111-62

NASA Contractor Report 187505

19748

TRANSLATING EXPERT SYSTEM RULES INTO ADA CODE WITH VALIDATION AND VERIFICATION

Lee Becker and R. James Duckworth

Worcester Polytechnic Institute
Worcester, Massachusetts

Peter Green

Real Time Intelligence Systems Corporation
Worcester, Massachusetts

Bill Michalson

Raytheon Company
Marlboro, Massachusetts

Dave Gosselin, Krishan Nainani, and Adam Pease

Worcester Polytechnic Institute
Worcester, Massachusetts

Grant NAG1-964
June 1991

NASA

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

TABLE OF CONTENTS

Table of Contents - continued

1. INTRODUCTION

This is the final report for work performed under NASA Grant NAG-1-964 during the period January 1989 to December 1989. This work is a continuation of the work started under NASA Grant NAG-1-809 as reported in NASA Contractor Report number 181810 "Evidence Flow Graphs for Validation and Verification of Expert Systems".

This work is part of an ongoing program to develop software tools which enable the rapid development, upgrading, and maintenance of embedded real-time artificial intelligence systems. Such systems are projected to become a critical component of the next generation air and space vehicles for NASA and the Air Force who supported this research. Realization of the potential of such systems for improving mission performance and reducing vehicular personnel is, however, being significantly impacted by long and costly development cycles and the problems of maintaining these systems. Of special concern is verifying that these systems will perform as expected and validating that upgraded versions do not inadvertently introduce catastrophic side effects.

Current AI technology uses expert systems shells to develop rules in a rapid prototyping environment. Then the expert systems shell is used as the run-time environment. There are two major problems with this:

a) The resultant systems often cannot execute rapidly enough to meet real-time decision making constraints. Often the rules have to be hand translated into Ada which solves the speed problem but results in a system which is difficult to expand and maintain.

b) There is a lack of good methods for the validation and verification of expert systems. The problems here stem from the fact that the reasoning algorithms can have apparently non-deterministic properties as they mimic the reasoning of people.

The Goals of this phase of the project were twofold:

1) To develop methods for automatically translating from high level knowledge representations, such as rules, to executable Ada code. This is so that rapid prototyping could be used for rule development and enhancement without the expense and time of repeatedly hand translating the rules to Ada code after each change to the rules.

2) To develop methods for the validation and verification of these knowledge representations by means of automatic test generation and evaluation. While the execution of rules appears to mimic non-deterministic human reasoning, it is a deterministic data-driven time-dependent process which can be tested using established techniques such as monte-carlo simulation. Effective testing, however, requires a large number of trials and the generation of tests and the interpretation of the results is a more complex process than in many more conventional systems. Hence automated test generation and evaluation tools are required.

These goals were successfully achieved and a prototype system was demonstrated which automatically translated rules from an Air Force expert system into Ada code. The resultant code was demonstrated to run in real-time. The prototype test generation and evaluation system was also demonstrated to be able to detect errors in the resultant system. The methods developed were based upon using Evidence Flow Graphs (EFGs) as an intermediate representation.

The development of embedded Artificial Intelligence (AI) systems such as the NASA Manned Maneuvering Unit (MMU) diagnostic system or the Air Force Adaptive Tactical Navigator[1] have shown that there is a conflict between the requirement for the system to respond rapidly to changes in the system's inputs and the ability to easily develop and maintain the system's knowledge base. Systems which have to respond rapidly, in real time, to external stimuli are typically coded in a language such as Ada so that they can execute code modules efficiently with a minimum of system overhead for control. This makes the development and modification of these systems a lengthy and costly process.

The knowledge bases of embedded intelligent systems are not static. They must be changed in response to changes in mission requirements, environments, and improved knowledge about how to most effectively use these systems. If these knowledge bases are coded in a language such as Ada then the time to change the software and verify that the system is still operating correctly can become undesirably long.

Ideally, the knowledge base should be maintained in an environment which allows rapid changes to be made and evaluated. Such an environment is provided by expert systems such as CLIPS[2]. These systems allow the rapid development of knowledge bases and rapid experimentation with changes and enhancements.

Unfortunately the resultant systems run slowly and are not capable of performing in many real-time situations. This is due to the sequential nature of their inference engines which precludes parallelizing their execution and the overhead associated with providing a highly interactive development environment.

When changes are made to the knowledge base of an intelligent system the changes may result in undesired side effects such as previous functions no longer working correctly or harmful output commands being produced (such as activating effectors at the wrong time). It is important to validate that a knowledge based system works according to specifications before it is released to the field and to verify that the system still works correctly whenever its knowledge base is updated as part of a maintenance procedure.

Testing of a knowledge based system involves extensive use of test meta-knowledge to ensure that testing covers all operating conditions in such a way as to ensure an adequate level of confidence in the system. It is also desirable that the evaluation of the test results be performed automatically as maintenance personnel may not have a high degree of familiarity with the knowledge base and its possible failure modes.

Current expert systems do not support such validation and verification methods. Knowledge based systems written in a standard programming language can be validated and verified using standard software techniques but these methods take a considerable amount of time and are not substantially automated.

At present, knowledge bases are typically developed using expert systems and then they are recoded in a language such as Ada so that they will run in a real time environment. This often involves stripping out much desirable functionality and has resulted in such system compromises as the replacement of sophisticated reasoning mechanisms with simple table lookups. Also it makes the subsequent maintenance and upgrading of these systems a difficult and time consuming process.

Research performed over the past year has shown that it is possible to automatically convert knowledge representations, such as those used by the MMU and Adaptive Tactical Navigator Systems, into Ada code which can be executed in real time. Further, this research project demonstrated that it was possible to test the performance of these systems using the evidence flow graphs which are used as an intermediate stage in the translation process.

This research has resulted in the software system called KRAM (Knowledge Representation into Ada Methodology) shown in figure 1.1. In this system high level knowledge representations, such as rules developed using an expert systems shell, are converted to Ada code modules called Activation Framework Objects (AFOs). These AFOs are linked with the Activation Framework (AF) execution environment to produce an intelligent run-time system which can be executed in real-time on a parallel processor. The resultant run-time system can also be tested to ensure that it will perform correctly in a mission environment.

