Case Western Reserve University

Center for Automation and Intelligent Systems Research

Technical Report TR 90-121 NAG3-1010 May 1990

ICE SYSTEM: INTERRUPTIBLE CONTROL EXPERT SYSTEM

James M. Vezina

Abstract

The ICE (Interruptible Control Expert) System, is based on an architecture designed to provide a strong foundation for real-time production rule expert systems. Three principles are adopted to guide the development of ICE. A practical delivery platform must be provided, no specialized hardware can be used to solve deficiencies in the software design. Knowledge of the environment and the rule-base is exploited to improve the performance of a delivered system. The third principle of ICE is to respond to the most critical event, at the expense of the more trivial tasks. Minimal time is spent on classifying the potential importance of environmental events with the majority of the time is used for finding the responses. A feature of the system, derived from all three principles, is the lack of working memory. By using a priori information, a fixed amount of memory can be specified for the hardware platform. The absence of working memory removes the dangers of garbage collection during the continuous operation of the controller.

This report describes research done at the Center for Automation and Intelligent Systems Research, Case Western Reserve University, Cleveland, Ohio 44106. Support for the Center's research is provided in part by the Cleveland Advanced Manufacturing Program and the State of Ohio.

ICE SYSTEM: INTERRUPTIBLE CONTROL EXPERT SYSTEM

By

JAMES M. VEZINA

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

Thesis Advisor: Leon Sterling

DEPARTMENT OF COMPUTER ENGINEERING AND SCIENCE

CASE WESTERN RESERVE UNIVERSITY

MAY 1990

ICE SYSTEM:
INTERRUPTIBLE CONTROL EXPERT SYSTEM

Abstract
By
JAMES M. VEZINA

The ICE (Interruptible Control Expert) System, is based on an architecture designed to provide a strong foundation for real-time production rule expert systems. Three principles are adopted to guide the development of ICE. A practical delivery platform must be provided, no specialized hardware can be used to solve deficiencies in the software design. Knowledge of the environment and the rule-base is exploited to improve the performance of a delivered system. The third principle of ICE is to respond to the most critical event, at the expense of the more trivial tasks. Minimal time is spent on classifying the potential importance of environmental events with the majority of the time is used for finding the responses. A feature of the system, derived from all three principles, is the lack of working memory. By using *a priori* information, a fixed amount of memory can be specified for the hardware platform. The absence of working memory removes the dangers of garbage collection during the continuous operation of the controller.

Acknowledgements

First I would like to thank the financial supporters of this work: NASA Lewis Research Center (under contract number NAG3-1010) and the Center for Automation and Intelligent Systems Research at Case Western Reserve University. Eric Bobinsky, at NASA, not only scrapped up the funding, but whose discussions inspired this research, thank you.

A thank you goes to the members of my committee, Professors George Ernst and Yoh-Han Pao for their comments, especially my advisor Professor Leon Sterling for his help along the way. I think it was Picasso who said it takes two people to create a painting. One to actually paint it, and another to hit the first with a stick to make him stop. Thank you Leon for being that second person.

To my friends who have helped in various ways, like the guys at AI WARE, Inc. for their patience during these last days. A special thanks goes to Farrokh Khatibi for our discussions and all the proofing you have done.

Thank you Susan, for your patience and understanding during the endless work on this thesis, now that it's over, we can get back to those wedding plans. Of course, a warm thank you goes to my family and God, for allowing all this to be possible.

To Susan,
who soon will be my bride,
and my family

Contents

List of Tables

List of Figures

Chapter 1

Terminology and Concepts

1.1 General Environmental Terminology

A coarse description of the system is needed before discussing the philosophical question of *"real-time."* The "whole world" is considered the **Environment**, as can be seen in the illustration, Figure 1.1. This includes the machinery, sensors and control parameters. The medium used to send information between the environment and control computer is called the **communications channel**. The real-time software runs on the **hardware platform** (or simply, platform). The software will be considered as the **controller**.

Events occur in the environment. They include everything that does or does not happen. One sensor value changing, and another remaining constant, are both considered events. Information about an event is sent to the controller. **Information** or **data** can represent the event itself (the affect), or be comprised of the effects of another event. If the environment groups data it transmits to the controller, then information is considered to be in a **report**. For example, due to limitations of the communication channels, remote locations in the environment may only send reports. As well as receiving data, the controller **responds** to particular events, or sets of events. A response alters the current environment in some way. The amount of time between an event occuring and the environment receives a response is called the **response time**.

The controller processes the environmental data with various **tasks**, leading to a response being issued. A **dependency path**, or bf path, is the order of tasks from the data representing an event to its response.

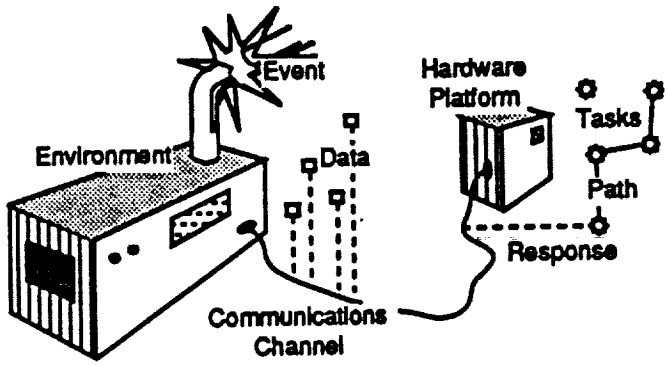


Figure 1.1: Model of the World

1.2 Model of the Environment and Controller

The model of the environment is shown, without the controller, in figure 1.2. The interface handles communications with the controller. The system state contains all the information of the environment, which is sent to the controller as a report when triggered by the internal clock. The triggering rate is set to be very fast, representing continuous data. Commands are received and placed into the command queue. If the command queue is full, new commands are accepted and discarded. It is the responsibility of the controller to be sure the environment can carry out a command. A STOP command is provided to empty the queue. In an emergency situation, the controller clears the queue so new actions can be carried out to correct the problem.

Three types of objects simulate the characteristics of the environment. **Active Controllable Agents** [Geo86] can be directly controlled by the expert system. These agents act upon the **Passive Agents** [Geo84], which cannot be directly controlled. The last set of agents are the **Active Uncontrollable Agents** [SH88]. This agent can act upon both the passive and active controllable agents. Consider the power system of the space station as

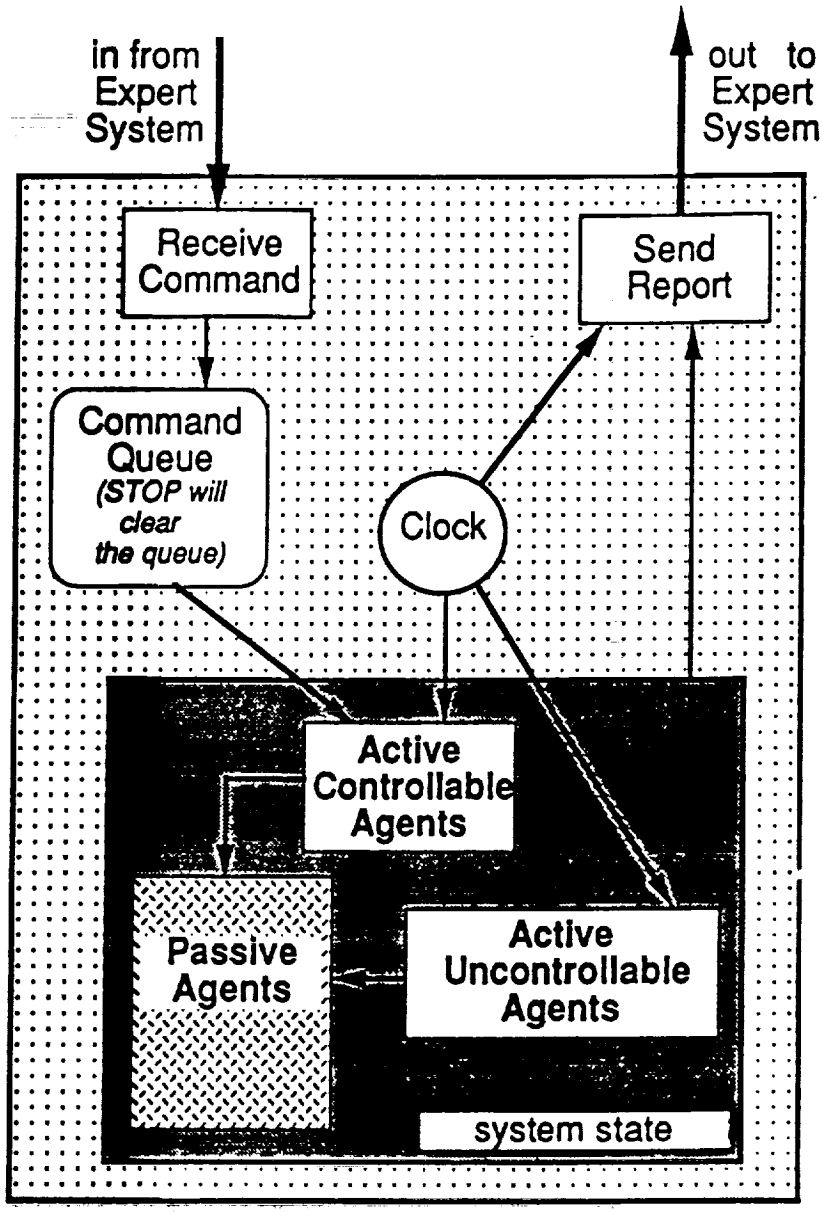


Figure 1.2: Model of the Environment

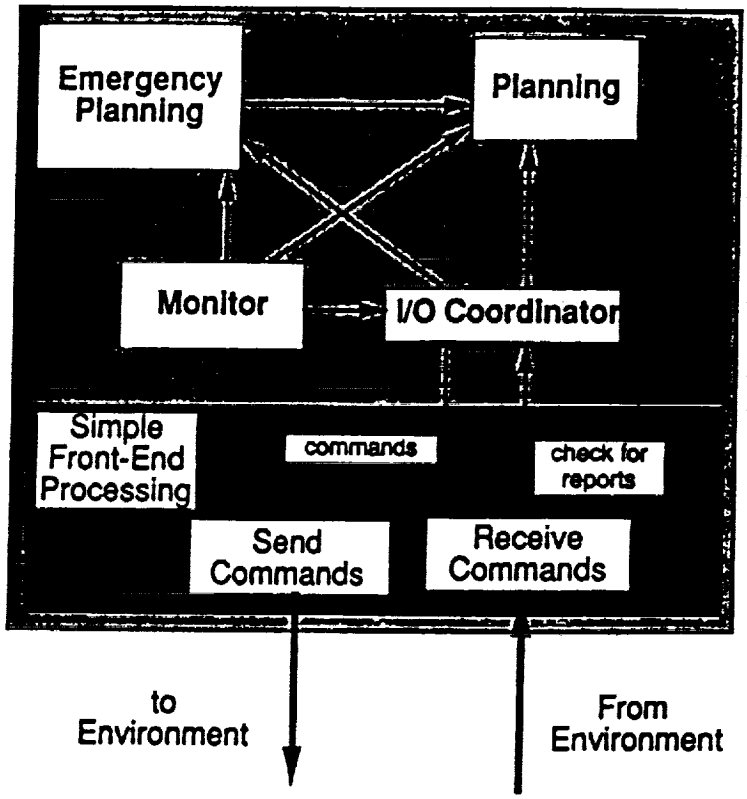


Figure 1.3: Model of the Controller

an example [Kus88][Dol87]. The breakers, active controllable agents, direct power flow over the transmission lines, which are passive agents. The last, active uncontrollable agents, are included in the environment. In this example there are two active uncontrollable agents are: experiments running in the space station (considered as black boxes with only their power requirements known), and small meteors bombarding the station. Both of these can greatly affect the operation of the power distribution system.

The controller can be implemented in many software architectures, some of which are presented in the following chapters. Regardless of the actual architecture, the controller must perform the following functions.

Figure 1.3 shows the controller also has a interface utility, the **I/O Coordinator.** It receives and processes the data from the environment. Commands sent to the environment must be managed to ensure that a response is not lost. **Monitor** verifies the operation of the environment. Planning was

divided into two parts. The **Planner** handles the typical, long term, planning for the environment. A second module, **Emergency Planner**, is added to handle the critical situations where a fast response must be issued without being concerned with all aspects of the system. Emergency rules can be made to respond to specific events. After stabilizing the environment, longer term planning restores productivity. The thesis concerns the software architecture implementing the controller, and uses the term **controller** when refering to the architecture.

1.3 Real-Time Systems

"Real-time" is often exaggerated and misused. It is used incorrectly to refer to "fast" systems, where fast can be as slow as seconds or minutes. This thesis, however, considers seconds as an upper bound, with milliseconds being used as the basic time unit. The response time is always a significant issue in definitions of real-time [KR88] [Shi87] [Ber88] [Moo86]. One definition [Ben84] concentrates on the environment controlling the actions of the software, while stressing the importance of continuous operation. Hard real-time [OC85] requires the software to respond within a designated time period. Too fast a response can be just as disastrous as one too slow.

There is a common denominator to the definitions. The software must respond to events in sufficient time to keep the environment running smoothly and minimize any further damage. Minimizing damage is crucial. There will be many situations where the environment is in a fatal state, and it is up to the controller to gracefully shut down the machinery to minimize any further damage. The software will continually be bombarded with data about various events. The controller must distinguish between possibly critical and non-critical events, and determine an appropriate plan of action. Planning must always consider the importance of a timely response.

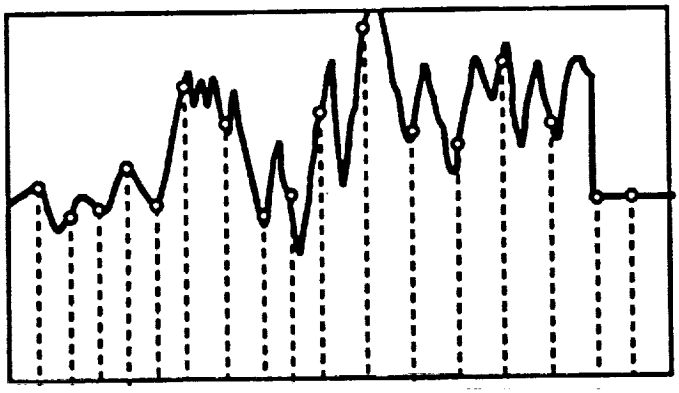


Figure 1.4: Variable Sampling Rate

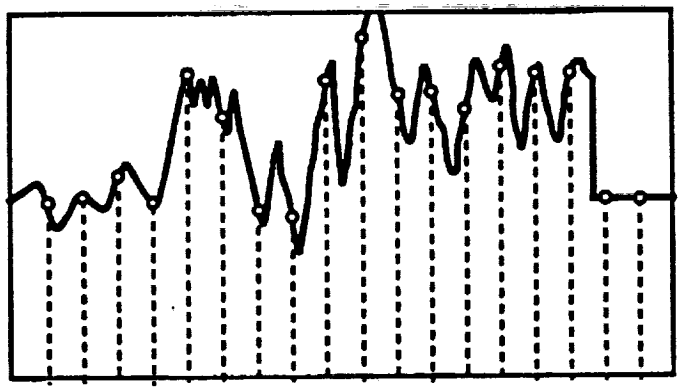


Figure 1.5: Fixed Sampling Rate

1.3.1 Data Processing

The environment is generally able to provide a continuous value for each datum, for example an analog sensor. Discrete computers must sample the signal in order to convert it to a digital value. Continuous sampling generates an enormous amount of data. The controller must use a scheme to decrease the data, while not effecting its integrity. There are four basic methods used: variable sampling rate, fixed sampling rate, fixed thresholding and dynamic thresholding. Figures 1.4 through 1.8 demonstrate the effect of each of these methods. The signal received from the environment is shown as the graph, while the points reflect the samples taken by the controller.

The first figure uses the variable sampling method [Kuo82], common to many expert systems. The data is accessed after the controller processes the

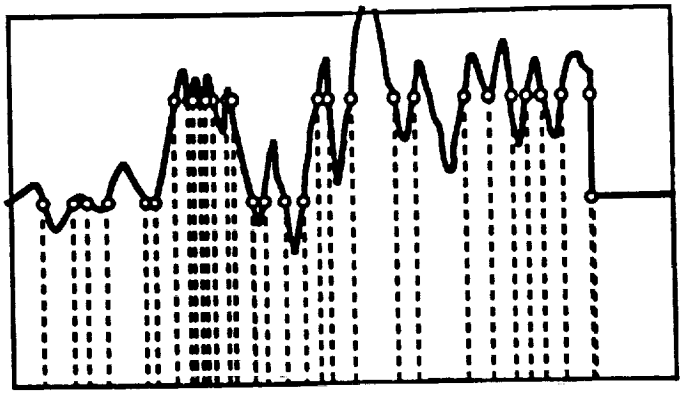


Figure 1.6: Fixed Thresholding

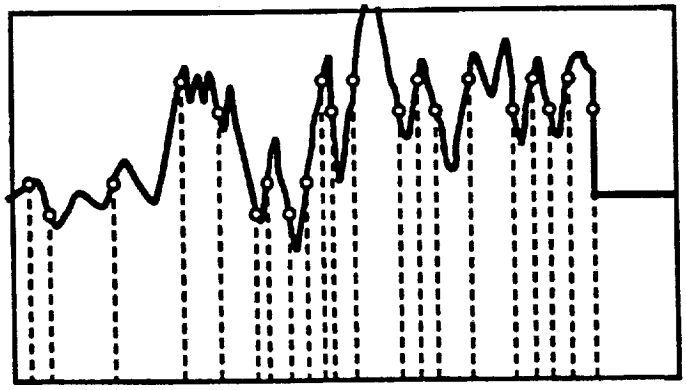


Figure 1.7: Fixed Thresholding with a Hysteresis Loop

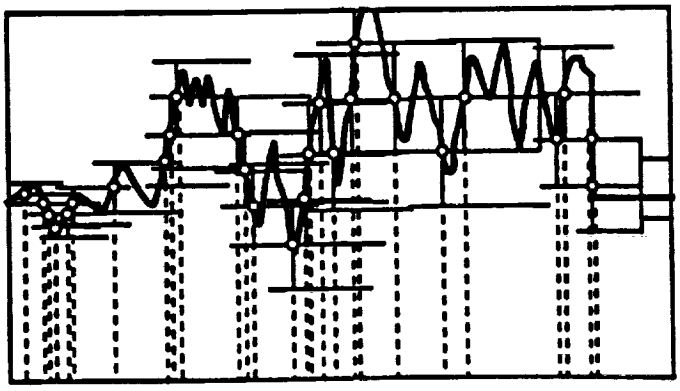


Figure 1.8: Variable Thresholding

previous samples. In this way, all of the accepted data can be considered. As the environment moves into a more dynamic or critical state, the controller requires more time to process the data, therefore the sampling rate decreases. Critical states may produce more data to be processed and the sampling rate again decreases. The controller can be literally blinded by the warning conditions of the environment and not see future fatal events.

By fixing the sampling rate, the controller is less likely to miss receiving the fatal events. Unfortunately the figure demonstrates how this situation can happen. A fixed sampling rate can still present the controller with much more data than it can handle. Figure 1.4 shows the rate at which the controller can process the data, while figure 1.5 provides much more than can be handled. A controller must then be able to distinguish which data are least important and ignore it. Another option is to retain all of the data until such time as the controller can process it, however the data validity is decaying. Validity decay is influenced by the elapsed time and responses issued by the controller. A response may invalidate the data entirely. If the controller processes data faster than the sampling rate, then it will remain inactive until new samples are received.

Thresholding provides a promising method to reduce the amount of insignificant data. As a data item remains in the current state (based on its value, trend or other aspects), it is considered to be constant and the new data is ignored. Upon entering another state, the controller recognizes the transition and the data is processed. The data is still initially received by one of the sampling techniques, but is processed by this thresholding method. This generally reduces the amount of data more than the previous two approaches alone, but the controller must still be able to cope with too much new data. Although fixed thresholding does generally reduce the incoming data, figure 1.6 provides a contradictory example. As data oscillates around a value, each pass across the threshold generates a new item to be processed by the controller. If the cycling rate and sampling rate are small, the controller receives a practically continuous signal. A similar problem arises in digital hardware as a signal changes state. The state transition is never clean,

and a certain amount of oscillation always occurs. One approach to alleviate the problem is to give the signal enough time to settle before accessing its state. Since the goal of real-time systems is to respond quickly, additional time to process data is not desired. Instead of defining the threshold as a single value, it can be defined as a band around the value, hysteresis loop. In order for the signal to move into a higher state, it must cross the higher threshold level. Correspondingly, to enter a lower state, the signal must cross the entire band. Oscillations can still occur, but their amplitude must now be greater than the threshold band. The technique is demonstrated in figure 1.7.

The last approach uses dynamic thresholding [WH89], 1.8. Instead of defining the threshold as a single value or even a range, a band surrounds the latest accepted data value. In doing so, a more accurate picture of the data can be seen, while avoiding oscillation problems. As the controller requires more processing time for a set of samples, the thresholds around data values can be expanded. In this way, the controller filters more of the new data. As more data can be processed, the thresholds are contracted. Unfortunately the problems associated with variable sampling rates appear. While the problem may not be as prevalent, the controller can still lose valuable data. This is most evident at the worst time, when a critical event is described by a tremendous amount of data. In this case, the controller must be its fastest, and be able to handle an unusually large amount of data. With all these methods, the controller must still be capable of determining the importance of incoming data. As less data can be processed, the unimportant and redundant information must be removed.

It is appropriate to consider the method used by production rule systems to determine the states of the data. Most rules map the data into one of several states. An engineer classifies a state as a range, usually with some error either way. For example, water boils over 212 degrees Fahrenheit. Due to the thermometer used and atmospheric conditions, the actual temperature might be plus or minus five degrees. Because the rules already define the thresholds of most states, the fixed thresholding method with hysteresis,

seems most logical. While variable thresholding has advantages, it does not consider the rules that are using the data. Also if a variable range grew too large, it may combine several states into one.

