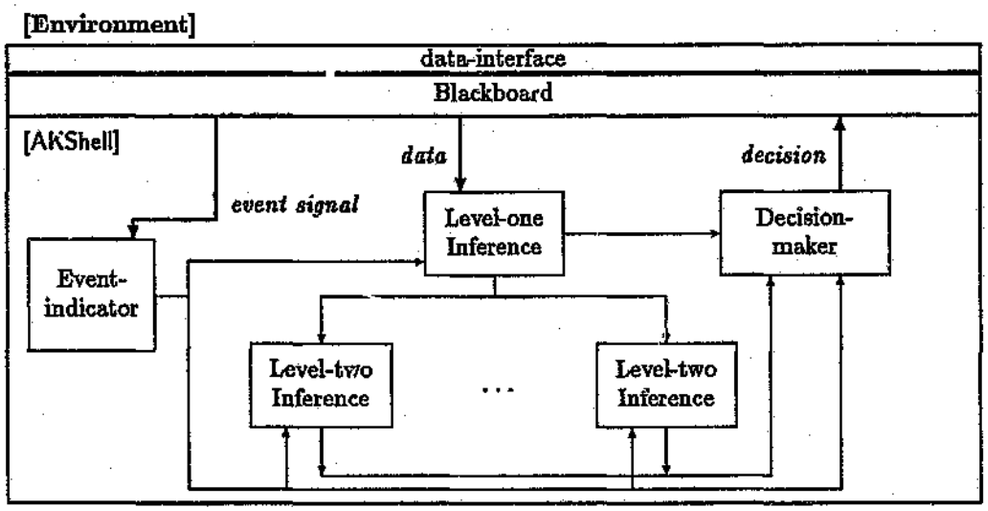


Figure 4.10: The block structure of AKShell

- a number of level-two inference engines;
- a decision-maker; and
- a data interface.

The overall structure is shown in Figure 4.10. The arrows indicate the logical links between the components.

The core of AKShell is a *blackboard* structure [25] [30] that maintains a CF table and the knowledge base. The blackboard knowledge base contains both the level-one knowledge base and the level-two semantic trees. The layout of the knowledge base is shown in Figure 4.2. The entry points into the appropriate subsets of the knowledge base are passed to the inference engines during an inference process. The progress of level-two inference is recorded in the CF table, that is, each active level-two inference engine updates its CF in the CF table. The blackboard is essentially a global database.

The event-indicator manages the overall system timing, and co-ordinates the inference processes and the decision-maker. It runs concurrently with these tasks. The event-indicator starts and ends inference-cycles by watching out for the event signal sent out to the blackboard by the data-interface as shown by the pseudo-code description in Figure 4.11.

```
Process event-indicator (input, output);

Variables:
   input variable: 'event signal' from data interface;

Begin
   Repeat for ever
      invoke level-one inference process;
      While (no new event) and
            (PIPs still incomplete) Do
         invoke PIPs;          /* level-two inference */
         Get: event signal;    /* check for new event */
      End While;
      invoke decision-maker;
   End Repeat;
End.
```

Figure 4.11: Event-indicator algorithm

This scheduling of inference-cycles is aligned to the system scan-cycles, that is, the start of an inference-cycle always coincides with the start of a scan-cycle. In worst-case conditions, an inference-cycle is scheduled for every scan-cycle; otherwise, an inference-cycle may last several scan-cycles (refer to Section 2.4.1). This is shown in Figure 4.9, that an inference-cycle may be longer or equal to a scan-cycle. The decision-maker is invoked at the end of every inference-cycle. Decisions improve as more time is given to an inference-cycle.

When a final decision is required as indicated by the event-indicator the decision-maker establishes a final solution based on the current performance of the PIPs. It takes from the blackboard a copy of the CF table and selects the partial solution with the highest CF value to be the final solution.

The data interface is a separate issue from the rest of AKShell. It links the expert system to the outside world. Its main functions include receiving data from, and routing decisions made by the expert system to, the outside world; and indicating the start of a new event. It runs independently of the rest of AKShell and synchronizes to its own timing mechanism.

The design of this interface is *ad hoc* and depends on the environment in which the expert system is applied. It is best to regard the data interface as a functional block that is customized for every application. For example, in a process control environment where data are obtained from programmable logic controllers (PLC's),

we will build an interface to read data from the particular makes of PLC's used. If the expert system directly controls the physical plant, this interface will also need to talk to the actuators.

Queuing of data, validity checking, ling of events and data history maintenance are subsidiary functions of the data interface. As data come in, they are checked for out-of-range errors, and validity time errors (e.g., time-stamping [20]). Data are kept in buffers and are *polled* as they are needed. This means that it is up to the internal AKShell tasks and the custom-tasks that drive output devices to poll the interface for data. Similarly, it is up to the decision-maker to write data to the interface whenever they become available.

The data interface will signal that a new event is detected if the new data changes by a predetermined threshold value. This signal is used by the event-indicator for task scheduling purposes.

4.5 Intertask communication

In a distributed system, message transmission delay is not negligible compared to the time between events in a single process [16], inter-process communication is therefore best kept to a minimum. Nodal activities are self-directed and each PIP uses its local estimate of the state of environment to control its processing. The only times interprocess communication takes places are:

- when an inference engine elicits a piece of information from the knowledge base;
- when an inference engine updates its confidence factor in the blackboard;
- when the decision-maker consults the CF table; and
- at the signalling of a new event.

In keeping with maintaining a structured and flexible design, communication between the component tasks is based on an *object-oriented* concept [27]: The internal AKShell tasks are treated as independent objects. When a task needs to communicate with another, information is sent via a piped message rather than using global variables. The message *pipes* can be read-only, read-and-write or write-only.

Table 4.1: Summary of the AKShell blackboard

Contents	structure
CF table	table of CF values
level-one knowledge base	production rules
level-two knowledge base	semantic trees, nodes
ev[illegible] indicator	event signals

We see from above that inter-process communication always takes place between the blackboard and another task, making the blackboard the central point of communication. Thus, each PIP will subscribe to a subset of the knowledge base by creating a read-only pipe to the blackboard and receive information via this pipe; it will create a write-only pipe to the blackboard CF table, and send updates of its CF through this pipe. Likewise, the decision-maker will create a read-only pipe to read off the CF table in the blackboard, and the data interface a read-and-write pipe to send data and event-signal, and to read the decision.

Finally, the salient features of AKShell are tabulated: the blackboard is summarized in Table 4.1, the inference engines in Table 4.2, the decision-maker in Table 4.3, and the data interface in Table 4.4.

4.6 Explanation feature

The explanation feature that has come to be accepted as standard component of conventional expert systems is omitted in this architecture. We reason that since the expert system must run in real-time, it will not be interrupted by the user at arbitrary points to explain its reasoning. Our aim in the design has been to trim the system down to the essentials in order to improve system response. The omission of an explanation function effectively relieves the processor from unnecessary overhead.

However, for debugging, development and maintenance purposes, it is important for the system to keep a record of its activities. Even though the explanation function had to be sacrificed, an independent module could be implemented to monitor and

Table 4.2: Summary of the inference engines

	description	input	output
level-one	– monotonic – data-driven – pattern-directed – sequential – cannot be interrupted	environment data	partial solutions
level-two	– monotonic – goal-driven – breadth-first search – node-based – demon-functions – progressive reasoning – can be interrupted	a partial solution from level one	confidence measure in the partial solution (CF)

Table 4.3: Summary of the decision-maker

	description
function	selects final decision based on the CF aggregate of the PIPs
input	CF values of the PIPs from the CF table
output	pointer to the partial solution with the highest CF value

Table 4.4: Summary of the data interface

function	interfaces AKShell to the outside world
internal features	data queuing (buffers) maintains data history checks time-validity of data checks against range errors signals new events
input from environment	environment variable values
output to environment	data to drive output devices
output to AKShell	variable values internal to AKShell; values can be absolute or qualitative; event signals
input from AKShell	decisions made by the expert system

log the reasoning process for later examination.

4.7 Analysis of the design

It was stated in Section 2 that in order to provide real-time response, the expert system must have an answer before time-out, even if this answer only confirms that no satisfactory solution can be given. In order to achieve this, the system must have at least generated the competing partial solutions by time-out. From these, it can then recommend, based on the initial certainty factors assigned to them, that either some of them can be tentatively accepted (because their CF values are greater than certain pre-defined threshold) or, none of them can be believed. Otherwise the system has no way of justifying its findings. Therefore this architecture cannot handle a response requirement quicker than the time taken to initiate partial solutions. We define this time period to the *critical response time* of the system. This is the time required by the system to complete the level-one inference process.

Critical response times vary with application and is primarily dependent on the implemented number of alternatives (hypotheses), and the number of preconditions

embedded in each alternative. Since level-one inference must examine every one of these in order to spawn partial solutions.

For systems that operate around the critical response time area, i.e., when scan-cycle equals the critical response time, decision making must rely solely on the confidence indicator (the initial values of CF's) produced by the level-one inference process. It is thus imperative that the inference engine and the knowledge base at level-one can perform when necessary, as a stand-alone system and produce reasonable results.

Real-time performance is therefore, guaranteed for worst-case response time requirements longer than the critical response time of the system. Because this is the bottle-neck in the system, care must be taken in the design, both of the inference engine and the knowledge representation at this level, to minimize the time overhead.

Once the second level inference process is initiated, the expert system can be stopped at *any time* to give the interim decision. The final solution could be the result of a completed inference, or it could be the result of the best effort in the allotted time. The quality of the solution is proportional to the time spent in finding the solution [32]. In this light, problem-solving by AKShell is viewed as a an incremental, opportunistic and asynchronous process.

Conspicuously absent in our design is the explanation feature that has come to be accepted as a standard component of conventional expert systems. We argue that since the expert system must run in real-time, it will not be interrupted by the user at arbitrary points to explain its reasoning. Our aim in the design has been to trim the system down to the essentials in order to improve system response. The omission of an explanation function effectively relieves the processor of an overhead.

However, for debugging, development and maintenance purposes, it may be desirable for the system to keep a record of its activities. Even though the explanation function had to be sacrificed, an independent module can be implemented to monitor and log the reasoning process for later examination.

Chapter 5

Implementation issues

The most important issue raised in the implementation of a real-time system is the question of efficiency: a real-time system has to make maximum use of time because time is a strictly limited resource. Algorithmic and architectural designs alone will not satisfy real-time requirements. A versatile and efficient programming language and an equally competent operating system are just as important in achieving real-time response.

There are several traditional *de facto* AI programming languages, among which, LISP and PROLOG enjoy the widest popularity: LISP for its list-processing ability and PROLOG for its built-in predicate logic structure, dynamic database management and backtracking facilities. In general, these AI languages were invented for research purposes, with the intention of illustrating certain aspects of AI programming (e.g. predicate logic in PROLOG) rather than for execution efficiency, e.g., the structure of PROLOG is based on predicate logic. Although AI concepts are more easily expressed in traditional AI languages, the trade-off involves a large computational overhead due to the prerequisite high level of abstraction. Execution speed and memory usage of programs written in these languages are unacceptable for real-time applications.

Fortunately AI techniques exist at a conceptual level and do not depend on particular implementations. These techniques can all be implemented in a general-purpose, procedural language such as C [31]. C was chosen to implement AKShell mainly for the following reasons:

- it is flexible enough for the implementation of functions ranging from low, machine-dependent levels to high levels of abstraction;

- C is available for a wide range of machines;
- C compilers consistently produce fast and efficient executable codes;
- the language is very popular among users and programmers alike, and highly portable between different systems down to the hardware level;
- a lot of existing software are written in C which makes interfacing with existing application programs easy — this is important if we want to integrate AI techniques in existing applications.

Portability is important because the design of AKShell is largely conceptual, and should not be machine-dependent. We would like AKShell to be implemented in different computer systems in order to rigourously evaluate the merit of the design. Just about every computer system, from personal micros to mainframes and transputers, supports a C compiler. Most of these compilers produce portable codes according to a generally accepted ANSI[1] standard.

A multitasking capability is a fundamental requirement of AKShell (to support the execution of the PIP.'s). With the obvious exception of transputers, which are intrinsically parallel machines, most of the contemporary computers use single processors and are largely serial machines. However, with proper management at the operating system level, concurrency can be simulated on a single-processor machine by time-slicing the processor between multiple tasks.

The prototype of AKShell was implemented on an IBM AT compatible machine. For this range of computers, multitasking operating systems are numerous, with varying degrees of sophistication.

From the design point of view, the basic support we require at the operating system level includes

- spawning concurrent tasks;
- suspending and killing running tasks;
- context switching and task scheduling — this refers to the actual sharing of the processor time;

[1]Draft Proposed American National Standard — Programming Language C.

- task synchronization — this refers to co-ordinating the parallel tasks that wish to access common data or resources [19];
- inter-task communication — this is usually done by creating file-like pipes [6], or via a globally defined message queue [7].

All or most of these requirements can be found in various commercially available operating systems. To name but a few, there are OS/2 from Microsoft Corporation, FLEXOS from Digital Research Inc., PC-MOS from The Software Link Inc., and executive shells that run on top of existing (conventional) operating systems like MS-DOS. The first versions of AKShell were developed under FLEXOS, version 1.3. FLEXOS is a full-blown multitasking operating system and provides the above supports plus a real-time kernel that supports real-time applications [6].

Using an off-the-shelf package means that we don't have to re-invent the wheel and write code that already exists. Unfortunately, it also means we have to limit our program designs within the provision and the constraints of the package. This is particularly apparent if the package was designed to be as general as possible. Thus, although FLEXOS answers all our basic requirements, it tends to be unwieldy to customize. Also, a large, full-fledged operating system like FLEXOS involves a large overhead in terms of memory requirement and internal system management. It was felt that AKShell's performance when running under such an operating system would not be a true reflection of the design. Unless a thorough study of the operating system is carried out, it is difficult to discern in the performance analysis exactly which elements are specific to the design, and which are dependent on the operating system used.

It was for these reasons that we decided to use a system executive developed with the Electrical Engineering Department [19] for the experimental implementation of AKShell. This system executive runs under MS-DOS and provides all the required features, yet it is small enough to impose only a low overhead. Also, the source codes of this system are available. Should alterations at the operating system level become necessary, they can be carried out with minimal effort. Experience gained in building the prototype allows more detailed specification of AKShell's operating system requirements.

Table 5.1: AKShell program files

implementation	file name	content
system header	AKSHELL.H	data types definitions and function prototypes.
	ERRMSG.H	define error return codes.
	USRFUNC.H	macros for implementing user-functions
	DATA_IF.H	macros for building data-interface funtions
system programs	AKSHELL.C	AKShell system functions, including `sys_akshell()`.
	INF1.C	Level-one functions.
	INF2.C	Level-two functions.
	DECISION.C	decision-maker functions
user programs	DATA_IF.C	data-interface related functions, including `data_if()`.
	USRFUNC.C	user-defined functions for use in knowledge base implementation.

5.1 Overview of the design

We have adopted a modular approach in the implementation of AKShell, adhering to the design decisions arrived at in Section 4. Each module is incorporated in a separate C file as summarized in Table 5.1.

In this chapter, we first give a brief introduction to the system executive, then we present the actual implementation in terms of the data structures and the various modules that make up this expert system shell.

Table 5.2: X.LIBtask control functions

function name	description
`x_thread()`	creates a task execution thread
`x_yield()`	currently executing task yields control to the system executive
`x_terminate()`	removes a task from the system

5.2 System executive

The system executive was designed as a programming tool for research in the area of distributed computing [19]. It recognizes the need for computer programs to respond timely and appropriately to external asynchronous events. The executive was implemented as a set of C functions contained in a C library called X.LIB, and a collection of header files that interfaces with the library functions. The executive effectively extends the C language to support concurrency.

No assembly language programming was used in the implementation of the executive, and most of the source code was written in ANSI C under Turbo C [36]. The library is therefore highly portable and complies with our portability requirements.

The executive supports non-preemptive round-robin task-scheduling. Each task consists of a task control data block, a private stack area and a pointer to the function that represents the task. This allows all the function codes to be *re-entrant*, implying that several identical tasks can be spawned and coexist with each other. This feature is essential in our implementation of concurrent inference engines at the second level in the AKShell architecture.

Concurrent tasks are spawned and co-ordinated by the system executive. The task control functions are provided in X.LIB. A description of these functions is summarized in Table 5.2.

Table 5.3: AKShell system tasks

task name	description
`data_if()`	Data-interface — input data are collected and put on the blackboard.
`sys_AKShell()`	This task maintains the system blackboard and continuously refreshes system constants (e.g. *bb_num_PIPs* — maximum number of active level-two inference processes, or PIP.'s.).
`inf1()`	Level-one inference engine.
`inf2()`	Level-two inference process; this task coordinates the execution of the inference engines at level-two (PIP.'s).
`PIP_`n`()`	Level-two inference engines, where $n = 0 \ldots$ *bb_num_PIPs* -1.
`decision_maker()`	Makes decisions based on the results of the inference engines.

5.3 AKShell tasks

Several concurrent tasks are spawned on initiating AKShell. This is done by asserting the appropriate function calls in the X.LIB procedure `x_initialise_user()` [19]. A brief description of the AKShell tasks thus spawned are given in Table 5.3.

Intertask communication is supported by a distributed blackboard structure that stores a defined set of state messages slots called *time-frames*. Any number of tasks can subscribe to any number of time-frames, which implies that each task can decide for itself which part of the blackboard to attach for its local usage. This blackboard concept coincides well with the architectural design of AKShell (Section 4.4); and the time-frame structure provides facilities for time-stamping and data validity checks, important in real-time applications (Section 2).

Intertask communication primitives as implemented in X.LIB can essentially be summarized under the following four categories:

1. Definition statements — these define the identification and type of a state-variable;

2. Naming statements — these establish the ownership of a state-variable;

3. Input/output statements — provide facilities for the subscriber tasks to read from and write to, state-variables;

4. General timing statements — these provide various timing primitives such as waiting for a certain time delay before carrying out the next task.

The scheduling and control of the spawned tasks are performed by a background system manager. Timing and synchronization of task scheduling is based on the hardware clock. Spawned like another task, the system manager provides the following functions:

- Checking for keyboard input and routing the input to the appropriate virtual window pertaining to an active task;
- Switching of active task windows;
- System status utilities, for system level maintenance;
- System clock maintenance — the clock resolution can be changed by the user to any multiple of clock ticks, where one second is equivalent to 18.2 clock ticks on an IBM personal computer.

Since this is a non-preemptive system, the onus is on each individual task to yield control to the executive, so that the next task may be carried out. Although this implies that some of the operating system-level responsibilities must be borne by the user program, the flexibility at the operating system-level lends us an opportunity to fine-tune AKShell performance-wise, and to specify exactly its requirements from an operating system.

5.4 AKShell blackboard

The AKShell blackboard was implemented under the blackboard system available in the system executive. Conceptually a blackboard is a global data region to which all tasks have access rights [25].

Table 5.4: AKShell blackboard state variables

owner	name	description
data_if()	bb_num_data	number of input variables
	bb_data_period	validity period of input data
	bb_data_spc	pointer to input data buffer
	bb_new_event_flag	signals a new event
sys_akshell()	bb_num_lvl1	number of level-one rules
	bb_num_lvl2	number of level-two specialists
	bb_num_PIPs	maximum number of PIP.'s to spawn
	bb_PIPs	record of the PIP.'s
	bb_lvl1_rules	pointer to level-one rules
	bb_k_tree	pointer to level-two knowledge trees
	bb_cf_threshold	a PIP. is spawned if its current CF is greater than this threshold
inf1()	bb_lvl1_flag	level-one status (active/suspended)
inf2()	bb_lvl2_flag	level-two status (active/suspended)

Under the system executive, a path is associated with the definition of a state-variable [9]. A path consists of the system-ID, group-ID and the owner-ID, and in effect, identifies the ownership of the state-variable. Thus the following statement declares the integer variable 'data' as owned by the task 'interface', of the group 'tasks' that belongs to the 'AKShell' system.

```
state_variable(int, data);
set_path(AKShell, tasks, interface);
   time_frame(data, d);
```

Table 5.4 gives a brief description of the AKShell state-variables and identifies their owner tasks.