KRAM uses the paradigm, shown in figure 1.2, of application modules which communicate by means of messages. These application modules can range in complexity from a single rule to a signal processing algorithm. Some modules are experts in a limited domain while others interface to sensors, actuators, and displays.

KRAM uses a data flow paradigm in which the arrival of data causes applications modules to become primed for execution. Applications modules can be executed on a distributed heterogeneous computer system with execution priority on any processor being given to the applications module which is most important to execute at that instant in time.

KRAM uses an abstract representation for systems called evidence flow graphs (EFGs) which consist of processing nodes interconnected by a directed graph[3]. Messages, which flow between nodes of this graph over arcs, trigger the execution of the nodes. Nodes perform four functions:

1) Initialization - before execution starts
2) Priming - determines whether the node has sufficient data on its input arcs to be fired - a node may require data on multiple arcs to fire
3) Importance - generates an estimate, based on locally available data as to the importance of executing this node - usually based on messages on its input arcs
4) Transfer - typically gets messages from the input arcs and sends messages out on output arcs.

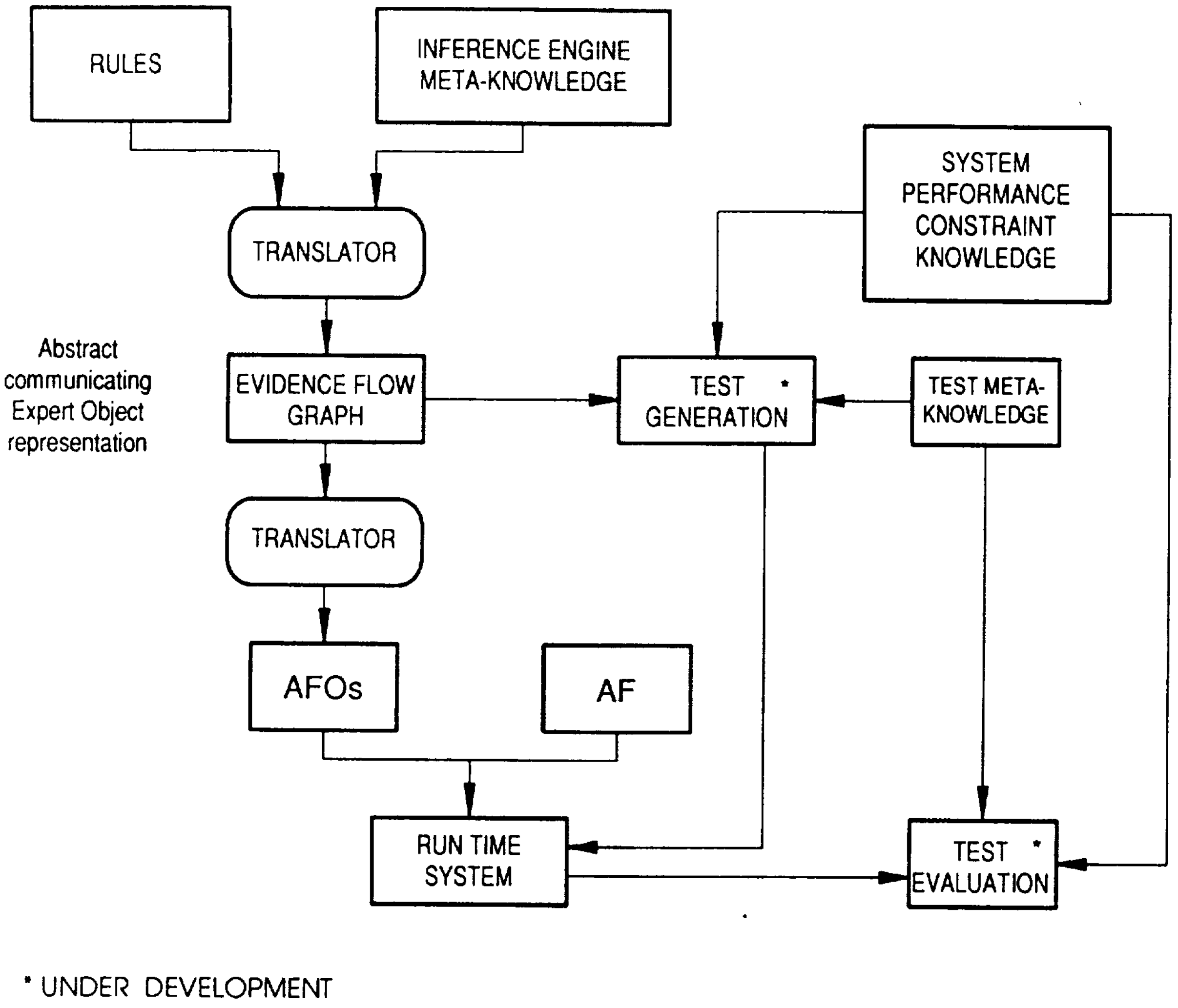


Figure 1.1 System for Converting Expert Systems Rules to a Run-Time System Capable of Real-Time Execution

These node functions are described in terms of the evidence flow graph syntax, for which a BNF description has been developed. An example of an EFG node description is:

```
node 1 rule_1 is            ; this node is called rule_1
  inputs (i1:sys_input) A:b; Arc 1 is an external boolean input called A
         (i2:sys_input) B:b; Arc 2 is an external boolean input called B
  outputs (o1:rule_2/i2)  C:b; Output 1 is a boolean called C
                            ; sent to input arc 2 of node called rule_2
  priming(i1 and i2);
  importance static(24);
  transfer
       get_msg(i1,A);
       get_msg(i2,B);
       if A=true and B=true then send_msg(24,o1) end if;
   end transfer;
end node rule_1;
```