1.3.2 Interruptability

Asynchronous operation is important when considering data processing. An interrupt generally indicates a severe event in the environment, the controller must focus on a response. A binary signal may alter its state and interrupt the controller. State transition, particularly in the fixed thresholding approach, also causes an interrupt. The software must be capable of accepting and processing these interrupts.

1.3.3 Responding to the Critical Event

Events are continuously occuring in a real world environment. The monitoring sensors will be providing the controller with a representation of the events. The software must identify the possible events depicted by the data. The response to the most critical event is the primary concern of the controller. The approach can be better explained by using an example from the planned space station Freedom, a sponsor of this work. In considering the space station, an astronaut might be aggravated by the lights not immediately returning. However he would be dead if the life support system restoration was delayed. Based on this scenario, the less important tasks are truly trivial as compared to the critical tasks.

1.3.4 Practical Issues

There are a few practical issues that need to be addressed by a real-time system. The software must be able to run continuously (at least as long as the environment is in operation). There are sophisticated real-time expert systems that perform well on specialized hardware (*e.g.* Lisp machines) [ST86] [OD87] [KM85]. Unfortunately, they cannot run continously. This limitation is inherent in single processor systems that must garbage collect. A

controller cannot ignore the environment while garbage collecting. The ability to interface to the environment and to conventional software is the next practical issue. This is especially true for an expert system attempting to be real-time. Knowledge is important, but so are the traditional algorithms in use today.

Guaranteeing response times is briefly mentioned in every description of real-time software. However there is never a practical solution to the problem and software. Real-time programmers will typically "hack" in assembly language until the software satisfies the given test conditions or current problem. Industry generally defines the requirements of real-time software, by defining a set of tests. If these tests are satisfied, the system is said to be verified and validated (V & V).

1.4 Expert Systems

1.4.1 Typical Prototype

This section briefly describes data driven production rule expert systems [WL83]. Each rule is made up of **antecedents** (IF-parts) which must be proven true, for the **consequences** (THEN-parts) to be executed, or **fired**. The input data (describing the events in the environment) is accepted by the expert system. The raw data is then processed and represented as **facts**. These facts lead to a rule firing in two ways.

The first method **matches** the facts to the antecedents of every rule. If all the antecedents of a rule are true, the rule is then placed (scheduled) into an **agenda** (queue). This is called **activating** a rule. The activated rule with the highest priority in the agenda is fired. If any new facts are created, the whole process is repeated, otherwise the next rule in the agenda is used. The Rete algorithm is the most common approach and will be described below.

The second method does not initially do the matching. It determines the rules that "might" be activated. These are **initiated**, placed into the agenda. The antecedents of the first initiated rule are attempted to be proven. If

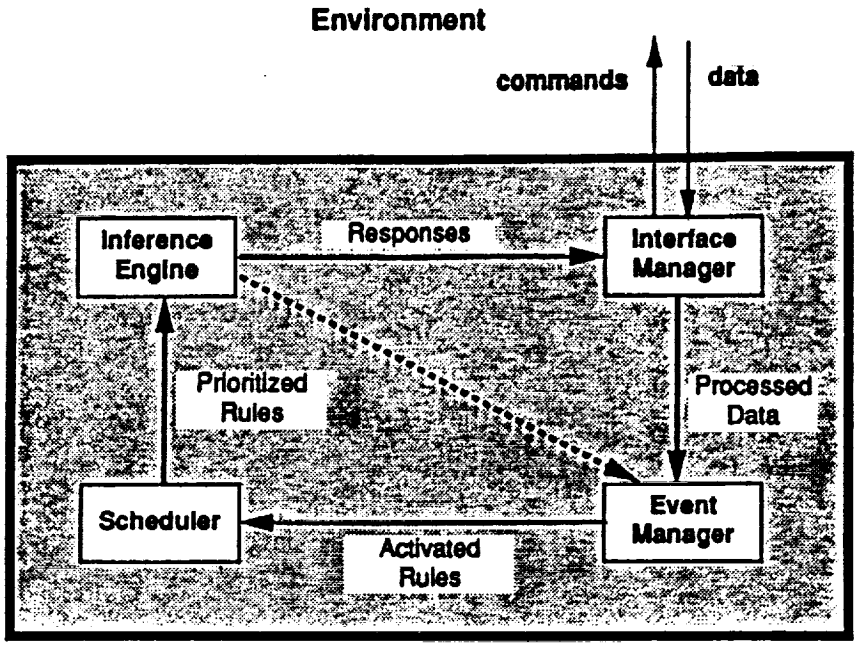


Figure 1.9: Typical Real-Time Production Rule Expert System

successful, the rule is immediately fired. Any new generated fact begins the cycle again.

Production rule systems, real-time or otherwise, follow the same basic architecture [SC88] [Ruo88]. Data is received from the environment or a user, and accepted by an Interface Manager, as seen in figure 1.9. This manager handles all communications with the environment. Processing incoming data includes one of the sampling or thresholding techniques from the previous section, though sampling may be done closer to the hardware level. The manager can also prompt for information from a user or software package, a database for instance. The environmental data must minimally be converted into data structures used by the expert system, such as facts.

The system considers the new data in the light of the previously analyzed data and determines the proper response. The Event Manager determines all possible avenues to pursue. The Scheduler, in turn, orders these possibilities.

The Inference Engine reasons about the most probable rule. Upon considering the rule, new information or facts may be created. These are passed to the Event Manager directly or through the Interface Manager. A necessary response is sent to the Interface Manager and appropriately directed to the environment. A common variation is to exclude the Interface Manager from receiving data while the current data is being analyzed. The manager can be activated by a timer, a command, or after finishing with the current data. In many cases, the interface manager accepts data based on a command from a rule firing in the Inference Engine. A command is issued after the current data is considered in enough detail to warrant the need for new data.

Event Manager

As stated, the Event Manager determines all possibilities to consider. This entails using all of the facts and activating the appropriate rules. As new facts are asserted, new rules may be activated. There are many variations on the single theme, the Rete algorithm [For82], described in the following section. Matching time is minimized by remembering all previous matches and partial matches. New assertions are compared with the minimum number of rules and previous facts. The variations generally tend to alter the amount of previous comparisons stored. The TREAT algorithm [Mir87] considers the Rete algorithm to use too much memory for the increase in performance, and therefore it saves less of the comparisons. Oflazer's algorithm [LG89] finds the two algorithms much too conservative, and requires more information to be recorded. Although much more memory is used, the performance should increase.

Matching is considered to require the most significant amount of processing time, as compared with the other three managers, approximately 85% [Gup86]. It is also one of the major obstacles in enabling an expert system to be interruptable. The system cannot be interrupted while a fact and all its effects are being matched to the antecedents of the rules and the previous facts, this will become clear when the discussion of the Rete algorithm is presented.

Scheduler

The activated rules are scheduled into an agenda, waiting to be fired by the inference engine. The ordering is based on a priority given to each rule. Some architectures group the rules into worlds [Fil88]. Each world is concerned with a different aspect of the environment. Only the rules in the current world are considered for scheduling. This topic is discussed below.

Inference Engine

This manager is also fairly straight forward. The first rule, with the highest priority, is taken from the agenda and its consequences fired. Firing creates new facts and respondes to the environment. Execution moves to the interface or event manager.

This discussion has concentrated on what is called **Data Driven**, or Forward Chaining, production rule expert systems. This means that the data or facts dictate all of the rules to be activated and fired. In general, this is the appropriate approach for control expert systems. **Goal Directed**, or Backward Chaining, is another approach. A goal is determined to be solved, or proven. The goal is a consequence of one or more rules. If the antecedents of one of these rules is proven true, then the goal is true. The antecedents of all these rules now become goals, and the process continues recursively. Goal driven expert systems are often used for diagnosis. Given information, the system determines why something will not work (the initial goal). This appears similar to controlling an environment, except the system is not informed of a problem, it must determine if one exists. Determining unknown faults is a problem for data driven expert systems.

1.4.2 The Rete Algorithm

The Rete algorithm [For82] is designed to minimize the total amount of matching time in a production rule expert system by avoiding unnecessary comparisons between facts and antecedents. The algorithm assumes the system contains a single set of rules and the facts remain relatively constant

throughout the course of a consultation. A consultation consists of starting the system and continuing until finished.

Each antecedent is made up of a number of elements. When compiling the network, each antecedent of every rule is broken down into its various elements. By combining similar elements, the amount of matching can be decreased. The algorithm can be better described by using the example in figure 1.10. Both antecedents of Rule 1 are similar to antecedents in Rule 2. Consider the network generated by Rule 1, shown in figure 1.11. The first antecedent is broken down into its elements: "a", "value", "?X" (assume a question mark denotes a variable that must be matched). The network is made a single path begining with "a" and ending with the variable X. The second antecedent generates a similar path. In this case however, the X variable must be matched to the same variable of the first antecedent. The combination of the two paths made by matching creates a join. The remaining element is then verified, to insure its existence. Upon reaching this point, the rule is activated. The power of the algorithm can be appreciated by turning our attention to Rule 2. The first and second antecedents are already mapped from Rule 1. The path of the last antecedent is similar to the one created by the second antecedent, except for the last join. Now the last element of this antecedent (the variable Z) can be joined against its corresponding element in antecedent "b", after the first join. The sets of facts that pass the new join activates Rule 2.

At each step, figure 1.11 shows the facts that currently match all of the constraints. Their addresses, or indexes, are stored in buckets. Upon considering the fact, *(a value 1)* , the first element matches the "a" bucket and is recorded there. Its second element matches "value" and it contains a third element. It is recorded in both buckets. Upon reaching the join, the corresponding bucket from the "b" path is empty so no further processing can be done. The next fact also matches the "a" bucket and is recorded. Its second element is not "value", and therefore is abandoned. Fact three is matched and recorded to the first three buckets until reaching the join. There is a matching fact in the corresponding bucket and the join is successful. Now

```
Rule 1:                          Rule 2:
     if                               if
         (a value ?X)                     (a value ?X)
         (b local ?X ?Y)                  (b local ?X ?Z)
                                          (c local ?W ?Z)
     then                             then
         { fire Rule 1 }                  { fire Rule 2 }
```

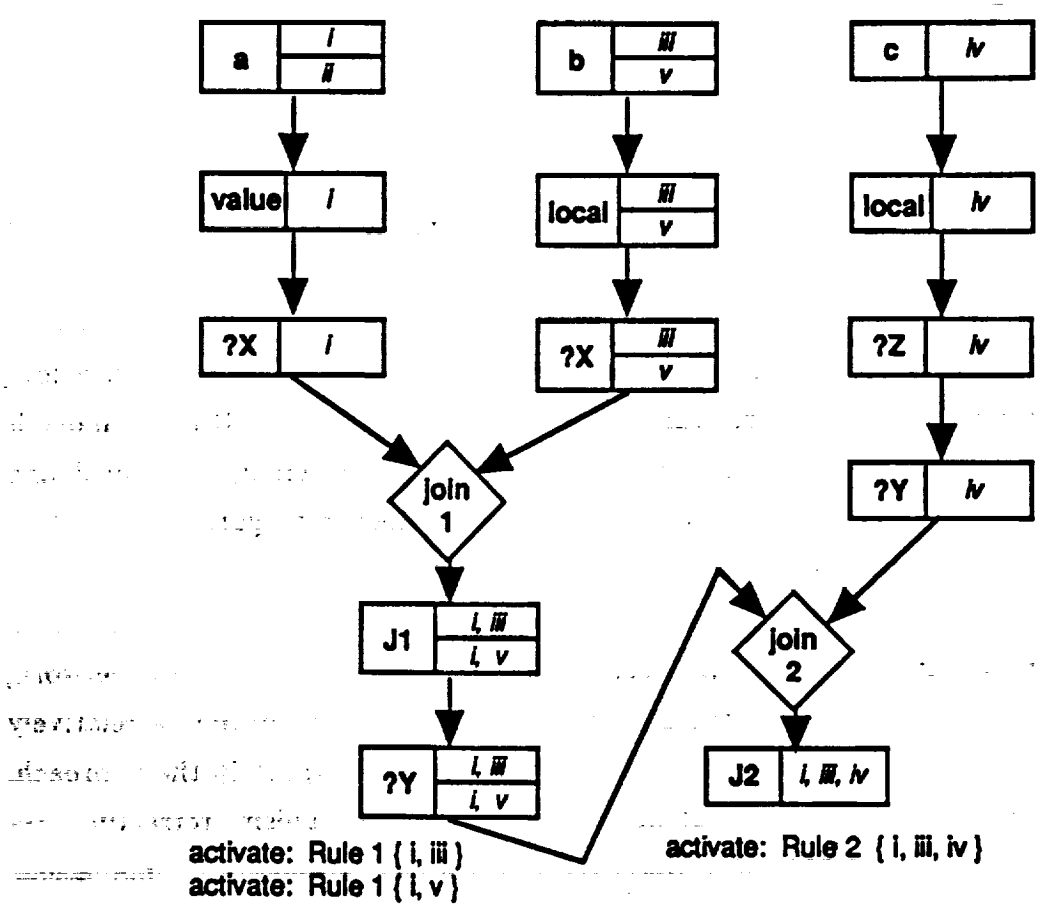
```
Facts:
        i:  (a value 1)          iv:  (c local 2 2)
       ii:  (a local 3)           v:  (b local 1 7)
      iii:  (b local 1 2)
```

Figure 1.10: Example System for the Rete Algorithm

both the first *(a value 1)* and third facts *(b local 1 2)* are recorded together, designated here by the set that contains them. The next bucket in the path verifies that the "b" fact has a fourth element. Since it is true, the two rules are recorded in the bucket, and Rule 1 is activated with this fact set.

The fourth rule is recorded into all of the buckets that it matches against. It is then joined against fact sets in the last bucket, which has activated Rule 1. The fact set is considered, and the last element of the "b" fact is joined. The "c" fact is recorded, with the previous set, in the next bucket. Rule 2 in now activated.

Assuming there were no more facts to consider, the rules (scheduled when activated) in the agenda can now be fired. After firing the first rule, assume a new fact is created, or rather asserted. Before the next rule is fired, the fact must be placed into the compiled network. The new fact *(b local 1 7)*, is now matched and Rule 1 is again activated in the same manner as with the previous "b" fact. The path continues to join 2, against "c" facts. In




```
Rule Activations

Rule 1:  { i, iii }
Rule 2:  { i, iii, iv}
Rule 1:  { i, v }
```

Figure 1.11: Network Generated by the Rete Algorithm

this case, the latest fact set does not match against any of those recorded in the other bucket. Now the next activated rule is fired. This continues until there are no more rules to be fired.

The network created by compiling the rules will fill a given amount of memory. Each node has a pointer to an area in memory serving as the bucket. A bucket can be dynamically increased or decreased. The memory used for the buckets is called **working memory**. A memory manager must distribute the available space from one bucket to another that needs it. It most likely frees the available memory from buckets that were decreased in size. The freed memory is placed into a pool that can be used for a bucket overflowing with new fact pointers. If the pool becomes too low, more stringent measures can be made in freeing memory. If all memory is being used in the buckets of the network, then the system is in deadlock and must halt. The memory manager here is analogous to garbage collection utilities in other systems.

It is evident that the total number of matches is minimized. The network generation must have a fixed set of rules to generate the data flow network, the first assumption in the algorithm. The second assumption, a relatively constant set of facts, is necessary due to a major drawback in the approach. While asserting a new fact fits nicely into the mechanism, retracting (removing) a fact does not. Upon retracting a fact, all possible combinations generated by the fact must be checked and any rules that have been activated must also be retracted. A large complex network would be unwound, and the buckets updated. The agenda also has to be searched for the rules to be removed. If a value in a fact was modified, the previous fact must be retracted and then the new fact can be asserted. Although the algorithm can handle a dynamic fact base, it is generally expected to remain fairly static.

Network compiling is not a trivial task. While the network generated by the two rules was simple, a large number of rules is much more complicated. The algorithm defines the type of network structure, but the exact graph is implementation specific. Each graph may produce dramatically different results.

The algorithm possess a more serious flaw when considering a real-time system. The fairness given to the facts is not appropriate. All facts (or data) are given the same weight. An important fact cannot preempt the matching of a less important fact. Before the critical fact is matched, it must wait until all possible combinations of a previous trivial facts are checked. All of the later facts must also be matched before any rule is fired. This forces a controller to execute the largest portion of its time before any response can be generated. Therefore the matching of an insignificant fact will postpone an important response. The first section of the results chapter shows this effect as compared to the ICE system. The second section demonstrates the results of this inefficiency.

1.4.3 Working Memory and Garbage Collection

Systems using working memory, like the Rete algorithm, reclaim used memory, garbage collection. Memory must be meticulously search for unused elements. There are techniques developed to make the job less painful, and allow the expert system to control the initiation of garbage collection. Practically speaking, memory will be scarce while the environment is in a fatal state. When the environment is in dire need of a response, the expert system is forced to hibernate until the garbage collector recovers enough memory to continue. This is a worst case scenario, but one that easily occurs. A real-time expert system should be designed to avoid the need for a memory reclamation facility.

Allowing the facts to be continuously asserted and retracted, the memory quickly becomes fragmented. The decrease in performance due to fragmentation may be solved with a memory reclamation utility. The time for garbage collection is already very expensive, and increasing the processing time is not wise. To compensate for the potential problems, an arbitrarily large amount of memory is provided with the hardware platform.

Large amounts of data compounds memory problems. As the amount of the data received from the environment increases, the need for memory management also increases. To lessen the chances of disastrous effects of garbage

collection during a crisis, more memory is arbitrarily added to the hardware platform. If the increase in memory is inadequate, then more is added. A particular prototype [HW89] uses 24 megabytes of memory to insure garbage collecting will not occur at an inopportune moment. By eliminating the need for a garbage collection facility, the controller has the additional advantage of being able to more accurately specify its requirements.

1.5 Parallel Architectures

Specialized hardware or parallel computers tremendously increase the cost of delivery and do not guarantee a definite improvement in performance. The controller becomes more complex on a parallel platform and various contention problems arise. Tasks are scheduled and managed across multiple queues and computers. Queue contention in a dynamic environment soon becomes evident.

Activated tasks are distributed across multiple computers. As concurrent processors are generating new tasks to be scheduled, each processor must wait for the master queue manager to accept and schedule these tasks. Many processing cycles will be lost because of scheduling. Similarly, a task may be forced to wait for the results of a pending task on another processor. Refering to figure 1.12, tasks T_a and T_b are needed by task T_c. Initially, task T_a, needing much processing, is scheduled to run on processor P_1; similarly, task T_b, requiring little computation, is sent to P_2. T_c is then scheduled to run on the third processor. As the first two processors are executing their respective tasks, P_3 is waiting for tasks T_a and T_b to finish. To improve this scenario, T_c is also sent to P1 and scheduled after T_a. Since T_a requires much more processing than T_b, T_a should finish much later. If T_b is preempted or unexpectedly took a long time, T_c again waits.

Increasing the number of processors also increases the amount of time necessary to manage and coordinate the system. At a certain point, adding another processor actually degrades performance. Beyond that point, adding

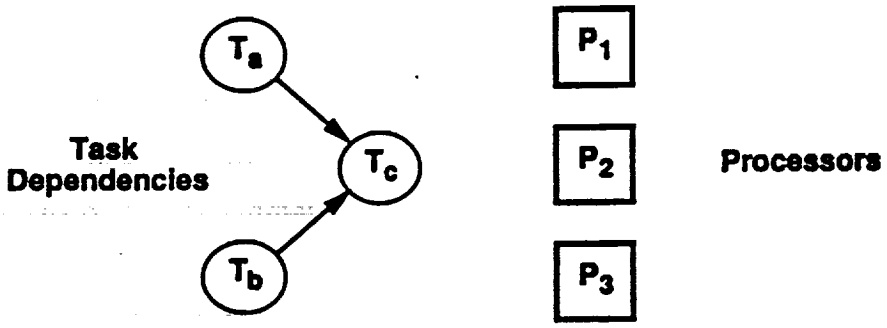


Figure 1.12: Task Dependency and Multiple Processor Example

more processors drastically decreases performance to being practically deadlocked by processor management. Improving the techniques used in the controller is a more promising solution to the problem.

1.5.1 Basic Approaches

The controller may be distributed across multiple machines. Each computer is responsible for one aspect of the software: Interface Manager, Event Manager, Scheduler or Inference Engine, each is independent. This approach is a common step in increasing performance and does not have as many of the previous contention problems. Unfortunately the bottleneck of the systems may not be affected. Production rule expert systems use approximately 85% of their time in the Event Manager, matching facts and antecedents. If the four managers were on separate computers, performance will not be improved by more than 15%, because the matching process holds up the rest of the processors. As new data is added, it will spend 85% of its time in the computer handling the matching process. The actual gain is lessened by the additional control needed to coordinate the four computers and the communication delay, to name only two. Later we will see that network communications can unexpectedly add minutes to communications delay to a system that must respond in only a few minutes.

1.5.2 Contention

Contention may arise as multiple processors access data in shared memory. As expected, one waits as the other accesses the data. Similar tasks access similar data, so it is likely that memory contention would arise many times. Read access does not contribute any constraints in resolving this problem; however updating the data would. For example, task T_a reads data D_i at the same instant as T_b requests to update the information. Task T_a reacts much differently depending on if it was allowed to access the data before or after T_b. If T_a wished to update D_i instead, then the final data value depends on which task was allowed to write last. Not only does memory contention need to be managed, but much more importantly, truth maintenance becomes a critical issue.

If a hardware bus is shared by more than one processor (as in many parallel computers) then contention arises again. A processor may wait to access data or control communications. The same problem occurs as multiple processors attempt to access the same device.