The state-variable time-frames exist in the global blackboard as specified in our design and supported by the system executive. Although the system executive allows any number of tasks to subscribe (with default read and write rights) to any time-

```
#define TRUE 1                    /* boolean true */
#define FALSE 0                   /* boolean false */

typedef unsigned char BOOLEAN;    /* boolean type */
typedef char QUALITY;             /* data quality type */
typedef char VAR_NAME[10];        /* variable identifier string */
typedef char FUNC_NAME[40];       /* function identifier */
typedef double real_t;            /* real number type */
```

Figure 5.1: AKShell-defined data types

frame, we have enforced in our implementation a one-reader-many-writer blackboard system such that only the owner of a state-variable may write to it, but any task may read from it.

5.5 Data structures

Four main (state-variable) data structures are implemented in AKShell: input data, the pattern-directed production rules of level-one, and a record to record the progress of the PIP.'s are all held in one-dimensional arrays, and a tree structure is used to maintain the level-two knowledge base. In addition, several new data types were defined for the implementation of these data structures. These are shown in Figure 5.1.

The record type for input data is defined by the structure `INP_DATA` (Figure 5.2). The input data record contains the name of the input variable, a slot for the qualitative value, a slot for the actual value, a flag that indicates whether the value is valid, and the lower and higher range of the variable. Of these, only the name of the variable and its lower and upper range values are specified by the user in the knowledge declaration file. Depending on the actual value read at run-time, if the data value falls within the upper one-third of the data range then its qualitative value is 'high'; 'normal' if the value falls within the middle one-third; and 'low' if it falls within the lower one-third.

For a system requiring x data elements, an x-element array of `INP_DATA` records is created. At run-time, the data-interface checks the following two conditions in order to decide whether a given data element is valid:

- If the data are time-valid (by checking its time-stamp against the system clock).

```
/*-----------------------------------*
   data type definition:
   - data range:
      range[0] = lower limit
      range[1] = higher limit
   -- data quality
      LOW_VAL = bottom 1/3 of range
      NORM_VAL = middle 1/3 of range
      HIGH_VAL = top 1/3 of range
 *-----------------------------------*/
#define LOW_LIM range[0]
#define HIGH_LIM range[1]
#define LOW_VAL 0
#define NORM_VAL 1
#define HIGH_VAL 2

typedef
struct {
   char name[10];
   real_t *value;        /* data address */
   BOOLEAN validity;     /* data validity */
   QUALITY quality;      /* data quality */
   real_t range[2];
} INP_DATA;
```

Figure 5.2: Type definition of input data structure

- If the data value falls within the specified range.

Only if both conditions are met will the given data be flagged valid. This validity verification is performed every time new data are received.

Level-two knowledge base is built on the record structure K_TREE (Figure 5.3). The address of the root of a semantic tree (Section 4.2.2) is held in a corresponding K_TREE record such that an array of m K_TREE records is created for a knowledge base containing m islands of expertise.

A semantic tree structure is built by chaining the knowledge nodes pertinent to the specialist knowledge represented by the tree. These nodes are represented by the NODE record structure. Each node is identified by a name, the CF weights of the node, its location in the tree (with reference to the 'sibling' and 'offspring' node pointers), and a list of predicate conditions pertaining to the node.

Predicate conditions are represented by a linked list of PREDICATE records. Each conditional expression, e.g. $A \leq B$, corresponds to a user-defined demon function (or procedural attachment) such that for the same example, the user would have written a function `leq()` that takes input variable value A and reference value B as

```
/*-------------------------*
   semantic tree structure
 *-------------------------*/

typedef
struct predicate {
   FUNC_NAME usr_func_name;
   BOOLEAN (*usr_func)(int *, real_t *, int, IMP_DATA *);
   int num_param;              /* parameter count */
   int *param_index;           /* data index (of data array) */
   real_t *ref_value;          /* reference value */
   struct predicate *next;
} PREDICATE;

typedef
struct node {
   VAR_NAME name;              /* node name */
   PREDICATE *assumption;      /* predicate list */
   real_t cf[2];               /* confidence factors */
   struct node *sibling;       /* horizontal link */
   struct node *offspring;     /* vertical link */
} NODE;

typedef
struct {
   FUNC_NAME specialist_name;
   NODE *root;
   FUNC_NAME cf_func_name;
   real_t (*cf_mode)(real_t, real_t);     /* user-defined cf function */
}  K_TREE;
```

Figure 5.3: Type definition of semantic tree structure

```
/* PIP records */
typedef
struct {
   int index;           /* K_TREE index */
   real_t cf;           /* record of confidence factor */
   BOOLEAN attached;    /* whether attached to an inf. eng. */
} PIP_REC;
```

Figure 5.4: Type definition of PIP. records

```
/*-----------------------------------------*
   Level 1 rules: the initial confidence
   is defined as the percentage of matched
   symptoms weighted by a user-defined value.
 *-----------------------------------------*/

typedef
struct {
   int specialist_index;   /* index to k_tree entry */
   int *symptoms;          /* array of causal facts */
   QUALITY *quality;       /* their values */
   int num_symptoms;       /* number of relevant symptoms */
   real_t cf;              /* confidence weight */
} DECISION_LIST;
```

Figure 5.5: Type definition of level-one rules

parameter, and returns the result ('true' or 'false') of the comparison.

The progress of the PIP.'s are recorded by an array of PIP_REC records. These PIP. records correspond to the CF table discussed in Section 4.4. A PIP_REC record is assigned to each specialist on the level-two knowledge tree. When a PIP. task is spawned, the PIP_REC record corresponding to the chosen specialist. The information contained in a PIP_REC record is shown in Figure 5.4.

The record structure DECISION_LIST is used to hold the level-one rules. A level-one rule consists of a fixed pattern, or template of qualitative values that are matched against input data. This template is stored in the array pair symptoms and quality, where symptoms indicates which data are being compared, and quality stores their corresponding template values.

Each level-one rule is associated with a level-two specialist, that is, as a rule is tested, the outcome of the test results in a certain degree of confidence toward

the associated level-two goal. The confidence weight of a particular level-one rule is stored in the record element CF and the location of the specialist is stored in specialist_index.

The bulk of the data type definition is declared in the header file 'AKShell.h' as listed in Appendix A.

5.6 Knowledge compiler

A knowledge compiler was implemented to compile the user-written knowledge file and enter the declared information into the AKShell blackboard. The source listing of this compiler is shown in Appendix A. The error codes and error messages generated on error by the compiler are summarized in Appendix B.

Knowledge compilation is divided into three parts: data declaration and level-two and level-one knowledge base declarations.

Data declaration is preceded by the keyword BEGIN_DATA:, and terminated by END_DATA:. Each data item is preceded by the keyword DATA:. Two types of data can be declared:

1. Interfaced data. The data-interface task makes available a list of data that can be read from the environment. If the user wants to use a particular interfaced data, it has to be declared by the keyword DATA:, the data identifier, and the lower and upper legal limits of the data value.

2. User constant. If needed, the user can declare a constant value to be referred to in the knowledge declaration. This is similarly declared by preceding the data identifier with the keyword DATA:, followed by the constant value.

As an example, the declaration of the variable control_signal that ranges from 0 to 10 volts, and a constant PI, is shown in Figure 5.6.

Level-two declaration is enclosed by the keywords BEGIN_LVL2: and END_LVL2:. Each level-two specialist is uniquely identified by the name of its goal (corresponding to the specialist_name of the K_TREE structure), declared after the keyword DECISION:.

The method by which CF is updated is declared by the keyword CF_MODE: and the CF-update function name. CF-update functions are user-defined functions and must

```
BEGIN_DATA:
   DATA: control_signal 0 10
   DATA: PI 3.141592654
      :
      :     ; other variables/constants
      :
END_DATA:
```

Figure 5.6: Example data declaration

have been pre-compiled and linked to AKShell before they can be called (otherwise the knowledge compiler will return an 'undefined function' error).

The nodes are declared immediately after the declaration of the specialist's goal and the CF-update mode. The information defining a node are categorized under the following keywords and their respective arguments (in the order of their appearance in the user-file):

- 'NODE: *identifier string*' — this identifies the node by a name
- 'CF_WEIGHT: *success weight, failure weight*' — this defines the confidence weight of failure and of success as discussed in Section 4.2.2;
- 'PARENT: *node identifier*' — the location of the node with respect to the semantic tree is specified by its parent node. If the parent of a node is the root of a tree, then the root identifier is specified in the argument;
- 'PREDICATE: *declaration of predicate conditions*' — defines the predicate conditions pertaining to the node.

A node can have any number of predicates. Each predicate definition consists of the following keyword-argument pairs:

- 'PREDICATE: *predicate function identifier*' — designates a user predicate function;
- 'PARAM: *data identifier list*' — defines the input parameter to the function;
- 'REF_VAL: *reference value list*' — defines the values of the reference parameter.

The predicate functions are user-written and constitute the *procedural knowledge* component of the AKShell design. Thus, a predicate function can be simple logic

```
/*-----------------------------------------*
   prototype of user-functions
   parameters:-
      inp: input data list
      ref: list of reference values
      n: number of input data/ref values
      p: memory address where data is kept
   returns TRUE or FALSE
 *-----------------------------------------*/
BOOLEAN (*func)(int *inp, real_t *ref, int n, INP_DATA *p);

/*-----------------------------------*
   prototype of CF-update functions
   parameters:-
      old_cf: current CF value
      cf_wgt: CF weight to apply
   returns the new CF value
 *-----------------------------------*/
real_t (*func)(real_t old_cf, real_t cf_wgt);
```

Figure 5.7: Prototype of user-defined functions

functions to complex, application-specific functions. As an example, for the `leq()` function mentioned in Section 5.5, we would declare the predicate statement

if $A \leq 20$

as

```
PREDICATE: leq
   PARAM: A
   REF_VAL: 20
```

As is the case for user-written CF-update functions, the predicate functions need to be pre-compiled and linked to AKShell before they can be called. Mixed programming environments are allowed, i.e. the user functions (CF-update and predicate) can be written in any language, as long as they can be compiled into object codes and linked with the existing AKShell objects [36]. Expressed in C the function prototype of the user-defined functions is shown in Figure 5.7

The header file 'usrfunc.h' (shown in Appendix A) contains macros that can be used for the implementation of user-functions. If the user-functions are written in C this should be included to simplify implementation.

To improve readability of the user knowledge file, comments are allowed by

```
state_variable(int, bb_num_data);
state_variable(INP_DATA *, bb_data_spc);
state_variable(int, bb_data_period);
state_variable(BOOLEAN, bb_new_event_flag);

initialise_blackboard;
initialise_time;

set_path(system_1, data_if, data_if);
   time_frame(bb_num_data, d)
   time_frame(bb_data_spc, p);
   time_frame(bb_data_period, d);
   time_frame(bb_new_event_flag, b);
```

Figure 5.8: Attach data-interface to blackboard

preceding them with a semicolon ';'. When the compiler encounters the semicolon, it will ignore the rest of the line and resume compiling on the next line.

The syntax diagrams defining the syntax rules for writing user-knowledge files is shown in Appendix E. The knowledge file used in the simulation of buffer-tank is listed in Appendix D.

5.7 Data-interface

As its name implies, the data-interface collects data from external sources and converts them into the AKShell-usable format. Because of its designation, the implementation details of the data-interface are *ad hoc* and application-dependent. But the overall design can be defined here in general terms.

First the data-interface is attached to the blackboard. The state-variables pathed (belonging) to the data interface are listed in Table 5.4. The code fragment attaching the data-interface to the blackboard is shown in Figure 5.8.

The calls to the X.LIB macro functions

`state_variable(`*data type*, *identifier*`);`

declares the state-variables, and the other two macro calls

```
initialise_blackboard;
initialise_time;
```

```
Process data-interface (input, output);

Variables:
   input variable: environment data;
   output variable: event flag;          /* signals new event */
                    data information;    /* data value, quality, time- */
                                         /* stamp and range-validity */
   local variable: threshold;            /* new event threshold */
Begin
   Repeat for ever
      Reset event flag;                  /* no new event, yet */
      Read input data;
      Time-stamp data;
      Check value range-validity;
      If data is valid then
         If difference between new and old data > threshold then
            Set event flag;
         If event flag is set then
            determine data quality;      /* high/normal/low */
            output event flag;
            output data information to blackboard;

      Yield to system executive;         /* non-preemptive yield */
   End Repeat;
End.
```

Figure 5.9: Data-interface algorithm

are standard X.LIB calls to initialize the blackboard and the system timer, respectively.

The bulk of the data-interface resides in a 'repeat-for-ever' loop that executes every time the round-robin yields to it. The actual interface functions are implemented in this loop. Essentially, the data-interface reads in data, processes the data by time-stamping the data and checking that it is within the legal (declared) value range, then sends the new data to the blackboard. The event flag is raised every time new data is read, to signal that a new event is detected. The general functions of the data-interface task are described by the pseudo-code in Figure 5.9.

The pseudo-code indicates that the state 'new event detected' is signalled whenever the newly read data differs from the old data by a certain threshold. The threshold is again, user-defined.

The source code listing, 'dataif.c' given Appendix A is an example writtten for interfacing a buffer-tank simulation to AKShell.

5.8 Event-indicator

The function of the event-indicator is to synchronize the system tasks based on the occurrence of new events. It was conceptually designed as a concurrently executing module in Section 4. Since Lun's system executive is non-preemptive and requires the concurrent tasks to yield by themselves in order to complete the round-robin, an explicit task that co-ordinates and synchronizes the other tasks is not practical. It is therefore not implemented as an explicit task. Instead, system status flags are used to signal the system states. Based on these system flags, the tasks can synchronize themselves accordingly.

These system flags are boolean variables posted to the blackboard. Each flag belongs to a particular task, and it is the responsibility of the owner task to set or reset its own flag. There are three such flags:

- `bb_new_event_flag`: belonging to the data-interface, this flag is set when new event is detected.
- `bb_lvl1_flag`: belonging to the level-one inference engine, this flag is set when the level-one inference engine is active, and reset when the inference engine suspends itself.
- `bb_lvl2_flag`: similar to `bb_lvl1_flag`, this flag belongs to the level-two inference process, and is set or reset depending on whether the level-two inference engines are active or suspended.

Synchronization is based on *edge-sensitive triggering*, which means that the tasks are synchronized according to the instant when the flags are set or reset. As the timing diagram given in Figure 5.10 shows, the rising edge of `bb_new_event_flag` will trigger the level-one inference engine to set its flag on. The level-two inference process becomes active on the falling edge of `bb_lvl1_flag`, and immediately suspends itself on the rising edge of `bb_new_event_flag`. That is, the level-two inference process either terminates naturally, or is suspended as soon as a new event is detected. This is in accordance with our design that the expert system will reason for as long as time permits.

The `bb_lvl2_flag` cycles shown in Figure 5.10 depicts the level-two inference

A	bb_new_event_flag
B	bb_lvl1_flag
C	bb_lvl2_flag

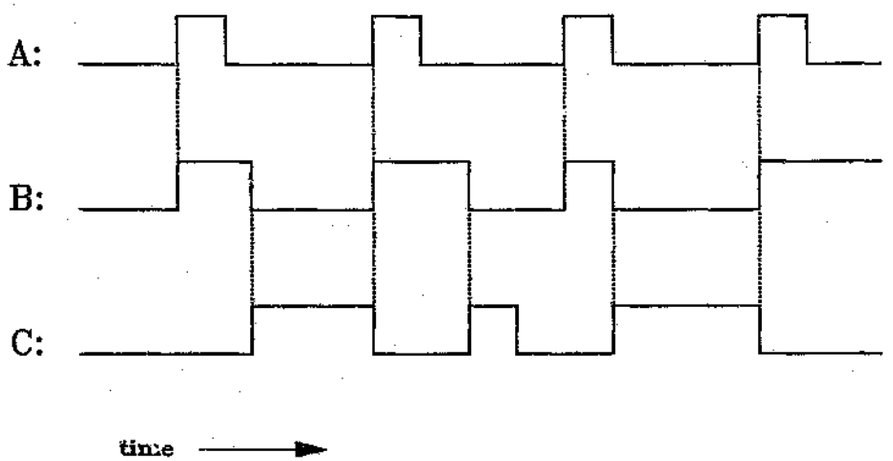


Figure 5.10: Timing diagram of the system flags

engine terminating due to a new event taking place in the first and third cycle, and naturally in the second cycle.

Finally, the decision-maker (`decision_maker()`) synchronizes itself to the falling edge of `bb_lvl2_flag`, which means that the decision-maker becomes active as soon as the level-two inference process is completed, or when the process is suspended due to a new event taking place. Thus, a decision is available even if the inference process did not reach natural completion.

5.9 Inference engines

The algorithmic implementation of the level-one and level-two inference engines follow closely the designs arrived at in Section 4.3. The source listing of the inference engines, viz. 'inf1.c' and 'inf2.c' are listed in Appendix A.

The level-one inference engine is activated as soon as a new event is detected (via the system flags, see Figure 5.10). As the inference engine tests each rule, it updates

the CF in the PIP. record corresponding to a particular specialist. The initial CF value is defined as the fraction of matched symptoms weighted by a user-defined confidence weight (defined in the user knowledge file), i.e.,

$$\text{initial CF} = (m/n) \times w$$

where,

m = number of symptoms matched;

n = total number of symptoms in the rule; and

w = user-defined CF weight.

As a result of the level-one inference, the CF's of the entire PIP. record array is initialized. A number of level-two inference engines, or PIP.'s then attach themselves to the PIP. records with the highest CF value. The actual number PIP.'s spawned is determined by setting the system constant, num_pips to the number required. (Refer to the listing of `AKShell.c` in Appendix A.)

The array of level-two inference engines are co-ordinated collectively by the task `inf2()`. This process owns the system flag `bb_lvl2_flag`, and so it is the responsibility of `inf2()` to start and stop the level-two inference process as is appropriate.

The PIP.'s are activated as soon as `bb_lvl2_flag` is set. Each PIP. then progressively updates the CF of the attached PIP. record as it traverses through the knowledge tree. The process stops as soon as `bb_lvl2_flag` is rese The entire inference process is then repeated on the next new event signal.

In order to ensure that all the PIP.'s get a chance to execute on the round-robin, (and hence to maintain concurrency,) the level-two inference engines are implemented in such a way that each PIP. yields to the next task as soon as it has completed a node. Thus, the time-slice resolution at level-two is the longest time it takes for a PIP. to complete reasoning with one node.

5.10 Decision-maker

Decision-making in AKShell is implemented as a concurrent task, `decision_maker()`. The source listing of the decision-maker task is shown in 'decision.c', in Appendix A.

As the timing diagram in Figure 5.10 shows, the task `decision_maker()` becomes active as soon as `bb_lvl2_flag` changes from 'on' to 'off'. At this point, the decision-

```
Process decision-maker (input, output);
   input variable:
      bb_PIPs : array of PIP_rec;  /* array of PIP records */
      bb_lvl2_flag : level-two active state flag;
   output variable:
      decision : index to final decision;
   local variable:
      max_cf : real;    /* current maximum CF value */
      i : integer;      /* looping index */

Begin
   Attach to blackboard PIP records;

   repeat for ever
      /* wait for the falling edge of the level-two flag */
      repeat until bb_lvl2_flag changes from on to off
         yield to next task;

      if data validity expired
         yield to next task;
      else
         /* reset max_cf and dec.      */
         max_cf = 0;
         decision = 0;

         /* choose PIP with maximum CF */
         for i = 1 to size of PIP array do
            /* compare cf values */
            if bb_PIPs[i].cf > max_cf then
               max_cf = bb_PIPs[i].cf;
               decision = bb_PIPs[i].index;
         End for;
      End else;
      yield to next task;
   End repeat;
End
```

Figure 5.11: Decision-maker algorithm

maker searches through the array of PIP. records for the one with the highest CF value. The goal pointed to by this PIP. record (bb_PIPs.index) is then chosen to be the final decision. A pseudo-code description of the decision-maker is shown in Figure 5.11.