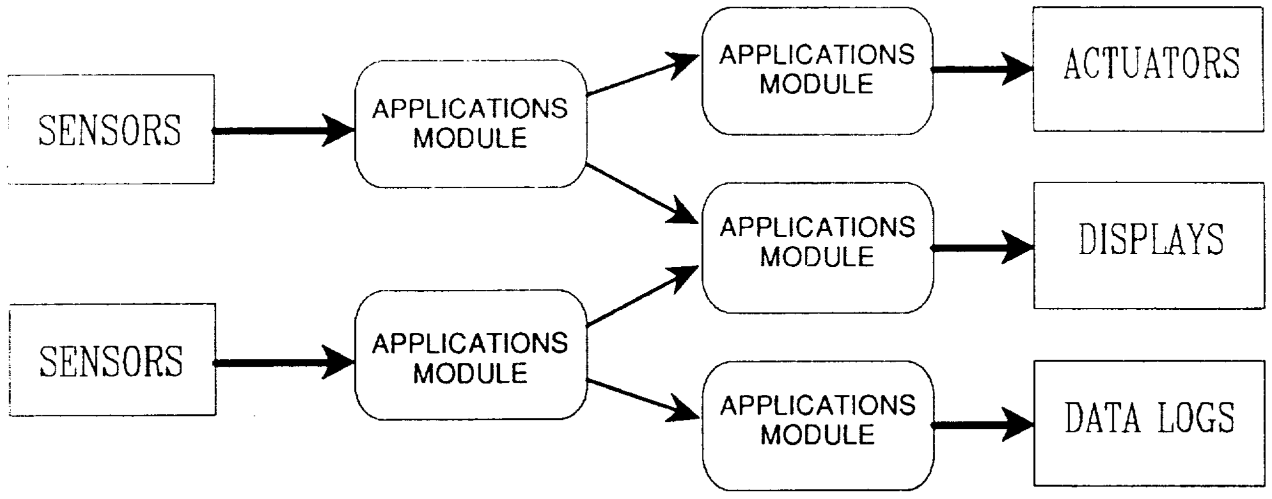


Figure 1.2 The KRAM Data Flow Model of a system

A Translator to EFG syntax has been developed for Adaptive Tactical Navigator rules and translators have been designed for ATN causal networks and for forward chaining CLIPS rules.

Expert systems shells typically contain extensive implicit knowledge. This makes the development of rules or other high level representations simpler because the user does not have to express this implicit knowledge. This implicit knowledge is incorporated explicitly into the evidence flow graph nodes during the translation process thus making the resultant EFG contextually independent of its source. The result is that EFGs from different sources can be joined together into a single EFG.

A translator has been developed to convert EFG nodes into Ada code modules called Activation Framework Objects. In this translation process the initialization, priming, importance, and transfer functions of each EFG node are converted to Ada code procedures and placed in a package. The packages for each node are then linked with an Ada version of the Activation Framework software package called AFA to form a run-time system.

AFA uses the paradigm of expert object code modules which communicate by means of messages. This paradigm was chosen based on the lessons of DSN [4]. In AFA, code module execution is triggered by the arrival of data in the form of messages. If a code module, called an Activation Framework Object (AFO), has no input messages then there is no requirement to schedule its execution. On the other hand, if an AFO has many important messages queued on its input port then this AFO should be given high priority for execution.

AFOs are written as procedures, but they are executed as independent code strips by the framework to which they are attached, as shown in figure 1.4. The framework performs AFO execution scheduling and performs message delivery. This message delivery is performed by in-memory copying for delivery to local AFOs or by use of network communications for delivery to AFOs in remote processors.

There is intended to be one AF framework per processor. This framework provides a uniform interface for the AFOs which use a template of the form:

```
get_message()
perform algorithm
send_message()
```

Conceptually, in some processors, the framework runs stand alone in place of the operating system, and in other cases the framework runs on top of an operating system such as Unix or VMS. In the case implemented, MSDOS, the framework uses the BIOS for performing some functions but largely runs as a stand alone multi-tasking operating system.