1.6 Scheduling

Although data may be properly received and validated, the time to issue all responses will most likely be much more than what is available. A critical issue is deciding which events and possible responses to pursue. An insignificant maintenance response may only take a millisecond to issue, while potential disaster recovery could involve hundreds of milliseconds. The quick response to the first event is unimportant. It is the second response that is important. The difference could be preventing a catastrophe. If the controller receives a report, the response time of the disaster recovery response is the time to consider. To complicate matters further, it may take several tasks (modules) or steps along a path [RS80] to reach a specific response. At a given time there will be many possible tasks to perform, and each is at a different stage in the development of a response.

Fairness, in operating system terms, gives each process (or set of tasks) an equal opportunity in computing time. However, fairness does not apply to real-time systems [Sta88]. A non-critical event should not interfere with the computing of a critical response. Unfortunately, it is hard to determine which is the most critical event. A path of steps leads to the highest priority response, but it may prove to be unnecessary. This could be the hardest aspect of programming real-time software.

Granularity, or size of a task, is as important as scheduling the next task to execute. Task granularity being too large might waste computing time by being involved with an unimportant task. On the other hand, very small steps consume resources because of the system overhead to plan and schedule the next task to execute. Some systems use parallel processing in an attempt to solve this problem, but it does not decrease the magnitude of the problem. Depending on the architecture of the hardware and the number of processes added, it could actually make the problem worse.

Upon processing the data, the software must determine the new tasks to be activated and schedule them along with those tasks that are still pending. While there are many scheduling techniques for a given (static) situation, dynamic environments are much more complex and requires the use of heuristics to schedule in a near optimum manner. Time is the trade off between using heuristics to optimally generate a schedule versus a simple method. While the simple method may not be optimal, it provides much more time for reasoning. A complex scheduler must also analyze the data to determine its importance. Giving priorities to the data states and tasks, may enable a simple method to produce satisfactory results. The scheduling mechanism also is constrained by the type of architecture used, and will be presented with the different architectures.

1.6.1 Task Scheduling

To make a task perform in real-time, many single queue expert systems rely on improving the scheduling algorithm. Perhaps the reason is the simple method initially used. Each rule or group of rules is given a priority. The

activated rules are sorted by their priorities and merged with the existing agenda. Variations appear in ordering rules with the same priority, either oldest first, newest first, or undetermined ordering.

Multiple queues scheduling cannot be generalized as easily. Some systems give each agenda a priority or range of priorities [Gut88]. The tasks are placed into the agenda with the equivalent priority. The Inference Engine looks to the highest priority agenda for a task. If none are present, then it continues with lower priority queues until a task is found.

Scheduling queues of multiple processors is much more complicated. The strategy executes the most likely critical tasks in parallel. The sets of tasks leading to the critical responses are spread across the processors. From before, figure 1.12 showed T_c requiring the results of T_a and T_b. So tasks T_a and T_b are each scheduled on different processors. If there were a similar task dependency for an alternate response, then its initial task is scheduled on the third processor. The queues should be dynamic enough to reschedule the tasks to take advantage of task completions. If all the tasks T_c depended upon were completed and a processor was free, then T_c is scheduled to run on the free processor.

1.6.2 Best Guess

Time plays an even more integral part of the searching method. Here we consider the data leading down various paths, where each path has a different response (responses similar to the previous method). A path is chosen based on the probability of being the best response and the time to determine if the response can be proven true or false. Using the previous example, the first operator had two choices: search for the cause or do an immediate shutdown. He determined it requires too much time to prove the corrective action. An immediate shutdown was a less optimal response, but could be accomplished in an acceptable time frame.

There are two basic approaches designed to respond in a given amount of time. They are a Best Guess [Kor87][Sor85], and searching based on the time available [PD88][Kai88]. A set of events invokes various responses

and one of these two methods are commonly used to determine which is the appropriate response.

In the course of discovering the optimal response, the controller continually updates the "best guess". When the allotted time is finished, the system uses the current (best guess) response. This can be illustrated by an example. Upon entering a smoke filled computer room, one operator may allot a small portion of time for the response and immediately shutdown all of the computer systems. Another operator may allot more time to discover the solution and look for the cause of the smoke. If the cause is quickly found, it is corrected, otherwise he too shuts down all of the computer systems. The example shows how necessary it is to correctly predict the amount of time to allocate for a response. While pursuing an optimal response, the environment may become unrecoverable, but responding too quickly may be ineffective or yield an inappropriate action. It should also be noted that there will be other important responses that must be determined at the same time. These other tasks might need to preempt the current path of tasks. Changing the focus of attention is necessary. By following a path to its completion, a response to a more critical event may be prevented.

1.7 Reasoning

Time is an important aspect in data analysis. Assuming the data is valid at the time it was sent to the controller (although faulty and noisy data must also be processed), as time passes, the validity of data may drastically decrease. The rate may be dependent on other factors in the system. The actual value may also change in the next moment. Nonmonotonic reasoning [Sho88] [MD80] [LR83] is necessary in this situation. It initially makes some assumptions (which include the validity of the data), but might revise its beliefs during reasoning about the event. The revision may be because of: data that follows, decaying data validity, or logic internal to the software. Monitoring a temperature gauge is a helpful example in explaining this concept. Monotonic reasoning has the operator record the temperature and

then go to his office to decide if the machine is working properly. There is no consideration that the gauge may have drastically increased or is continually fluctuating. Nonmonotonicity considers the changing value while reasoning. Temperature increases, or decreases, are an important trend, just as how it may fluctuate. The knowledge that the temperature tends to be lower at night and even lower during the winter months, can also put the value or trend into proper perspective. Temporal Reasoning [VK86] [HA87] [MF86] considers the aspect of the data changing with respect to time. The order of events is considered. The order can be sequential or events can overlap. Event A and B can occur during event C, but A and B may be sequential. Past events may appear in a different light when new events occur. If the temperature gauge increases slightly, it might be ignored. However, if the machinery suddenly breaks down, overheating is diagnosed based primarily on the previously ignored sensor.

1.7.1 Truth Maintenance

The second, more intricate, truth maintenance problem is much harder to solve. There are three basic methods to handle this situation [WH88]: Forward Tracing, Backward Tracing and Dynamically Setting Censors. Given the example in figure 1.13, assume that T_1 started the execution, with T_4 and T_5 now being the current tasks. If an antecedent of T_1 now becomes false, then the two current tasks need to be deactivated.

In **Forward Tracing**, the system *chases* through the paths, setting the tasks to false until deactivating the current tasks. By giving this function the ability to preempt all other processing on a single computer, the tasks can be properly deactivated. This, of course, slows the system down. If the problem is before a long trivial chain, the critical tasks are preempted for quite awhile. The importance of the chain cannot be determined from its root. T_1 may be unimportant, but it may lead to a problem that can cause a major catastrophe.

The second method is invoked when a rule is about to fire. **Backward tracing** checks the antecedents of all the rules leading to the current one.

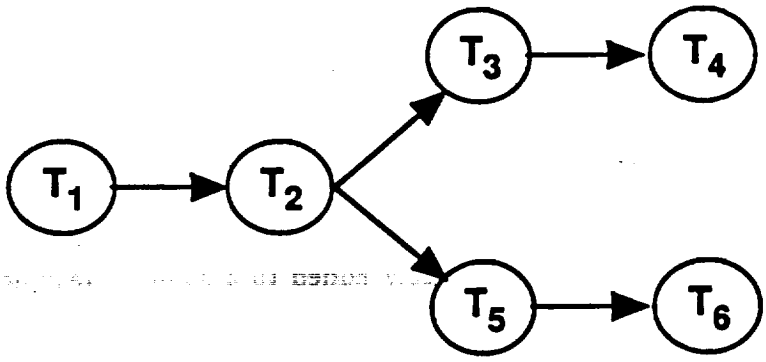


Figure 1.13: Example of Task Dependency

In cases where a response must be issued immediately, backward tracing is ignored. While this may produce an inaccurate response, it is the best choice within the given amount of time. To further enhance this approach, both methods can be used simultaneously. Forward tracing is not given the ultimate priority, it shares the processing time with the tasks. As it moves down the path to the rule to be fired, backward tracing moves in the reverse direction. Rules that are no longer valid are determined much faster, with the combined effort.

The preceding approaches to truth maintenance take time and are very cumbersome. Matching occurs again and again. **Dynamically setting censors** [MW86] [Had86] overcomes this problem, although it is not as accurate. As a rule fires, a particular censor may be set. If the rule then becomes invalid, the censor is updated. Rules farther down the path check the censor to determine the validity of the preceeding rules. The censor could also be represented as a fact. A sensor value may initiate a reasoning process and also serve as the mechanism for validity of pursuing the problem. While the sensor is in an abnormal state, the event should still be explored. The accuracy of the method is entirely determined by the censors choosen, and even then it may not be valid in every circumstance.

Tracing is geared for finding incorrect rules. A rule that is no longer

valid can be proven much faster than an invalid one. Censors allow the valid responses to be generated in the least amount of time. A censor is analogous to checksums used to determine data validity on a hardware platform. The checksums do not ensure that the is perfectly correct, but give a high probability of accuracy. The censors represented by the data from the environment are already present in the Data Table. Other censors can encode a much more complicated value, these are placed into the System State Table. The approach can be seamlessly added to a system designed with the ICE System architecture.

Chapter 2

Related Work

2.1 Parallel Implementations

When discussing real-time software, high performance is always required. Unfortunately current technology is unable to perform satisfactorily in a complex environment. Lockheed is developing one of the most well known real-time expert systems, Pilot's Associate [LG89] [LP87]. It is comprised of a Digital Equipment Corporation VAX-11/780 networked, via ethernet, with three Symbolic Lisp machines (more computers are being considered). The performance of the system is shown in table 2.1, with response times on the order of hundreds of milliseconds. However, this is reported as two to three orders of magnitude too slow. A clear solution to the performance problem is not clear, and it is hoped that the addition of more processing power will help. Another approach is improving upon one of the techniques used in the system, task scheduling for instance.

Objective	Response Time
Pilot Modeling	450 msec
Determining Pilot Intent	50 msec
Defining Threat Objects	100 msec
Assessing Target Value	500 msec
Generating Plans	450 msec

Table 2.1: Performance Requirements of Lockheed's Pilot's Associate

```
(defrule  RuleName  "comment string"
    (first antecedent)
    (other antecedents)
 =>
    (first consequence)
    (other consequences))
```

Figure 2.14: Rule Structure Used in the CLIPS Expert System Shell.

2.2 Production Rule Expert Systems

2.2.1 CLIPS - NASA's Expert System Shell

NASA developed the C Language Production System, CLIPS, [GR89] [Gia87a] [Gia87b] to provide a forward chaining production rule system based on the Rete algorithm. It is designed to be a low cost, highly portable platform to develop and deliver expert systems. The low cost is accomplished by developing the shell internally, thus eliminating profit margins and subsidizing the development costs with NASA funds. Easy integration with external systems is the third design criterion, enabling it to be embedded in applications. Although the previous section describes the matching algorithm, there are a few other points that must be mentioned.

The example from the preceeding section demonstrates the fact structure used in CLIPS. The rule structure is very similar, as can be seen in figure 2.14. Each rule is delimited by defrule and must have a unique name. The arrow (⇒) separates the antecedents from the consequences. The antecedents are matched against the fact base. Each consequence performs some kind of action. The action could assert/retract a fact, interact with the user or perform a user defined function, to name a few.

Each rule has a priority, but most use the default of zero. The priority is used to schedule an activated rule into the agenda. The rules with the same priority are scheduled as *last in first out*, a LIFO queue.

By carefully organizing the antecedents of the various rules, different effects can occur. If a rule is intended to process only one of a group of facts, the antecedent order would determine the order of the rule activations. If we further assume that the first rule fired would deactivate the other rules, then the antecedent order is very important. Consider the example of deciding on a formal outfit. If first you choose the tie to wear, then the rest of the apparel is limited. However, deciding a shirt would reversely limit the ties that can be worn. In choosing a tie to wear, a number of possibilities arise, and each would activate the rule. After firing one of the rules, the tie is selected and all of the other rules are deactivated.

2.2.2 TREAT

The TREAT algorithm [Mir87] was designed to increase the performance of the Rete approach, by improving on the method used to retract invalid facts. The Rete algorithm saves all of the successful joins in buckets throughout the network. Retracting a fact must traverse through the network for all the possible matches the fact may have. TREAT does not save all of the comparisons. The facts are initially separated and stored. The matching process proceeds in the same manner, but only the end results are saved.

An example can demonstrate the approach. Figure 2.15 shows a network created by the Rete algorithm. By adding another fact, *A2*, to the network, the results are shown in figure 2.16. As can be seen, it moves to the first join and is compared to the three 'B' elements. The successful matches are compared to the 'C' facts in the second join. When removing the fact, the system traverses the network removing all instances involving *A2*.

The intermediate buckets are not saved in the TREAT algorithm. The example of its network is shown in figure 2.17, only the initial and final results are stored. When adding the fact to the system, two steps occur. The first generates a network for that fact in the same manner as the Rete algorithm, shown in figure 2.18. Figure 2.19 represents the state once matching completes, the intermediate steps are lost. A simpler process occurs when removing the fact, figure 2.20. The fact is removed from the initial 'A'

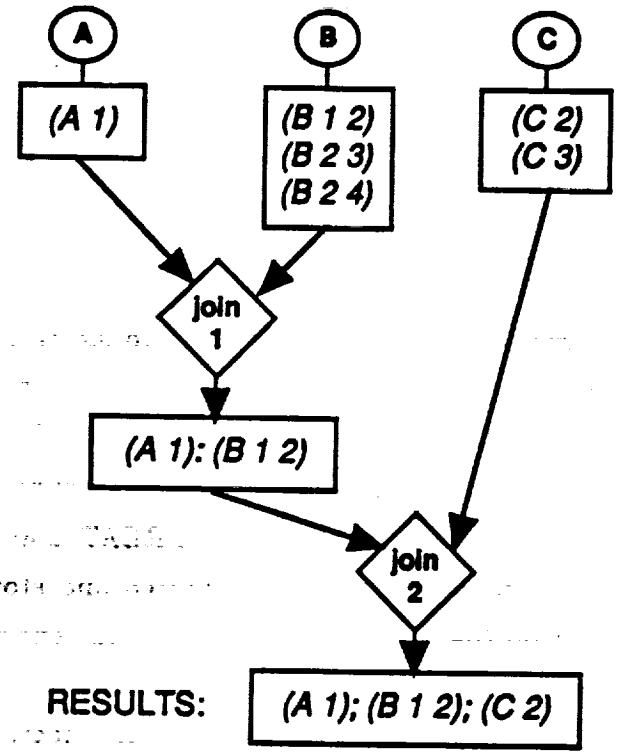


Figure 2.15: An Example Rete Network

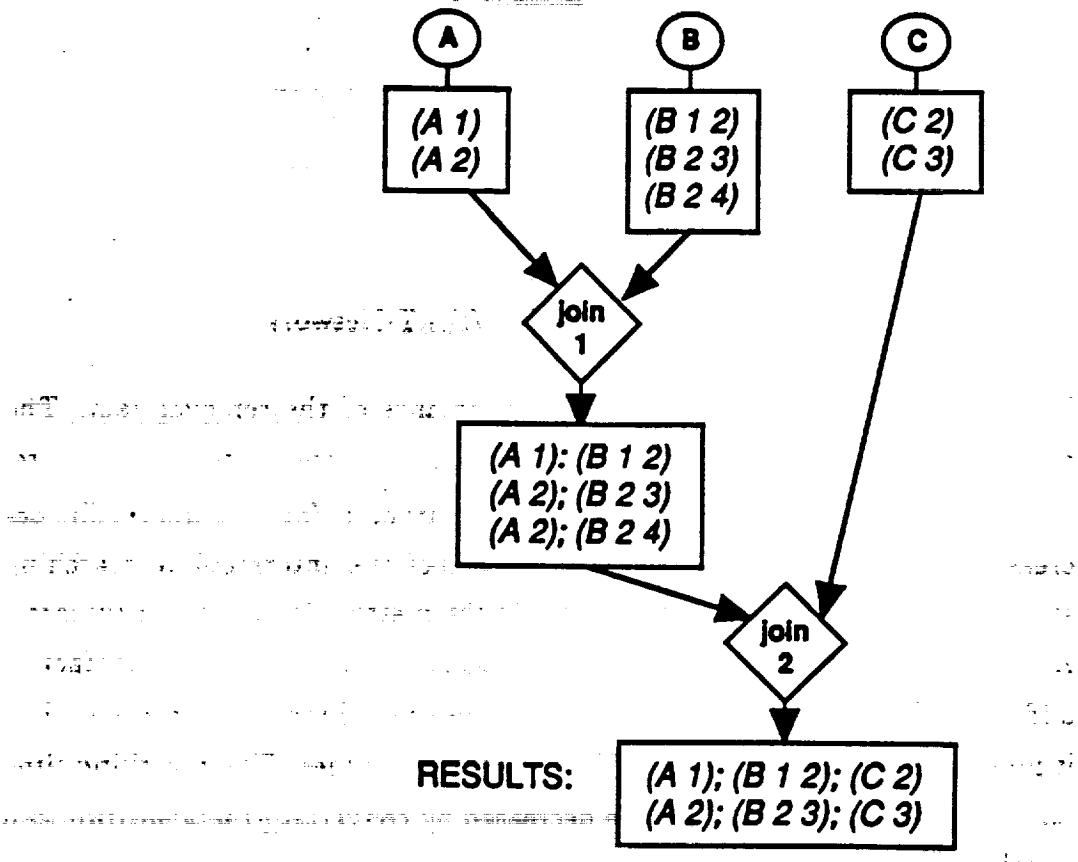


Figure 2.16: Adding a Fact to the Rete Network

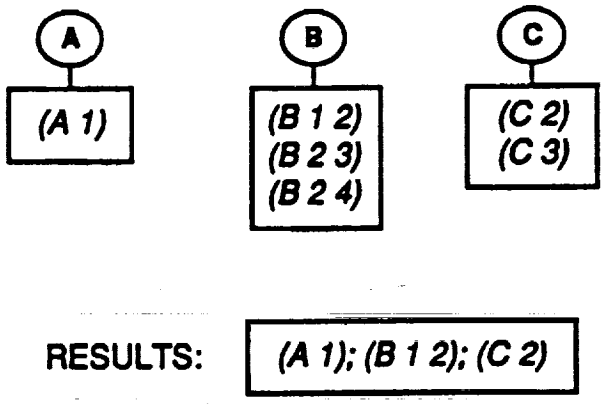


Figure 2.17: Example TREAT Network

bucket, and the results are search for an instance of the removed fact. The appropriate results are removed and the network returns to its original state.

The amount of matching necessary to retract a fact is drastically decreased. Memory is not needed to save all of the intermediate matching stages, so it too is decreased. The flaw in the method is handling new facts. A new fact requires matching to occur again. Consider a new 'C' fact is added to the example system. Not only does matching need to occur, but it must also backtrack to discover the previous matches. The matching time for new facts is considered to be decreased by converting the algorithm to a parallel machine.