5.11 Conclusion

In this chapter, we have presented the implementation details of AKShell. The component tasks of AKShell consists of

- a knowledge compiler;

- a data interface;
- level-one and level-two inference engines; and
- a decision-maker.

These tasks run concurrently under the system executive that extends MSDOS to support multitasking. Synchronization of the concurrent tasks is achieved through the use of system flags. Intertask communication is supported by the blackboard system provided by the system executive .

Chapter 6

Performance analysis

The system as implemented bases its final decisions on a set of competing partial solutions, and is capable of yielding one of these partial solutions as the final decision any time during its inference process. That is to say that the system can offer partial results as they are needed. The performance of the system can thus be seen to be independent of the timing constraints stemming from the environment. The quality of the results, however, depends on the quality of the knowledge base engineered for the application, but this is beyond the scope of this research. But we can, at this stage, conduct a performance analysis of AKShell in terms of its responsiveness with respect to timing constraints. For this purpose, a simple fault diagnosis application based on a buffer-tank simulation was used.

6.1 Experimental set-up and parameters

The buffer-tank simulation used for the performance analysis was implemented in C, and spawned in parallel to AKShell under Lun's system executive. The physical process model used represents an actual laboratory experiment. The schematic representation of this buffer-tank is shown in Figure 6.1. Based on the measurement of h, the inlet valve of the buffer-tank is controlled by a proportional-integral PI controller. To make the simulation more realistic, a noise generator that adds *Gaussian noise* [4] to the simulated h was also incorporated. The derivation of the required equations for the simulation is given in Appendix C.

The simulation is realized through three concurrent tasks. These are

cross-sectional diameter	0.2 m
max. level (h_max)	1 m
cross-sectional area (A)	0.03 m^2
control signal (u)	0 – 1 V
max. input flow (q_1)	3.1×10^{-5} m^3/s

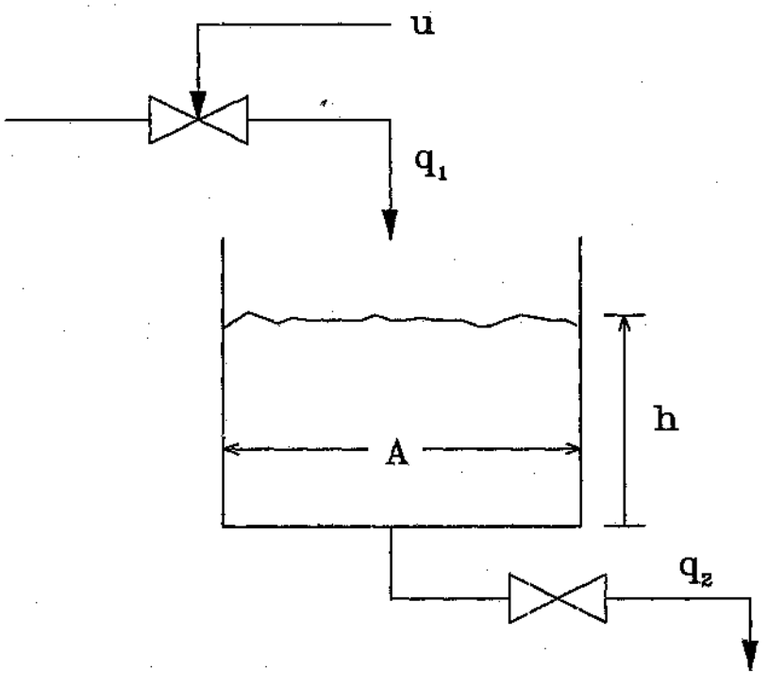


Figure 6.1: Schematic diagram of the buffer-tank

1. Discrete model of the buffer-tank, including the PI-controller.

2. Gaussian noise generator — generates tank-level noise up to a level defined by an external variable `noise_lvl`.

3. Fault-monitor — this task makes available several buffer-tank parameters. Via this task, arbitrary values may be forced on these parameters in order to simulate faults.

The simulation module generates tank data continuously which is then captured and time-stamped by the data-interface. The data-interface implemented provides the following data:

- level of the tank, h (0 – 1 m);
- control valve signal, u (0 – 1 V);
- (level) set point, sp (0 – 1 volt);
- controller parameters, i.e., the proportional gain (Kc) and the integral time constant (TI).

In addition, two historical values are provided for each of the tank parameters. Thus, the values of h, u and sp at time t, $t-1$ and $t-2$ are available.

In real life, and reflected in the simulated model, the open-loop step response of the buffer-tank is about 400 seconds. The sampling period of the discrete model, Δ is set at 10 seconds (see Appendix C. But for such a slow response, we decided that the interface need not read the buffer-tank value at every sampling point. A fairly accurate picture of the buffer-tank activities can be obtained at 5 – 10 times Δ, so we decided on a scan period of 50 seconds at the interface, i.e., the interface updates the buffer-tank data on the system blackboard at 50 second intervals.

This sampling interval of the system means that the shortest inference cycle lasts no longer than 50 seconds. This is according to the sample-and-hold technique discussed in Section 2. In order to test the system performance under more demanding time constraints, we can use the feature provided by Lun's system executive which allows the system timing to be synchronized to the system clock of the computer [19]. On an IBM PC, there are about 18.2 clock ticks per second. At the limit, we can

synchronize the system timing to every clock tick. In this instance, the scan period is effectively scaled down to about three seconds. Likewise the shortest inference cycle can be no more than three seconds.

The design of AKShell is such that the shortest inference cycle consists exclusively of a completed level-one inference process, i.e., the cycle terminates before the level-two PIP.'s are spawned. The time taken to complete this cycle was defined in Section 4.7 as the critical response time of the system. In this case, the critical response time of the expert system is three seconds — the system must have at least completed level-one inference within the first three seconds of every inference cycle.

6.2 Using the expert system

The source codes specific to the building of an expert system for this experiment are listed in Appendix C. For the purpose of this experiment, AKShell was attached to the pro edural knowledge defined in `UsrFunc.C`. The declarative part of the knowledge base is separately defined in `EXAMPLE.AKS`. This knowledge base contains knowledge designed to detect a small number of faults. The object of this knowledge base is not to provide a comprehensive tool for fault-diagnosis, but to provide a means of testing the performance of AKShell.

An IBM AT compatible machine with the following features is required to run the expert system:

- 640 kilobytes of RAM.
- One 360Kb or 1.2Mb disk-drive (to load the program).
- MS-DOS operating system.
- 80287 math-coprocessor.
- Enhanced Graphics Adaptor and monitor.

6.2.1 Starting the system

The system is started by entering the following command at the DOS prompt:

```
AKSHELL EXAMPLE.AKS
```

This command line invokes the AKShell compiler to compile the knowledge file EXAMPLE.AKS. If compilation is successful, the compiled knowledge is loaded into AKShell and the expert system is ready to operate. The user is then prompted for the following information:

1. ">> Enter number of PIPs (1 ¡= num ¡= n):" — where n is the number of level-two goals declared in the knowledge file. The user specifies here how many of these can be active concurrently. Thus, if 10 specialists exists in the knowledge base and the user specifies 5 PIP.'s, AKShell will upon initiating level-two inference, spawn up to five PIP.'s with the most promising CF's. These five PIP.'s will then compete toward the final decision.

2. ">> Enter CF threshold (0 ¡= num ¡ 1):" — The CF threshold is the minimum CF value at which a level-two goal/knowledge island will be attached to a PIP.. For example, if the CF threshold is set at 0.05, then a PIP. will not be assigned to this goal unless it receives a CF rating greater than 0.05 through level-one inference. In other words, the CF threshold sets the minimum level of confidence at which a goal will be given further consideration.

Once the number of PIP.'s and the CF threshold are established, the main screen as shown in Figure 6.2.1 will be displayed, and the data-interface will prompt the user to enter the new-event threshold and the data validity period.

The new-event threshold defines the minimum change in the interfaced data before the data-interface signals a new event taking place. Thus, the new-event flag will be set as soon as one of the tank variables interfaced to AKShell varies by more than this threshold.

The data validity period defines how long the data values remain valid in the blackboard. This period also defines the time the expert system has to make a decision, since a decision is valid only if its associated data remains valid.

6.2.2 AKShell windows

As shown in Figure 6.2.1, the main display consists of several virtual windows. These windows divide the screen into two parts: the upper two windows, namely, 'Buffer-tank' and 'Fault-monitor', belong to the simulation module; and the lower windows

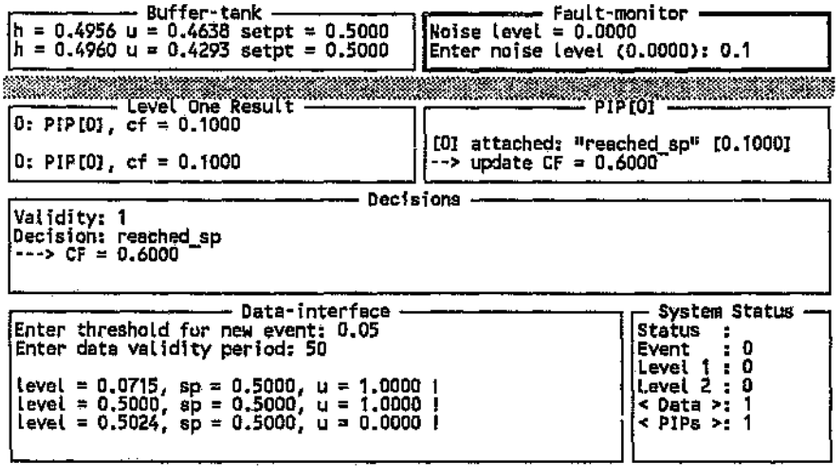


Figure 6.2: AKShell main screen

belong to the expert system. The data-interface of the expert system serves as the link between the two parts.

Each virtual window belongs to a concurrent task that needs to access the console for read and write purposes. A task can write to, and read from, only its own virtual window [19].

An active window is the window that accepts user input from the keyboard. This is denoted by the double-line border around the window. In Figure 6.2.1, the 'Fault-monitor' window is shown to be the current active window. Pressing the 'Control' key in combination with either the left ('CTRL-←') or the right arrow key ('CTRL-→') will switch the active window to the previous or the next virtual window, respectively.

With the buffer-tank window as the active window, a graphics window can be invoked by pressing the 'ALT-F10' key combination. The graphic window displays a plot of the tank level, set-point and control valve signal versus time (scan points) as shown in Figure 6.2.2. Pressing the 'ALT-F10' combination in the graphics window will restore the screen to text display mode.

With the system in text display mode, pressing the key combination 'CTRL-F1' (with any window as the active window) will invoke the system manager window. As Figure 6.2.2 shows, several operation system level functions, including status displays

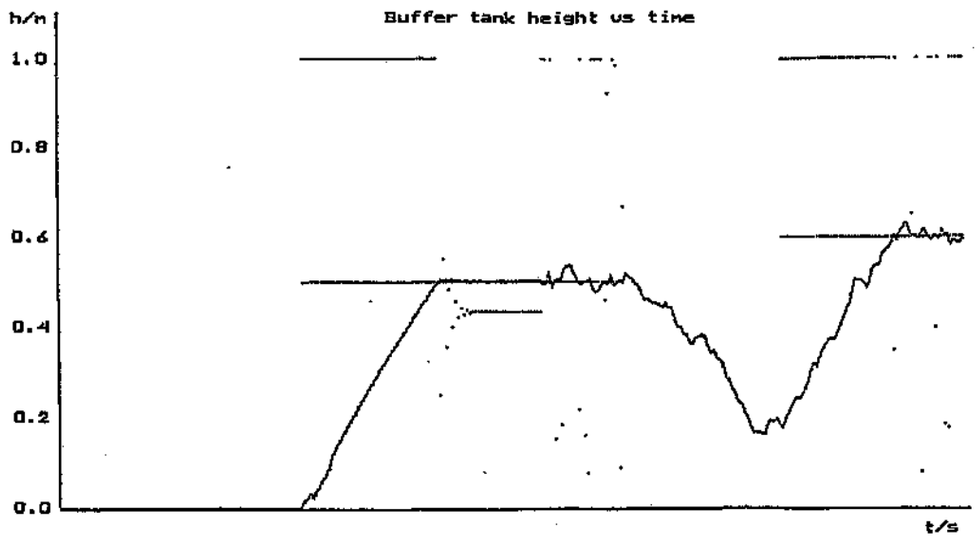


Figure 6.3: AKShell graphics window

and terminating the program, can be performed through the system manager [19]. We invoke the system manager in order to change the timing resolution (option 9 of the system manager).

The 'System Status' window is shown in the lower right of the screen. This window shows the running status of the system. The flashing symbol under 'Status' indicates that the system is going through the round-robin task switching successfully. The state of the system flags are shown under 'Event', 'Level 1' and 'Level 2'. A '1' indicates that a flag is on, '0' if it is off, and 'X' if the validity of a flag has expired. Similarly for data and PIP. status, '1' and 'X' under '< Data >' and '< PIPs >' indicates whether the validity of the respective item has expired.

The 'Data-interface' window displays the value of the tank-level (level), set-point (sp) and the value control signal (u) received by the data-interface. Based on the user-specified new event threshold, the data-interface will raise the new-event flag if the change in a received data value exceeds this threshold. An '!' appended to the end of a display indicates that a new-event was detected (this can be cross-referenced with the new-event flag in the 'System Status' window).

The progress of the inference engines are displayed in the 'Level One Results' and the 'PIP' virtual windows. 'Level One Results' shows the CF's in the PIP.'s as derived by the level-one inference engine. Each PIP. (denoted by 'PIP[n]', where n

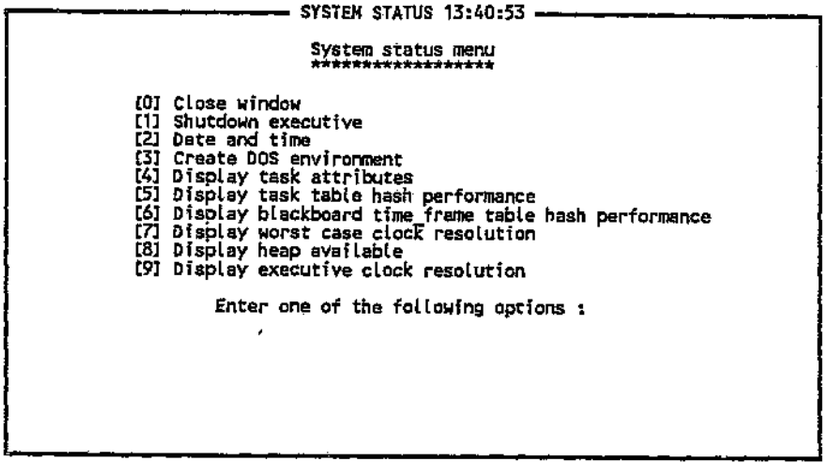


Figure 6.4: System manager window

is the index of the PIP.'s) then displays its progress in terms of the CF in its own virtual window.

The 'Decisions' window shows the decision reached by the expert system (e.g. 'reached_sp' if the set-point has been reached), the final CF in the decision, and whether the decision is still valid.

Faults can be simulated by forcing a parameter to take a user-entered value through the 'Fault Monitor' window. A parameter value is forced by pressing an appropriate function key and entering the desired value. A summary of the function key assignments is given in Table 6.1

Thus selecting function key 'F1' allows the user to change the noise level of the Gaussian noise generator, 'F2' the valve control-signal, etc., and 'F10' to force the parameters to assume the initial default value.

6.3 Experimental results

The expert system was set up using the knowledge base provided by UsrFunc.C and EXAMPLE.AKS (see Appendix A. The correct operational settings of the data validity period should be 50, since the scan period of the data-interface is set at 50, and data should be valid for the duration between scan cycles.

Table 6.1: Fault monitor function keys

function key	parameter
F1	Gaussian noise level
F2	buffer tank level (h)
F3	set-point
F4	valve-signal (u)
F5	proportional gain (Kc)
F6	integral time constant
F10	force default parameter values

First, AKShell was tested under different settings of the system clock resolution (under option 9 of the system manager). The higher the clock resolution the less time each task gets to operate, and vice versa. Thus choosing the higher clock resolutions, and therefore speeding up the system timing, can be seen as effectively 'slowing' down the computer's operating speed.

We found that even at the highest setting, when the system clock is synchronized to every clock tick (at 18.2 per second), the expert system still produced decisions at the anticipated intervals even if the available time is not sufficient to complete the inference cycle. This effectively proves that the performance of AKShell is independent of the hardware performance. In this respect, the performance of AKShell was satisfactory.

By design of AKShell, the inference engines and the decision-maker will not operate if data validity has expired. This allows us to use different settings of the data validity period to test the system performance. Data validity period is set as a multiple of the system clock resolution. For instance, if the clock resolution is set to one second, then setting data validity period to 10 means that data is valid for 10 seconds after its creation.

We found that for the experimental set-up, the shortest data validity period for which the expert system is still capable of making a decision is 5, measured at the highest clock resolution. The minimum time required to complete one inference

cycle can therefore be calculated to be:

$$5 \times \frac{1}{18.2} \approx 0.3$$

This means that the critical response time of the system is approximately 300 milliseconds, ten times within range of the three-second specification.

Finally, we tested the inference capability of the expert system. Manipulating the tank parameters through the 'Fault-monitor' window, we could simulate faults such as

- leak — by forcing the tank level to a low value.
- stuck valve — by keeping the valve control signal at some arbitrary value.
- faulty controller — by changing the values of the PI-controller constants.
- random faults — by setting the noise level to a high value (e.g. 0.5).

We found the expert system capable of producing the correct decisions as defined by the knowledge base. We therefore conclude that the performance of AKShell is satisfactory in meeting the real-time requirements as specified in this research.

Chapter 7

Conclusion

The most critical concern in designing a real-time expert system is the management of time. In this research, we have analyzed several key concepts and architectural design issues such as

- parallelism for risk reduction,
- a node structure that combines procedural and declarative knowledge to improve response,
- a multi-layered and partitioned structure that enables the system to zoom in on relevant knowledge rapidly,
- the 'competitive reasoning' technique; and
- the incremental approach to problem solving that allows the system to progressively improve the solution until the time available for reasoning is exhausted.

We have assumed an approximate model of the real-world by asserting that snap-shots (sample-and-hold data) of the world are adequate descriptions. Data consistency and validity checking then becomes a simple task and only monotonic reasoning mechanisms are needed in the system.

The main modules that make up AKShell are

- Knowledge-compiler: compiles and loads the user domain-knowledge into the shell's blackboard.
- Data-interface: interfaces the expert system to the outside world.

- Level-one and level-two inference engines.
- Decision-maker.

These modules consists of concurrent tasks spawned under the system executive that provides the system with multitasking capabilities. Intertask communication is maintained through the use of a global blackboard. A mechanism is embedded within the AKShell architecture that uses system status flags to co-ordinate task execution. The concurrent tasks are thus synchronized according to the state of the system.

The implemented shell allows the user to configure an expert system by

- attaching an appropriate data-interface for the intended application
- building a custom knowledge base for the application

The two-level architecture of AKShell supports quick-reflex, meta-level reasoning at level-one, where superficial initial hypotheses are generated; and more in-depth reasoning at level-two, where these hypotheses are subject to more in-depth analyses. At level-two, PIP.'s are assigned to prove the hypotheses concurrently. Decisions are made by resolving uncertainty in the hypothese, i.e., when a decision is requested, the hypothesis with the highest CF value will be chosen to be the final decision.

By this two-level design, a tentative decision can be made as soon as level-one inference is complete. The critical response time of the system was thus defined as the time it takes to complete level-one inference.

The critical response time is a fixed and measurable quantity, and is application and hardware dependent. The results of this investigation have shown that, for as long as the timing constraint does not require the system to produce results in less than the critical response time, the AKShell architecture is able to guarantee real-time response.

However, a guaranteed real-time response by itself does not imply satisfactory results. All that AKShell offers is the mechanism for a knowledge engineer to develop and fine-tune the knowledge contained in the expert system. A satisfactory real-time expert system is possible only when the timely response of AKShell is coupled with a well-engineered knowledge base.