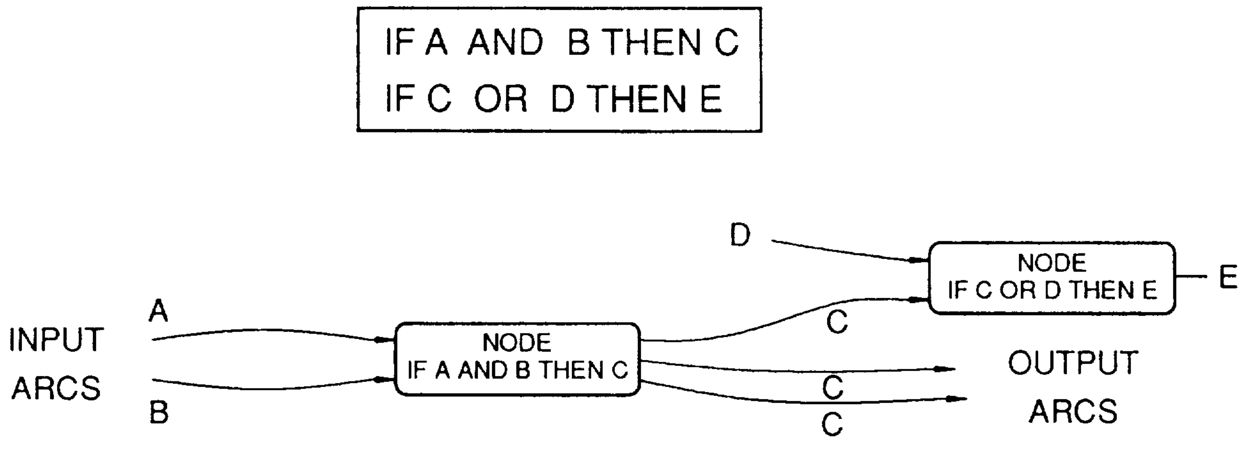


Figure 1.3 Conversion of Expert Systems Rules to Evidence Flow Graphs

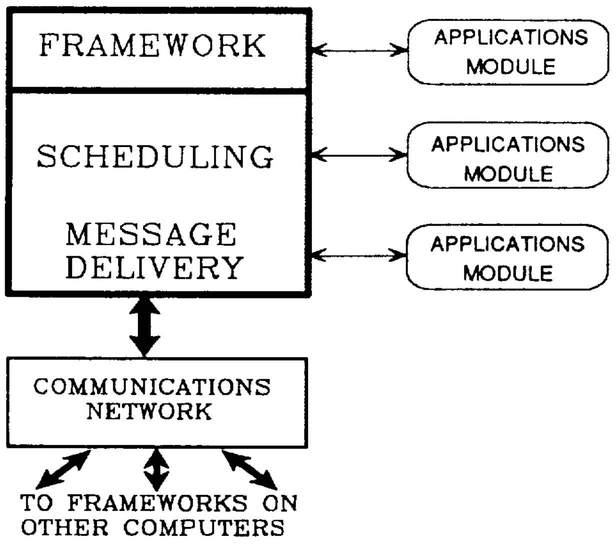


Figure 1.4. AFO Application Code Modules are attached to a framework.

A key feature of AF is that applications AFOs can control the prioritization of messages and remote AFOs. When an AFO sends a message to another AFO, it specifies how important this message is within its own domain of knowledge. This local importance is multiplied by the global importance (as in management hierarchy) of the AFO to form the activation level of the message. Message transmission is prioritized by activation level. More importantly, the prioritization of the AFOs within a given processor is based upon the activation levels of the messages on their input queues. By default the importance of an AFO is its global importance multiplied by the sum of the activation levels of the messages queued on its input ports. This means that an AFO can trigger the execution of a code module on a remote processor simply by sending it a message, and by controlling message importances, the relative execution priorities of remote processes can be controlled.

The first version of AF[5] was written in the C language and did not allow AFO pre-emption when another AFO sent a message. This version was used in a sequence of experiments to write a multi-layer mobile robot controller. While usable in this form, it was found that the lack of AFO pre-emption caused significant programming problems as the execution of any AFO code thread had to be kept short enough not to cause a significant delay in execution of other AFOs waiting on its completion. Messages were also coded as ASCII strings which required significant programming labor and processor cycles for conversion of data to and from an ASCII format.

A LISP language version of AF was used as the basis of the laboratory prototype of the Adaptive Tactical Navigator(ATN)[6]. This real-time AI system was developed by The Analytic Sciences Corporation. The ATN system is designed to replace the functions of the navigator in the next generation of Air Force attack aircraft. Several experimental versions of AF have been developed subsequently. One version supported AFO pre-emption upon message delivery and another version supported dynamic AFO instantiation and the use of distribution nodes to which messages can be delivered for distribution to a group of AFOs. The Real-Time Intelligent Systems Corporation has developed a completely new C language version of AF called AFC [7] which features AFO preemption and uniform treatment of events. The design and user interface for AFA was based upon AFC.

A major emphasis of this research project is on the capability to verify that the resultant system is able to perform according to some specification and to validate that the system still performs correctly after modifications as part of a maintenance procedure. There are several major issues here:

1) How to specify how an expert system should perform. During this research phase, a constraint-based language has been developed which allows the specification of the expected relationships between the inputs and outputs of an expert system. This language includes the ability to specify:
a) Expected ranges of input values
b) Outputs which must occur for certain ranges of input values
c) Outputs which must not occur and the conditions under which they are prohibited.
d) The order in which inputs can occur.
e) The order in which outputs must and must not occur
f) Likely and possible ranges of input values

2) How to determine whether the system performs as specified. Here there are several choices:
a) Evaluation of the rules (or other representation) for logical consistency. This is being studied by other investigators and was not investigated as part of this study.
b) Testing of the system using random inputs which are chosen based on the specification language. This monte-carlo approach was investigated and used during the research reported here.
c) Evaluation of the system's EFG to determine whether errors can be detected by graph analysis techniques. In the research reported here the use of Petri networks was investigated to determine whether Petri network methods would enable the detection of such problems as unreachable nodes, deadlock, and livelock.

3) When converting a rule base or other representation into an evidence flow graph there is a choice as to whether to force the resultant system to follow the serial execution constraints of the original inference engine or whether to relax these constraints. Relaxation of the serialization constraints can result in a system with considerable parallelism which is capable of execution at high speed on a parallel processing system. This is very important to related projects such as KRAPP[8] which are trying to minimize the execution time of intelligent systems through the use of parallel processors.

If the predicate clauses of rules in a rule base are incompletely specified, and hence rely on rule ordering for correct execution, then such a system will not execute correctly when translated to executable Ada code unless serialization constraints are specifically incorporated by the use of inhibitor links in the EFG. As these constraints are highly undesirable from a performance standpoint, the approach taken in developing the rule to EFG translator described in this report is to maximize parallelism. This does, however, require a thorough testing of the resultant system to ensure that it still performs correctly even after translation to an EFG.

In prior research, testing was performed by translating the Evidence Flow Graph into a Simscript simulation model and testing the performance using a random selection of input values. This approach has became problematic for several reasons:

a) EFG syntax has become much more complex as the number of cases to be incorporated in its generalized format has increased. This made the translation into Simscript much more complicated.

b) The Simscript simulation model ran very slowly.

c) The simulations produced a large volume of output which was hard to use to detect errors.

This problem has been overcome by using AFA in place of Simscript. AFA, by its nature, provides most of the queued execution model features of Simscript and executes much more rapidly. AFA has been enhanced by the development of interfaces to input test data and to capture resultant output data. This version of AFA has been used with the test generator to demonstrate how faults can be detected using monte-carlo techniques.

As shown in figure 1.1, the test generation program uses system performance constraint knowledge, along with test metaknowledge and the EFG, to generate randomly selected test inputs for the run-time Ada version of the system. These are used as successive inputs to the system and the outputs are captured into a disk file. These test outputs are evaluated using a rudimentary test evaluator program.