Before leaving the discussion on the TREAT algorithm, the reference provides results of comparing it to the Rete algorithm. Three of the results are presented here in figure 2.21. The bars have been normalized to the Rete algorithm. The black represents the matches necessary for adding facts, and the white refers to the matches necessary for removing facts. "T1" and "T2" refer to two different searching strategies used by the TREAT implementations. The first uses lexical order when searching, while the second uses seed-ordering. The three benchmarks are briefly described as:

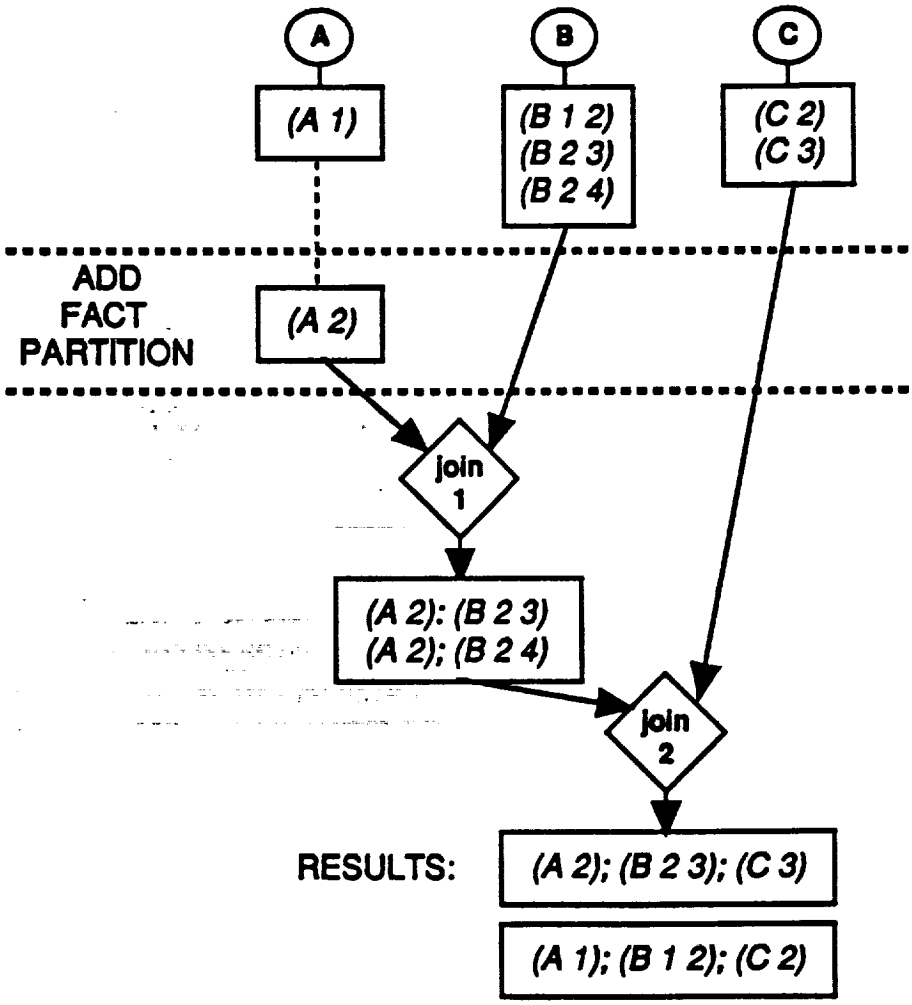


Figure 2.18: Adding a Fact to the TREAT System

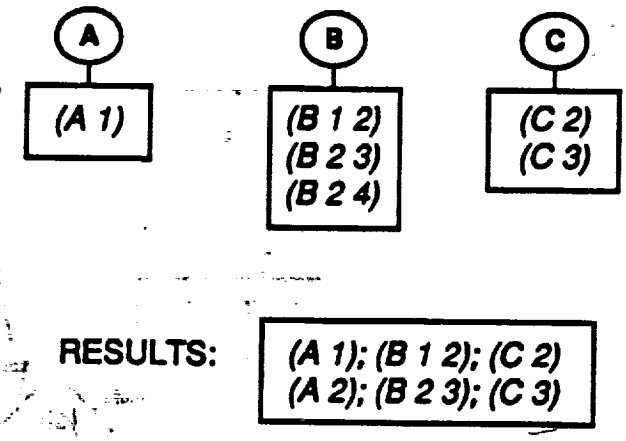


Figure 2.19: Updated TREAT Network, by Adding a Fact

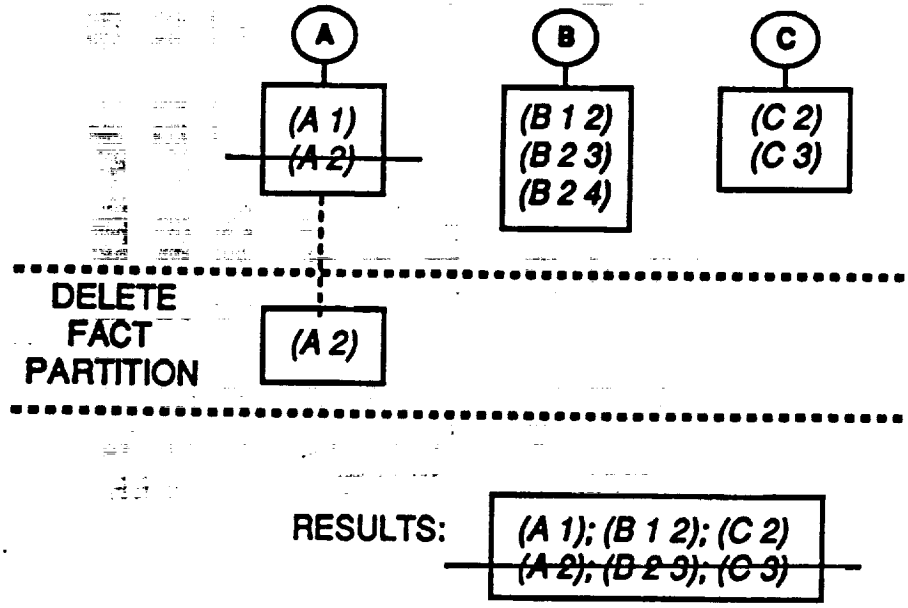


Figure 2.20: Removing a Fact from the TREAT System

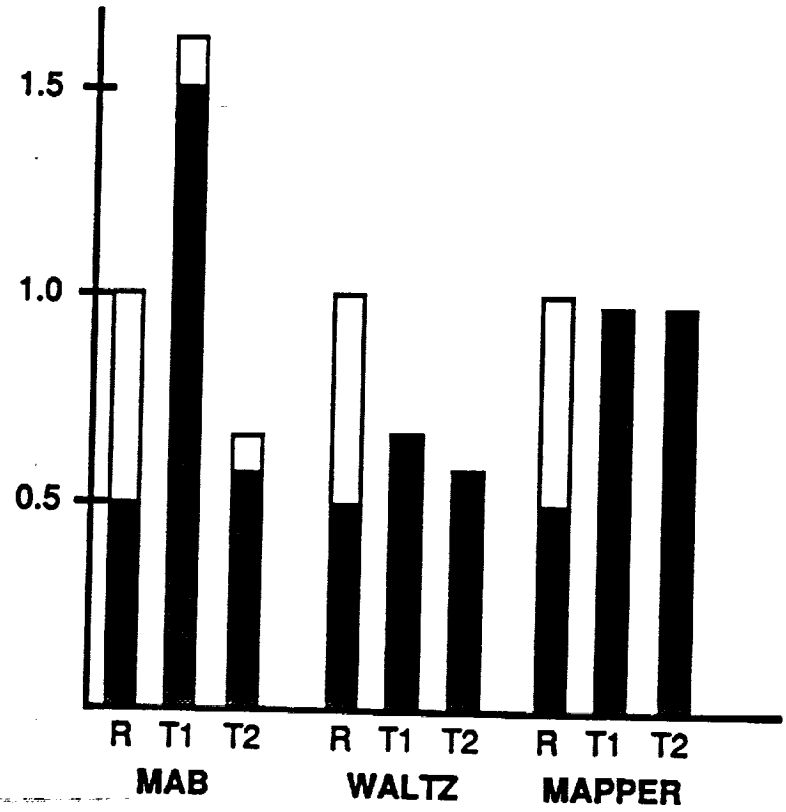


Figure 2.21: TREAT Results

1. **MAB** - 13 rules with 34 antecedents - the familiar monkeys and bananas problem [Bea85].

2. **WALTZ** - 33 rules with 130 antecedents - performs Waltz constraint propagation [Win79].

3. **MAPPER** - 237 rules with 771 antecedents - assist travelers using public transportation in Manhattan, New York. The system contains most of the bus and subway information.

These results may be encouraging for using a Rete-type algorithm on a parallel machine, but do not overcome the problems of using this approach

for real-time systems. All of the facts are still matched activating all the rules to be fired, including those for the insignificant events.

2.2.3 YES/MVS

YES/MVS [Gea84], Yorktown Expert System for MVS operators, [1] is a continuous real-time production rule expert system to continuously maintain a large IBM mainframe. The system is to dynamically maintain the mainframe by adjusting internal processing parameters to prevent a system crash. While monitoring the computer, it is also capable of analyzing performance and make recommendations. An experienced operator can perform these tasks, such operators are hard to come by and are not always available.

The system, shown in figure 2.22, is networked to the mainframe in question so that it can run as independently as possible. The MCCF, MVS Communications Control Facility, receives the filtered information from a separately developed facility called the CCOP. This external utility handles all direct communications with the mainframe being monitored and filters the messages for the expert system. Upon receiving the data, the MCCF alters the format into a fact structure.

The Operator Interface provides an operator with status of the mainframe and makes recommendations. The operator can then approve or cancel the recommended actions. If canceled, an explanation is requested. Other commands can be given to YES/MVS to send to the computer; an explanation would also be expected. The purpose of this module is to enable the operator to validate the expert system. Once a type of action is certified, YES/MVS would automatically carry it out. After proving the operation of the expert system, the operator interface would be removed.

The heart of the system is the Expert Machine. OPS5 [Bea85] is the architecture of this module. The software was ported to the IBM computer in Lisp, with some interesting enhancements. Other improvements are presented as YES/OPS [LT86]. While modifications have been made, the

[1] YES/MVS was developed by the IBM T.J. Watson Research Center in Yorktown Heights, New York.

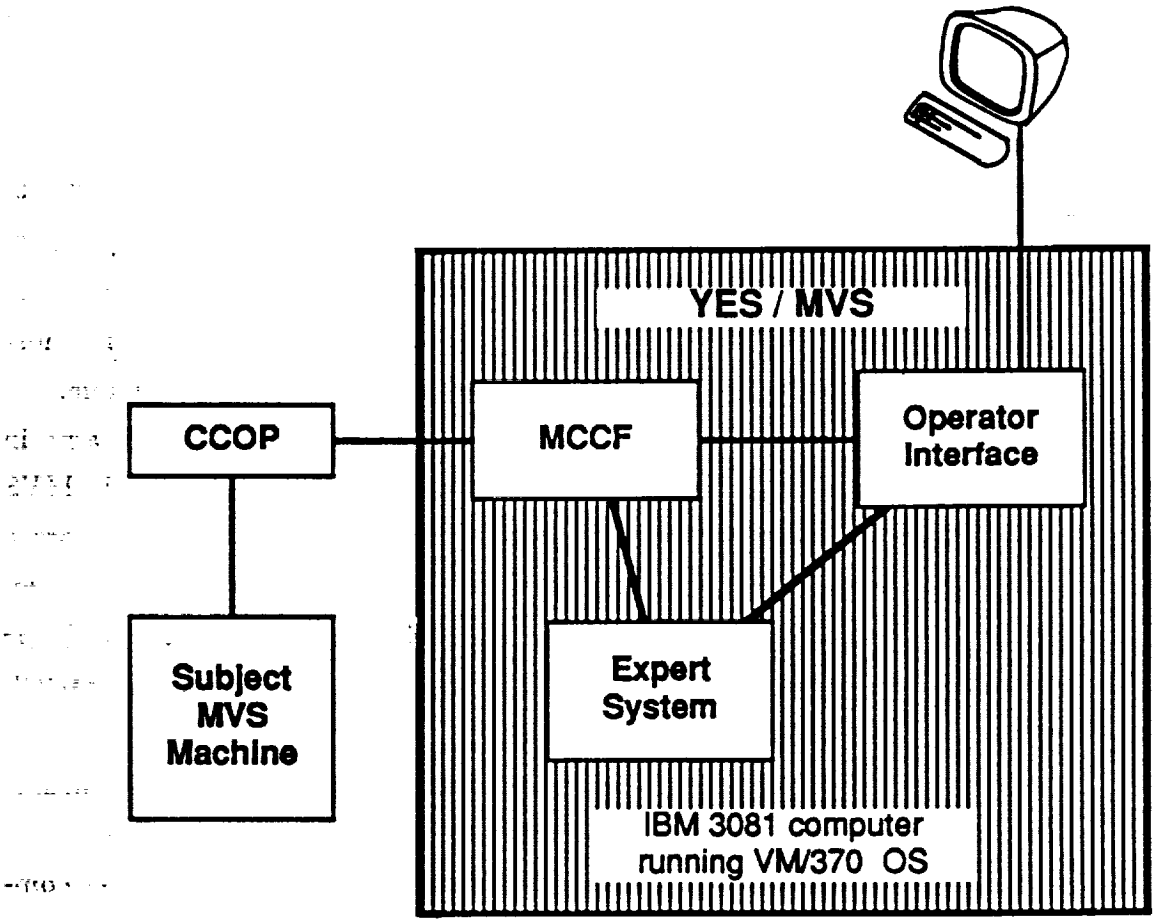


Figure 2.22: Architecture of YES/MVS

architecture follows the previous discussion, including using the Rete algorithm. These enhancements point out further deficiencies and solutions with using the Rete algorithm in a dynamic environment. However, some of these problems can only be partially solved.

When reasoning in a real-time situation, a plan will generate several responses that must be sent to the environment at specific times. On several computers, one can be dedicated to handling this, but the problem had to be solved on a single processor. The result was a new consequence command, **TIMED-MAKE**, and a **Timer Queue**. As a rule generates a command or other assertion, it could either be immediately sent to the environment, asserted or placed in the timer queue with the TIMED-MAKE instruction. The command, time to execute, and other parameters would be placed into the timer queue. At the appropriate time, it would be asserted into the fact base. If it was a command, a rule would be activated to send it to the environment.

Rules in the system are each given a priority and are associated with a task. The task also possesses a priority. When deciding the rules to be scheduled, and executed, the rules in the highest priority task would be considered first. These rules would then be ordered by their individual priority. In this way the system can easily change its focus of attention, while limiting the number of rules to be focused on. To further enhance the performance of the system, the consequences of each rule are also compiled, as is done in OPS83 [For85] and YAPS [All87]. Functionally the consequences are not changed, they are just not interpreted.

There have also been changes to matching of antecedents in the rules. Modifying a fact is one of the most glaring inefficiencies with the Rete algorithm in a dynamic environment. A sensor value changing first retracts the previous fact containing data and then asserts a new fact with the current value. Rules that do not use the element of the fact that is being altered, would be deactivated and immediately reactivated. YES/OPS allows facts to be modified. The process would follow the path of the invalid fact until reaching the bucket of the altered element. Parsing continues removing the

fact index from buckets that use the previous element but cannot use the new value. Rules that were activated by these are now deactivated. However, if the new value is also valid in the bucket, then nothing is changed. All of the rules that may be effected are left activated. The new value may also follow other paths and activate new rules that were previously not used. In this manner, there would not be any unnecessary deactivations or activations. A point to note is a side effect produced in the original method, by deactivating and then activating a rule. If rules A and B have the same priority, then the newer rule, A, would be scheduled ahead of B, the older one. Now if B is removed and scheduled because of a modified fact, B would now be younger than A. The order of the two rules would be reversed in the agenda.

While matching the antecedents of a rule, a searching method could be useful. A rule may select the highest, or lowest, value of certain sensors. The original algorithm would compare every fact to every other, searching for the extreme case. YES/OPS has implemented a mechanism to perform this type of search. The set of facts being considered is defined and the maximum or minimum would be returned.

The ordering of the antecedents in rules could impose redundant comparisons. Consider the first three rules in figure 2.23, each letter refers to antecedent. In Rule 1 and Rule 2, the 'a' and 'b' are mapped together as one. Unfortunately, Rule 3 cannot take advantage of the fact that its first two antecedents, 'c' and 'd', cannot be mapped with those in Rule 1. The reason follows from the discussion of the Rete algorithm. The data flow network would be created by parsing each antecedent of each rule, one at a time. Antecedents 'a' and 'b' would be matched, and their results are used to match the last two, 'c' and 'd'.

The antecedents can be defined in the YES/OPS system by the last three rules in the figure. Here we specify that 'c' and 'd' should be mapped independently of 'a' and 'b'. The results of the two mappings would then be joined together. In this way, Rule 3 can take advantage of the matching caused by Rule 1. The method follows the philosophy of the Rete algorithm, it is only building the data flow network that differs.

```
Antecedents of Rules in OPS5

Rule  1:  IF  a,  b,  c,  d   THEN  . . .
Rule  2:  IF  a,  b,  e       THEN  . . .
Rule  3:  IF  c,  d,  f       THEN  . . .

Antecedents of Rule in YES/OPS

Rule  1:  IF  a,  b,  (c,  d)  THEN  . . .
Rule  2:  IF  a,  b,  e        THEN  . . .
Rule  3:  IF  c,  d,  f        THEN  . . .
```

Figure 2.23: Antecedent Matching Problems in the Rete Algorithm

The problems with garbage collection systems have been brought out before. Any system using the Rete algorithm must provide some utility to handle these problems. It is done here by defining a task of three rules that would initiate and execute the garbage collecting processes. That task would receive the lowest priority so that it would not interfere with any other task. When the system has no tasks to evaluate, the focus of attention would turn to garbage collection. A rule would initiate the process, and another would actually carry the process out. The third and last rule terminates memory reclamation when finished. This task would be preempted by any other activated task in the system.

The approach should proceed smoothly with enough memory, and the ability to quickly respond to all of the data. A critical event occuring in the environment, would generate many alarms. These alarms would literally flood the controller with data. Even if the controller were able to respond immediately, the effect of the response would be delayed as the environment carries out the command. While this is going on, the controller would continue to be flooded with data. YES/OPS could be deadlocked if there were any task to be executed (other than garbage collecting) and no working memory available. The highest priority task could not execute, because of the lack of available memory. The garbage collection task would not be executed

because it would be scheduled to run after the current task.

The deadlock would be solved by allowing the garbage collection task to preempt any other task, if the available working memory was below a certain value. However this would greatly delay the response time to the critical event. Assuming that the amount of information is proportional to the severity of the event, then the memory reclamation task would be most likely to preempt the most critical tasks. To make matters worse, the task would have the most to do when it preempted the other tasks, and therefore take the longest time. To aid this situation, after interrupting the system, the garbage collection task could continue until reclaiming a specified amount of memory and then returning to its dormant state. This last approach would only lessen the harm of garbage collection.

2.3 Blackboard Systems

Data processing and response has always been the primary aspect in discussing real-time expert systems. Production rule expert systems using a data driven, or forward chaining, engine follow this approach. Blackboard architectures follow a similar approach and are also used in developing real-time systems [CH87].

A blackboard is the common area for information, but there are specific *sections* where the information can be posted. A problem would be broken down into loosely coupled subproblems, and each of these would be a section in the blackboard. In general, sections are ordered into layers, constituting intermediate solutions to the problem. Tasks are associated with each section, or rather, information within a section. A task would be a specialist in the section it was associated with and also be independent of the other specialists. While it may need information from another section, the actual operation is independent. In blackboard terms, these tasks are known as KS or Knowledge Sources. A KS can alter the data, post new data into the section, or post new data into another section. When new data is posted into a higher layer, the current layer is said to have provided a solution to

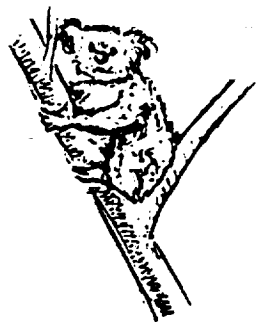


Figure 2.24: Koala in its Natural Habitat

its subproblem. The higher layer would use this solution in determining an answer to its sub-problem. Flexibility of the architecture allows a KS to be a procedural component or a set of rules. A controller mechanism is necessary to determine which KS should be activated.

An example will be used to describe the architecture in more detail. Consider the problem of finding and classifying a koala [CH87]. The koala has specific physical characteristics and habits. It has the basic look of a teddy bear, the four limbs, head, their orientation, *etc.* Figure 2.24 shows the particular look of the koala in its normal habitat. The koalas prefer to sit in the crook of branches and move up or down the tree depending on the time of day. When looking for one of the little animals, an observer would explore an area where they have been seen and look 30 to 50 feet in the trees. When seeing an animal, the observer must then classify it as a koala or not. While only a few would take their computer along and look for any animal in the trees, the problem is simple and can illustrate how blackboard systems operate.

The blackboard would be divided in the manner displayed in figure 2.25. The top layer determines if a koala is in the scene. The lower layers determine aspects of the animal. The points in the area represent information in a section, while the lines show how a KS uses one fact to determine another. The different KS illustrate how new facts can be created from information of a section. The new facts can be in the same section, as in the *Color KS*

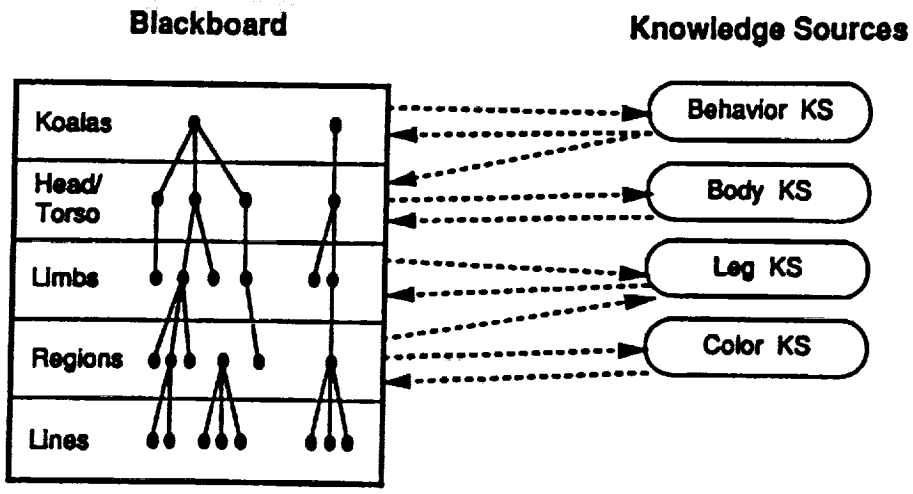


Figure 2.25: Basic Blackboard Architecture

or into a higher level, as in the *Leg KS*. When looking into an area of a view in the forest, a partial shape of an animal can be extracted by color. A patch of color can be distinguished as separate from the trees, and then associated with the animal in question. If a patch of the right color and orientation is determined, a specialist would place the information into the leg section. This new information would be processed and determined if it indeed was a leg. The figure subtly shows how the different sections and KS are independent. The body specialist does not have to know how the arm specialist determined that something is an arm, the fact that it is an arm is enough.

Now that the basic functioning of the blackboard is seen, the question arises on how a KS is activated, and which one should be executed first? Each specialist knows how it contributes to the solution of the problem and what information is needed for it to be useful. A KS can be considered as a very large rule, and its antecedents (pre-conditions) must be met before it is activated. Controllers are added in figure 2.26 to our previous example. The dotted lines refer to the flow of information, while the solid lines refer to the flow of control. The controllers monitor the new data entering the

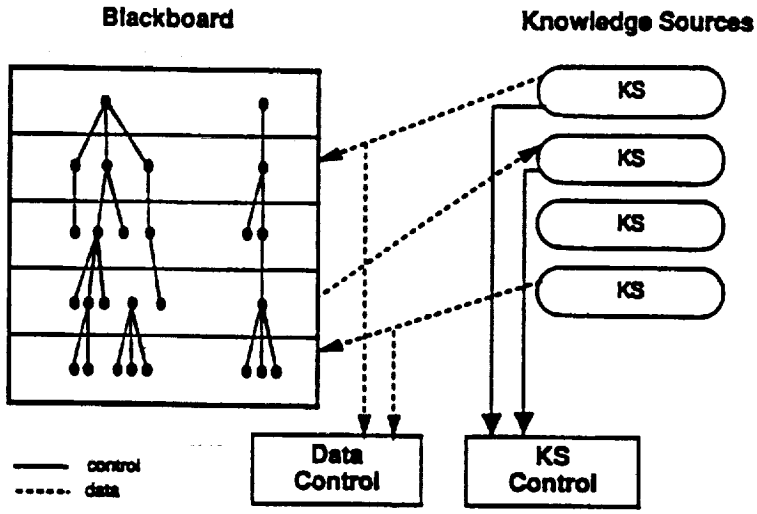


Figure 2.26: Blackboard with Controllers

blackboard and also the KS being executed. This information is then used in determining the plan of action.