7.1 Recommendations for further development

Several points were observed during the investigation which lead to the following recommendations for further research:

- When implemented using X.11B, each time a PIP. gets a chance to run, it will not yield control of the CPU until it has finished processing a node. This is a drawback of non-preemptive systems, since within a node, there may be an undesirably long loop that may upset the overall system timing. True parallel processing is possible only with a pre-emptive multitasking operating system, with incrased synchronization overhead.

- We have mentioned in the design that a significant advantage could be gained by allowing the operating system to assign higher priorities to the more promising PIP.'s. The current version of the system executive does not provide priority-based task scheduling, and hence this feature of the design was not implemented and properly investigated.

- The knowledge compiler as it is implemented allows a user to design a knowledge base in a semi-high level, declarative language. A more sophisticatd compiler is desirable to facilitate knowledge engineering. A higher level or more abstract language would result in a more intelligible and maintainable knowledge base.

- Besides the interfaced data provided by the data-interface, the current version of AKShell also allows user-defined constants in data declaration. A further development in this area would be to allow user-defined variables, as this may be useful in knowledge engineering.

- The system task synchronization is based on three system flags. It was felt during implementation that introducing more system flags (e.g., assigning one per PIP.) would improve the task coordination.

- Finally, for development of large systems, we propose that a formal communication protocol that specifies how data may be interfaced, how the blackboard is organized, and how the tasks communicate with each other should be clari-

fied. We feel that a well-designed protocol would further improve the system performance.

Bibliography

[1] Alty L.J. & Coombs M.J., *Expert Systems Concepts and Examples*, NCC Publications, 1984.

[2] Barnes J.G.P., *An Introduction to Real-time Programming*, SPL International, The Charter, England.

[3] Bennett M.E., Real-time Continuous AI systems, *IEE Proceedings*, Vol. 134, Pt. D, No. 4, July 1987.

[4] Bentley J.P., *Principles of Measurement Systems*, Construction Press, Longman Group Ltd., 1983.

[5] Duda R.O. et al, A Computer-based Consultant for Mineral Exploration, *Technical Report*, SRI International, Sep. 1979.

[6] *FlexOS Programmer's Guide*, Version 1.3, Digital Research, 1986.

[7] Foard M.F., Multitasking Methods, *PC Tech Journal*, Mar. 1986.

[8] Frey P.W., A Bit-mapped Classifier, *BYTE*, Nov. 1986.

[9] Friedman L., Reasoning by Plausible Inference, *Proc. 5th Conf. Automated Deduction*, 1981.

[10] Hayes-Roth F., The Role of Partial and Best Matches in Knowledge Systems, in: Waterman D.A. & Hayes-Roth F. ed., *Pattern-directed Inference Systems*, Academic Press, 1987.

[11] Ishida Y., Integrating Model-based and Syndrome-based Processing Plant Fault Diagnosis: A Qualitative Approach to Process Diagnosis, *Engineering Applications of AI*, Vol. 1, 1989.

[12] Jackson P., *Introduction to Expert Systems*, Addison-Wesley, 1986.

[13] Kinnucan P., Computers That Think Like Experts, *High Technology*, Jan. 1984.

[14] Kopetz H. et al, Distributed Fault-tolerant Real-time systems: The Mars Approach, *IEEE Micro*, Feb. 1989.

[15] Kornfeld W.A. & Hewitt C.E., The Scientific Community Metaphor, in: Bond A.H. & Gasser L. ed., *Readings in Distributed Artificial Intelligence*, Morgan Kaufmann Publishers, Inc., 1988.

[16] Lamport L., Time, Clocks and the Ordering of Events in a Distributed System, *Communications of the ACM*, Vol. 21, No. 7, July 1978.

[17] Lun V., MacLeod I.M. & Hanrahan H.E., *An Introduction to Expert Systems*, Dept. of Electrical Engineering, University of the Witwatersrand, 1988.

[18] Lun V. & MacLeod I.M., Distributed Knowledge Systems in Real-time Applications, *5th IFAC/IFIP Symposium on Software for Computer Control*, Johannesburg, Apr. 1988.

[19] Lun V. & MacLeod I.M., *The Development of a Microcomputer-based Non-preemptive System Executive*, Dept. of Electrical Engineering, University of the Witwatersrand, 1990.

[20] Macleod I.M., *Automatic Control by Distributed Intelligence*, Inaugural Lecture, Dept. of Electrical Engineering, University of the Witwatersrand, Aug. 1986.

[21] MacLeod I.M. & Lun V., Expert Systems In Engineering, *Elektron*, Nov./Dec. 1986.

[22] Michaelsen R.H., Michie D. & Boulanger A., The Technology of Expert Systems, *BYTE*, Apr. 1985.

[23] Minsky M., A Framework For Representing Knowledge, in: Winston P. ed., *The Psychology of Computer Vision*, McGraw-Hill, 1975.

[24] Myers W., Introduction to Expert Systems, *IEEE Expert*, Spring 1986.

[25] Nii H.P. & Brown H., Blackboard Architectures, *Sixth National Conference on Artificial Intelligence*, July 1987.

[26] Nilsson N.J., *Principles of Artificial Intelligence*, Tioga Publishing Co., 1980.

[27] Pascoe G.A., Elements of Object-oriented Programming, *BYTE*, Aug. 1986.

[28] Rainho M., *Uncertainty Management in Expert Systems*, MSc. project report, Dept. of Electrical Engineering, University of the Witwatersrand, 1988.

[29] Reddy Y.V.R., Fox M.S. & Husain N., The Knowledge-based Simulation System, *IEEE Software*, 1986.

[30] Rich E., *Artificial Intelligence*, McGraw-Hill, 1983.

[31] Schildt H., *Artificial Intelligence Using C*, Osborne McGraw-Hill, 1987.

[32] Sharma D.D. & Sridharan N.S., *Knowledge-based Real-time Control: A Parallel Processing Perspective*, FMC Corporation, Artificial Intelligence Centre, Santa Clara, California.

[33] Shortliffe E.H., *Computer-based Medical Consultations: MYCIN*, Elsevier, 1976.

[34] Smuts W.B. & MacLeod I.M., *Real-time Issues When Modelling Expert Control Supervision*, Dept. of Electrical Engineering, University of the Witwatersrand.

[35] Thompson T.F. & Clancey W.J., A Qualitative Modelling Shell for Process Diagnosis, *IEEE Software*, 1986.

[36] *Turbo C Reference Guide*, version 2.0, Borland International, 1988.

[37] Uhr L., *Multi-computer Architectures for Artificial Intelligence*, John Wiley & Son, Inc., 1987.

[38] Walker A., Knowledge systems: Principles and practice, *IBM J. Res. Develop.*, vol.30, no.1, Jan. 1986.

[39] Winston P.H., The Commercial Debut of Artificial Intelligence, in: Quinlan, J.R. ed., *Applications of Expert Systems*, Turing Press, 1987.

[40] Winston P.H., Artificial Intelligence: An Engineering Perspective, *IEE Proceedings*, vol. 134, Pt. D, No. 4, July 1987.

[41] Wright, M.L., HEXCON: A Hybrid Mircrocomputer-based Expert System for Real-time Control Applications, *IEEE Western Confererence on Knowledge-based Engineering and Expert Systems*, June 1986.

[42] Yourdon E., *Design of On-line Computer Systems*, Prentice-Hall, Inc., 1972.