In the monte-carlo analysis method, sets of inputs are generated by random choice within the ranges specified in the test language specification. These are applied in a randomly selected order and messages are propagated between nodes until all activity in the network ceases. As execution progresses, messages generated at the output nodes are captured in a disk file. Execution is repeated for a large number of sets of input until the network has been tested an adequate number of times for the desired level of confidence.

The monte-carlo analysis is capable of detecting problems such as:

a) Outputs do not meet specification

b) Outputs are inconsistent

c) Outputs sensitive to small changes in input values

d) Outputs sensitive to node (rule) execution order

e) Outputs sensitive to input data order

In addition it has been found that problems such as reachability can be detected during conversion of a rule set into an EFG.

During the past year the following project objectives have been successfully achieved:

1. Development of a comprehensive BNF language specification for Evidence Flow Graphs which encompasses the features found in all the intelligent systems studied including many expert systems.
2. Acquisition of Adaptive Tactical Navigator system causal network and rule knowledge representations and their conversion into Evidence Flow Graphs.
3. Study of the CLIPS expert system shell and the development of techniques to convert forward and backward chaining rule representations into EFGS.
4. Development of an automatic translator to convert ATN rules in a CLIPS-like syntax to EFGs. This translator was written in Ada.
5. Development of a translator from EFGs to Ada code modules which, when linked with AFA, form an executable system. This translator is written in Ada.
6. Development of an Ada version of the Activation Framework called AFA.
7. Development of a test specification language.
8. Development of a test generator program written in Common Lisp.
9. Study of methods for test evaluation.
10. Study of Petri Network technology as a way of analyzing EFGs to detect problems.

The most important achievement was the demonstration of the viability of the concept of starting with rules and automatically converting these into executable Ada code. Not only was the resultant code executable in real-time but it was demonstrated that the system could be tested to ensure that it performed to required performance specifications for validation and verification purposes. Also, the associated KRAPP project has demonstrated that the code is executable with a high degree of parallelism on a fault tolerant parallel processor.

It was found that random monte-carlo testing was a viable approach to evaluating performance but this required large amounts of processor time. It was also found that a large volume of output data could be generated which indicated that it was highly desirable to use a much more sophisticated computer program to analyze the output.

Petri network technology was studied in the hope that it would lead to techniques for the analysis of EFGs thereby avoiding the processor time spent in monte-carlo testing. A technique for converting EFGs into Petri Nets was developed but the resultant networks were so complex as to make them intractable from an analytic viewpoint. It was concluded that it was desirable to extend Petri network techniques to be able to include EFGs so that graph analysis could be performed directly on the EFG to detect possible execution cycles resulting in livelock or deadlock.

An important byproduct of the research was an analytical understanding of the functioning of rule-based expert system shells. In the process of studying how to convert rules and their associated shell meta-knowledge into EFGs, many potential problems that could cause incorrect execution became apparent. Intelligent systems are easy to develop using expert system shells because of the meta-knowledge contained within the shells. Users only have to provide the needed additional knowledge in the form of rules. The disadvantage is that users may be unaware of the constraints on or actions of the inference engine resulting in unexpected side effects under certain conditions.

As a result of this research we have come to believe that the meta-knowledge in expert systems shells should be explicitly stated and specified. Further, we have come to realize that minor changes to an inference engine could result in significant changes in the execution of a rule base. This implies that validation and verification procedures for rule-based expert systems must include the shells themselves as well as the rules they process.

2. EVIDENCE FLOW GRAPHS

This section is an exploration into the use of evidence flow graph representations for expert systems. An EFG is a common graphical representation for intelligent systems. There are several benefits of converting an expert system into an evidence flow graph. By using this representation it is more feasible to perform validation and verification than if the system were left in its original form. This model also provides a simpler form for parallel execution. Lastly, it is a unifying representation for a number of dissimilar systems.

This report provides a certain amount of theoretical background to facilitate understanding of the material presented, and then presents a number of chapters that deal with the research undertaken in an increasingly more specific manner. Section 2.1 examines a number of expert systems, describes the various types of meta-knowledge found in an expert system, and explains the mathematics of evidential reasoning. Section 2.2 describes the evidence flow graph concept, how it functions, and theories associated with it. Section 2.3 discusses the translation of an expert system into the evidence flow graph model.

2.1 Background

This section provides an examination of the major areas of existing work which either motivated or gave a foundation for the research presented in this report. Expert systems are the central topic in this report. Section 2.1.1 defines and discusses expert systems and provides a context of historical research. In addition, it examines some of the current shortcomings in existing systems. Section 2.1.2 discusses the large body of implicit knowledge contained in expert systems. Section three further expands on one of the types of meta-knowledge introduced in section two: confidence calculation or evidence mathematics. Extensive work has been done in this area, and for the most part, only those methods which have been specifically applied to existing expert systems are examined.

2.1.1 Expert Systems

An expert system is a software system which models the reasoning process of an human who has knowledge about a certain domain[9]. A human expert uses past experiences and rules of thumb to deal with present problems. For example, a physician uses his training and experience to diagnose his patients problems. An expert system attempts to model this kind of knowledge in a limited but intuitive way.

Expert systems are one of the first practical and widespread applications of Artificial Intelligence (AI) research[9]. Part of the success of expert systems is that they model human intelligence in a way that is useful, and also that knowledge is relatively easily captured in this form.

Much of the function of a typical expert system could be duplicated by a conventional programming language. However, an expert system is a far more specialized tool than a general purpose language. In giving up flexibility, much less effort is required to achieve the same results. An expert system is usually not intended to perform graphics intensive operations, or complicated mathematical calculations, so it is not designed with this in mind.

Expert systems also differ from much other AI work. The factor once again that separates expert systems from other representations is that they are designed to represent knowledge in a specific limited format. They do not handle the very general symbolic manipulation that LISP does, for example.

With the extensive abstraction that make expert systems so easy to use there is often an additional penalty: speed. Expert systems are optimized for human understanding and not computer execution.