In determining the solution to our problem a control plan would provide the basic direction to pursue. Based on the current information in the blackboard, a specific subproblem may be addressed. On the other hand, a new piece of information may influence the plan to change direction. From our example, the plan may suggest looking for anything that does not resemble a tree. Upon spotting a patch of color, the plan may be directed at analyzing the color or determining if it belongs to the head, limb or torso of a koala. A new direction may be decided upon based on the KS currently being used. While the color may influence the *color KS* to be activated, the *color KS* could then influence the control plan to include the *leg KS*. When working on one particular aspect of a problem, other aspects may naturally follow.

If a new piece of information is the reason for the next action, then a KS is chosen to process the data. Once chosen, the information and the KS are instantiated, paired, and executed. This is known as **event-centered scheduling.** Determining if a black spot was an eye of the koala would be an example.

When a KS is the cause of the current plan, an information object must be selected as its context. The two are instantiated using knowledge-centered scheduling. After deciding the black spot was indeed an eye, now we can proceed in trying to decide if the area around the spot is the head.

Both the new information and a KS may provoke the scheduler to choose them. In this case, the scheduler would instantiate the data as the context of the KS and proceed. While looking for the head, if a leg was seen, then we could proceed in searching for an arm or the torso.

2.3.1 The Guardian System

The Guardian system [HS89b] [WH89] is a typical example of a blackboard architecture being used as a real-time system. The purpose of the system is to monitor patients who have recently had major surgery on one or more of their vital organs. Life-support systems provide the fundamental function of the failing organs. The machines must not only keep the patient alive, but also allow the person to be weaned from the device. If a patient uses a life-support system for too long, the ailing organs will never recover. Guardian's mission is to adjust the life-support machines to the patient's current needs, while following the weaning plan. Each patient would have his or her own unique complications, so Guardian must be aware of general care and of the patient in question.

Each life-support and patient sensor (total of fifteen) is polled every twenty seconds. Two example sensors are breaths per minute and gas pressure. Lab test data is given as another five values. The lab tests are requested by Guardian, and after an appropriate amount of delay, their value is returned. Guardian would present a user with the current scenario and advice on altering parameters of the life-support machines. While Guardian could run autonomously, it currently only provides advice.

The platform of the system covers several TI Explorer Lisp machines, each with a unique function, as can be seen in figure 2.27. The lowest level machine simulates the environment, this would not exist in a real scenario. The next processor provides the interface to the environment. It transmits the

commands, and accepts and preprocesses the data, by the variable thresholding method mentioned earlier. The communications processor handles the coordination of incoming (preprocessed) data and outgoing commands. The commands may have already been issued and held until the appropriate time. The reasoning mechanism is then freed of the task of managing the already planned responses. The Lisp processors for the user interfaces provide a detailed picture of everything going on in Guardian. This includes the internal perspective of the environment, the current plan, possible solution, *etc.* Each user would only be presented with the appropriate information on their individual work station. The remaining processor is the major concern here. It is the **reasoning system**, which has been adapted from the dynamic control architecture [Hay85] implemented as BB1 [CH87], blackboard architecture. With the processed data, the reasoning system determines a plan to analyze the data, does the analysis, and responds in a timely fashion.

The reasoning system has three major processes, the **Agenda Manager, Scheduler, and the Executor.** These follow the basic blackboard strategy presented in the previous section. The events include the data from the environment and the information generated by the reasoning process. The Agenda Manager analyzes the new information and provides an agenda of possible operations. The Scheduler in turn uses the current plan of action, or control plan, and determines the focus of attention, the next operation. The Executor would execute this operation and generate more events. These events include altering the current control plan. The average time of one of these cycles is fifteen seconds, based on running the system over a forty-five minute run.

The control plan includes the aspect of the environment Guardian is concerned with and the reasoning process to use. **Associative and Model-Based Reasoning** are the two basic mechanisms. Knowledge base reasoning would be used by the first to quickly provide a response, on the order of seconds. A designer would spell out the method to determine the cause of anticipated events and appropriate responses. This is similar to production rule systems, except here the reasoning is not very deep. It is intended to

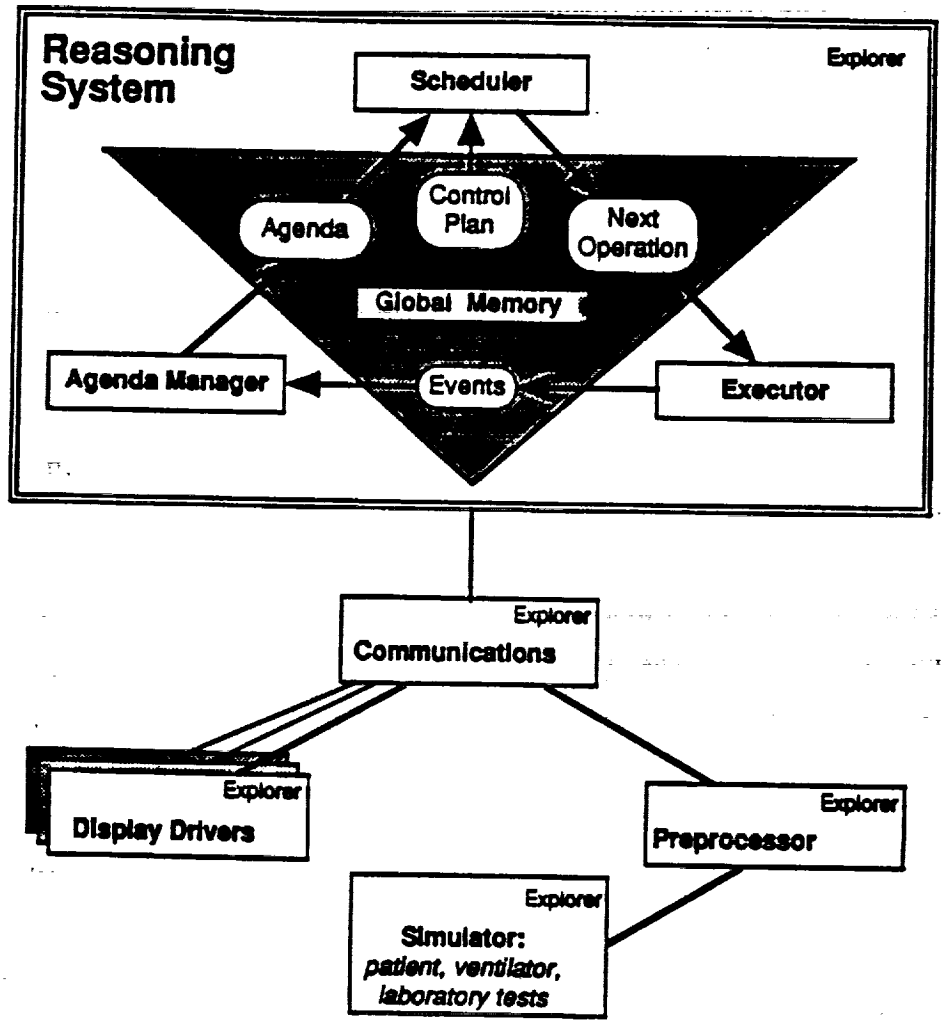


Figure 2.27: Architecture of Guardian

provide a fast response to an immediate problem. Model-based reasoning would require much more time, on the order of minutes, and attempt to determine the cause of an unknown event. This method compensates for unfamiliar problems and can also be used to correct previously wrong assertions by Guardian. The approach would be more appropriate for problems requiring a long term plan of action.

Guardian distinguishes between reasoning about the environment and determining the control plan. The two are interleaved so that the system can take as much advantage of the current state of the blackboard as possible. The control plan may use either an associative or model-based response, or a combination of the two. In this way the system has the capability to respond very fast while also being able to consider the whole environment over time.

Depending on the reasoning process being used, various aspects of the knowledge base would be used. Because of this, the knowledge of the environment is modularized into three types: **Reasoning, Domain, and First-principles.** The first, reasoning knowledge, contains information on the actual reasoning processes and the various options available to a control plan. This would present the case for one of the reasoning methods. Associative reasoning uses the domain knowledge to determine a problem in the environment. The domain also handles the specific environmental information, like the breathing rate. The basic model of a general environment would be placed into first-principle knowledge. It is the text book scenario of how various aspects of the environment operate. This is independent of the specific case, but can provide a model for the corresponding environment. By modularizing the knowledge in this manner, a module can be altered to better serve its function without affecting the rest of the knowledge base. Another reasoning scheme can be added to the system without affecting the domain and first-principle knowledge.

Guardian provides a promising direction for a real-time expert system architecture. The system contains task planning over time and data compression mechanisms. The parallel platform allows interruptability and simplifies the design of the various processes in figure 2.27. Unfortunately, the

computer network also provides transmission delays, reaching into the order of minutes [WH89]. These delays occur because of unforeseen network communications. Network transmission, no matter how fast, also slows the changes to data thresholding. As the reasoning process is overrun by new data, the change to limit new data will not go into effect immediately. Conversely, as the system can process more data, a delay would occur again. Some reported aspects of Guardian, such as model-based planning, have not yet been implemented. Other properties, such as temporal reasoning, are deemed necessary although have not been addressed.

2.4 Commercial Real-Time Expert System Shells

There are several expert system shells that are marketed as being *real-time*, but unfortunately this is more of a sales ploy rather than a reality. A typical production rule expert system is OPS-83[2]. The company boasts third-party benchmarks showing OPS-83 running faster and in less memory than other leading expert system tools. Compiling the consequences of the rules does increase the speed of the system. It is not recommended for autonomous control, rather it is meant for traditional consulting systems and as an aid to operator monitoring. Although it may be faster than many of the other products available, it is not real-time.

2.4.1 Gensym's G2

The most well known real-time expert system shell in the United States market is G2 [Wol87], by Gensym in Cambridge, MA. The product appears to be geared toward process engineers, rather than computer engineers. It enables the user to create a model of a real-time environment and simulate it

[2]OPS-83 was created by Dr. Charles Forgy, who worked in the development of Digital Equipment Corporation's OPS5. It is a product of Production Systems Technology, Inc. in Pittsburg, PA.

in non-real-time. It also has many built in capabilities for creating impressive graphical prototypes of controllers.

The user defines objects to represent various aspects of the environment using an object oriented approach. Before continuing with this package, a brief description is necessary on the standard object oriented environment. A class has a description of the attributes and methods, functions. Objects in a class all share the same types of attributes and methods, but each can be tailored. For example, a class may be a model of automobile. One of the attributes may be the color of the body and a method could be its accelerating capabilities. An object of the class could be a black car. If the black car is accelerated for two minutes, it is said that the method, accelerate, receives the message *accelerate for two minutes.* It then responds with a given speed and distance. Another object in the same class may be red and/or accelerate slower.

These methods can be defined using heuristics or in a more traditional approach. Rules are associated with a class of objects and may also be associated with related classes. The system focuses its attention on a class of objects or a problem type (although the distinction is unclear). Rules are inferenced in a data-driven, forward chaining, approach. Although the literature gives the option of backward chaining, goal driven, it is most likely done by defining the rules in a reverse manner. This same technique, and sales pitch, is common among commercial expert system shells. The data received by the system is time-stamped and the period in which it is valid.

The system can also schedule alarms at various intervals. These intervals are based on number of seconds, and there is no guarantee when the task will be acted upon, only that is should be scheduled at the given number of seconds.

The software is targeted at a very high level. The system assumes that there would be low level controllers to compensate for the performance limitations. It also appears to be more of a process monitoring system. Rather than autonomously control the environment, it would present a higher level description to an operator.

The software does enable an engineer to prototype a real-time expert system and simulate the test environment. The graphics capabilities also present an impressive demonstration. The software would be very useful in specifying the requirements of an autonomous expert system, and perhaps simulate the knowledge base using its modeling capabilities. It would be even more useful to internally sell the concept of a real-time expert system. Even though the product can be used to specify and sell the concept, it would probably not meet the demands of controlling an environment.

2.4.2 NEMO from S_2O

The Paris, France company S_2O is now marketing their product in the United States. NEMO is meant to approach the solution of real-time expert systems. It generates decision support systems that can possess some of the aspects of a real-time, similar to G2. The product has many built in graphic capabilities and is useful for high level monitoring. Temporal and nonmonotonic reasoning can be embedded into these rules and compiled into a tree-structure. The inference engine operates by forward chaining through the groups of rules. The product also has primitives to build user interfaces, access databases, and perform data acquisition. Although it may be satisfactory for the types of solutions it is being marketed to, it cannot autonomously control a complex environment.

Chapter 3

ICE System

3.1 Design Principles

The primary goal of the ICE System, Interruptible Control Expert System, is to design a production rule expert system architecture for control while maintaining a practical approach. Minimizing the response time of the most critical event is of utmost importance. As we have seen in other systems, a more general architecture was used and the rule priorities guide the system toward the critical events. Systems based on the Rete algorithm will generally minimize the response time for all of the events, including the less important ones. Minimizing the response time of the most critical event was designed into the architecture of the ICE System, at the expense of the less important responses.

The ICE System follows a different matching strategy. Instead of minimizing the total number of comparisons, it attempts to minimize the comparisons before responding to the most critical event. The approach assumes the more vital the response, the less available time. The usual approach minimizes the time for all of the responses. The first response is actually issued later, as can be seen by the results. For a given set of responses, the ICE System responds much faster to the initial responses, but the last responses may be slower. The results show that the other enhancements to the system increase performance to the point where all responses are issued much more quickly.

The design allows the environment to interrupt its operation. Interruptability was deemed necessary for a controller to be able to quickly respond to the environment. One or more alarms will generally indicate a fatal event in the environment and the expert system must immediately be made aware

of the situation. If polling the environment, the sampling rate is added to the response time of the controller. Even if a polling controller was able to immediately respond to an event, the environment does not receive any commands until the expert system requests the information. By allowing the alarms to preempt the current reasoning process of the expert system, the response is issued that much faster. The time difference may be the difference between a valid response and a catastrophe.

While striving to minimize the response time, practical issues were always kept in mind. An autonomous expert system controller is active for an extended period of time. ICE can run continuously for any period of time, as long as the hardware platform is running. To insure continuous operation, working memory was not a design option. When using working memory, there is a point where the amount of free space must be increased and a garbage collection utility is run. The memory reclamation places the controller off-line, unable to respond to environmental events, or at least delaying the necessary reasoning process. In a critical situation, the expert system will be bombarded with data from the environment. When deciding the proper response, memory may quickly become a premium. Garbage collection is used to recover memory. In this situation, the response time dramatically increases.

A fixed memory size leads into the next design criteria, deliverability. To overcome the above garbage collection scenario, some systems use specialized hardware and massive amounts of memory. Neither may be necessary or able to satisfy the control requirements of an environment. Fixing the memory size and using a single processor were considered the hardware platform of choice. A large amount of memory may be necessary, but a ceiling can be placed on the system requirements rather than an arbitrarily large size. The single processor was adopted because of cost considerations and portability. Before designing a massively parallel architecture, a more general, and cheaper, processor must be considered. If the performance is inadequate, then specialized hardware may or may not solve the problem.

By not using specialized hardware, the ICE System can more easily be

embedded into an environment and access traditional algorithms. To truly be capable of integrating with the environment, a real-time expert system must be able to use functions that were written in traditional languages. These routines may be used to communicate with the environment, or analyze the data, to name only two examples the code can provide.

When discussing real-time software, speed always comes to mind. Pure performance may be inadequate to satisfy the environmental constraints, but it is usually necessary. ICE had to be fast. The results examine the overall performance of the system, but more importantly the time necessary for critical responses. If the rate of polling is increased, a controller will be receiving much more data from the environment. The increase in data might overwhelm the expert system and cause the response time to increase. Instead of only designing the system to be fast, ICE also minimizes the amount of time necessary for a response. Another system of equal speed performance, may not be able to respond in the same amount of time. The results chapter compensates the CLIPS implementation for the speed difference, and shows the response time will still be in favor of the ICE System.

3.2 Architecture

The ICE System uses rules and forward chaining like other production rule expert systems. However, there have been modifications to the typical approach, enabling ICE to respond to a real-time environment. Figure 3.28 represents the basic architecture of the software, briefly described below. The following sections explore each of its components.

The **Interface Manager** accepts and processes the incoming data, into the **Environmental Data States**. An item from the environment is encoded into one or more states, or several pieces of data combine into a single state [Pau88]. Most systems match the new data against the rule antecedents and previous facts, a very costly process. Instead, the ICE System determines, *a priori* [FP88] all of the rules a state will influence. Rules are

grouped into tasks, and these tasks are associated with the states. When entering, exiting or remaining in a state, the appropriate tasks are initiated. An initiated task may preempt the current task or wait for the Scheduler to place it into the Agenda.

The Inference Engine analyzes the rules of the highest priority task. Antecedents of each rule are verified and fired. The firing of a rule results in responses to the environment via the Interface Manager, or initiating another task. A rule can also alter an internal state, System States. These states reflect aspects of the reasoning process that are in addition to the environmental data. System States operate in the same fashion as the Environmental Data States. The two are considered separately because of their nature, external and internal aspects of the environment. Encoding a system state also initiates tasks.

3.3 Interface Manager

All communications with the environment are the responsibility of the Interface Manager. Not only must it access data, it must also coordinate the responses to be sent out. The prototypes, described in the results chapter, insure a message is sent and properly received. The interface does not guarantee that the environment can handle a message. One of the test environments can only accept a small subset of the commands making up the long term plan. The Interface Manager will send any number of commands at a time, and the environment will accept the messages, but is unable to record the command to be carried out. The reasoning process, rules and system states, coordinates the number of commands that the environment can carry out. These are given to the interface to be transmitted.

Environmental data is accessed in two ways: by polling or allowing the environment to interrupt its operation. Polling is a common approach to monitor the environment. The test systems use a variable polling rate, a simpler implementation. A specific rate can be chosen that provides the controller with enough information to accurately reflect the environment.

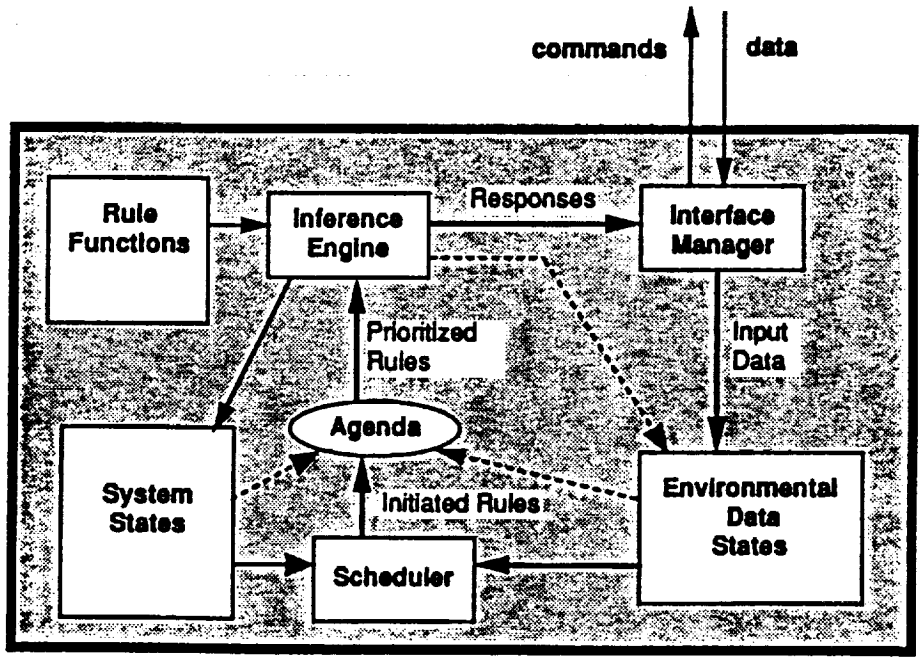


Figure 3.28: Architecture of the ICE System

Events are able to interrupt ICE, typically these are related to catastrophic events. Expert systems that use an approach based on the Rete matching algorithm cannot realistically be interrupted. Interruptability is another benefit of using the ICE architecture.

Polled data is either kept or discarded by using fixed thresholding with hysteresis. By further limiting the environmental data, more time is left for the reasoning process. The thresholds and their range is defined by the data states. Each state has a method, or function, that converts raw data into a state. The conversion process eliminates some of the unnecessary data. The next section provides a detailed description of this process.

3.4 The Facts of the System

The environment may have surprising behaviors, but the data sent to the controller is from specific sensors. Therefore the type of data is specificied, only the rate and values cannot be provided for all cases. Likewise, the rules of the controller, once verified and validated, are also fixed. We can assume that new rules will not be added to the knowledge base while the expert system is in operation. Based on these assumptions, the amount of memory for incoming data can be precisely determined.

Assume the environment only possesses a sensor with a single value, and the controller has three rules. One rule determines the environment is in a critical state and must be stopped. The next confirms the environment is operating normally. The last rule analyzes the previous three values of the sensor to determine the trend of the environment. To specify the amount of memory necessary in the expert system, we consider our single piece of data with respect to the knowledge base. Two rules define the thresholds for a state as normal or critical. The last rule uses the trend of the sensor, it must have the previous three readings and the state of the current trend. Summing up our requirements: one memory element for the state, and four for trend analysis. Therefore, five memory elements satisfy all of the data requirements of the rules. A critical or normal state change initiates the

corresponding rule. The state of the current trend will initiate the last rule.

New environmental information can quickly be converted into its appropriate states. By associating rules with data states, the initiated rules can be quickly determined. A state and initiated rule are coupled together when being sent to be scheduled. In blackboard terminology, the state is the context of the rule. While the rule may not be valid, the ICE system is made aware of possible problems in the environment. Attention may be focused on the new information or not. This will be brought up in later sections.