Appendix A

AKShell program listing

```
/*===== AKShell.h =====*/

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

#ifndef X_TASK
#define X_TASK void
#endif

/* timing macro */
#define before(timer, delay) while (!x_timer(&timer, delay)
#define DEFAULT_PERIOD 100


/*-------------------------------*
   confidence factors:
      cf[1] = success cf (ADD_CF)
      cf[0] = failure cf (DED_CF)
 *-------------------------------*/
#define ADD_CF cf[1]
#define DED_CF cf[0]

#define FOREVER for (;;)

typedef unsigned char BOOLEAN;
typedef char QUALITY;
typedef char VAR_NAME[20];
typedef char FUNC_NAME[40];
typedef double real_t;          /* real number type */


/*-------------------------------------*
   data type definition:
   - data range:
      range[0] = lower limit
      range[1] = higher limit
   - data quality
      LOW_VAL = bottom 1/3 of range
      NORM_VAL = middle 1/3 of range
      HIGH_VAL = top 1/3 of range
 *-------------------------------------*/
#define LOW_LIM range[0]
#define HIGH_LIM range[1]
#define LOW_VAL 0
#define NORM_VAL 1
#define HIGH_VAL 2

typedef
struct {
   VAR_NAME name;
   real_t *value;       /* data value */
   BOOLEAN validity;    /* data validity */
   QUALITY quality;     /* data quality */
   real_t range[2];
} INP_DATA;


/*-------------------------*
   semantic tree structure
 *-------------------------*/

typedef
struct predicate {
   FUNC_NAME usr_func_name;
   BOOLEAN (*usr_func)(int *, real_t *, int, INP_DATA *);
   int num_param;               /* parameter count */
   int *param_index;            /* data index (of data array) */
   real_t *ref_value;           /* reference value */
   struct predicate *next;
} PREDICATE;

typedef
struct node {
   VAR_NAME name;               /* node name */
   PREDICATE *assumption;       /* predicate list */
   real_t cf[2];                /* confidence factors */
   struct node *sibling;        /* horizontal link */
   struct node *offspring;      /* vertical link */
} NODE;

typedef
struct {
   FUNC_NAME specialist_name;
```

```
   NODE *root;
   FUNC_NAME cf_func_name;
   real_t (*cf_mode)(real_t, real_t);     /* user-defined cf function */
} K_TREE;


/* record of PIP CF */

typedef
struct {
   int index;           /* K_TREE index */
   real_t cf;           /* record of confidence factor */
   BOOLEAN attached;    /* whether attached to an inf. eng. */
} PIP_REC;


/* stack for building tree */

typedef
struct node_stack {
   NODE *node;
   struct node_stack *next;
} NODE_STACK;


/*-------------------------------------------------------------------*
   Level 1 rules: the initial confidence is defined as the percentage
   of matched symptoms weighted by a user-defined value.
 *-------------------------------------------------------------------*/

typedef
struct {
   int specialist_index;                /* index to k_tree entry */
   int *symptoms;                       /* array of causal facts */
   QUALITY *quality;                    /* their values */
   int num_symptoms;                    /* number of relevant symptoms */
   real_t cf;                           /* confidence weight */
} DECISION_LIST;


/*-----------------------------------------*
   CF update function (as defined in UsrFunc.c)
 *-----------------------------------------*/
extern real_t cf_mode(real_t, real_t);

/*-----------------------------*
   [AKShell.c] ... main system
 *-----------------------------*/
extern X_TASK sys_AKShell(void);

/*-------------------------------------------------------------------*
   [KBComp.c] ... Knowledge compiler
   User knowledge file is compiled into the following AKShell
   global variables:
      extern int num_data;              : number of input data
      extern int num_lvl1;              : number of level 1 rules
      extern int num_lvl2;              : number of level 2 specialists
      extern int num_pips;              : number of available PIPs
      extern INP_DATA *data_spc;        : data space
      extern DECISION_LIST *lvl1_rules; : level one rules
      extern K_TREE *k_tree;            : level two knowledge trees
      extern real_t cf_threshold;       : threshold to activate PIPs
   Must include "usrfunc.h" for user-function & cf_mode() declarations:
      extern int usr_func_declare(void);
      extern real_t cf_mode(real_t, real_t);
 *-------------------------------------------------------------------*/
extern int kb_compile(char *);

/*-------------------------------*
   [Data_IF.c] ... Data Interface
 *-------------------------------*/
extern X_TASK data_if(void);

/*---------------------------------------*
   [DispFlag.c] ... display status flags
 *---------------------------------------*/
extern X_TASK disp_status_flags(void);

/*---------------------------------------*
   [Inf1.c] ... level-1 inference engine
 *---------------------------------------*/
extern X_TASK lvl1_inference(void);
extern X_TASK wake_PIPs(void);
extern X_TASK disp_lvl1(void);

/*---------------------------------------*
   [Inf2.c] ... level-2 inference engine
 *---------------------------------------*/
```

```
extern X_TASK inf2(void);
extern X_TASK inf2_pip(void);

/*--------------------------------*
   [Decision.c] ... Decision-maker
 *--------------------------------*/
extern X_TASK decision_maker(void);
```

```
/*===== Data_IF.h ========================*
   Macros for data-interface implementation
   Must be included after
   - stdlib.h
   - AKShell.h
   - errmsg.h
 *========================================*/

#define ONE_THIRD 0.33333333
#define TWO_THIRD 0.66666667

typedef struct {
   VAR_NAME name;
   real_t *data_loc;
} if_data_t;

extern if_data_t *if_data_list;
extern int if_data_declare(void);


/*----- if_data_declare() ------------------------------*
   Copies interface data into the interface-data list, to
   be used by kb_comp().
   - Interface data declaration:
      begin_if_data_list(x)
         define_if_data(interface_data_1, &data_1);
                .
                .
         define_if_data(interface_data_X, &data_X);
      end_if_data_list
   - Main call:
      int num;
      num = if_data_declare;
   where num is returned by this function. If successful,
   the number of interface data mapped is returned; else
   an error code is returned.
 *------------------------------------------------------*/

#define begin_if_data_list(n) \
   if_data_t *if_data_list; \
   \
   int if_data_declare(void) \
   { \
      int i = 0; \
      int num_if_data = n; \
   \
      if ((if_data_list = (if_data_t *) calloc(n, sizeof(if_data_t))) \
            == NULL) \
         return(MEM_ERR);

#define define_if_data(d, data) \
      if (i >= num_if_data) return DATA_LIST_FULL; \
      strcpy(if_data_list[i].name, #d); \
      if_data_list[i++].data_loc = &data

#define end_if_data_list() \
      return(i); \
   }  /* if_data_declare() */
```

```
/*===== Usrfunc.h ======================*
   Macros for user function declaration.
   Must be included after
   - stdlib.h
   - AKShell.h
   - errmsg.h
 *======================================*/

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

typedef
struct {
   FUNC_NAME name;
   BOOLEAN (*func)(int *, real_t *, int, INP_DATA *);
} func_rec_t;


/*----- usr_func_declare() -------------------------------*
   Copies user functions into the user function list, to
   be used by kb_comp().
   - User function declaration:
      usr_func_prototype("function name");
                  :
                  :       (x user functions)
                  :
      begin_usr_func_list(x)
         define_usr_func(user_function_1);
                  :
         define_usr_func(user_function_x);
      end_usr_func_list
   - Main call:
      int num;
      num = usr_func_declare;
   where num is returned by this function. If successful,
   the number of user functions mapped is returned; else
   an error code is returned.
 *--------------------------------------------------------*/

extern int usr_func_declare(void);

#define usr_func_prototype(f) BOOLEAN f(int *, real_t *, int, INP_DATA *)

#define begin_usr_func_list(n) \
   func_rec_t *usr_func_list; \
   \
   int \
   usr_func_declare(void) \
   { \
      int i = 0, max_func = n; \
   \
      if ((usr_func_list = (func_rec_t *) calloc(n, sizeof(func_rec_t))) \
            == NULL) \
         return(MEM_ERR);

#define define_usr_func(f) \
      if (i >= max_func) return FUNC_LIST_FULL; \
      strcpy(usr_func_list[i].name, #f); \
      usr_func_list[i++].func = f

#define end_usr_func_list \
      return(i); \
   }  /* usr_func_declare() */ \


/*-------------------------------------------------------------------------*
                  Macros for CF-function declarations
 *-------------------------------------------------------------------------*/

typedef
struct {
   FUNC_NAME name;
   real_t (*func)(real_t, real_t);
} cf_func_rec_t;


/*----- cf_func_declare() --------------------------------*
   Copies user functions into the CF function list, to
   be used by kb_comp().
   - User function declaration:
      cf_func_prototype("function name");
                  :
                  :       (x CF functions)
```

```
                :
      begin_cf_func_list(x)
         define_cf_func(cf_function_1);
                :
         define_cf_func(cf_functic )_x);
      end_cf_func_list
   - Main call:
      int num;
      num = cf_func_declare;
   where num is returned by this function. If successful,
   the number of CF functions mapped is returned; else
   an error code is returned.
 *-------------------------------------------------------*/

extern int cf_func_declare(void);

#define cf_func_prototype(f) real_t f(real_t, real_t)

#define begin_cf_func_list(n) \
   cf_func_rec_t *cf_func_list; \
   \
   int \
   cf_func_declare(void) \
   { \
      int i = 0, max_func = n; \
   \
      if ((cf_func_list = (cf_func_rec_t *) \
            calloc(n, sizeof(cf_func_rec_t))) == NULL) \
         return(MEM_ERR);

#define define_cf_func(f) \
      if (i >= max_func) return FUNC_LIST_FULL; \
      strcpy(cf_func_list[i].name, #f); \
      cf_func_list[i++].func = f

#define end_cf_func_list \
      return(i); \
   } /* cf_func_declare() */ \
```

```
/*===== AKShell.c =====*
  The calling program
 *====================*/

/* Token passing system */

#include <ctype.h>

#include <x.h>
#include <xtime.h>
#include "AKShell.h"
#include "errmsg.h"
#include "buftank.h"

/* global variables */
int num_data;                /* number of input data */
int num_lvl1;                /* number of level 1 rules */
int num_lvl2;                /* number of level 2 specialists */
int num_pips;                /* number of available PIPs */
INP_DATA *data_spc;          /* data space */
DECISION_LIST *lvl1_rules;   /* level one rules */
K_TREE *K_tree;              /* level two knowledge trees */
real_t cf_threshold;         /* threshold to activate PIPs */

/*--------------------*
  function prototypes
 *--------------------*/

void main(ir  char *[]);
X_TASK sys_/ hell(void);


void
main(int argc, char *argv[])
{
   int err;

   if (argc < 2) {
      printf("Usage: AKShell <knowledge file name>\n");
      exit(1);
   }

   clrscr();
   err = kb_compile(argv[1]);
   if (err < NO_ERR)
      exit(2);
   clrscr();

   x_initialise_system();
}  /* main() */


void
x_initialise_user(void)
{
   char pip name[80];
   char cmd_l n[20];
   int i;
   extern int num_pips;    /* number of available PIPs */

   /* system tasks */
   x_thread(sys_AKShell, "sys_AKShell", 1024, 6, "");
   x_thread(disp_status_flags, "sys_flags", 1024, 5, "");
   x_thread(lvl1_inference, "inf1", 2048, 8, "");
   x_thread(disp_lvl1, "disp_lvl1", 1024, 3, "");
   x_thread(inf2, "inf2", 1024, 5, "");

   /* spawn PIPs */
   for (i = 0; i < num_pips; i++) {
      sprintf(pip_name, "pip_%d", i);
      sprintf(cmd_line, " PIP[%d] ", i);
      x_thread(inf2_pip, pip_name, 5120, 10, cmd_line);
   }

   x_thread(decision_maker, "decisions", 1024, 6, "");

   /* simulation module */
   x_thread(fault_monitor, "fault_monitor", 1024, 10, "");
   x_thread(gauss_noise, "noise", 1024, 2, "");
   x_thread(buf_tank, "Buffer_tank", 10240, 10, "");

   /* data interface */
   x_thread(data_if, "data_if", 2048, 10, "");
}  /* x_initialise_user() */


/*----- sys_AKShell() ----------*
   AKShell system task:
```

```
   - maintains system constants
   and the knowledge base.
 *-----------------------------*/

X_TASK
sys_AKShell(void)
{
   int i;

   state_variable(int, bb_num_lvl1);
   state_variable(int, bb_num_lvl2);
   state_variable(int, bb_num_PIPs);
   state_variable(DECISION_LIST *, bb_lvl1_rules);
   state_variable(K_TREE *, bb_k_tree);
   state_variable(real_t, bb_cf_threshold);

   initialise_blackboard;
   initialise_time;

   set_path(system_1, blackboard, sys_AKShell);
      time_frame(bb_num_lvl1, d);
      time_frame(bb_num_lvl2, d);
      time_frame(bb_num_PIPs, d);
      time_frame(bb_lvl1_rules, p);
      time_frame(bb_k_tree, p);
      time_frame(bb_cf_threshold, f);

   /*-----------------------------*
      initialise system variables
    *-----------------------------*/
   bb_num_lvl1 = num_lvl1;
   bb_num_lvl2 = num_lvl2;
   bb_num_PIPs = num_pips;
   bb_lvl1_rules = lvl1_rules;
   bb_k_tree = k_tree;
   bb_cf_threshold = cf_threshold;

   /*-------------------------------------*
      continuously refresh the blackboard
    *-------------------------------------*/
   FOREVER {
      X_begin
         with_validity(DEFAULT_PERIOD) output(bb_num_lvl1);
         with_validity(DEFAULT_PERIOD) output(bb_num_lvl2);
         with_validity(DEFAULT_PERIOD) output(bb_num_PIPs);
         with_validity(DEFAULT_PERIOD) output(bb_lvl1_rules);
         with_validity(DEFAULT_PERIOD) output(bb_k_tree);
         with_validity(DEFAULT_PERIOD) output(bb_cf_threshold);
      X_end
   }
} /* sys_AKShell() */
```

```
/*===== inf1.c ============*
  level 1 inference engine
 *=========================*/

#include "x.h"
#include "xtime.h"
#include "AKShell.h"

X_TASK lvl1_inference(void);
real_t update_lvl2_cf(real_t, real_t, int, int);
void sort_to_n(PIP_REC *, int, int);
X_TASK disp_lvl1(void);
X_TASK wake_PIPs(void);


/*----- lvl1_inference() -----*
  Level one inferece engine
 *----------------------------*/

X_TASK
lvl1_inference(void)
{
   #define NUM_STATIC 4       /* number of static bb data imported */

   BOOLEAN old_event_flag = FALSE;
   int i, fact_index;
   int num_matched;
   DECISION_LIST curr_rule;
   INP_DATA curr_fact;
   real_t cf_weight;

   state_variable(int, bb_num_lvl1);
   state_variable(int, bb_num_lvl2);
   state_variable(int, bb_num_PIPs);
   state_variable(DECISION_LIST *, bb_lvl1_rules);
   state_variable(INP_DATA *, bb_data_spc);
   state_variable(BOOLEAN, bb_new_event_flag);
   state_variable(BOOLEAN, bb_lvl1_flag);
   state_variable(PIP_REC *, bb_PIPs);

   initialise_blackboard;
   initialise_time;

   set_path(system_1, blackboard, sys_AKShell);
      time_frame(bb_num_lvl1, d);
      time_frame(bb_num_lvl2, d);
      time_frame(bb_num_PIPs, d);
      time_frame(bb_lvl1_rules, p);
   set_path(system_1, data_if, data_if);
      time_frame(bb_data_spc, p);
      time_frame(bb_new_event_flag, b);
   set_path(system_1, level_1, inf1);
      time_frame(bb_lvl1_flag, b);
      time_frame(bb_PIPs, p);

   /*---------------------------------*
      initialise and attach to system
    *---------------------------------*/
   for (i = 0; i != NUM_STATIC;) {
      /* get static data */
      input(bb_num_lvl1) perform i++;
      input(bb_num_lvl2) perform i++;
      input(bb_num_PIPs) perform i++;
      input(bb_lvl1_rules) perform i++;
      if (i == NUM_STATIC) {
         /* allocate space for PIP record array */
         if ((bb_PIPs = (PIP_REC *) calloc(bb_num_lvl2, sizeof(PIP_REC)))
                  == NULL)
            abort();       /* memory error -> abort program */
      }
      else
         i = 0;
      x_yield();
   }

   /*-----------------*
      inference loop:
    *-----------------*/
   bb_lvl1_flag = FALSE;
   FOREVER {
      /*---------------------------------*
         Start a new inference cycle if
         new event flag is set.
       *---------------------------------*/
      input(bb_new_event_flag) perform {
         if (bb_new_event_flag && !old_event_flag) { /* start of new event */
            x_begin
               /* set level-one flag */
```

```
            bb_lvl1_flag = TRUE;
            with_validity(DEFAULT_PERIOD) output(bb_lvl1_flag);
         x_end

         /* reset and detach PIP records */
         for (i = 0; i < bb_num_lvl2; i++) {
            bb_PIPs[i].index = i;
            bb_PIPs[i].cf = 0;
            bb_PIPs[i].attached = FALSE;
         }

         /* qualitatively examine facts */
         input(bb_data_spc) perform {
            for (i = 0; i < bb_num_lvl1; i++) {
               curr_rule = bb_lvl1_rules[i];
               num_matched = 0;
               for (fact_index = 0;
                     fact_index < curr_rule.num_symptoms;
                     fact_index++) {
                  curr_fact =
                     bb_data_spc[curr_rule.symptoms[fact_index]];
                  if (curr_fact.validity &&
                        curr_rule.quality[fact_index] ==
                        curr_fact.quality) {
                     num_matched++;
                  }
               }
               /* update PIP's cf value */
               bb_PIPs[curr_rule.specialist_index].cf =
                  update_lvl2_cf(
                     bb_PIPs[curr_rule.specialist_index].cf,
                     curr_rule.cf, num_matched,
                     curr_rule.num_symptoms);
            }
         }
         /* sort the PIP record array */
         sort_to_n(bb_PIPs, bb_num_PIPs, bb_num_lvl2);

         /* time-stamp PIP records with the end-validity time
            of bb_data_spc, then update the blackboard */
         start_validity = X_CLOCK;
         end_validity = bb_data_spc_time_frame->end_validity;
         output(bb_PIPs);
      } /* if start of new event */
      else {
         /* if data still valid, extend validity of PIPs */
         input(bb_data_spc) perform {
            start_validity = X_CLOCK;
            end_validity = bb_data_spc_time_frame->end_validity;
            output(bb_PIPs);
         }
      }
      old_event_flag = bb_new_event_flag;

      /* update blackboard */
      bb_lvl1_flag = FALSE;
      with_validity(DEFAULT_PERIOD) output(bb_lvl1_flag);
   }
   x_yield();
}

#undef NUM_STATIC

} /* lvl1_inference() */


/*----- update_lvl2_cf() -------------------------*
   Updates the cf value of a level-two specialist.
   Formula:
      cf = old_cf + frac * wgt
   where
      old_cf = old cf value
      frac = fraction of predicates matched
      wgt = cf weight of current rule
   Returns the updated cf value
 *------------------------------------------------*/

real_t
update_lvl2_cf(real_t old_cf, real_t wgt, int num_matched, int num_symptoms)
{
   if (num_matched == 0 || num_symptoms == 0)
      return old_cf;

   return old_cf + wgt * (real_t) num_matched / (real_t) num_symptoms;
} /* update_lvl2_cf() */


/*----- sort_to_n() ------------------------------*
```

```
   Sort the PIP_REC array up to the nth element so that
   the first n elements will contain the highest CFs.
 *-------------------------------------------------------*/

void
sort_to_n(PIP_REC *base, int n, int size)
{
   PIP_REC dummy;
   int i, j;

   /* (partial) bubble sort algorithm */
   for (i = 0; i < n; i++)
      for (j = i + 1; j < size; j++)
         if (base[i].cf < base[j].cf) {
            dummy = base[i];
            base[i] = base[j];
            base[j] = dummy;
         }
}  /* sort_to_n() */


/*----- disp_lvl1() -------------------------*
   Displays the result of level-one inference.
   Inferenced results are invalid if validity
   of data (or any other relevant blackboard
   variable) expires.
 *-------------------------------------------*/

X_TASK
disp_lvl1(void)
{
   int i;
   BOOLEAN old_lvl1_flag = FALSE;

   state_variable(int, bb_num_PIPs);
   state_variable(BOOLEAN, bb_lvl1_flag);
   state_variable(PIP_REC *, bb_PIPs);

   initialise_blackboard;
   initialise_time;

   set_path(system_1, blackboard, sys_AKShell);
      time_frame(bb_num_PIPs, d);
   set_path(system_1, level_1, inf1);
      time_frame(bb_lvl1_flag, b);
      time_frame(bb_PIPs, p);

   x_create_window(6, 1, 5, 40, 0, 0, " Level One Result ", X_VISIBLE);

   FOREVER {
      input(bb_lvl1_flag) perform {
         /* display result once level-1 inference finishes */
         if (!bb_lvl1_flag && old_lvl1_flag) {
            /* display inferenced results */
            input(bb_PIPs) and(bb_num_PIPs) perform {
               x_printf("\n");
               for (i = 0; i < bb_num_PIPs; i++)
                  x_printf("\n%d: PIP[%d], cf = %1.4f",
                        i, bb_PIPs[i].index, bb_PIPs[i].cf);
            }
         }
         old_lvl1_flag = bb_lvl1_flag;
      }

      x_yield();
   }
}  /* disp_lvl1() */
```

```
/*========= inf2.c =========*
   level 2 inference engine
 *==========================*/

#include <string.h>

#include "x.h"
#include "xtime.h"

#include "AKShell.h"

X_TASK inf2(void);
X_TASK inf2_pip(void);
int unify(PREDICATE *, INP_DATA *);


/*----- inf2() --------------------*
   Activate level-2 PIPs via system
   flags. PIPs are waken on the
   falling edge of the level-2 flag
   and suspended on the rising edge
   of the new-event flag.
 *---------------------------------*/

X_TASK
inf2(void)
{
   BOOLEAN old_lvl1_flag = FALSE;
   BOOLEAN old_event_flag = FALSE;

   state_variable(BOOLEAN, bb_new_event_flag);
   state_variable(INP_DATA *, bb_data_spc);
   state_variable(BOOLEAN, bb_lvl1_flag);
   state_variable(PIP_REC *, bb_PIPs);
   state_variable(BOOLEAN, bb_lvl2_flag);
   state_variable(int, bb_PIP_status);
   state_variable(int, bb_num_PIPs);

   initialise_blackboard;
   initialise_time;

   set_path(system_1, data_if, data_if);
      time_frame(bb_new_event_flag, b);
      time_frame(bb_data_spc, p);
   set_path(system_T, level_1, inf1);
      time_frame(bb_lvl1_flag, p);
      time_frame(bb_PIPs, p);
   set_path(system_T, level_2, inf2);
      time_frame(bb_lvl2_flag, b);
      time_frame(bb_PIP_status, d);
   set_path(system_T, blackboard, sys_AKShell);
      time_frame(bb_num_PIPs, d);

   bb_lvl2_flag = FALSE;
   FOREVER {
      input(bb_new_event_flag) and(bb_lvl1_flag) perform {
         if (!bb_lvl2_flag) {
            bb_lvl2_flag = (old_lvl1_flag && !bb_lvl1_flag);

            /* reset PIP status */
            if (bb_lvl2_flag) bb_PIP_status = 0;
            with_validity(DEFAULT_PERIOD) output(bb_PIP_status);
         }
         else {
            /*-------------------------------------------------*
               1) Turn level-2 flag off if all PIPs have completed
               2) Check if data & PIP records are still valid with
                  reference to the system flags.
             *-------------------------------------------------*/
            input(bb_PIPs) and(bb_data_spc) and(bb_num_PIPs) perform {
               bb_lvl2_flag = ((bb_PIP_status < bb_num_PIPs) &&
                               ((!old_event_flag && bb_new_event_flag) ||
                               (bb_lvl2_flag && !bb_new_event_flag)));
            }
            else
               bb_lvl2_flag = FALSE;
         }
         old_event_flag = bb_new_event_flag;
         old_lvl1_flag = bb_lvl1_flag;
         with_validity(DEFAULT_PERIOD) output(bb_lvl2_flag);
         with_validity(DEFAULT_PERIOD) output(bb_PIP_status);
      }

      x_yield();
   }
} /* inf2() */
```

```
/*----- inf2_pip() -------------------------*
   Level 2 inference engine (PIP).
   - PIPs become active when bb_level_2_flag
   becomes TRUE, and suspend themselves
   when the flag is set to FALSE.
 *------------------------------------------*/

X_TASK
inf2_pip(void)
{
   int pip;
   NODE *curr_node, *curr_level, *curr_choice;
   real_t max_cf, curr_cf;
   real_t (*cf_mode)(real_t, real_t);
   BOOLEAN result;

   state_variable(K_TREE *, bb_k_tree);
   state_variable(int, bb_num_PIPs);
   state_variable(real_t, bb_cf_threshold);
   state_variable(PIP_REC *, bb_PIPs);
   state_variable(BOOLEAN, bb_lvl2_flag);
   state_variable(INP_DATA *, bb_data_spc);
   state_variable(int, bb_PIP_status);

   initialise_blackboard;
   initialise_time;

   set_path(system_1, blackboard, sys_AKShell);
      time_frame(bb_k_tree, p);
      time_frame(bb_num_PIPs, d);
      time_frame(bb_cf_threshold, f);
   set_path(system_T, level_1, inf1);
      time_frame(bb_PIPs, p);
   set_path(system_T, level_2, inf2);
      time_frame(bb_lvl2_flag, b);
      time_frame(bb_PIP_status, d);
   set_path(system_T, data_if, data_if);
      time_frame(bb_data_spc, p);

   x_create_window(6, 41, 5, 40, 0, 0,
         X_CURRENT_TCB->command_line, X_VISIBLE);

   FOREVER {
      pip = -1;      /* initialise to invalid PIP index */
      /* initialise and attach to system */
      input(bb_lvl2_flag) and(bb_cf_threshold) and(bb_num_PIPs)