In many expert systems, the knowledge is summarized in the form of rules. These rules are typically acquired in interviews conducted by a knowledge engineer. It is his job to collect and interpret information provided to him and convert this data into a form that is usable by the expert system that he has chosen.

The application of these rules is controlled by a portion of the expert system called the inference engine. The inference engine must decide which rule to apply given a set of facts and previous conclusions.

Therefore, a typical expert system consists of three parts (see figure 2.1), a knowledge base, an inference engine, and a data base (or fact base). The knowledge base is the collection of rules. It provides the knowledge to reach conclusions based on given information on a certain topic or set of topics. The data base consists of the information that is provided to the rule base. The inference engine controls evaluation of the rules. It determines what information is used first, and the order in which rules are examined.

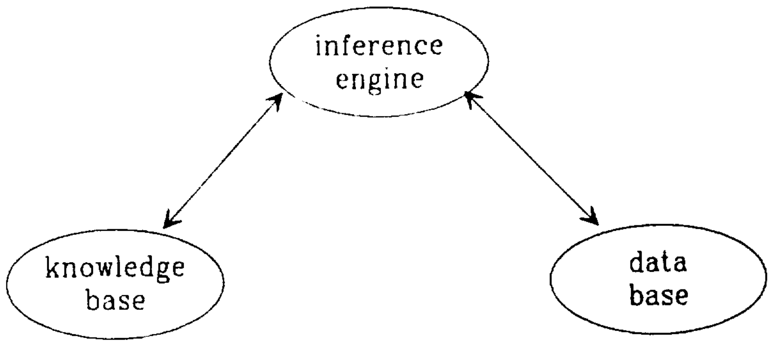


Figure 2.1 A Typical Expert System.

A rule is generally of the form if <preconditions> then <postconditions>.
For example,

```
if has_wings and can_fly then
     animal_is_bird
```

The postconditions typically form the preconditions for subsequent rules.

One of the first working expert systems was the Dendral system developed at Stanford University[10]. This early system combined procedural modules, algorithms and production rules to provide a decision making tool for an organic chemist. From this system, many others followed, most notably Mycin[11]. Both of these were highly specialized systems that evolved over several years of research. The most interesting development from the Mycin research was the Emycin shell. This system was the first expert system shell. A shell is essentially an inference engine without a rule base or data base. It is helps to separate the expert system developer from programming. One shell may be suitable for solving a number of different types of problems. Emycin was applied further for medical diagnosis, but was also used for non- medical diagnosis. Until this point, expert systems had to be created virtually from scratch. Many other shells have since been created by other researchers and companies.

There are two types of shells to consider. One type is the application-specific shell such as Emycin. In this type of system, a certain amount of knowledge is embedded in the inference engine provided, along with task-specific functions that the rule base may access. The second type is the generic shell. It contains no domain specific knowledge and as a result is applicable to a wide variety of problems.

In examining the commercial shells it became clear that a great deal of knowledge is embedded in the inference engine. Any attempt to translate an expert system to an EFG must take into account this metaknowledge. This will be described in the following sections.

2.1.2 Metaknowledge

In the broadest sense, metaknowledge is knowledge about knowledge. Metaknowledge may be knowledge about how knowledge is organized, what it means or how to obtain more knowledge. The base knowledge in this case is that which is contained in the facts and rule base of an expert system. The metaknowledge is about what the rule base knowledge means and how it is to be used. Some of this metaknowledge tells how the inference engine functions. Other metaknowledge shows how the performance of an expert system may be improved within the constraints of functionality. Still other metaknowledge provides information on testing the system. For the most part, this metaknowledge has not, in the past, been categorized or examined in any systematic fashion, if it has even been recognized at all.

It is important to be conscious of this metaknowledge for a number of reasons. Being conscious of the differences between expert systems, and being able to categorize these differences, allows more efficient comparison and evaluation of the systems. To attempt a common representation of different expert systems, their differences must be categorized to preserve their original functionality. Lastly, in developing classifications for the capabilities of different expert systems, it should be more possible to point further expert system development in the most fruitful direction.

A number of different types of knowledge have been identified which do not form part of the rule base or data base of an expert system (see figure 2.2). There are three categories of this knowledge. Some is contained in the inference engine and is necessary for proper execution of the system. It is called functional knowledge. Other information is called optimization knowledge. It does not affect the correct operation of a system, but is applied to a parallel implementation of a system to improve performance. The last kind of knowledge is called simulation knowledge. This information is used in validation and verification of the system.