The encoding methods replace some of the matching computation, allowing ICE to quickly determine initiated rules. In a typical system, only the data value is stored, and the lengthy matching process determines the data state. As new data is added, the memory containing old values has to be reclaimed, garbage collected. Ignoring the need to garbage collect, the design trade-offs are between memory and processing time. The challenge in real-time systems is to respond quickly, hence time is a motivating factor. Therefore, memory should be used to replace as much processing time as possible. While the amount of memory may be large, it is fixed. In delivering a system, specifying the necessary amount of memory is preferred to guessing an amount that may satisfy all situations. On an autonomous space station, the cost of memory is insignificant compared to the possibility of the controller stopping due to a lack of resources.

Each object has an associated method to convert the raw data into the appropriate states. The method uses thresholding to determine the state of the raw data value. Multiple values can be combined into a single state. In the case of remembering previous values of the sensor, a simple circular queue is provided. Methods can do further analysis, even use neural networks to process the data. The purpose is to quickly provide the rules with the environmental data in as usable a form as possible.

The intent is not to run long computations on every new value. The purpose is to convert a raw value into a state that reflects one or more antecedents. A rule, in a real-time controller, typically determines if a value (represented as a fact) is within a certain range (also represented as facts).

Instead of considering the state of the data a number of times in the matching process, it is only determined once. Each rule now has to consider a single state value rather than comparing, matching, several values and ranges. Instead of initially performing a sophisticated function, using resources that might serve a much more critical rule, processing should be as simple as possible to determine the direction to proceed. When time is scheduled for this rule, the more sophisticated algorithm can continue. It may be even more advantageous to break down the processing further and only take steps toward the final solution. The long processing can be verified along the way. This depends upon the situation.

Rules are associated with the various data states. Upon updating a state, the method also initiates rules. Initiation occurs on entering, leaving or remaining within a state. Combinations of the three also exist. Since rules are used to create the states, the states in turn, know which rules they might cause to fire. In one step, their respective tasks are sent to be scheduled. This replaces the lengthy matching process found in other systems. The more trivial data takes a miniscule amount of time to initiate their tasks. The critical tasks, also quickly recognized, can immediately be acted upon by preempting the currently executing rules.

Expert systems use facts other than those representing the environmental data values. These can likewise be represented as **System State Objects.** Created in the same fashion, it has all of the functional characteristics of environmental data objects. The only actual difference between the two is access by the Interface Manager. Both can be accessed and updated during inferencing, but only the Data Table can be updated by the Interface Manager. The distinction follows from the nature of the data.

3.5 Rules

A set of rules are grouped as a task. These rules are related by the states that initiate them. The rules are ordered in a list and do not need a specific priority. Individual rules can belong to any number of tasks. Previously we

have referred to groups of rules, being initiated for example, actually tasks were being discussed.

The number of rules in a task is dependent on the environment and the task in question. The granularity, or size of tasks [MS83][Hob85], has a few trade-offs. Few rules per task closely controls the inference process. A small invalid task completes very quickly, thus allowing the next task to start. Very large invalid tasks generate a long delay before the next response. However, small tasks create much more overhead, than a few large ones. A general heuristic in defining tasks, is to create groupings that seem natural. While this may not produce the optimum configuration, it is much easier to develop and maintain.

The rules are functions in the C programming language which verify all of their antecedents before firing the consequences. Most of the necessary matching is accomplished by the data and system state objects. An antecedent has the option of performing additional analysis on the information. The antecedents and consequences are able to perform user-defined functions. There are no convoluted "hooks" in which to call a given procedure. It is a normal function call.

The consequences generally alter various states in the System State Table. When altering these values, the associated method is used in the same manner as by the interface manager. While the method is processing the update, it will initiate other tasks. A new task can even preempt the currently executing task.

An antecedent may need to find a related fact. The state that initiated the task can be used to directly reference other information. A sensor value indicating a dangerous level tells the controller a device must be shutdown. The state and task are initiated and inferenced upon. While inferencing, the device to halt must be found. In Rete algorithms, the device configuration is compiled into the data flow network, and found by matching against the corresponding bucket. In the ICE System, the sensor and device configurations are known *a priori* and therefore directly associated. Given a sensor, the rule can immediately reference its device.

A second searching and matching problem occurs within a rule, and was examined by the YES system. For a given situation, a rule finds it necessary to obtain the greatest, or smallest, value of a specific type. For example, in assigning a job to a processor, it is generally sent to the one least used. There is no direct relationship between the job and processors. A search must occur to discover the least active computer. In the Rete algorithm, all of the processor activities are compared to each other to discover the minimum. The YES system implements a special searching mechanism to handle the situation. The same mechanism is used here in ICE.

Matching might also look for an element that must satisfy criteria based on the initiating element. A data flow network generally has done much of the matching, and the answer quickly determined. The same solution to the previous problem is used here. Instead of searching for a minimum, the search looks for an element satisfying the given constraints. To speed up the process, one or more most likely choices can be associated with the state. These choices are checked first. Failing to find a solution, the rest of the possibilities are examined.

When inferencing on rules in any type of real-time expert system, truth maintenance becomes an issue. Because the environment is nonmonotonic, the received data may change at any moment. The data validity is also decreasing as time goes on. If not processed in a specific amount of time, the data may be invalid. There are two cases where this will cause problems. The first considers the individual rules. A rule might be correctly activated. Before it is fired, one of the antecedents changes and invalidates the rule. The rule may fire before the system deactivates it. The second problem is much more intricate. A task can invoke other tasks to be inferenced upon. A dependency path of tasks is made. If one of the previous tasks becomes invalid, then the current task must be deactivated.

A rule that is ready to fire, but whose antecedents are not true, is particularly a problem with multiple processors. The rule may be on a given processor, while another computer discovers the error. The Rete algorithm on a single processor solves the problem by having a mode to handle all

matching, activations and deactivations. No rule will fire while the matching mode is executing. The ICE System has the rules verify their antecedents immediately before being executed. A fact might correctly initiate a rule, but become invalid before the rule is inferenced upon. When the rule is ready to fire, the fact is checked. If true, the rule fires, otherwise the rule is abandoned.

Maintaining the integrity of a dependency path of tasks is much harder. Dynamic censors were previously discussed, and can be used within the ICE architecture. The censors are presented as system states and can be referenced, and de-referenced, at any time. This is currently used on a limited scale. A task initiates other tasks by using the system state objects. When these new tasks are inferenced, the state is verified by each rule. By using the state as an antecedent of rules further down the dependency path, high level truth maintenance can be maintained.

3.6 Scheduling the Agenda

The scheduling is broken down into two processes, as seen in figure 3.29. The Initial Scheduler accepts the initiated task and state, and determines its priority. A new task will preempt the execution of a lower priority task. Otherwise, it is placed in the temporary queue. The highest priority task is allowed to execute with the minimum amount of interference from the other tasks. The current task is only slowed down by receiving new data and determining its priority. If a new task is more critical, then it becomes the current focus of attention of the inference engine. Fairness, as mentioned earlier, is not appropriate for real-time systems. When in outer space, if the lights have been off for a long time and the life-support only recently failed, the lights are still low priority. An astronaut can live in the dark.

It is necessary to use the importance of a state along with the priority of the task, determining how vital their combination is. A number of combinations are possible, but the current prototypes sum the two priorities. These systems were first developed in CLIPS, which only gives a priority to

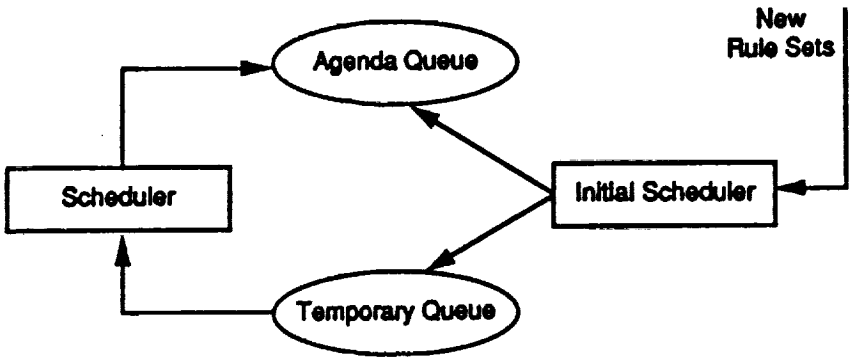


Figure 3.29: ICE System Agenda

individual rules. In transfering the system into the ICE architecture, certain data states were given a priority to fine tune the scheduling mechanism. The simple summation is the fastest method to combine the priorities and provided a satisfactory solution. Other techniques are possible, but the intent is to have a very quick determination of the task/state priority.

In the YES/OPS architecture, both tasks and rules were given a priority. ICE uses the task priority to determine the *world*, or rules to use. All of these rules are scheduled together. The ICE System uses a finer granularity when defining the tasks. The rules, in a task, can be ordered *a priori*. Two rules with the same importance can be placed in an appropriate order, otherwise the order does not matter. Scheduling time is therefore saved in determining proper responses to the environment.

The Initial Scheduler places a priority on a task/state pair, and the current task priority is checked. A critical event can thus preempt the controller and alter its focus of attention. There is no complex scheduling, the two values are simply, and quickly, compared. Less important tasks are also checked, and passed onto the temporary queue. This mechanism provides a rapid procedure to allow the controller to change its focus at any moment.

The temporary queue is a FIFO, first in first out, queue. By using a

FIFO list for the temporary queue, the age of the tasks is implicitly stated. When the scheduler executes, the first element of the temporary queue is placed into the agenda queue, a prioritized list. The highest priority task is the first to be considered by the Interface Engine. Tasks with the same priority place the younger first.

The younger tasks are considered before older tasks with the same priority. This choice is dependent on the environment and the tasks used to control it. Due to the validity decay of the data, older states are less likely of being true than younger states. CLIPS also uses this method of conflict scheduling and was another reason to adapt the approach for similar results.

Scheduling can be done in a number of ways. A typical approach schedules after a specific number of task executions, it may be after each task. The current prototypes inference all of the tasks in the agenda before rescheduling. The scheduler can be started based on a command from a task, or the priority of the current task. If task response was not critical, the scheduler has time to merge the two queues. Scheduling can take place immediately, regardless of the priority, but that drastically conflicts with the philosophy of the ICE System to respond to the most critical event.

Another method considered in the design of the ICE System invokes scheduling based on the priorities of the waiting tasks. There can be a slot to hold the value of the highest priority task in the temporary queue. Tasks are added in the same fashion, but also compare their priority with the highest one in the queue. The value is updated appropriately. Before the Inference Engine begins processing the next task, it compares the priority to the highest priority in the temporary queue. If a higher priority task is waiting, rescheduling takes place. The purpose of this method allows the higher priority tasks to execute without waiting for less important tasks to be scheduled. While it is currently not implemented, it is a useful feature for future systems.

Before leaving the discussion on the agenda and scheduling, an important point in memory management needs to be addressed. While the ICE System has removed the need for working memory, it still must have a pool of nodes

to use in scheduling the tasks. Each node is made up of pointers to the state and initiated task, along with the combined prioriy. These are needed in any real-time software, for task scheduling, be they rules or procedural code. The pool of elements is a fixed size, and therefore a contingency plan must exist, in case the pool emptied.

The issue becomes which task/state nodes to remove, forget. The least significant nodes are the most likely candidates, but are spread across the two queues. Since the agenda is already prioritized and its elements are more likely to be older than those in the temporary queue, the last elements would be used for new higher priority task/data elements. If the system is to the point of losing tasks in the agenda, then an effort must be made to schedule the elements in the temporary queue. A possibility is to schedule tasks more important than the last element in the agenda. The other nodes are freed for new initiations.

3.7 Inferencing

The Inference Engine considers the first, highest priority, task in the Agenda. The rules in the task are sequentially executed. A rule function verifies its antecedents and is responsible for executing the consequences. The verification process can check states, combine states or perform other types of processing (*i.e.* searching). The execution of the consequences generates responses and alters states within the controller. The commands are sent to the environment via the Interface Manager. States are updated with their respective methods and might initiate other tasks to execute, just as if the data came from the environment.

The approach allows the system to be interrupted at any time. Matching algorithms used in other systems do not perform well in an interruptable environment. While these systems may be interrupted outside of the matching process, it is not worth the effort. The matching process consumes so much of the execution time, the left over time only allows a very small window for interrupts to appear. In processing ICE rules, the matching is done locally.

If the rule was interrupted, it can restart the matching of its antecedents. The time lost is fairly insignificant. Interrupting in the prototypes is disabled during rule execution and only allowed between the rules. This enables the system to finish its current "thought."

The latency period is the amount of time between the environment sending an interrupt signal and being accepted by the controller. The worst case is the environment generating a signal the moment after inferencing began on a rule. The average amount of time to process a rule is the average worst case latency time. The absolute worst case considers the rule requiring the longest time. As is the norm, rules are fairly quick and therefore the latency period will be acceptable. In cases where a rule consumes too much time, interrupting may be left enabled.

Common approaches either use interrupting between rules or not at all. In cases where an interrupt signifies a highly critical event, the ICE architecture is capable of always allowing interrupts. The interruptability scheme depends on the environment. With the space station Freedom, data is received as reports from a sophisticated power distribution system. If interrupts are possible, they would only indicate life or station threatening situations. These must be responded to immediately and therefore their interrupts are always allowed.

Only a minimal amount of processing time is taken away from the most critical reasoning. If the resulting response was necessary, the system is performing perfectly. Unfortunately, it may not be the case. An alarm may incorrectly point to a catastrophic event, and the controller determines the error while a valid response is waiting. This point is unfortunate, but necessary.

The greatest strength and weakest link of the ICE System is in initiating tasks. The approach provides the minimum response time for the critical events in the environment. The least important event is not guaranteed a fast response. The weak aspect occurs because invalid tasks are being initiated and other tasks becoming invalid. In both these cases the Inference Engine uses the precious processor time to iterate through the rules and determine

that they are invalid. This presents a case for using small, fast, rules and fine granularity when creating the tasks. The time loss is the greatest for the least important tasks. In designing the system, this was seen as an unfortunate side effect of quickly generating vital responses. However, the critical response time is considered to be more important than minimizing all of the response times, including the trivial ones. A point to notice in the results is that the system performance is high enough to respond much faster to all of the events, as compared to the more traditional production rule system done in CLIPS.

3.8 Knowledge Engineering

There are several stages in developing an expert system in the ICE architecture. The first stage defines and validates all of the rules in the system. These can be created using a commercial expert system shell, which usually provide many developement features.

The second phase builds the rules, the environmental and system states of the ICE System. For any given rule, the user specifies which antecedents can initiate the rule, or the *hot* antecedents. When a hot antecedent enters (or remains, or leaves) a state, the rule is initiated (sent to be scheduled). The antecedents that are not hot will still be checked during inferencing, but cannot initiate this rule. After deciding on at least one hot antecedent for each rule, the environmental data and system states can be determined by the rules. The states will reflect those used for comparisons in the rules. A function must be defined to convert the raw data (a boolean, single value, or many values) into it corresponding state. To illustrate this point, consider a rule has an several antecedents stating a sensor is in its warning state, and another antecedent is concerned with the past history of the machine. Assume further, that the knowledge engineer has elected to only use the sensor fact to initiate the rule. By knowing the range defining the sensor in the warning state, the state tables can be built. The rules can also be defined in the C programming language, with respect to the states in the

various tables.

The more antecedents selected to invoke the rule, the greater the chances of the rule being initiated multiple times. This also leads to the possibility of using all of the scheduling memory pool. However, by not using enough of the antecedents, the rule may not be initiated as often as the user would like. These points are very specific to the problem, and the best solution is not always easy to determine.

The last phase specifies the sets of rules, which are the tasks. To further increase system performance, rules are grouped into sets or tasks. When the hot antecedent wishes to fire a particular rule, it actually fires the task, (a small set of rules). These are usually broken down by the initiating states. If a sensor state intiates five rules, then these five rules make up a task. This does not have to be the case, for a faster response the knowledge engineer may elect to break the rules down into two tasks, those with higher and lower priority (priorities will be discussed shortly). A rule may be included in a task because it is always fired after the current set of rules. In this manner, the new rule does not have to be initiated to fire. However, a user must be cautioned, a system state should be used to insure that the rule should fire, the initiated rules may not be true.

In determining the tasks, the rules within the task can be prioritized. The inference engine sequentially tests and fires each rule in the task. The task is therefore an ordered list of rules. The ordering is of the discretion of the user.

The tasks and data states must each be given a priority. The higher the priority reflects the more critical the task or data is. There are several methods to combine these priorities, currently the two are simply added. Consider the following illustration, there is a breaker going to a life support module in the space station, and another handling the lights within another module. A rule states that if a breaker will potentially fail, the other redundant breaker must be used to insure continuous operation. This rule is given a priority, as in many expert system shells. If both breakers fail, the rule is initiated

twice, but the breaker going to the life support system is much more important than the other, controlling the lights. By giving each breaker state the appropriate priority, this can be reflected in prioritizing the task-state pairs.

By using two priorities, the knowledge engineer has a great deal of flexibility and power in ordering the tasks and states. With these features also comes more complexity. The knowledge engineer must be very careful of unexpected effects in the system. In the test prototypes, most data states were given a zero priority, so that only the task priorities were really considered. For tasks that handled both high and low critical tasks, the critical states were given a priority. Even by using this simple scenario in developing a large system, problems may still arise unexpectedly.

A last point for the knowledge engineer to determine is the size of the scheduling memory pool, the problem also occurs in many other types of software systems. The pool must be large enough to handle any situation. Some systems will crash if the pool is emptied, which may be fine for their situation. A continuous controller cannot be halted because of a shortage of memory. Here the pool size is determined to be more that is expected. A contingency plan must be devised for a situation where the pool is emptied, so that the controller can still operate. The plan depends upon the implementation, but a few methods were mentioned earlier while discussing the scheduler and the agenda.

As in any system, care must be taken to prevent unexpected events. For real-time controllers, contingency plans must be determined for every possible flaw in the design, the scheduling pool problem for example.

Chapter 4

Results

The ICE System is compared to a typical expert system based on the Rete algorithm, we have used CLIPS. There are faster systems on the market, but it is readily available and able to provide an adequate medium of comparison after compensating for the overall performance differences. Two sets of tests are used to analyze ICE. A smaller knowledge base, of 17 rules, tests the general concepts of the thesis. The performance difference between the two systems is found along with an analysis of how time is used to discover the proper responses. The second system, of 80 rules, provides a more sophisticated environment to further scrutinize the response times of the architectures.

The following sections describe the test bed environment and an analysis of the results. The appendix contains specific information on the tests.

Both have been executed on a Digital Equipment Corporation VAX computer running the VMS operating system. Time is measured in ten millisecond units of processor time, but presented as milliseconds. Because of the basic time unit, time measurements less than ten milliseconds will be more sensitve to noise.

4.1 Test System 1: Machine Monitoring

A machine monitoring problem [GR89] determines the difference in speed and the response time, due to the matching algorithms. An abstract view of the environment is shown in the figure 4.30. There are four devices, each with one or two sensors, for a total of six. Every sensor has unique ranges indicating the device state:

1. critically high

2. high warning level

3. normal condition

4. low warning level

5. critically low.

The two critical states indicate the machine will soon fail and must be immediately shutdown, to prevent any further damage. A normal condition defines the safe, expected, operation of the machine. Warning states can be entered for short periods of time, with no effect on the system. An operator must be notified of any machine leaving the normal operating conditions. The sensor will remain in a warning state for a period of time while the operation of the machine is degrading. If a sensor remains in a warning state for a specific amount of time, the machine is shutdown to correct the problem before any damage is done.

The initial startup time (*e.g.* compiling the rules into the Rete network) is not considered when comparing the two systems. One hundred of the one hundred and sixty cycles are considered, and averaged over one thousand runs of the environment. The other cycles compensate for startup time and validate the operation of the system. A cycle starts by retrieving predefined data from each of the sensors. The data is analyzed and acted upon accordingly. After all of the responses are issued, the cycle repeats. Both systems operate in the same manner, *i.e.* the ICE System is not allowing any interrupts. This presents an accurate comparison of the overall speed performance of the two systems.

The first test analyzes the difference in the cycle speeds between the ICE System and the typical approach. In the first case, the environment is operating normally without any unexpected situations. All sensors remain in the warning state during the one hundred test cycles of the second case. Table 4.2 presents the average cycle times. A dramatic difference can be seen. At this point, the increase is primarily attributed to writing the rules in the C programming language. The average of the speed differences is used to compensate the second set of test results.

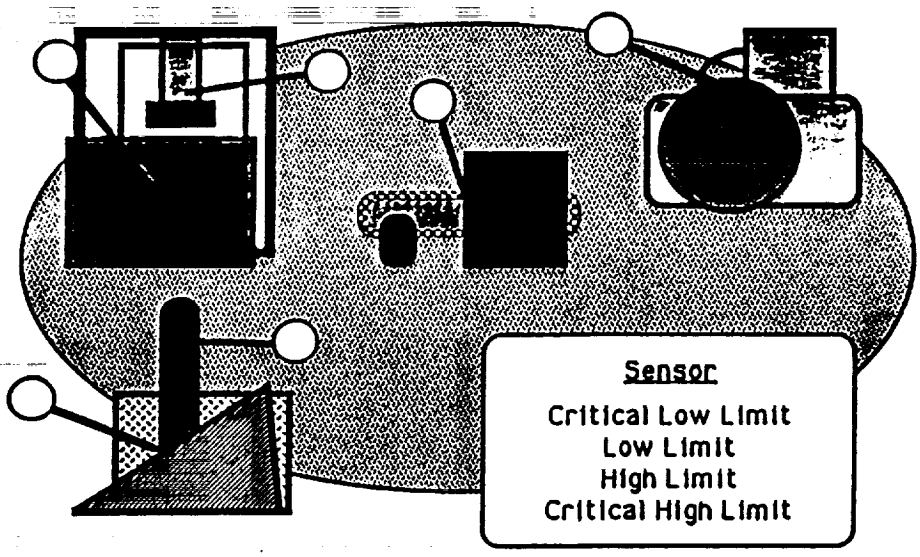


Figure 4.30: Machine Monitoring Problem

Test Case	Typical	ICE
Normal Operation	60	3
Warning Operation	78	8

Table 4.2: Benchmark: Average Cycle Time

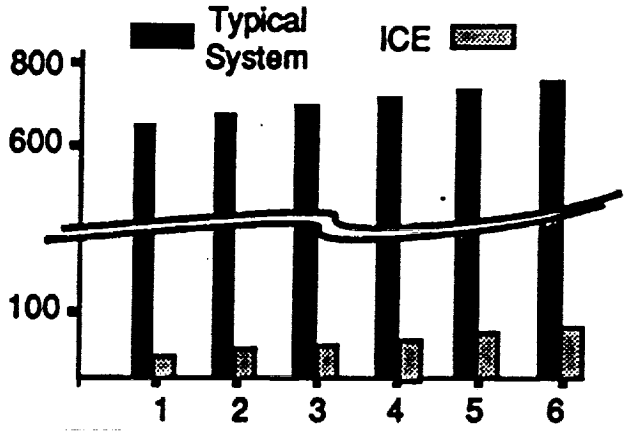


Figure 4.31: Benchmark: Response Times for the 6 Warning Responses

The typical system shows a relatively small increase in the cycle time when handling all of the warning states. The typical approach performs all of the matching for both cases, and therefore the time period is similar. ICE, on the other hand, quickly recognizes a sensor has remained in the normal state and does not initiate any tasks. The raw data is only placed into its proper state, normal. The warning states initiate tasks for inferencing. Scheduling, inferencing and responding constitute the five millisecond increase.