            and(bb_PIPs) and(bb_k_tree) and(bb_data_spc) perform {
         if (bb_lvl2_flag == TRUE) {
            /*-----------------------------------*
               bid for a PIP goal, check that:
               - there is a valid goal
               - goal not attached to a PIP yet
               - cf of the goal is > threshold
             *-----------------------------------*/
            for (pip = 0; pip < bb_num_PIPs; pip++) {
               if (bb_PIPs[pip].attached == FALSE &&
                     bb_PIPs[pip].cf >= bb_cf_threshold) {
                  /* attach to this PIP goal */
                  bb_PIPs[pip].attached = TRUE;
                  with_validity(DEFAULT_PERIOD) output(bb_PIPs);
                  x_yield();
                  break;
               }
            }
         }
      }

      if (pip >= 0 && pip < bb_num_PIPs) {
         /* attach specialist */
         curr_level = bb_k_tree[bb_PIPs[pip].index].root;
         x_printf("\n\n[%d] attached: \"%s\" [%1.4f]", bb_PIPs[pip].index,
                  curr_level->name, bb_PIPs[pip].cf);
         cf_mode = bb_k_tree[bb_PIPs[pip].index].cf_mode;
         curr_level = curr_level->offspring;    /* go to first level */

         /* inference algorithm (loop) */
         while (bb_lvl2_flag && curr_level) {
            /* first choose the 1st node */
            curr_choice = curr_level;
            result = unify(curr_choice->assumption, bb_data_spc);
            max_cf = curr_choice->cf[result];

            /* check others on the same level */
            curr_node = curr_choice->sibling;
            while (curr_node) {
               result = unify(curr_node->assumption, bb_data_spc);
               curr_cf = curr_choice->cf[result];
               if (curr_cf > max_cf) {
```

```
                max_cf = curr_cf;
                curr_choice = curr_node;
             }
             curr_node = curr_node->sibling;
          }

          /*-----------------------------------------*
             made final choice on this level-> update
             confidence factor using user-defined
             function for this tree and move to next
             level:
          *-----------------------------------------*/
          bb_PIPs[pip].cf = cf_mode(bb_PIPs[pip].cf, max_cf);
          with_validity(DEFAULT_PERIOD) output(bb_PIPs);
          x_printf("\n--> update CF = %1.4f", bb_PIPs[pip].cf);

          input(bb_lvl2_flag) perform {
             curr_level = curr_choice->offspring;   /* go to next level */

             /* increment status if end of tree is reached */
             if (curr_level == NULL) {
                input(bb_PIP_status) perform {
                   bb_PIP_status++;
                   with_validity(DEFAULT_PERIOD) output(bb_PIP_status);
                }
             }
          else
             bb_lvl2_flag = FALSE;
          }
          x_yield();
       }
    }

    x_yield();
  } /* forever loop */
} /* inf2_pip() */


/*----- unify()-----------------------------------------------------*
   Check the validity of a node by calling predicate functions,
   usr_func(), which returns a boolean value to indicate whether
   predicate conditions are met. If all predicates are satisfied
   then return TRUE (1); else return FALSE (0). The returned value
   is an index to the node's confidence array (ADD_CF & DED_CF).
   The indexed cf value is compounded with the inferenced cf.
 *-----------------------------------------------------------------*/

int
unify(PREDICATE *assumption, INP_DATA *data)
{
   PREDICATE *curr_assume;
   BOOLEAN (*curr_func)(int *, real_t *, int, INP_DATA *);

   curr_assume = assumption;
   while (curr_assume) {
      curr_func = curr_assume->usr_func;
      if ((curr_func(curr_assume->param_index, curr_assume->ref_value,
           curr_assume->num_param, data)) == TRUE) {
         curr_assume = curr_assume->next;
      }
      else
         return FALSE;
   }
   return TRUE;
} /* unify() */
```

```
/*===== Decision.c =====*
   Decision-maker
 *======================*/

#include "x.h"
#include "xtime.h"
#include "AKShell.h"

X_TASK decision_maker(void);


/*----- decision_maker ------------------------------*
   Select a decision based on the CF results of the
   level-2 PIPs. Activates on the negative edge of
   bb_lvl2_flag. Decision is valid if data is still
   valid.
 *---------------------------------------------------*/

X_TASK
decision_maker(void)
{
   real_t max_cf;
   int i, max_index;
   BOOLEAN valid;
   char cf_str[10];

   state_variable(K_TREE *, bb_k_tree);
   state_variable(int, bb_num_PIPs);
   state_variable(PIP_REC *, bb_PIPs);
   state_variable(INP_DATA *, bb_data_spc);
   state_variable(BOOLEAN, bb_lvl2_flag);

   BOOLEAN old_lvl2_flag;

   initialise_blackboard;
   initialise_time;

   set_path(system_1, blackboard, sys_AKShell);
      time_frame(bb_k_tree, p);
      time_frame(bb_num_PIPs, d);
   set_path(system_1, data_if, data_if);
      time_frame(bb_data_s[ ], p);
   set_path(system_1, level_1, inf1);
      time_frame(bb_PIPs, p);
   set_path(system_1, level_2, inf2);
      time_frame(bb_lvl2_flag, b);

   x_create_window(11, 1, 6, 80, 0, 0, " Decision-maker ", X_VISIBLE);

   /* initialise and attach to system */
   bb_num_PIPs = 0;
   valid = FALSE;
   x_write(1, 1, 0, "Validity: X");
   x_write(2, 1, 0, "Decision: ");
   x_write(3, 1, 0, "---> CF = ");
   do {
      input(bb_num_PIPs) and(bb_k_tree)
            and(bb_lvl2_flag) and(bb_data_spc) perform {
         /* do nothing until attached to system */
      }
      x_yield();
   } while (bb_num_PIPs <= 0);

   FOREVER {
      old_lvl2_flag = bb_lvl2_flag;

      /* make a decision on negative edge of level-2 flag */
       nput(bb_lvl2_flag) perform {
        if (old_lvl2_flag && !bb_lvl2_flag) {
           input(bb_data_spc) and(bb_PIPs) perform {
              /* make a decision */
              valid = TRUE;
              max_index = bb_PIPs[0].index;
              max_cf = bb_PIPs[0].cf;
              for (i = 1; i < bb_num_PIPs; i++) {
                 if (max_cf < bb_PIPs[i].cf) {
                    max_c    bb_PIPs[i].cf;
                    max_index = bb_PIPs[i].index;
                 }
              }
              sprintf(cf_str, "%1.4f", max_cf);
              x_clear_row(2, 11);
              x_write(2, 11, 0, bb_k_tree[max_index].specialist_name);
              x_clear_row(3, 11);
              x_write(3, 11, 0, cf_str);

              /* implement decision action here */
```

```
            x_yield();
          }
          else
            valid = FALSE;
        }
        x_write_char(1, 11, 0, valid+48);
      }
      x_yield();
    } /* FOREVER */
} /* decision_maker() */
```

```
/*===== UsrFunc.c ========*
  user-defined functions
 *========================*/

#include <stdlib.h>
#include "AKShell.h"
#include "errmeg.h"
#include "usrfunc.h"


/*--------------------*
   CF update function
 *--------------------*/
real_t cf_mode(real_t, real_t);


/*---------------------------*
   initialise user functions
 *---------------------------*/
usr_func_prototype(less_than);
usr_func_prototype(greater_than);
usr_func_prototype(equal_to);
usr_func_prototype(not_equal_to);
usr_func_prototype(increasing);
usr_func_prototype(decreasing);

begin_usr_func_list(6)
   define_usr_func(less_than);
   define_usr_func(greater_than);
   define_usr_func(equal_to);
   define_usr_func(not_equal_to);
   define_usr_func(increasing);
   define_usr_func(decreasing);
end_usr_func_list


/*-------------------------*
   initialise CF functions
 *-------------------------*/
cf_func_prototype(additive);
cf_func_prototype(associative);
cf_func_prototype(conjunctive);
cf_func_prototype(disjunctive);

begin_cf_func_list(4)
   define_cf_func(additive);
   define_cf_func(associative);
   define_cf_func(conjunctive);
   define_cf_func(disjunctive);
end_cf_func_list


/*-------------------------------------------------------------------------*
                        user-defined functions
 *-------------------------------------------------------------------------*/

BOOLEAN
less_than(int *x, real_t *ref, int num, INP_DATA *data)
{
   int i;

   for (i = 0; i < num; i++) {
      if (*(data[x[i]].value) > ref[i]) return FALSE;
   }

   return TRUE;
}  /* less_than() */


BOOLEAN
greater_than(int *x, real_t *ref, int num, INP_DATA *data)
{
   int i;

   for (i = 0; i < num; i++) {
      if (*(data[x[i]].value) < ref[i]) return FALSE;
   }

   return TRUE;
}  /* greater_than() */


BOOLEAN
equal_to(int *x, real_t *ref, int num, INP_DATA *data)
{
   int i;

   for (i = 0; i < num; i++) {
```

```
      if (*(data[x[i]].value) != ref[i]) return FALSE;
   }

   return TRUE;
}  /* equal_to() */


BOOLEAN
not_equal_to(int *x, real_t *ref, int num, INP_DATA *data)
{
   int i;

   for (i = 0; i < num; i++) {
      if (*(data[x[i]].value) == ref[i]) return FALSE;
   }

   return TRUE;
}  /* not_equal_to() */


BOOLEAN
increasing(int *x, real_t *ref, int num, INP_DATA *data)
{
   int i;

   for (i = num - 1; i; i--) {
      if (*(data[x[i-1]].value) < *(data[x[i]].value)) return FALSE;
   }

   return TRUE;
}  /* increasing() */


BOOLEAN
decreasing(int *x, real_t *ref, int num, INP_DATA *data)
{
   int i;

   for (i = num - 1; i; i--) {
      if (*(data[x[i-1]].value) > *(data[x[i]].value)) return FALSE;
   }

   return TRUE;
}  /* increasing() */


/*--------------------------------------------------------------------------*
                    user-defined cf-update functions
 *--------------------------------------------------------------------------*/

/*-------------------------------------*
   User-defined cf modes:

   Input:- old_cf: value of current CF
           wgt: CF weight
   Ouput:- new CF as modified by wgt.

   Default functions to modify cf:-
      additive: addition
      associative: multiplication
      conjunctive: chooses max
      disjunctive: chooses min
/*-------------------------------------*/


real_t
additive(real_t old_cf, real_t cf_weight)
{
   return old_cf + cf_weight;
}       /* additive() */


real_t
associative(real_t old_cf, real_t cf_weight)
{
   return old_cf * cf_weight;
}       /* associative() */


real_t
conjunctive(real_t old_cf, real_t cf_weight)
{
   return (real_t) min(old_cf, cf_weight);
}       /* conjunctive() */


real_t
disjunctive(real_t old_cf, real_t cf_weight)
```

```
{
   return (real_t) max(old_cf, cf_weight);
}       /* disjunctive() */
```

```
/*====== KB_Comp.c ==========*
   AKSHELL Knowledge compiler
 *===========================*/

#define DEBUG

#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <ctype.h>
#include <string.h>
#include <math.h>

#include "x.h"
#include "xtime.h"
#include "AKShell.h"
#include "UsrFunc.h"
#include "Data_IF.h"
#include "ErrMsg.h"

#define LABEL_LEN 20
#define NUM_LEN 20

#define High_token "high"
#define Low_token "low"

/*--------*
   tokens
 *--------*/
#define Begin_data "BEGIN_DATA:"
#define End_data "END_DATA:"
#define Begin_lvl1 "BEGIN_LVL1:"
#define End_lvl1 "END_LVL1:"
#define Begin_lvl2 "BEGIN_LVL2:"
#define End_lvl2 "END_LVL2:"
#define Data_token "DATA:"
#define Data_set_token "DATA_SET:"
#define Decision_token "DECISION:"
#define CF_weight_token "CF_WEIGHT:"
#define CF_mode_token "CF_MODE:"
#define Node_token "NODE:"
#define Parent_token "PARENT:"
#define Pred_token "PREDICATE:"
#define Param_token "PARAM:"
#define Ref_val_token "REF_VAL:"

#define Begin_comment ';'     /* marks start of comment */

/*--------------------*
   real number defines
 *--------------------*/
#define DEC_POINT '.'
#define EXPONENT 'E'
#define POS_SIGN '+'
#define NEG_SIGN '-'

/*-----------------*
   global variables
 *-----------------*/
extern int num_data, num_lvl2, num_lvl1, num_pips;
extern INP_DATA *data_spc;
extern DECISION_LIST *lvl1_rules;
extern K_TREE *K_tree;
extern real_t cf_threshold;
extern func_rec_t *usr_func_list;
extern cf_func_rec_t *cf_func_list;
extern if_data_t *If_data_list;
extern int num_if_data;

NODE_STACK *node_stack = NULL, *stack_top = NULL;

/*--------------------*
   function prototypes
 *--------------------*/
int kb_compile(char *);
int data_declare(FILE *, int *);
int lvl1_declare(FILE *, int *);
int lvl2_declare(FILE *, int *);

int count_item(FILE *, char *, char *);
NODE *find_parent(int, char *);
int match_data(int, char *);
int match_specialist(int, char *);
int attach_usr_func(int, PREDICATE *);
real_t *attach_data(int, char *);

int get_token(FILE *, int *, char *);
int get_label(FILE *, int *, char *);
```

```
int get_real_t(FILE *, int *, real_t *);
int get_int(FILE *, int *, int *);
int skip_white_space(FILE *, int *);
BOOLEAN is_white_space(char, int *);
BOOLEAN is_bad_char(char);
void skip_word(FILE *, int *);

void echo_error(int, int, char *);
char *errmsg(int, char *);

int push_node(NODE *);
NODE *pop_node(void);
void clear_stack(void);

#ifdef DEBUG
   void debug1(int);
   void debug2(int);
#endif


/*----- kb_compile() ----------------------------------*
   Rule compiler: returns NO_ERR if successful, else it
   calls "echo_msg" to print out an error message as
   defined in ErrMsg.TXT, and returns the error code.
 *-----------------------------------------------------*/

int
kb_compile(char *name)
{
   FILE *aks_file;
   char token_str[10];
   int err, i;
   int line_no = 1;     /* line number of input file */

   if ((aks_file = fopen(name, "r")) == NULL) {
      printf("Could not open file %s\n", name);
      exit(-1);
   }

   printf("\n<< AKShell Knowledge Compiler >>\n\n");

   /* initialise global variables */
   num_data = num_lvl2 = num_lvl1 = num_pips = 0;
   data_spc = NULL; lvl1_rules = NULL; K_tree = NULL;
   cf_threshold = 0;

   /* compile data definition */
   printf("Compiling data declaration ... ");
   if ((err = num_data = data_declare(aks_file, &line_no)) < NO_ERR) {
      echo_error(err, line_no, name);
      return err;
   }

   /* compile level-two tree definition */
   printf("Compiling level-two declaration ... ");
   if ((err = num_lvl2 = lvl2_declare(aks_file, &line_no)) < NO_ERR) {
      echo_error(err, line_no, name);
      return err;
   }

   /* compile level-one rule definition */
   printf("Compiling level-one declaration ... ");
   if ((err = num_lvl1 = lvl1_declare(aks_file, &line_no)) < NO_ERR) {
      echo_error(err, line_no, name);
      return err;
   }

   fclose(aks_file);
   printf("\n\n");

   do {
      printf(">> Enter number of PIPs (1 <= num <= %d): ", num_lvl2);
      scanf("%d", &num_pips);
      if (num_pips < 1 || num_pips > num_lvl2)
         printf("Entered number out of range!\n\n");
   } while (num_pips < 1 || num_pips > num_lvl2);

   do {
      printf(">> Enter CF threshold (0 <= num < 1): ");
      scanf("%lf", &cf_threshold);
      if (cf_threshold < 0 || cf_threshold >= 1)
         printf("Entered number out of range!\n\n");
   } while (num_pips < 1 || num_pips > num_lvl2);
   printf("\n");

   #ifdef DEBUG
      printf("\n... Compiled information:\n\n");
      debug1(num_lvl1);
```

```
      debug2(num_lvl2);
   #endif

   return err;
}  /* kb_compile() */


/*----- data_declare() -----------------*
   Data declaration. Returns the number
   of data compiled, or an error code.
 *--------------------------------------*/

int
data_declare(FILE *i_file, int *line)
{
   int index, err;
   int num_data = 0;
   int num_if_data;
   char dummy_str[80];

   /* initialise interfaced data */
   if ((err = num_if_data = if_data_declare()) < NO_ERR)
      return (num_if_data == 0) ? NO_IF_DATA : err;

   /* find starting label */
   if ((err = get_token(i_file, line, Begin_data)) != NO_ERR)
      return err;

   /* find number of data */
   if ((err = num_data
         = count_item(i_file, End_data, Data_token)) <= 0)
      return (num_data == 0) ? NO_DATA : err;

   /* read definition */
   if ((data_spc = (INP_DATA *) calloc(num_data, sizeof(INP_DATA)))
       == NULL)
      return MEM_ERR;
   for (index = 0; index < num_data; index++) {
      if ((err = get_token(i_file, line, Data_token)) != NO_ERR)
         return err;
      if ((err = get_label(i_file, line, dummy_str)) != NO_ERR)
         return err;
      strcpy(data_spc[index].name, dummy_str);

      if ((data_spc[index].value =
           attach_data(num_if_data, dummy_str)) != NULL) {
         /* interfaced data */
         if ((err = get_real_t(i_file, line, &data_spc[index].LOW_LIM))
             != NO_ERR)
            return err;
         if ((err = get_real_t(i_file, line, &data_spc[index].HIGH_LIM))
             != NO_ERR) return err;
      }
      else {
         /* user-constant */
         if ((data_spc[index].value =
              (real_t *) malloc(sizeof(real_t))) == NULL)
            return MEM_ERR;
         if ((err = get_real_t(i_file, line, data_spc[index].value))
             != NO_ERR)
            return err;
         data_spc[index].LOW_LIM = data_spc[index].HIGH_LIM
                                 = *(data_spc[index].value);
      }
   }

   /* find end label */
   if ((err = get_token(i_file, line, End_data)) != NO_ER
      return err;

   printf("OK\n");
   return num_data;
}  /* data_declare() */


/*----- lvl1_declare() ------------------------------------------*
   Level-one declaration. If successful, returns the number of
   level-one rules compiled (num_lvl1), else return an error code
   as defined in errmsg.
 *---------------------------------------------------------------*/

int
lvl1_declare(FILE *i_file, int *line)
{
   int index, i, err;
   int num_lvl1 = 0;
   int specialist_number;
   int num_symptoms;
```

```
    int *symptoms;
    char dummy_str[80];
    real_t dummy_real_t;
    QUALITY *qualities;

    /* find start label */
    if ((err = get_token(i_file, line, Begin_lvl1)) != NO_ERR)
       return err;

    /* count number of level-1 rules */
    if ((err = num_lvl1
          = count_item(i_file, End_lvl1, Decision_token)) <= 0)
       return (num_lvl1 == 0) ? NO_LVL1 : err;

    /* read definition */
    if ((lvl1_rules = (DECISION_LIST *)
          calloc(num_lvl1, sizeof(DECISION_LIST))) == NULL)
       return MEM_ERR;
    for (index = 0; index < num_lvl1; index++) {
       /* get rule label */
       if ((err = get_token(i_file, line, Decision_token)) != NO_ERR)
          return err;
       if ((err = get_label(i_file, line, dummy_str)) != NO_ERR)
          return err;

       /* match specialist number */
       if ((err = specialist_number
             = match_specialist(num_lvl2, dummy_str)) < 0)
          return err;
       lvl1_rules[index].specialist_index = specialist_number;

       /* get initial CF weight */
       if ((err = get_token(i_file, line, CF_weight_token)) != NO_ERR)
          return err;
       if ((err = get_real_t(i_file, line, &dummy_real_t)) != NO_ERR)
          return err;
       lvl1_rules[index].cf = dummy_real_t;

       /* count number of predicates */
       if ((err = get_token(i_file, line, Data_set_token)) != NO_ERR)
          return err;
       if (index < num_lvl1 - 1) {
          if ((err = num_symptoms = count_item(i_file, Decision_token, ""))
                < NO_ERR)
             return err;
       }
       else {
          if ((err = num_symptoms = count_item(i_file, End_lvl1, "")) < NO_ERR)
             return err;
       }
       num_symptoms /= 2;

       /* get predicate */
       lvl1_rules[index].num_symptoms = num_symptoms;
       if ((symptoms = lvl1_rules[index].symptoms = (int *)
             calloc(num_symptoms, sizeof(int))) == NULL)
          return MEM_ERR;
       if ((qualities = lvl1_rules[index].quality = (QUALITY *)
             calloc(num_symptoms, sizeof(QUALITY))) == NULL)
          return MEM_ERR;

       for (i = 0; i < num_symptoms; i++) {
          /* specify data */
          if ((err = get_label(i_file, line, dummy_str)) != NO_ERR)
             return err;
          if ((err = symptoms[i]
                = match_data(num_data, dummy_str)) < NO_ERR)
             return err;

          /* specify quality */
          if ((err = get_label(i_file, line, dummy_str)) != NO_ERR)
             return err;
          if (!strcmp(dummy_str, High_token))
             qualities[i] = HIGH_VAL;
          else if (!strcmp(dummy_str, Low_token))
             qualities[i] = LOW_VAL;
          else
             qualities[i] = NORM_VAL;   /* default = normal quality */
       }
    }

    /* find end label */
    if ((err = get_token(i_file, line, End_lvl1)) != NO_ERR)
       return err;

    printf("OK\n");
    return num_lvl1;
 } /* lvl1_declare() */
```

```
/*----- lvl2_declare() ----------------------------------------*
   Build level-two knowledge trees. Returns the number level-two
   specialists compiled (num_lvl2) if successful, else returns
   an error code as defined in errmsg.
 *-------------------------------------------------------------*/

int
lvl2_declare(FILE *i_file, int *line)
{
   int num_lvl2, index;
   int node_cnt, pred_cnt, param_cnt;
   int err;
   int num_usr_func, num_cf_func;
   int num_nodes, num_preds, num_param;
   NODE *curr_node, *parent, *a_node;
   PREDICATE *curr_pred = 0;
   char dummy_str[80];
   real_t dummy_real_t;

   /* initialise user/CF functions */
   if (((err = num_usr_func = usr_func_declare()) < NO_ERR) ||
       ((err = num_cf_func = cf_func_declare()) < NO_ERR))
      return (num_usr_func == 0 || num_cf_func == 0) ?
         NULL_FUNC_LIST : err;

   /* find start label */
   if ((err = get_token(i_file, line, Begin_lvl2)) != NO_ERR)
      return err;

   /* count number of level-2 specialists */
   if ((err = num_lvl2
         = count_item(i_file, End_lvl2, Decision_token)) <= 0)
      return (num_lvl2 == 0) ? NO_LVL2 : err;

   /* read definition */
   if ((k_tree = (K_TREE *) calloc(num_lvl2, sizeof(K_TREE))) == NULL)
      return MEM_ERR;

   /* build knowledge trees */
   for (index = 0; index < num_lvl2; index++) {
      /* match decision name */
      if ((err = get_token(i_file, line, Decision_token)) != NO_ERR)
         return err;
      if ((err = get_label(i_file, line, dummy_str)) != NO_ERR) return err;
      if ((a_node = k_tree[index].root = (NODE *) calloc(1, sizeof(NODE)))
            == NULL)
         return MEM_ERR;
      strcpy(a_node->name, dummy_str);
      strcpy(k_tree[index].specialist_name, dummy_str);

      /* attach CF function */
      if ((err = get_token(i_file, line, CF_mode_token)) != NO_ERR)
         return err;
      if ((err = get_label(i_file, line, dummy_str)) != NO_ERR) return err;
      strcpy(k_tree[index].cf_func_name, dummy_str);
      if ((err = attach_cf_func(num_cf_func, &k_tree[index])) != NO_ERR)
         return err;

      /* count number of nodes */
      if ((err = num_nodes
            = count_item(i_file, Decision_token, Node_token)) <= 0)
         return (num_nodes == 0) ? NULL_NODE : err;

      /* link nodes */
      for (node_cnt = 0; node_cnt < num_nodes; node_cnt++) {
         if ((err = get_token(i_file, line, Node_token)) != NO_ERR)
            return err;
         if ((err = get_label(i_file, line, dummy_str)) != NO_ERR)
            return err;
         if ((a_node = (NODE *) calloc(1, sizeof(NODE))) == NULL)
            return MEM_ERR;
         strcpy(a_node->name, dummy_str);

         /* define CF weights */
         if ((err = get_token(i_file, line, CF_weight_token)) != NO_ERR)
            return err;
         if ((err = get_real_t(i_file, line, &a_node->ADD_CF)) != NO_ERR)
            return err;
         if (fabs(a_node->ADD_CF) > 1)
            return INVALID_CF;
         if ((err = get_real_t(i_file, line, &a_node->DED_CF)) != NO_ERR)
            return err;
         if (fabs(a_node->DED_CF) > 1)
            return INVALID_CF;

         /* find parent node */
```

```
        if ((err = get_token(i_file, line, Parent_token)) != NO_ERR)
           return err;
        if ((err = get_label(i_file, line, dummy_str)) != NO_ERR)
           return err;
        if ((parent = find_parent(index, dummy_str)) == NULL)
           return NULL_NODE;

        /* link current node to current tree */
        if (parent->offspring) {
           curr_node = parent->offspring;
           while (curr_node->sibling) curr_node = curr_node->sibling;
           curr_node->sibling = a_node;
        }
        else
           parent->offspring = a_node;

        /* count number of predicates */
        if ((err = num_preds
                 = count_item(i_file, Node_token, Pred_token)) < NO_ERR)
           return err;

        /* link predicates */
        for (pred_cnt = 0; pred_cnt < num_preds; pred_cnt++) {
           if ((err = get_token(i_file, line, Pred_token)) != NO_ERR)
              return err;
           if ((err = get_label(i_file, line, dummy_str)) != NO_ERR)
              return err;
           if (a_node->assumption) {
              if ((curr_pred->next = (PREDICATE *)
                    calloc(1, sizeof(PREDICATE))) == NULL)
                 return MEM_ERR;
              curr_pred = curr_pred->next;
           }
           else if ((curr_pred = a_node->assumption =
                    calloc(1, sizeof(PREDICATE))) == NULL)