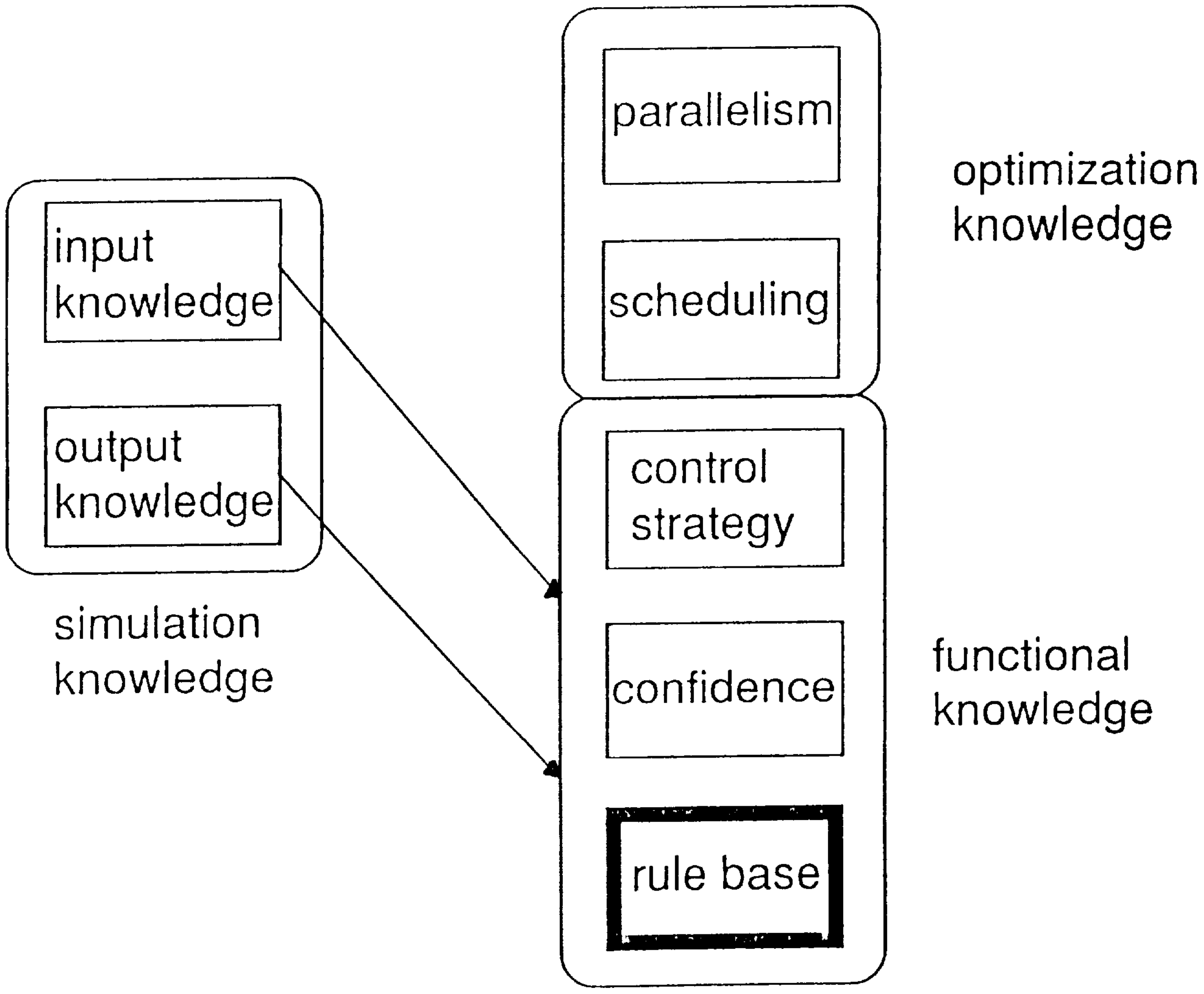


Figure 2.2 Types of Metaknowledge.

Functional knowledge encompasses the rule base itself and confidence and control knowledge. Confidence refers to the system by which conclusions are associated with a degree of certainty. Control refers to the ordering of rule execution.

Optimization knowledge consists of scheduling and parallelism knowledge. Scheduling refers to how the execution of rules may be ordered within the functional restraints of the system. Parallelism refers to how a system may be rendered and executed in a parallel form.

Simulation knowledge may be broken down into input and output knowledge. Input knowledge is information about the range and ordering of inputs. This knowledge is necessary to limit the number of test cases and to ensure that test cases are given in a useful domain. Output knowledge consists of a set of results that should occur given a set of inputs. Simulation knowledge, and how it relates to the process of validation and verification, is discussed more fully in section 3.3. Specific expert systems, and the metaknowledge that governs their operation, are described in section 2.2.2.

2.1.2.1 Rule Base

This type of metaknowledge is the least obvious of all. It has to do with the interpretation of each individual rule - the rule semantics. The highest level of knowledge in this area is that the rule base is, in fact, a rule base instead of, for example, a shopping list, or poetry. Beyond this, there are more useful observations. For example, given the following rule,

```
if A and B then
    C
```

if the expression is true, then do we assume that the symbol C has a value of true? If the expression is false, may we then assume that C is false, or is it not assigned a value and may therefore contain a random value?

For the most part, this issue will be left here. Most semantic attributes of any given expert system will be stated in a general discussion of the language or be implicit in its syntax. However, it is important to keep this issue in mind to avoid missing something important, or declaring an aspect of a given expert system language as obvious and not worthy of mention.

2.1.2.2 Confidence

Much knowledge has a measure of uncertainty connected with it. Facts may be confused or they may contain errors in measurement. A Rule may not always be true for the information with which it is defined. This situation is most obviously true in those systems which deal with medical diagnosis such as [11]. Limited data may be available at any one time, yet useful conclusions may be reached as long as it is known that the conclusion is not completely certain.

Different systems have different ways of dealing with uncertainty. Some are very simple and others quite complex. The complexity of the confidence system should be carefully considered. Human experts are given to provide conclusions on a small number of conditions[12]. An excessively complex system may force the designer into adding information into the system purely to satisfy its mathematical requirements, rather than to add legitimate data.

The systems described in section 2.2.2 are all quite simple. The causal network confidence system described in section 2.3.1 is an example of a more complex method.

2.1.2.3 Control

At any one time, a number of rules in an expert system may be able to reach a conclusion. Some procedure must be used to choose which one to execute first. Different systems and shells utilize different methods to do this. Some emphasize simplicity, executing rules in the order that they were entered by the user. Others are extremely complex, considering the recency of facts used by rules, the size of rules, confidence in rules and data, and other factors.

In a simple system all knowledge must be encoded explicitly in the rules. In a more complex system, this is not the case. Let us take for example a system in which rules which all have enough information to reach a conclusion are known to be executed in a sequential order. If we have two rules

```
if A and F then
    C,
```

and

```
if D and E then
    F,
```

and given that as input to the system we have A, D, and E as true and F as false, the system will not conclude C. Given that this is the result that the designer wishes, everything appears correct. However, if a strict sequential ordering of the rules was not necessarily the case, the inference engine might execute rule #2 before rule #1 and conclude C instead. Therefore, what the designer really means is:

```
Q is false

if A and F and not Q then
     C
```

and

```
if D and E then
     F and Q.
```

Any system designer should be aware of this "hidden" knowledge, as must any researcher attempting a common expert system representation. Unfortunately the details of this control are sometimes inadequately documented.