During the second test, all of the samples remain in the warning state. The validity periods of each of the sensors was greatly increased to allow the devices to remain operating. Warning messages are issued for each of the sensors. Again a dramatic difference is seen in the speed, show in table 4.31. After examining the first table, the speed increase is expected.

The more interesting point is the difference in performance due to the matching algorithm. Because Rete initially matches all of the available data, it requires much more time at the begining of the cycle. The first response is then delayed for a long period of time. Figure 4.32 plots the response times as the percentage of the total cycle time, showing the processing time used to accomplish each of the responses. The implemented typical approach uses

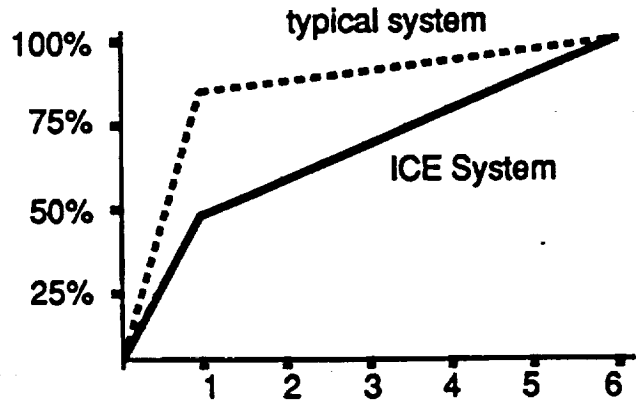


Figure 4.32: Benchmark: Percentage of Time Used for Each Response

85 percent of its time in responding to the first event. The exciting aspect of the tests is the response times of the ICE System. Less than 45 percent of the processing time is used to respond to the first event.

Included in the initial response time, is the time used to simulate the devices and sensors. The simulation involves accessing data report tables and presenting the sensor data for all of the currently operating devices. The other five responses already have all of the data available and processed. These responses only require a confirmation of their actions, the same amount of *work*, and therefore show a constant increase in response time.

The typical system performs very quickly for the later responses, because all of the matching has already been accomplished. The graphs represent a major aspect of the ICE architecture. The system spreads the matching process over the whole cycle, therefore more time is needed for the later responses. The figure also shows ICE not only outperforms the typical approach in the first response, but all of the other responses as well.

Other expert system shells based on the Rete algorithm may be much faster than the CLIPS software, and may not show as dramatic a difference in the pure speed aspect of ICE. Because these systems are based on Rete,

they exhibit the same effect when comparing the percentage of time used to accomplish the various tasks.

4.2 Test System 2: Monkeys, Bananas and Zombies

The next set of tests separate the environment from the controller. A simulation process was created to handle the generate data reports and process the commands received from the controller process. The two communicate through a data channel, termed a mailbox in the VMS operating system. The environment is described in terms of the model presented in the concepts chapter. The analysis of the results is then presented, specific test information can be found in the appendix.

4.2.1 Monkey, Bananas and Zombies Description

The monkey and bananas problem is used to benchmark many expert systems. The monkey represents an active controllable agent and the boxes, to be stacked, reflect the passive agents. The active uncontrollable agents were not present in the scenario. "Zombies" were added to fulfill this feature. A Zombie will appear in the environment, and wander about thr room for a period of time. The monkey must stay away from these monsters. The procedure is given with the description of the test.

A room contains many boxes, as demonstrated in figure 4.33. A number of these must be stacked in a tower, under the bananas, for the monkey to reach the bananas on the ceiling. The controller determines a long-term plan to stack the objects and manages the commands sent to accomplish the mission. There are four tests based on variations to this system described in the following paragraphs.

The first is the standard situation for all of the tests. A dozen, or so, boxes are stacked to reach the bananas, without anything going wrong. The monkey is monitored to determine the completed commands, then an appropriate

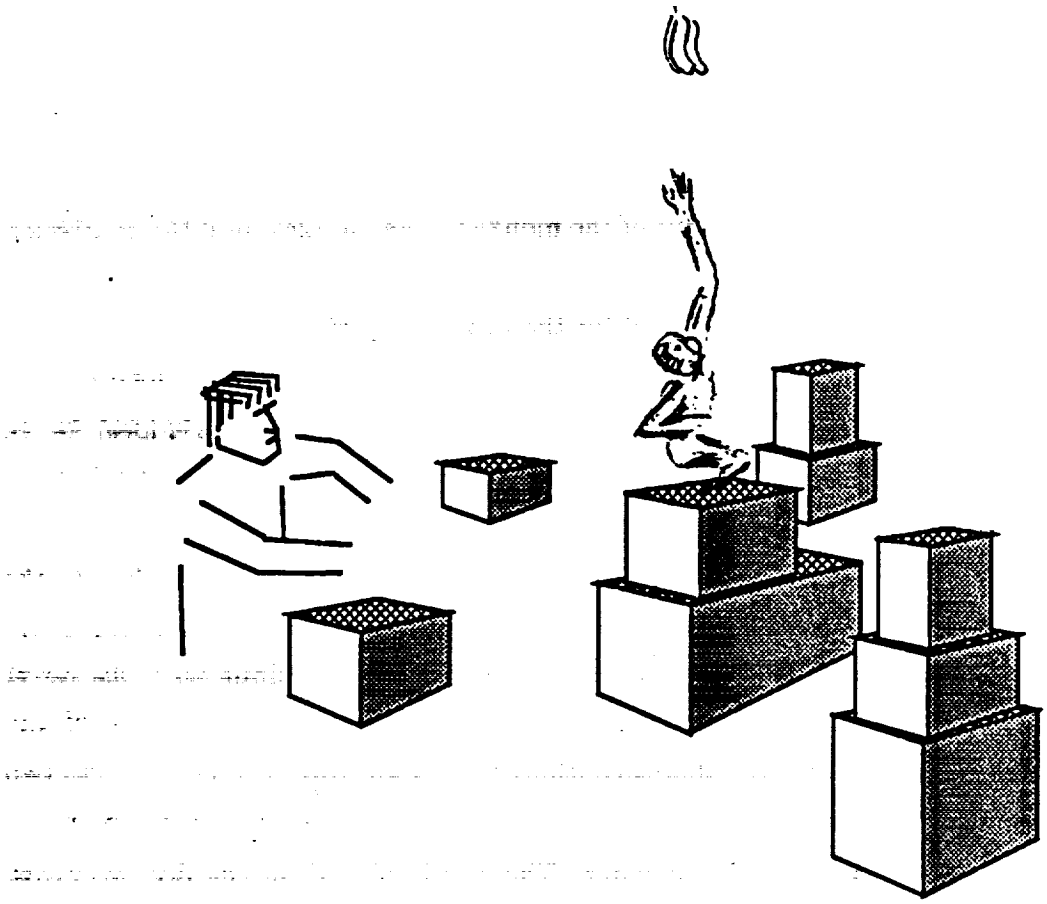


Figure 4.33: Monkey, Bananas and Zombie Problem

number of new commands can be sent.

The next test represents a flaw in the active controllable agent. The monkey cannot lift the heavy boxes and has no warning alarms to indicate the failure. The controller must monitor the tower construction to determine the error. Once a box is found to be too heavy, the current plan must be augmented to accomodate for the missing object.

The third test simulates passive agent failures. Some of the boxes are unable to support the weight of the monkey. The monkey, and the controller, are unable to determine this characteristic until after the failure has occured. The long-term plan is updated for the broken objects.

Active uncontrollable agents, zombies, are considered in the last type of test. To allow the two systems to be compared, the zombies appear based on objects the monkey picks up. Certain boxes, unknown to the controller, will trigger the appearance of the zombie. The monster at one end of the room has no effect on the actions of the monkey. When it is too close to the monkey, the controller must initiate a plan to move the monkey far away. Once the monkey has been frightened, it does not continue with the tower construction until after the zombie leaves. After the disappearence of the monster, the long-term plan is continued at an appropriate place. If the two are within a warning area, the monkey has time to place the box on the floor, and move away from the monster. The box is placed on the floor so boxes lower in the stack can still be accessed without trigger the appearance of the zombie. As the two active agents become very, critically, close to each-other, the monkey drops the box as soon as possible and runs. If the monkey is currently on the top of a stack, the box is left on the top. This results in all of the boxes in the stack not being accessible. When the abandoned box is not placed on the tower, the controller must update the plan to compensate for the lower tower height. After the disappearence of the monster, the long-term plan is continued.

Boxes are dropped in the previous test situation so the monkey can run from the zombie. The monkey can walk while carrying an object, but can run at faster pace, when its hands are free.

4.2.2 Test Results

Each of the three agents generate important and insignificant events. The monkey, an active controllable agent, is monitored to determine the time and number of commands to be sent. A change in the carrying state of the monkey reflects either a box has been picked up or put down. The events can be mapped directly to the corresponding command in the currently sent subset of the long term plan. When the controller determines the monkey has nothing to do, standing still, it assumes all of the sent commands are completed. The maximum number of new commands the environment can accept is determined and sent one at a time.

The passive agents, boxes, can fail and be used to diagnose a failure of the active controllable agent. Both of these events represent a problem with the current plan of the tower construction, and an update must be added. There are no explicit responses sent to the environment. This case only determines recognizing the event. Because the ICE system is interruptable, replanning takes place over several cycles and is often preempted by more critical events.

The zombies, active uncontrollable agents, present another complication to the scenario. If a zombie is currently in the room and close to the monkey, the controller must first send the STOP command to the monkey, to free up the command space. Planning a solution to the problem begins and the responding commands are sent. In this test, planning takes place after the first response.

The responses to events in the test cases are broken down into four segments: receiving the data, recognizing the events, the initial response to the event, and subsequent responses. The controller does not analyze all of the continuous data, instead it considers samples taken from the signal. The average rate the software accepts data is the average worst case delay in receiving the data of the most critical event. The time to recognize events from the data report, and issue the proper responses are dependent on the event and controller. To determine these times, the controller records the actions with a time stamp. The timing of the subsequent responses are also recorded with a time stamp. Many of the events require multiple commands

Architecture	Report Acceptance
Typical	2,502.0
ICE	5.0
Compensated ICE	75.0

Table 4.3: Average time between accepting reports

for a complete response, and therefore it is important to know the expected amount of time delay between the commands. After recognizing an event, the reasoning process determines a set of commands for the response. Time for the first command to be sent includes the planning time for the following commands. Hence the first response is considered separately from the other commands making up the response.

Delays in Accepting Data Reports

Both controllers require at least a minimal amount of time to process a report. Table 4.3 shows the average time between accepting reports for all the tests. Even when compensating for the speed performance of ICE, a dramatic difference is still seen. The reason for this affect is interruptability.

The ICE system allows the environment to interrupt with new data. Interrupts are enabled after the tasks in the agenda are inferenced, only a few tasks are typically in the agenda at a time. The rational allows the inference engine to complete "its current thought" before new data enters the system. By decreasing the time to accept a report, the rest of the response times will be increased. The increase is due to more data entering the system and less available time for scheduling and inferencing. However the controller must accept as much data as possible to ensure a minimum amount of time to respond to the most critical event.

Recognizing Events and the Initial Response

To respond to the most critical event, the controller must first recognize the events represented in the data report, table 4.4. The further processing of

Event	Typical	ICE	
		Actual	Compensated
Object Dropped	1,373.5	5.1	76.5
Object Picked Up	961.8	5.1	76.5
Monkey Still	1,073.9	5.2	78.0
Object heavy	1,599.9	8.5	127.5
Object Broke	1,575.1	5.7	85.5
Zombie Too Close	905.3	2.6	39.0
Zombie Gone	1,691.5	9.2	138.5

Table 4.4: Times to Recognize the Events

Event	Typical	ICE	
		Actual	Compensated
Object Dropped	51.4	7.3	109.5
Object Picked Up	52.7	7.3	109.5
Monkey Still	62.1	6.3	94.5
Zombie Gone	285.0	5.2	78.0

Table 4.5: Times to Respond to the Events

the events can then be ordered and the proper responses determined, table 4.5. The matching algorithm plays a significant role in the difference in these times. Since the system size has increased, the Rete algorithm has more data to match with more rules, and therefore the time increases. The ICE system event recognition time increases due only to the data, it does not match the data against all of the rules.

After the event is recognized, the typical system issues the response faster than the compensated ICE time. Because most of the matching in the typical approach has taken place, during the recognition phase, it can respond quickly. The ICE system however, verifies the antecedents of the rules making up this phase. The passive objects do not issue a response to the environment, replanning is done. The zombie becoming dangerously close to the monkey represents a slightly different problem and is discussed separately.

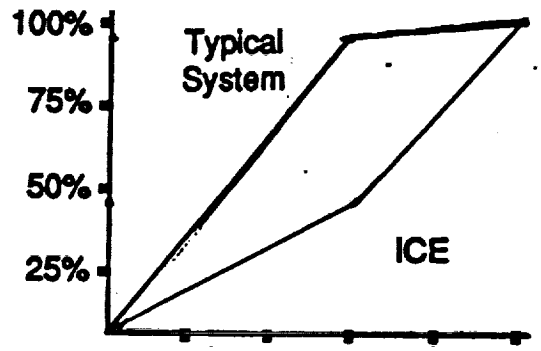


Figure 4.34: Percentage response time of active controllable agents

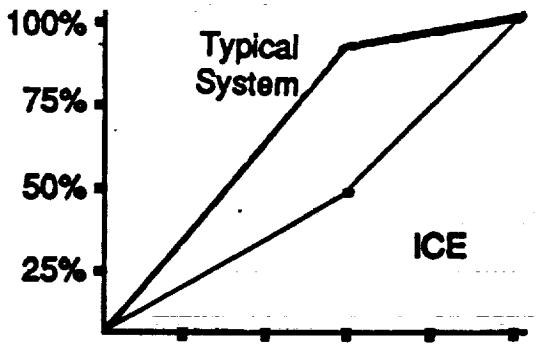


Figure 4.35: Percentage response time of active uncontrollable agents

Recognizing and responding to events are considered together, because matching occurs in the different phases. Figure 4.34 and 4.35 demonstrate the combined effect as a percentage of time from receiving the data to issuing the response. As demonstrated, the typical system uses practically all of its time recognizing the event, while ICE uses less than half. After recognizing the most significant event, the ICE system strives to respond to it and only matches the rules in the path toward a response. For this reason, the second phase of the graph is not more significant. The rest of the matching, for the less significant events, is done after the response. This matching increases the time to receive a data report. The degradation is lessened by allowing the environment to interrupt the inferencing process.

This is the major problem with using Rete types of algorithms in a dynamic environment. As the complexity of the environment increases, the amount of data and rules increase. The response time of the typical approach is increased by the new data and their combination with all of the rules. The ICE architecture does not possess this drawback, it need only be concerned with the additional data. Only rules added to the path to a given event will influence the particular response.

Recognizing and Responding to the Zombies

As mentioned, the monkey will run if the zombie gets too close. The controller recognizes if the zombie is too close to the monkey. When it occurs, the first response is to stop the monkey so the situation will not degrade and to free command queue of the environment for the new commands. The controller then determines an emergency plan of action for the monkey to escape from the monster. The second response time includes this planning. The average response times for the phases are presented in table 4.6, and figure 4.36 represents the percentage of time used for each phase until the second response. The same pattern occurs again, a majority of the typical approach uses most of its time for the initial matching while the ICE time is used to plan the escape.

Event	Typical	ICE	
		Actual	Compensated
Recognized Zombie	905.3	2.6	39.0
STOP Response	3.5	0.6	9.0
Escape Response	196.0	10.7	160.5
Response Time	1,104.8	13.9	208.5

Table 4.6: Response time of controller to the zombie

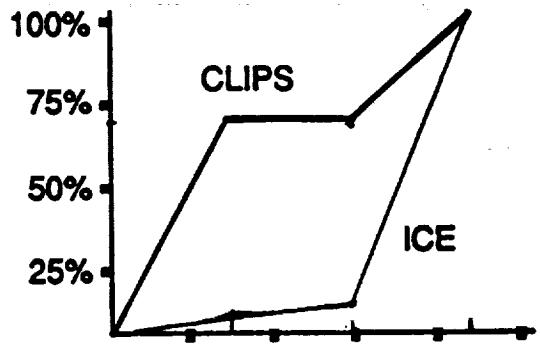


Figure 4.36: Percentage response time of the zombie

Event	Typical	ICE	
		Actual	Compensated
Intermediate Command	46.5	1.3	19.5
Initial Planning	1,638.5	9.1	136.7

Table 4.7: Time to send next command

Other Results

Two other time periods have as yet not been presented. The first is the time between sending commands. The time, for the controller to determine a set of commands can be sent to the environment, is discovered by the time of the first command sent. Table 4.7 presents the average time to send the rest of the commands. Also found in the table are the initial planning time of the two systems. The planning time period begins after the expert system receives the first report from the environmental simulator. The period ends with the first command being sent to the monkey.

The effect of compiling the rules, into C functions impacts on the these figures. When planning and finding the next command to send, the typical approach performs its general search over the data for a specific fact. ICE rules can perform a more specific search, this approach was found earlier in YES/MVS and OPS-83. These two systems found significant improvement by compiling the consequences and using special searching strategies.

Consider the first entry in the table, sending the next command. The typical system knows the index of the next command, but must search every command for that index. Commands will be searched after the correct command is found, there is no mechanism to stop this search. The ICE system uses the index to directly retrieve the next command.

The second set of figures represents the initial planning phase. Here again the special searching mechanism is used to find boxes for the tower. The typical system searches all of the boxes looking for an appropriate one. This activates the rule to find a suitable box with all of the possible boxes, after firing with the first instance, all of the other instantiations are retracted.

Much time is spent on performing all of the initial matching, and then more time is needed to retract the fact that looks for a box from the data network. By retracting this fact, all of the rule instantiations are also retracted. The extra overhead consumes much more time than was saved by recording the matches to prove a box is valid. The ICE system search considers each box until it finds an appropriate one. Once found, the system continues with the consequences of the rule and continues. No overhead is present, only the search. If searching is very complex, special mechanisms can be used.

Chapter 5

Conclusions and Future Directions

5.1 Concluding Remarks

The prototypes show the advantages of ICE over CLIPS in controlling the dynamic test environments. The comparison can be generalized to include expert systems based on types of the Rete algorithm. The fundamental problem of systems, like CLIPS, for real-time control is the matching process handles all of the data from the environment before reasoning on the most important response. The ICE system shows it is able to quickly recognize the potential events from the data and can therefore direct the reasoning process much faster.

The data processing method of fixed thresholds with hysteresis is a sound approach in converting the raw data into its associated states. By using the antecedents of the rules to determine these thresholds, the data states not used by the controller will be immediately discarded. The point is readily demonstrated by the first test of the sensors and devices prototype. The normal sensor conditions were not needed in any of the rules and was discarded, providing a much faster cycle time. The same point can be made on the Rete approaches, but it would not be as evident to a developer.

The prototypes proved the concept of not using working memory in a data-driven production rule expert system. The delivered system is able to specify the exact amount of memory necessary for the expert system. This may be a large number, but is the absolute ceiling for continuous operation.

The ICE system architecture shows great potential in delivering an embedded real-time expert system. Those systems using a form of the Rete algorithm must overcome the initial matching problem, changing data values (so as not to continually retract and assert facts), and dynamic memory

management. Some of these other systems have developed features used in ICE, but it was deemed worth abandoning Rete for a new architecture. The Rete algorithm is a natural approach for off-line systems. Stretching its limits to meet the demands of a dynamic environment appears to be building a real-time system on a weak foundation.

5.2 Future Directions

The ICE system represents a first step in providing an architecture for real-time expert systems. There are many directions that can be taken, internal mechanisms for the delivery system and those to aid in the development of a system.

The first internal mechanism presented uses dynamic censors to aid in truth maintenance. A developer already can use the system states and his/her rules to accomplish the same result, but a more automatic approach is highly desirable.

An external clock for scheduling tasks needed after a specific time. The clock is used to solve a number of problems in real-time software, initiating a polling cycle for instance. Real-time clocks are often interfaced to the hardware platform to achieve the desired effect. Another approach takes advantage of the defined tasks to be used as a clock. Knowing the time for a task to complete and its start time, the time when the task finishes is easily determined. This knowledge and by modifying the scheduling mechanism, a task can be scheduled to run at a particular time, without interrupting the software. The method is termed quantum scheduling [Gut88].

The *a priori* information on the rules and the environment is mentioned several times when referring to building a controller with the ICE architecture. Obviously a compiler to generate an ICE system from another system designed for the development process [HS89a]. The compiler must allow the developer influence over the sizes of the tasks, priorities and other features more specific to the problem.