              return MEM_ERR;

           /* attach to user-functions */
           strcpy(curr_pred->usr_func_name, dummy_str);
           if ((err = attach_usr_func(num_usr_func, curr_pred)) != NO_ERR)
              return err;

           /* read parameter names and set data indices in predicate */
           if ((err = get_token(i_file, line, Param_token)) != NO_ERR)
              return err;

           /* count number of parameters */
           if ((err = num_param
                    = count_item(i_file, Ref_val_token, "")) < NO_ERR)
              return err;

           /* place predicates */
           curr_pred->num_param = num_param;
           if ((curr_pred->param_index = (int *)
                 calloc(num_param, sizeof(int))) == NULL)
              return MEM_ERR;
           if ((curr_pred->ref_value = (real_t *)
                 calloc(num_param, sizeof(real_t))) == NULL)
              return MEM_ERR;

           /* attach data to parameter list */
           for (param_cnt = 0; param_cnt < num_param; param_cnt++) {
              if ((err = get_label(i_file, line, dummy_str)) != NO_ERR)
                 return err;
              if ((err = curr_pred->param_index[param_cnt]
                       = match_data(num_data, dummy_str)) < NO_ERR)
                 return err;
           }

           /* read reference values */
           if ((err = get_token(i_file, line, Ref_val_token)) != NO_ERR)
              return err;
           for (param_cnt = 0; param_cnt < num_param; param_cnt++) {
              if ((err = get_real_t(i_file, line, &dummy_real_t)) != NO_ERR)
                 return err;
              curr_pred->ref_value[param_cnt] = dummy_real_t;
           }
        } /* pred_cnt loop */
     } /* node_cnt loop */
  } /* index loop */

  /* find end label */
  if ((err = get_token(i_file, line, End_lvl2)) != NO_ERR)
     return err;

  free(usr_func_list);
  free(cf_func_list);
  printf("OK\n");
```

```
   return num_lvl2;
}  /* lvl2_declare() */


/*-------------------------------------------------------------------------*
                           Support Functions
 *-------------------------------------------------------------------------*/

/*----- count_item() ------------------------------------------*
   Counts and returns the number of specified item. If no item
   is specified (when item = ""), then the number of words
   (labels) counted is returned. Counting continues until "end"
   label is found. At the end of count, the file pointer
   "i_file" is reset to where counting started. An error code
   is returned if counting is unsuccessful.
 *-----------------------------------------------------------*/

int
count_item(FILE *i_file, char *end, char *item)
{
   int i, err, num = 0;
   char buffer[80];
   long pos = 0;
   int line = 0;     /* dummy line number */

   pos = ftell(i_file);

   /* count number of item */
   do {
      if (feof(i_file)) return IS_EOF;
      get_label(i_file, &line, buffer);
      if (!strcmp(end, buffer)) break;
      else if (*item == NULL) num++;
      else if (!strcmp(item, buffer)) num++;
   } while (!feof(i_file));

   fseek(i_file, pos, SEEK_SET);
   return num;
}  /* count_item() */


/*----- find_parent() -----------------------------------------*
   Find and return the parent node in the current tree. If not
   found, NULL is returned. (Breadth-first search)
 *-----------------------------------------------------------*/

NODE *
find_parent(int index, char *name)
{
   NODE *curr_node;
   int err;

   curr_node = k_tree[index].root;

   while (curr_node) {
      while (curr_node) {
         if (!strcmp(curr_node->name, name)) {
            clear_stack();
            return curr_node;
         }
         err = push_node(curr_node);
         if (err != NO_ERR) printf("Warning: memory full!\n");
         curr_node = curr_node->sibling;
      }
      while (!curr_node && node_stack) {
         curr_node = pop_node();
         curr_node = curr_node->offspring;
      }
   }

   clear_stack();
   return NULL;
}  /* find_parent() */


/*----- match_data() ------------------------------------------*
   Checks if data exists. Returns index to the data list if found,
   else returns an error code (INVALID_DATA).
 *-------------------------------------------------------------------------*/

int
match_data(int num_data, char *target)
{
   int i;

   for (i = 0; i < num_data; i++) {
      if (!strcmp(data_spc[i].name, target)) {
         return i;
```

```
      }
   }

   return INVALID_DATA;
}  /* match_data() */


/*----- match_specialist() ----------------------------------------*
   Matches level-one decision name with those in level-two. Returns
   index to the knowledge tree if a match is found, else an error
   code is returned.
 *-----------------------------------------------------------------*/

int
match_specialist(int num_lvl2, char *specialist_name)
{
   int i;

   /* linear search k_tree to check if specialist exists */
   for (i = 0; i < num_lvl2; i++) {
      if (!strcmp(k_tree[i].specialist_name, specialist_name))
         return i;
   }

   return UNDEFINED_ACTION;
}  /* match_specialist() */


/*----- attach_usr_func() -----------------------------------------*
   Searches in the user-function list for the one specified. If found,
   the function is attached to node; else an error code is returned.
 *-----------------------------------------------------------------*/

int
attach_usr_func(int n, PREDICATE *curr_pred)
{
   int i;

   /* search and match the function list */
   for (i = 0; i < n; i++) {
      if (!strcmp(curr_pred->usr_func_name, usr_func_list[i].name)) {
         /* attach matched function */
         curr_pred->usr_func = usr_func_list[i].func;
         return NO_ERR;
      }
   }

   /* not matched, return error */
   return UNDEFINED_USR_FUNC;
}  /* attach_usr_func() */


/*----- attach_cf_func() ------------------------------------------*
   Searches in the cf-function list for the one specified. If found,
   the function is attached to node; else an error code is returned.
 *-----------------------------------------------------------------*/

int
attach_cf_func(int n, K_TREE *curr_tree)
{
   int i;

   /* search and match the function list */
   for (i = 0; i < n; i++) {
      if (!strcmp(curr_tree->cf_func_name, cf_func_list[i].name)) {
         /* attach matched function */
         curr_tree->cf_mode = cf_func_list[i].func;
         return NO_ERR;
      }
   }

   /* not matched, return error */
   return UNDEFINED_CF_FUNC;
}  /* attach_cf_func() */


/*----- attach_data() ---------------------------------------------*
   Searches in the data-list for an existing interface variable.
   If found, returns address of data; else returns 0.
 *-----------------------------------------------------------------*/

real_t *attach_data(int n, char *var)
{
   int i;

   for (i = 0; i < n; i++) {
      if (!strcmp(if_data_list[i].name, var))
         return if_data_list[i].data_loc;
```

```
   }

   return NULL;
}  /* attach_data() */


/*----- get_token() ----------------------------------------*
   Given a token, this function attempts to find it in the
   rule file. If found, 0 is return. If the next string
   in the file is not the desired token, 2 is returned. 1 is
   returned if the token is not found by the end of file.
 *----------------------------------------------------------*/

int
get_token(FILE *i_file, int *line, char *token_str)
{
   char ch;
   int index, token_len;

   token_len = strlen(token_str);

   /* skip white spaces */
   if (skip_white_space(i_file, line) == IS_EOF) return IS_EOF;

   for (index = 0; index < token_len; index++) {
      ch = fgetc(i_file);
      if (feof(i_file)) return IS_EOF;
      if (ch != token_str[index]) return NO_TOKEN;
   }

   /* delimit token by white space */
   ch = fgetc(i_file);
   if (feof(i_file)) return IS_EOF;
   if (!is_white_space(ch, line)) return NO_TOKEN;

   return NO_ERR;
}  /* get_token() */


/*----- get_label() ----------------------------------------*
   Get the name of a token label. If found, return 0; else return
   an error code. Note that only alpha-numeric characters, ':' and
   '_' are allowed in a label.
 *----------------------------------------------------------*/

int
get_label(FILE *i_file, int *line, char *label_str)
{
   int count, dummy;
   char ch;
   char buffer_str[LABEL_LEN + 1];

   /* skip white spaces */
   if (skip_white_space(i_file, line) == IS_EOF) return IS_EOF;

   for (count = 0; count < LABEL_LEN; count++) {
      ch = fgetc(i_file);
      if (feof(i_file))
         break;
      if (is_bad_char(ch)) {
         if (is_white_space(ch, &dummy))
            /* dummy is used to avoid double count of line number */
            break;
         else
            return BAD_CHAR;
      }
      buffer_str[count] = ch;
   }
   buffer_str[count] = 0;     /* NULL termination */
   strcpy(label_str, buffer_str);

   /* Read until delimited by white space. Extra chars are ignored. */
   while (!feof(i_file) && !is_white_space(ch, line))
      ch = fgetc(i_file);

   return NO_ERR;
}  /* get_label() */


/*----- get_real_t() ---------------------------------------*
   Reads in a word (delimited by white space) then use strtod() to
   convert it into a to a real number. Returns an error if a bad
   character or an overflow condition is encountered.
 *----------------------------------------------------------*/

int
get_real_t(FILE *i_file, int *line, real_t *value)
{
```

```
   int i;
   char ch, real_t_str[NUM_LEN], *str;
   char *endptr;

   *value = 0.0;     /* initialise value */

   /* skip white spaces */
   if (skip_white_space(i_file, line) == IS_EOF) return IS_EOF;

   /* read in a string */
   for (i = 0; i < NUM_LEN; i++) {
      ch = fgetc(i_file);
      if (is_white_space(ch, line)) {
         real_t_str[i] = 0;
         break;
      }
      real_t_str[i] = ch;
   }

   /* convert string to real number */
   *value = strtod(real_t_str, &endptr);
   if (*value == HUGE_VAL || *endptr != NULL)
      return BAD_REAL_T;

   return NO_ERR;
}  /* get_real_t() */


/*----- get_int() -----------*
   Read in an integer value.
 *---------------------------*/

int
get_int(FILE *i_file, int *line, int *value)
{
   int count;
   char ch, int_str[NUM_LEN];

   *value = 0;    /* initialise value */

   /* skip white spaces */
   if (skip_white_space(i_file, line) == IS_EOF) return IS_EOF;

   /*-----------------------------------------*
      Read in a word (delimited by white space)
      then use atoi() to convert.
    *-----------------------------------------*/

   count = 0;

   ch = fgetc(i_file);
   while (isdigit(ch)) {
      if (count < NUM_LEN)
         int_str[count] = ch;
      else break;
      count++;
      ch = fgetc(i_file);
   }
   if (!is_white_space(ch, line))
      return BAD_INT;

   *value = atoi(int_str);
   return NO_ERR;
}  /* get_int() */


/*----- skip_white_space() ----------------------------------------*
   Advace file pointer until end of white space is reached. If the
   comment marker is found on a line, skip to rest of the line.
 *-----------------------------------------------------------------*/

int
skip_white_space(FILE *i_file, int *line)
{
   char ch;

   do {
      if (feof(i_file)) return IS_EOF;
      ch = fgetc(i_file);

      /* if comment, skip to rest of line */
      if (ch == Begin_comment) {
         while (ch != '\n')
            ch = fgetc(i_file);
      }
   } while (is_white_space(ch, line));

   /* push back the last non-space character */
```

```
   ungetc(ch, i_file);

   return NO_ERR;
}  /* skip_white_space() */


/*----- is_white_space() --------------------*
   Determine if given char is a white space.
   If ch == \newline, increment line count.
 *-------------------------------------------*/

BOOLEAN
is_white_space(char ch, int *line)
{
   switch(ch) {
      case '\n':
         (*line)++;
      case 32 :   /* space */
      case  9 :   /* tab */
         return TRUE;
   }
   return FALSE;
}  /* is_white_space() */


/*----- is_bad_char() -----------------*
   If an illegal character is detected,
   return 1; else return 0.
 *-------------------------------------*/

BOOLEAN
is_bad_char(char ch)
{
   if (isalpha(ch) || isdigit(ch) || (ch == '_') || (ch == ':'))
         return FALSE;
   else return TRUE;
}  /* is_bad_char() */


void
skip_word(FILE *i_file, int *line)
{
   char ch;

   do {
      ch = fgetc(i_file);
   }  while (!is_white_space(ch, line) && !feof(i_file));
}  /* skip_w---") */


/*-------------------------------------------------------------------------*
                        Error message functions
 *-------------------------------------------------------------------------*/

/*----- echo_error() -------------*
   Echoes compiler error message.
 *--------------------------------*/

void
echo_error(int err, int line_no, char *filename)
{
   char msg[80];

   printf("\n! %s(%d) : error %d: ", filename, line_no, -err);
   printf("%s\n", errmsg(err, msg) ? msg : "no error message");
}  /* echo_error() */


/*----- errmsg() ----------------------------------------------------*
   Returns the error message corresponding to the input error number.
   Returns NULL if the error message is not found.
 *-------------------------------------------------------------------*/

char *
errmsg(int err_no, char *errstr)
{
   int i;
   FILE *errfile;
   char ch, *str = 0;

   if ((errfile = fopen("errmsg.txt", "r")) == 0)
      return 0;

   while (!feof(errfile)) {
      i = 0;
      do {
         ch = fgetc(errfile);
      } while (ch != '$' && !feof(errfile));
```

```
      if (feof(errfile)) {
         fclose(errfile);
         return 0;
      }
      ch = fgetc(errfile);
      while (isdigit(ch) && !feof(errfile)) {
         i = i*10 + ch - 48;     /* convert ch to integer */
         ch = fgetc(errfile);
      }
      str = fgets(errstr, 80, errfile);
      if (i == -err_no) {
         fclose(errfile);
         return str;
      }
   }
   fclose(errfile);
   return NULL;
}  /* errmsg() */

/*-------------------------------------------------------------------------*
                  node-stack push and pop functions
 *-------------------------------------------------------------------------*/

int
push_node(NODE *node)
{
   if (stack_top) {
      stack_top->next = (NODE_STACK *) calloc(1, sizeof(NODE_STACK));
      stack_top = stack_top->next;
      if (stack_top == NULL) return MEM_ERR;
   }
   else {
      stack_top = node_stack = (NODE_STACK *) calloc(1, sizeof(NODE_STACK));
      if (stack_top == NULL) return MEM_ERR;
   }
   stack_top->node = node;

   return NO_ERR;
}  /* push_node() */


NODE *
pop_node(void)
{
   NODE *top_node;
   NODE_STACK *curr;

   if (stack_top)
      top_node = stack_top->node;
   else return NULL;

   curr = node_stack;
   if (curr == stack_top) {
      free(curr);
      stack_top = node_stack = NULL;
   }
   else {
      while (curr && curr->next != stack_top)
         curr = curr->next;
      free(curr->next);
      curr->next = NULL;
      stack_top = curr;
   }

   return top_node;
}  /* pop_node() */


void
clear_stack(void)
{
   while (node_stack)
      pop_node();
}  /* clear_stack() */

/*-------------------------------------------------------------------------*/

#ifdef DEBUG

void
debug1(int num_lvl1)
{
   int *symptoms;
   QUALITY *qualities;
   int i, j;

   printf("\n%d data found.\n", num_data);
```

```
   for (i = 0; i < num_data; i++) {
      printf("  %10s: ", data_spc[i].name);
      printf("%10.4lf %10.4lf %10.4lf\n", *(data_spc[i].value),
            data_spc[i].LOW_LIM, data_spc[i].HIGH_LIM);
   }

   printf("\n");
   printf("%d level one rules found.\n", num_lvl1);
   for (i = 0; i < num_lvl1; i++) {
      symptoms = lvl1_rules[i].symptoms;
      qualities = lvlT_rules[i].quality;
      printf(" Specialist number %d\n", lvl1_rules[i].specialist_index);
      printf("  %d number of predicate symptoms:\n",
            lvl1_rules[i].num_symptoms);
      for (j = 0; j < lvl1_rules[i].num_symptoms; j++) {
         printf("   symptom: %d -- %3d\n", symptoms[j], (int) qualities[j]);
      }
      printf("   CF weight = %1.4f\n", lvl1_rules[i].cf);
   }
} /* debug1() */


void
debug2(int num_lvl2)
{
   NODE *curr_node;
   PREDICATE *curr_pred;
   int i, err;

   for (i = 0; i < num_lvl2; i++) {
      clear_stack();
      printf("\n%s\n", k_tree[i].specialist_name);
      curr_node = k_tree[i].root;
      while (curr_node) {
         while (curr_node) {
            err = push_node(curr_node);
            if (err != NO_ERR)
               printf("Warning: memory full!\n");
            curr_node = curr_node->sibling;
         }
         while (!curr_node && node_stack) {
            curr_node = pop_node();
            printf("--> %s\n", curr_node->name);
            curr_pred = curr_node->assumption;
            while (curr_pred) {
               printf("    f = %s()\n", curr_pred->usr_func_name);
               curr_pred = curr_pred->next;
            }
            curr_node = curr_node->offspring;
         }
      }
   }
} /* debug2() */

#endif

/*-------------------------------------------------------------------------*/
```

```
/*====== buftnkif.c ==================================*
   Data interface to buffer-tank simulation (buftank.c)
 *====================================================*/

#include <stdlib.h>
#include <math.h>
#include <x.h>
#include <xtime.h>
#include "AKShell.h"
#include "buftank.h"
#include "data_if.h"
#include "errmsg.h"

#define NUM_IF_DATA 11  /* number of interfaced data */
#define IF_DELTA 50     /* interface period */

/* receive data macro */
#define receive_data(d) \
   dummy = d; \
   input(d) perform \
      bb_new_event_flag = (exceed_threshold(d, dummy, threshold)) ? \
         TRUE : bb_new_event_flag; \
   else d = dummy


/* interfaced data */
real_t if_tank_h[3];    /* tank level */
real_t if_setpt[3];     /* set-point */
real_t if_tank_u[3];    /* valve control signal */
real_t if_Kc, if_TI;    /* controller constants */

/* prototype of support functions */
BOOLEAN exceed_threshold(real_t, real_t, real_t);
void qualify_data(INP_DATA *, int);


/*---------------------------------*
   Declare interface data using the
   macros provided in data_if.h.
 *---------------------------------*/

begin_if_data_list(NUM_IF_DATA)
   define_if_data(tank_h0, if_tank_h[0]);
   define_if_data(tank_h1, if_tank_h[1]);
   define_if_data(tank_h2, if_tank_h[2]);
   define_if_data(setpt0, if_setpt[0]);
   define_if_data(setpt1, if_setpt[1]);
   define_if_data(setpt2, if_setpt[2]);
   define_if_data(tank_u0, if_tank_u[0]);
   define_if_data(tank_u1, if_tank_u[1]);
   define_if_data(tank_u2, if_tank_u[2]);
   define_if_data(tank_Kc, if_Kc);
   define_if_data(tank_TI, if_TI);
end_if_data_list()


/*----- data_if() ------------------------------*
   Data-interface: reads in new data values at
   regular intervals.

   data quality defined by:
      'high' = within top 1/3 of data range
      'normal' = middle 1/3 of data range
      'low = within bottom 1/3 of data range
 *----------------------------------------------*/

X_TASK data_if(void)
{
   /* process state variables */
   state_variable(real_t, tank_hgt);    /* tank height */
   state_variable(real_t, tank_setpt);  /* tank height set-point */
   state_variable(real_t, tank_u);      /* valve control signal 0 - 1 V */
   state_variable(real_t, tank_Kc);     /* proportional gain */
   state_variable(real_t, tank_TI);     /* integral time constant */

   /* interface state variables */
   state_variable(int, bb_num_data);            /* number of system data */
   state_variable(INP_DATA *, bb_data_spc);     /* ptr to system data area */
   state_variable(int, bb_data_period);         /* validity period of data */
   state_variable(BOOLEAN, bb_new_event_flag);  /* new event flag */

   /* kb-compiler provided variables */
   extern int num_data;                 /* number of compiled data */
   extern INP_DATA *data_spc;           /* physical location of data area */

   /* internal variables */
   time_t timer;                        /* interface scan timer */
   real_t threshold;                    /* new event threshold */
```

```
BYTE buffer[20];                    /* user-input buffer */
real_t dummy;                       /* used for receiving data */

initialise_blackboard;
initialise_time;

set_path(system_2, buftank, buftank);
   time_frame(tank_hgt, f);
   time_frame(tank_u, f);
set_path(system_2, buftank, monitor);
   time_frame(tank_Kc, f);
   time_frame(tank_TI, f);
   time_frame(tank_setpt, f);
set_path(system_1, data_if, data_if);
   time_frame(bb_num_data, d);
   time_frame(bb_data_spc, p);
   time_frame(bb_data_period, d);
   time_frame(bb_new_event_flag, b);

x_create_window(17, 1, 9, 60, 0, 0, " Data Interface ", X_VISIBLE);

/* initialise and attach to system */
x_begin
   /* initialise interfaced data */
   if_tank_h[0] = if_tank_h[1] = if_tank_h[2] = tank_hgt = 0;
   if_setpt[0] = if_setpt[1] = if_setpt[2] = tank_setpt = 0;
   if_tank_u[0] = if_tank_u[1] = if_tank_u[2] = if_tank_u[3] = tank_u = 0;
   if_Kc = tank_Kc = if_TI = tank_TI = 0;

   bb_new_event_flag = FALSE;
   bb_num_data = num_data;
   bb_data_period = 0;
   bb_data_spc = data_spc;
   with_validity(DEFAULT_PERIOD) output(bb_num_data);
   with_validity(0) output(bb_data_period);
   with_validity(0) output(bb_data_spc);

   /* establish new event threshold */
   do {
      x_printf("Enter threshold for new event: ");
      x_read(buffer);
      threshold = atof(buffer);
      if (threshold <= 0) x_yield();
   } while (threshold <= 0);

   /* establish data validity length */
   do {
      x_printf("Enter data validity period: ");
      x_read(buffer);
      bb_data_period = atoi(buffer);
      if (bb_data_period <= 0) x_yield();
   } while (bb_data_period <= 0);

   /* kick off data interface */
   with_validity(DEFAULT_PERIOD) output(bb_data_period);
x_end

reset_timer(timer);     /* reset sampling period */
x_timer(&timer, IF_DELTA);

FOREVER {
   /* receive data: check if new event threshold exceeded */
   receive_data(tank_hgt);
   receive_data(tank_setpt);
   receive_data(tank_u);
   receive_data(tank_Kc);
   receive_data(tank_TI);

   x_printf("\nlevel = %1.4lf, sp = %1.4lf, u = %1.4lf ",
            tank_hgt, tank_setpt, tank_u);
   if (bb_new_event_flag) x_printf("! ");

   /* cycle historical data */
   if_tank_h[2] = if_tank_h[1]; if_tank_h[1] = if_tank_h[0];
   if_setpt[2] = if_setpt[1]; if_setpt[1] = if_setpt[0];
   if_tank_u[2] = if_tank_u[1]; if_tank_u[1] = if_tank_u[0];
   if_tank_h[0] = tank_hgt;
   if_setpt[0] = tank_setpt;
   if_tank_u[0] = tank_u;
   if_Kc = tank_Kc; if_TI = tank_TI;
   qualify_data(bb_data_spc, bb_num_data);   /* verify data */

   /* update blackboard at set intervals */
   x_begin
      with_validity(bb_data_period) output(bb_data_spc);
      with_validity(DEFAULT_PERIOD) output(bb_num_data);
      with_validity(DEFAULT_PERIOD) output(bb_data_period);
      with_validity(DEFAULT_PERIOD) output(bb_new_event_flag);
```

```
      x_end

      /* reset event flag if set */
      if (bb_new_event_flag) {
         x_begin
            bb_new_event_flag = FALSE;
            with_validity(DEFAULT_PERIOD) output(bb_new_event_flag);
         x_end
      }

      /* wait for interface time */
      before(timer, IF_DELTA) perform {
         x_yield();
      }
      reset_timer(timer);
      x_timer(&timer, IF_DELTA);
   }
} /* data_if() */


/*----- qualify_data() ------------------------*
   Checks data range and defines data quality.
      NB: range == 0 implies that the data
   being checked is a user-constant/variable,
   and is therefore always valid.
 *---------------------------------------------*/

void qualify_data(INP_DATA *data, int n)
{
   real_t range, fill;
   int i;

   for (i = 0; i < n; i++) {
      range = data[i].HIGH_LIM - data[i].LOW_LIM;
      data[i].validity = (*(data[i].value) >= data[i].LOW_LIM &&
                          *(data[i].value) <= data[i].HIGH_LIM) ||
                          range == 0;
      fill = (*(data[i].value) - data[i].LOW_LIM) / range;
      if (fill < ONE_THIRD)
         data[i].quality = LOW_VAL;
      else if (fill < TWO_THIRD)
         data[i].quality = NORM_VAL;
      else
         data[i].quality = HIGH_VAL;
   }
} /* qualify_data() */


/*----- exceed_threshold() --------------------------------*
   Returns TRUE if δ(interfaced data) > threshold, where
   threshold (>= 0) is expressed as fractional change
   from previous data value, eg 0.1 for 10% change. Returns
   FALSE if δ is less than the expressed threshold.
 *-------------------------------------------------------*/

BOOLEAN exceed_threshold(real_t new_val, real_t old_val, real_t threshold)
{
   real_t divisor;

   if (!old_val) {
      if (!new_val)
         return FALSE;
      else
         divisor = new_val;
   }
   else
      divisor = old_val;

   if (fabs((new_val - old_val) / divisor) > threshold) return TRUE;
   else return FALSE;
} /* exceed_threshold() */
```