The development can use an existing commercial expert syste shell, or one

designed specifically for developing real-time expert systems. In either case, a modelling utility to simulate the environment allows the developer an easy mechanism for testing various scenarios. This point may be obvious, but the modelling facility should not be converted over to the delivery system [YM83]. If the controller must model the environment, a catastrophe may take its toll before the model determines its existence.

In discussions on future work of real-time systems, a few points usually surface. The first is defining a precise mechanism for implementing real-time software rather than ad-hoc attempts to solve an instance of an environmental problem. However the same discussions are echoed in building expert systems. A strong issue in real-time software is the ability to guarantee a satisfactory response withing a specified time window. Perhaps automatic program verification is the first step in solving this last issue.

Appendix A

Sensors and Devices Test

The tests on monitoring sensors and devices [GR89] is comprised of four devices with one or more sensors. Table A.8 and A.9 contain all of the sensors and their associated device and thresholds. Below the "Low Critical" level, the device is said to be in a critical state and must be immediately shutdown. The warning state is between the two low thresholds, meaning a potential problem may exist. If the device is in this state for a short time, it is not considered a problem. After a fixed period of time, shown in table A.10, the device is considered unstable and is shutdown. The normal operation of a device is between the two warning thresholds. No action needs to be taken by the controller. The "High Warning" and "High Critical" states operate in the same manner as their corresponding low states.

Data from both the tests is found at the end of this section. Cycles 3 to 103 are used to compare the two systems in the results chapter.

Sensor	Device	Low		High	
		Critical	Warning	Critical	Warning
1	1	60	70	80	130
2	1	20	40	85	180
3	2	60	70	85	130
4	3	60	70	85	130
5	4	65	70	85	125
6	4	110	115	85	130

Table A.8: Ranges for Testing Normal Operation

Sensor	Device	Low		High	
		Critical	Warning	Critical	Warning
1	1	60	70	120	130
2	1	20	40	160	180
3	2	60	70	120	130
4	3	60	70	120	130
5	4	65	70	120	125
6	4	110	115	125	130

Table A.9: Ranges for Testing Warning Operation

Sensor	Device	Warning Period	
		Normal	Warning
1	1	3	120
2	1	5	120
3	2	4	120
4	3	4	120
5	4	4	120
6	4	2	120

Table A.10: Ranges for Warning Period

	SENSOR NUMBER					
Cycle	1	2	3	4	5	6
1 :	100	100	100	100	100	120
2 :	100	100	100	100	100	120
3 :	100	100	100	100	100	120
4 :	100	100	100	100	100	120
5 :	100	100	100	100	100	120
6 :	100	100	100	100	100	120
7 :	100	100	100	100	100	120
8 :	100	100	100	100	100	120
9 :	100	100	100	100	100	120
10 :	100	100	100	100	100	120
11 :	100	100	100	100	100	120
12 :	100	100	100	100	100	120
13 :	100	100	100	100	100	120
14 :	100	100	100	100	100	120
15 :	100	100	100	100	100	120
16 :	100	100	100	100	100	120
17 :	100	100	100	100	100	120
18 :	100	100	100	100	100	120
19 :	100	100	100	100	100	120
20 :	100	100	100	100	100	120
21 :	100	100	100	100	100	120
22 :	100	100	100	100	100	120
23 :	100	100	100	100	100	120
24 :	100	100	100	100	100	120
25 :	100	100	100	100	100	120
26 :	100	100	100	100	100	120

27 :	100	100	100	100	100	120
28 :	100	100	100	100	100	120
29 :	100	100	100	100	100	120
30 :	100	100	100	100	100	120
31 :	100	100	100	100	100	120
32 :	100	100	100	100	100	120
33 :	100	100	100	100	100	120
34 :	100	100	100	100	100	120
35 :	100	100	100	100	100	120
36 :	100	100	100	100	100	120
37 :	100	100	100	100	100	120
38 :	100	100	100	100	100	120
39 :	100	100	100	100	100	120
40 :	100	100	100	100	100	120
41 :	100	100	100	100	100	120
42 :	100	100	100	100	100	120
43 :	100	100	100	100	100	120
44 :	100	100	100	100	100	120
45 :	100	100	100	100	100	120
46 :	100	100	100	100	100	120
47 :	100	100	100	100	100	120
48 :	100	100	100	100	100	120
49 :	100	100	100	100	100	120
50 :	100	100	100	100	100	120
51 :	100	100	100	100	100	120
52 :	100	100	100	100	100	120
53 :	100	100	100	100	100	120
54 :	100	100	100	100	100	120
55 :	100	100	100	100	100	120

56 :	100	100	100	100	100	120
57 :	100	100	100	100	100	120
58 :	100	100	100	100	100	120
59 :	100	100	100	100	100	120
60 :	100	100	100	100	100	120
61 :	100	100	100	100	100	120
62 :	100	100	100	100	100	120
63 :	100	100	100	100	100	120
64 :	100	100	100	100	100	120
65 :	100	100	100	100	100	120
66 :	100	100	100	100	100	120
67 :	100	100	100	100	100	120
68 :	100	100	100	100	100	120
69 :	100	100	100	100	100	120
70 :	100	100	100	100	100	120
71 :	100	100	100	100	100	120
72 :	100	100	100	100	100	120
73 :	100	100	100	100	100	120
74 :	100	100	100	100	100	120
75 :	100	100	100	100	100	120
76 :	100	100	100	100	100	120
77 :	100	100	100	100	100	120
78 :	100	100	100	100	100	120
79 :	100	100	100	100	100	120
80 :	100	100	100	100	100	120
81 :	100	100	100	100	100	120
82 :	100	100	100	100	100	120
83 :	100	100	100	100	100	120
84 :	100	100	100	100	100	120

85 :	100	100	100	100	100	120
86 :	100	100	100	100	100	120
87 :	100	100	100	100	100	120
88 :	100	100	100	100	100	120
89 :	100	100	100	100	100	120
90 :	100	100	100	100	100	120
91 :	100	100	100	100	100	120
92 :	100	100	100	100	100	120
93 :	100	100	100	100	100	120
94 :	100	100	100	100	100	120
95 :	100	100	100	100	100	120
96 :	100	100	100	100	100	120
97 :	100	100	100	100	100	120
98 :	100	100	100	100	100	120
99 :	100	100	100	100	100	120
100 :	100	100	100	100	100	120
101 :	100	100	100	100	100	120
102 :	100	100	100	100	100	120
103 :	100	100	100	100	100	120
104 :	100	100	100	100	100	120
105 :	100	100	100	100	100	120
106 :	100	100	100	100	100	120
107 :	100	100	100	100	100	120
108 :	100	100	100	100	100	120
109 :	100	100	100	100	100	120
110 :	100	100	100	100	100	120
111 :	100	100	100	100	100	120
112 :	100	119	100	100	100	120
113 :	101	80	90	80	75	124

114 :	90	30	90	80	90	120
115 :	119	100	90	80	123	120
116 :	65	170	90	80	123	120
117 :	65	170	90	90	123	120
118 :	100	170	65	100	110	120
119 :	101	170	65	90	68	120
120 :	100	170	90	100	68	120
121 :	100	100	90	110	100	120
122 :	120	100	90	100	123	120
123 :	100	100	100	100	100	120
124 :	100	100	100	100	100	120
125 :	100	100	100	100	100	120
126 :	100	100	100	100	100	120
127 :	100	100	100	100	100	120
128 :	100	100	100	100	100	120
129 :	100	100	100	100	100	120
130 :	100	100	100	100	100	120
131 :	100	100	100	100	100	120
132 :	100	100	100	100	100	120
133 :	100	100	100	100	100	120
134 :	100	100	100	100	100	120
135 :	100	100	100	100	100	120
136 :	100	100	100	100	100	120
137 :	100	100	100	100	100	120
138 :	100	100	100	100	100	120
139 :	100	100	100	100	100	120
140 :	100	100	100	100	100	120
141 :	100	100	100	100	100	120
142 :	100	100	100	100	100	120

143 :	100	100	100	100	100	120
144 :	100	100	100	100	100	120
145 :	100	100	100	100	100	120
146 :	100	100	100	100	100	120
147 :	100	100	100	100	100	120
148 :	100	100	100	100	100	120
149 :	100	100	100	100	100	120
150 :	100	100	100	100	100	120
151 :	100	100	100	100	100	120
152 :	100	100	100	100	100	120
153 :	100	100	100	100	100	120
154 :	100	100	100	100	100	120
155 :	140	85	90	90	123	120
156 :	100	100	125	110	123	120
157 :	90	100	125	100	123	120
158 :	100	200	90	100	100	100
159 :	101	100	90	100	100	100
160 :	100	100	90	100	100	100

Appendix B

Monkey, Bananas and Zombies Tests

The environment of the monkey, banana and zombies tests, described in the results chapter, is presented here. The environment uses a euclidean coordinate space of 150 by 150 to record the location of each agent. The bananas are always found at location $(25, 25)$ at a height of 20 units. The monkey always starts on the floor at $(30, 31)$ and not holding a box. It can walk, while carrying a box, at a speed of 2 units per step nd run empty handed at 8 units per step. Each step takes one simulation clock cycle. To climb up or down a single box requires a full clock cycle of the monkey, regardless of whether it is carrying anything.

The 19 boxes of the environment are specified in table B.11, and shown in figure B.37. These are the initial conditions of the boxes for all four of the tests. The tower construction is initially the same for the tests, but the CLIPS and ICE systems generate a slightly different plan.

Some of the boxes have slightly different attributes in each of the tests, table B.12. Table B.13 shows the extra boxes added to the tower. In the faulty box test, the boxes break when the monkey stands on them, their height becomes zero. Re-planning must add boxes to replace the lost height. In the heavy box case, the heavy boxes are replace along with all of the covered boxes. The heavy object 7 is on top of box 1, thus making the later box inaccessible to the monkey.

Box	Location		Height	On	Under
	X	Y			
0	7	50	2	floor	nothing
1	1	2	3	floor	7
2	25	100	5	floor	nothing
3	65	140	1	floor	nothing
4	100	100	1	11	5
5	100	100	2	4	nothing
6	1	2	1	7	nothing
7	1	2	2	1	6
8	145	95	1	floor	nothing
9	100	100	2	15	11
10	75	3	1	floor	16
11	100	100	3	9	4
12	60	67	1	floor	nothing
13	107	20	1	floor	14
14	107	20	1	13	nothing
15	100	100	1	floor	9
16	75	3	2	10	nothing
17	70	71	1	18	nothing
18	70	71	3	floor	17

Table B.11: Box Characteristics

Box	Faulty	Heavy	Zombie
2	*	*	*
7	*	*	
12		*	
16	*		
17			*

Table B.12: Boxes for the Last Three Tests

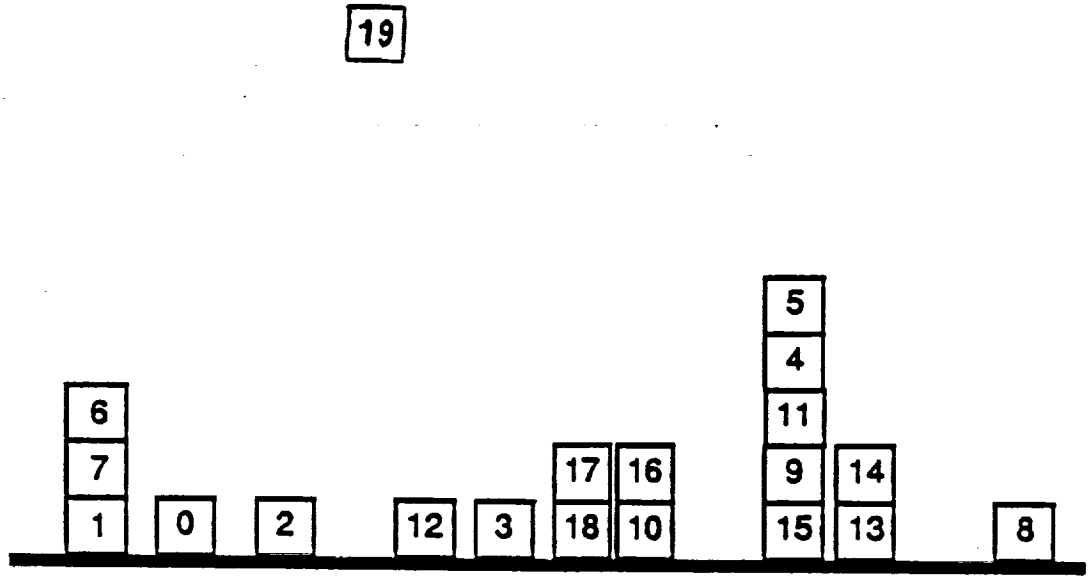


Figure B.37: Picture of the Monkey, Bananas and Zombie World

Faulty		Heavy	
CLIPS	ICE	CLIPS	ICE
13	1	13	1
10	4	10	4
18	10	18	10
11	11	11	11
1	13	1	13
			9

Table B.13: Additional Boxes Needed for the Tower

THIS PAGE WAS INTENTIONALLY LEFT BLANK

Bibliography

[All87] Elizabeth Allen. Yaps: a production rule system meets objects. In *AAAI*, pages 5–7, 1987.

[Bea85] Lee Brownston and et. al. *Programming Expert Systems in OPS5: an introduction to rule-based programming.* Addison-Wesley, Reading, Massachusetts, 1985.

[Ben84] S. Bennett. *Construction of Real-Time Software*, pages 87–100. 1984.

[Ber88] John A. Bernard. Use of a rule-based system for process control. *IEEE Control Systems Magazine*, 8(5):3–12, October 1988.

[CH87] Alan Garvey; Craig Cornelius and Barbara Hayes-Roth. Computational costs versus benefits of control reasoning. In *AAAI*, pages 110–115, 1987.

[Dol87] James L. Dolce. *A Proposal to Investigate Space Station Power System Autonomous Control.* Technical Report POP 87-2, NASA Lewis Research Center, 1987.

[Fil88] Robert E. Filman. Reasoning with worlds and truth maintenance in a knowledge-based system shell. *Communications of the ACM*, 31(4):382, April 1988.

[For82] Charles L. Forgy. Rete: a fast algorithm for the many object pattern match problem. *Artificial Intelligence an International Journal*, 19():17–37, 1982.

[For85] Charles L. Forgy. *OPS-83 User's Manual.* Technical Report , Carnegie Mellon University, 1985.

[HS89a] David Handelman and Robert Stengel. Combining expert systems and analytical redundancy concepts for fault-tolerant flight control. *Journal of Guidance, Control and Dynamics*, 12(1):39–45, January / February 1989.

[HS89b] Barbara Hayes-Roth; Richard Washington; Rattikorn Hewette; Micheal Hewett and Adam Seiver. Intelligent monitoring and control. In *IJCAI*, pages 243–249, 1989.

[HW89] B. Freitag; B. Huber and W. Womann. An integrated knowledge based assembly control system for automobile manufacturing. In *IJCAI*, pages 1369–1374, 1989.

[Kai88] Hermann Kaindl. Minimaxing. *AI Magazine*, 9(3):69, Fall 1988.

[KM85] L.B. Hawkinson; M.E. Levin; C.G. Knickerbocker and R.L. Moore. A paradigm for real-time inference. In *AI and Advanced Computer Technology*, pages 51–56, 1985.

[Kor87] Richard Korf. Real-time hearistic search: first results. In *AAAI*, pages 133–138, 1987.

[KR88] Thomas J. Laffey; Preston A. Cox; James L. Schmidt; Simon M. Kao and Jackson Read. Real-time knowledge-based systems. *AI Magazine*, 9(1):27–45, Spring 1988.

[Kuo82] Benjamin C. Kuo. *Automatic Control Systems*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.

[Kus88] G.L. Kusic. *A System Security Monitor for Space Station Power Systems*. Technical Report TR-PO-88-1, NASA Lewis Research Center, 1988.

[LG89] Thomas Laffey and Anoop Gupta. Real-time knowledge-based systems. In *IJCAI Tutorial: MAS*, page , 1989.

[LP87] Tim A. Nguyen; A. Perkins; Thomas J. Laffet and Deanne Pecora. Knowledge base verification. *AI Magazine*, 8(2):69, Summer 1987.

[FP88] Stuart R. Faulk and David L. Parnes. On synchronization in hard-real-time systems. *Communications of the ACM*, 31(3):274, March 1988.

[Gea84] J.H. Griesmer and et. al. Yes/mvs: a continuous real time expert systems. In *AAAI*, pages 130–136, 1984.

[Geo84] Michael Georgeff. A theory of action for multi-agent planning. In *AAAI*, pages 121–, 1984.

[Geo86] Michael P. Georgeff. The representation of events in multi-agent domains. In *AAAI*, pages 70–, 1986.

[Gia87a] Joseph C. Giarrantano. *CLIPS Reference Guide.* September 13, 1987.

[Gia87b] Joseph C. Giarrantano. *CLIPS User's Guide.* September 13, 1987.

[GR89] Joseph C. Giarranto and Gary Riley. *Expert Systems: Principles and Programming.* PWS - Kent, 1989.

[Gup86] Anoop Gupta. *Parallelism in Production Systems.* PhD thesis, Carnegie Mellon University, 1986.

[Gut88] Scott B. Guthery. Self-timing programs and the quantum scheduler. *Communications of the ACM*, 31(6):696–, June 1988.

[HA87] Patrick J. Hayes and James F. Allen. Short time periods. In *IJCAI*, pages 981–983, 1987.

[Had86] Peter Haddawy. Implementation of and experiments with a variable precision logic inference systems. In *AAAI*, page 238, 1986.

[Hay85] Barbara Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence an International Journal*, 26():251–321, 1985.

[Hob85] J. Hobbs. Granularity. In *IJCAI*, page , 1985.

[LR83] William Long and Thomas Russ. A control structure for time dependent reasoning. In *IJCAI*, pages 230–232, 1983.

[LT86] Marshall I. Schor; Timothy P. Daly; Ho Soo Lee and Beth R. Tibbitts. Advances in rete pattern matching. In *AAAI*, pages 226–, 1986.

[MD80] D. McDermott and J. Doyle. Non-monotonic logic 1. *Artificial Intelligence an International Journal*, 13():41–72, 1980.

[MF86] Bruce Leban; David D. McDonald and David R. Forster. A representation for collections of tempored intervals. In *AAAI*, page 367, 1986.

[Mir87] Daniel P. Miranker. Treat: a better match algorithm for artificial intelligent production systems. In *AAAI*, pages 42–47, 1987.

[Moo86] R. Moore. Expert systems in on-line process control. In *Proceedings of CDC Conference*, page , 1986.

[MS83] Shoichi Masui; John McDermott and Alan Sobel. Decision-making in time critical situations. In *IJCAI*, pages 233–235, 1983.

[MW86] R. Michalski and P. Winston. Variable precision logic. *Artificial Intelligence an International Journal*, 29(2):, 1986.

[OC85] C.A. O'Reily and A.S. Cromarty. Fast is not real-time; designing effective real-time artificial intelligent systems. In *Proceeding SPIE*, pages 249–257, 1985.

[OD87] L.L. Odette and W.B. Dress. Engineering intelligence into real-time applications. *Expert Systems*, 4(4):228–237, November 1987.

[Pau88] L.F. Pau. Sensor data fusion. *Journal of Intelligence and Robotic Systems*, 1():103–116, 1988.

[PD88] Victor R. Lesser; Jasmina Paulin and Edward Durfee. Approximate processing in real-time problem solving. *AI Magazine*, 9(1):49–61, Spring 1988.

[RS80] Chuck Rieger and Craig Stanfill. Real time causal monitors for complex physical sites. In *AAAI*, pages 215–217, 1980.

[Ruo88] Corinne C. Ruokangas. Real-time control for manufacturing space shuttle main engines work in progress. In *AISA*, page 5, 1988.

[SC88] Ralph P. Sobek and Raja G. Chatila. Integrated planning and execution control for an autonomous mobile robot. *Artificial Intelligence an International Journal*, 3(2):103–113, April 1988.

[SH88] James C. Sanborn and James A. Hendler. A model of reaction for planning in dynamic environments. *Artificial Intelligence an International Journal*, 3(2):95–102, April 1988.

[Shi87] Richard S. Shirley. Some lessons learned using expert systems for process control. *IEEE Control Systems Magazine*, 11–15, December 1987.

[Sho88] Yoav Shoham. Chronological ignorance: experiments in nonmonotonic temporal reasoning. In *Artificial Intelligence an International Journal*, pages 279–331, April 1988.

[Sor85] Sorrells. Time-constrained inference strategy for real-time expert systems. In *IEEE Proceeding WESTEX*, pages 1336–1341, 1985.

[ST86] A. Paterson; P. Sacho and M. Turner. Escort: the application of causal knowledge to real-time process control. *Expert Systems*, 31(1):22, January 1986.

[Sta88] John A. Stankovic. Misconceptions about real-time computing: a serious problem for next-generation systems. *IEEE Computer*, 21(10):10, October 1988.

[VK86] Marc Vilain and Henry Kautz. Constraint propagation algorithms for temporal reasoning. In *AAAI*, pages 377–, 1986.

[WH88] Show-Way Yeh; Chuan-lin Wu and Chaw-Kwei Hung. Solutions to time variant problems of real-time expert systems. In *AISA*, page , 1988.

[WH89] Richard Washington and Barbara Hayes-Roth. Input data management in real-time ai systems. In *IJCAI*, pages 250–255, 1989.

[Win79] P.H. Winston. *Aritificial Intelligence.* Addison Wesley, Reading, Massachusetts, 1979.

[WL83] Frederick Hayes-Roth; Donald A. Waterman; and Douglas B Lenat. *Building Expert Systems.* Addison-Wesley, Reading, Massachusetts, 1983.

[Wol87] A. Wolfe. An easier way to build a real-time expert system. *Electronics*, ():71–73, March 1987.

[YM83] Naoyuki Yamada and Hiroshi Motoda. A diagnosis method of dynamic systems, using the knowledge of system description. In *IJCAI*, pages 225–229, 1983.