Appendix B

Compiler error messages

Error code	Message
1	out of memory.
2	unexpected end of file.
3	token/label expected.
4	bad character in label.
5	illegal floating point.
6	illegal integer.
7	invalid data.
8	undefined action.
9	failed to link node.
10	invalid knowledge base declaration.
11	function list overflow.
12	data undefined.
13	level-one undefined.
14	level-two undefined.
15	undefined user-function.
16	empty function list.
17	cf value out of range.
18	undefined CF-function.
19	interface-data list overflow.
20	empty interface-data list.

Appendix C

Derivation of the buffer tank simulation

A schematic diagram of the buffer tank modelled here is shown in Figure C.1.

1. Rate of accumulation of the buffer tank (m^3/s):

$$\frac{d}{dt}Ah = q_1 - q_2$$

2.

$$q_2 \propto \sqrt{\rho g h}$$

Thus, for constant ρ,

$$q_2 = C_1\sqrt{h}$$

where C_1 is a constant. Therefore,

$$A\frac{d}{dt}h = q_1 - C_1\sqrt{h}$$

For laboratory system, $q_1 \approx 2$ litres in 64 seconds, therefore,

$$q_1 = 3.1 \times 10^{-5}$$

Put $\frac{dh}{dt} = 0$,

$$C_1 = \frac{q_1}{\sqrt{h}} = \frac{3.1 \times 10^{-5}}{\sqrt{0.5}} = 4.4 \times 10^{-5}$$

So, an estimate for C_1 is $C_1 = 1 \times 10^{-4}$.

cross-sectional diameter	0.2 m
max. level (h_{max})	1 m
cross-sectional area (A)	0.03 m^2
control signal (u)	0 – 1 V
max. input flow (q_1)	3.1×10^{-5} m^3/s

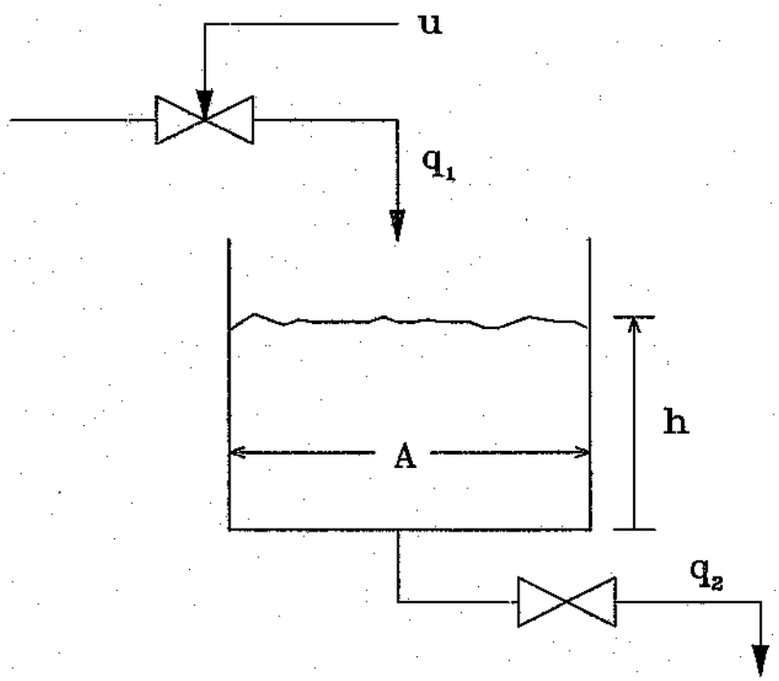


Figure C.1: Schematic diagram of the buffer-tank

3. Linearized model:

$$\frac{d}{dt}A\tilde{h} = \tilde{q}_1 - \frac{C_1}{2\sqrt{h_s s}}\tilde{h}$$

Therefore,

$$\frac{2A\sqrt{h_s s}}{C_1}\frac{d}{dt}\tilde{h} + \tilde{h} = q_1(\frac{2\sqrt{h_s s}}{C_1}$$

i.e.

$$\tau\frac{d}{dt}\tilde{h} + \tilde{h} = gain(q_1)$$

$$\tau = \frac{2A\sqrt{h_s s}}{C_1} = 424 gain = \frac{2\sqrt{h_s s}}{C_1} = \frac{2\sqrt{0.5}}{10^{-4}}$$

Thus, $\tau = 424$ seconds (7 minutes) and $gain = 1.4 \times 10^4$ m/m^3/s.

4. Forward difference (Euler) approximation: For digital simulation, if $D \equiv \frac{d}{dt}$, then,

$$\begin{aligned} Dx(t) &\approx \frac{x(t+\Delta) - x(t)}{\Delta} \\ &= \frac{q-1}{\Delta}x(t), t = k\Delta \\ &= \delta x(t) \end{aligned}$$

Implementation:

$$y = \delta^{-1}x = \frac{\Delta}{q-1}x$$

$$y(k) = y(k-1) + \Delta x(k-1)$$

5. Valve:

$$q_1 = (5 \times 10^{-5}/1)u$$

6. Tank plus value:

$$\frac{d}{dt}\tilde{h} = \frac{1}{A}(q_1 - \frac{C_1}{2\sqrt{h_s s}}\tilde{h}$$

$$q_1 = 5 \times 10^{-5}u$$

Put in s-domain:

$$s \cdot h = \frac{1}{A}(5 \times 10^{-5}u - C\sqrt{h})$$

Therefore,

$$h(k) = h(k-1) + \frac{\Delta}{A}(5 \times 10^{-5}u(k-1) - C\sqrt{h(k-1)})$$

7. **PI controller:**

$$e = h_s - h$$
$$u(s) = K_c(1 + \frac{1}{T_I s}E(s)$$
$$u_2 = K_c \cdot e$$
$$s \cdot u_1 = \frac{K_c}{T_I}e$$

8. **Digitize:**

$$e(k-1) = h_s(k-1) - h(k-1)$$
$$u_1(k) = u_1(k-1) + \Delta\frac{K_c}{T_I}e(k-1)$$
$$e(k) = h_s(k) - h(k)$$
$$u(k) = u_1(k) + K_c e(k)$$
$$h(k) = h(k-1) + \frac{\Delta}{A}(5 \times 10^{-5}u(k-1) - C\sqrt{h(k-1)})$$

Appendix D

Knowledge declaration for buffer tank simulation

```
BEGIN_DATA:
   DATA: tank_h0   0 1   ; tank level @ t
   DATA: tank_h1   0 1   ; tank level @ t-1
   DATA: tank_h2   0 1   ; tank level @ t-2
   DATA: tank_u0   0 1   ; valve control signal @ t
   DATA: tank_u1   0 1   ; valve control signal @ t-1
   DATA: tank_u2   0 1   ; valve control signal @ t-2
   DATA: setpt0    0 1   ; set-point @ t
   DATA: setpt1    0 1   ; set-point @ t-1
   DATA: setpt2    0 1   ; set-point @ t-2
END_DATA:

BEGIN_LVL2:
   DECISION: leak     ; leak in the tank
   CF_MODE: additive
      NODE: not_enough   ; not reached set-pt yet
         CF_WEIGHT: 0.3 -0.2
         PARENT: leak
```

```
      PREDICATE: equal_
         PARAM: setpt0 setpt1 setpt2
         REF_VAL: 0.5 0.5 0.5

      PREDICATE: less_than
         PARAM: tank_h0 tank_h1 tank_h2
         REF_VAL: 0.5 0.5 0.5

   NODE: decreasing  ; level is dropping
      CF_WEIGHT: 0.3 -0.4
      PARENT: not_enough

      PREDICATE: decreasing
         PARAM: tank_h0 tank_h1 tank_h2
         REF_VAL: 0 0 0       ; dummy values

DECISION: stuck_valve
CF_MODE: additive
   NODE: too_much
      CF_WEIGHT: 0.5 -0.4
      PARENT: stuck_valve

      PREDICATE: equal_to
         PARAM: setpt0 setpt1 setpt2
         REF_VAL: 0.5 0.5 0.5

      PREDICATE: greater_than
         PARAM: tank_h0 tank_h1 tank_h2
         REF_VAL: 0.5 0.5 0.5

   NODE: flooding
      CF_WEIGHT: 0.3 -0.2
```

```
    PARENT: too_much

  PREDICATE: increasing
     PARAM: tank_h0 tank_h1 tank_h2
     REF_VAL: 0.0 0.0 0.0        ; dummy values

NODE: cannot_empty
    CF_WEIGHT: 0.5 -0.5
    PARENT: stuck_valve

    PREDICATE: equal_to
       PARAM: setpt0 setpt1 setpt2
       REF_VAL: 0.0 0.0 0.0

    PREDICATE: greater_than
       PARAM: tank_h0 tank_h1 tank_h2
       REF_VAL: 0.0 0.0 0.0

DECISION: faulty_controller
CF_MODE: additive
NODE: no_response
    CF_WEIGHT: 0.4 -0.4
    PARENT: faulty_controller

    PREDICATE: equal_to
       PARAM: setpt0
       REF_VAL: 0.5

    PREDICATE: less_than
       PARAM: tank_h0
       REF_VAL: 0.5
```

```
            PREDICATE: less_than
               PARAM: tank_u0
               REF_VAL: 0.8
END_LVL2:

BEGIN_LVL1:
   DECISION: leak
      CF_WEIGHT: 0.1
      DATA_SET:
         tank_h0 low
         tank_h1 low
         tank_h2 low

   DECISION: stuck_valve
      CF_WEIGHT: 0.1
      DATA_SET:
         tank_u0 low
         tank_u1 low
         tank_u2 low

   DECISION: stuck_valve
      CF_WEIGHT: 0.1
      DATA_SET:
         tank_u0 normal
         tank_u1 normal
         tank_u2 normal

   DECISION: stuck_valve
      CF_WEIGHT: 0.1
      DATA_SET:
         tank_u0 high
         tank_u1 high
```

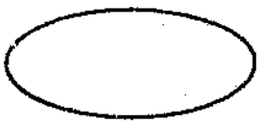
```
            tank_u2 high

      DECISION: faulty_controller
         CF_WEIGHT: 0.05
         DATA_SET:
            tank_u0 low
            tank_u0 high
            tank_u0 normal
   END_LVL1:
```

Appendix E

Knowledge declaration syntax diagrams

AKShell Knowledge Declaration Syntax Diagrams

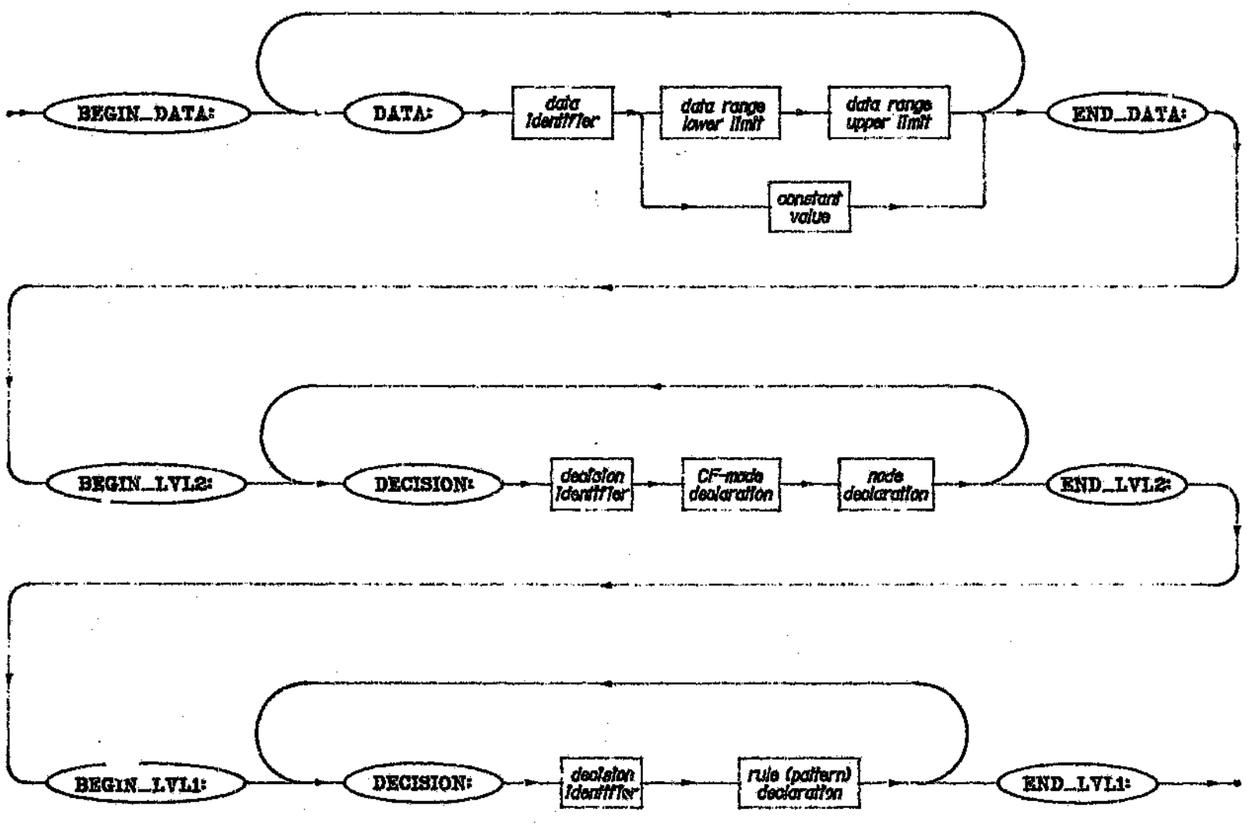
Legend



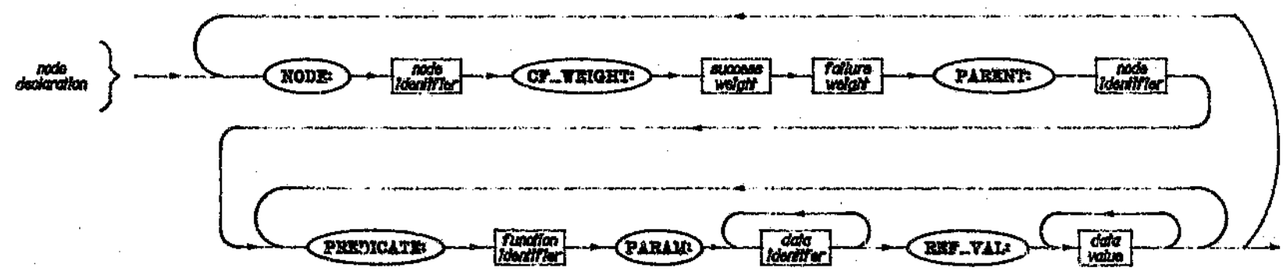
keyword entry
(reserved word)

user entry
(argument)

syntactic flow


BEGIN_DATA:
DATA:
data identifier
data range lower limit
data range upper limit
END_DATA:
constant value
BEGIN_LVL2:
DECISION:
decision identifier
CF-mode declaration
node declaration
END_LVL2:
BEGIN_LVL1:
DECISION:
decision identifier
rule (pattern) declaration
END_LVL1:


CF-mode declaration
CF_MODE:
CF function identifier


node declaration
NODE:
node identifier
CF_WEIGHT:
success weight
failure weight
PARENT:
node identifier
PREDICATE:
function identifier
PARAM:
data identifier
REF_VAL:
data value


rule (pattern) declaration
CF_WEIGHT:
CF value
DATA_SET:
data identifier
data quality (low/normal/high)