2.1.2.4 Scheduling

Scheduling refers to what liberties may be taken within the bounds of a given control strategy. Some paths of reasoning in a given system may be more promising than others, and it may be possible to give priority to these paths without disrupting the functionality of a system. This topic will not be examined further.

2.1.2.5 Parallelism

Parallelism refers to the method and extent to which a set of decision processes may be caused to execute simultaneously. The need for speed in computation has increased dramatically in the past decade. The increase in hardware or processor speeds and software and algorithm improvement has not kept pace with this need. Normally, computer instructions are executed sequentially. These instructions can not be evaluated at a speed any faster than the computer's processor is able to handle. This is known as the von Neumann bottleneck.

A solution to this problem is to have multiple processors working at the same time on different pieces of code. For example, given the calculation,

$$a = b \times c + d \times e$$

a conventional computer system would find b×c, then d×e and add the results. A parallel system would calculate b×c and d×e simultaneously, each on a different processor. The processors would then report their results and one processor would add those results to arrive at the answer.

In this simple situation it would seem that parallelization would be a wonderful solution, and one without penalty. However, this is not the case. Suppose that multiplication were a very quick operation, and that message passing between processors were quite lengthy. The result would be that the time saved in computing the two products at the same time would be lost in the time taken to report the results of the calculation.

The tradeoff then is time saved in parallel execution versus the time to pass whatever messages are required. While these times can be estimated, they most likely will not be exact. Therefore, the solution found to this tradeoff may be less than optimal.

One problem then, is to determine to what extent code should be broken up into parallel sections so as to balance the savings of parallel execution against the penalty of message passing. This is referred to as the grain of parallelism. A related issue is, given a certain desired grain of parallelism, how should code sections be grouped so as to minimize message passing. This is known as partitioning.

There is an additional difficulty in rendering an expert system in parallel. The control strategy must also be functional in parallel. This may be done in a number of ways. The most cumbersome method is to pass global importance values to each rule and only allow the one with the highest value to execute. However, this eliminates parallelism. A better method is to allow as many rules as possible to execute, and order the outputs, allowing only the output from the rule with highest importance to be passed along.

2.1.3 Evidence Mathematics

Evidence mathematics consists of mathematical techniques for deriving a measure of confidence for a given piece of data. It is this body of mathematical theory which forms the basis for the confidence systems and confidence metaknowledge incorporated into many expert systems. The basic concepts are much the same as those found in an introductory course in probability.

For example, given that we have an animal which is a bird, what is the probability that it will fly? This might be expressed as P(fly|bird). This expression states that given only the data "bird", and that no other evidence exists that may affect the probability of "fly", we may conclude that "fly" has a certain probability. However, according to traditional models of probability, if we acquire new evidence such as "broken_limb", which may affect "fly", we must consider the old expression invalid.

The Bayesian scheme for evidence combination is the first method to be examined here. It relaxes slightly the constraints of probability theory [12]. It assumes that we may state that a given set of variables or preconditions affects another given set of variables or postconditions, regardless of all other variables. It requires that for every precondition we must have a conditional probability value to pair with any affected postcondition.

This system is seldom used. It requires a large set of values and extensive computation. The conditional probabilities are rarely accurate except in the case where large amounts of statistical data are available. The causal network discussed in Sections four and five is an example of this type of system. The computation is organized in a graphical manner and computations involving different dependencies may take place in parallel. The computations propagate through the network until all the data is consistent with the mathematics of probability.

Some of the problems with the Bayesian method may be illustrated by the following example [13]. Let us say that we have questioned an expert and that we have derived the following rule:

```
if animal has wings and animal lays eggs then
     conclude with certainty 0.9 that animal is bird.
```

The expert may well agree with this statement, but is he willing to agree with

```
if animal has wings then
     conclude with certainty 0.45 that animal is bird.
```

Bayesian methods would require a mathematical statement of the extent to which each component of the preconditions affects the postcondition.

A newer theory, called the Dempster-Shafer method, increases the complexity of this method to allow for the incomplete data on conditional probability inherent in most knowledge bases. It considers each probability to be an interval. It requires two confidence values to be present for each variable. They are termed support and plausibility. Support corresponds to our normal definition confidence. It is the extent to which given information shows the datum to be true. Plausibility is the extent to which given information fails to refute the datum.

In the case where equal, strong evidence indicates that a fact is true and that it is not true, support would be one, and plausibility would be zero. This allows for a new measure of uncertainty which is quite powerful.

This method is consistent with the axioms of probability and specifically accounts for uncertainty in the confidence data. The math involving a full and practical implementation of this system is beyond the scope of this report. Further explanation may be found in [14] and[9].

Most expert systems use a far more lenient scheme. They treat the confidence in any result or conclusion to be a function of the inputs and the confidence in the rules which yield these conclusions. Calculations are simple multiplication, addition, and subtraction. In the preceding example, the first rule would be sufficient. In the case of a conjunction or set of "and"s, the minimum of the confidence levels of the inputs would be used and multiplied with the confidence in the rule to achieve the confidence in the conclusion. For example, given the rule

```
        if A and B then
             C,
```

and that A has confidence 0.9, B has confidence 0.8, and the rule has confidence value 0.7, the confidence in C would be min (0.9,0.8) * 0.7 = 0.56. In the case of a disjunction or set of "or"s, the maximum of the inputs would be used.

The systems examined in section 2.2.2 are all based on this relaxed model of confidence calculation.

2.2 Theoretical Method

This Section discusses the theoretical issues involved in developing the common representation for expert systems called the evidence flow graph (EFG). Section one describes the evidence flow graph, what it is, and why is it needed. Section two describes specific expert systems, expert system shells, and methods. This examination reinforces the classification of different types of metaknowledge and shows what different potential systems a common representation must encompass. Section three discusses methods proposed for validation and verification (V&V) of expert systems using the EFG representation. V&V issues affect many of the design decisions for the EFG representation. Section four examines a few special issues that may affect a translation of an expert system into an EFG. These topics are discussed in this section because they do not arise in the translation of the two systems chosen for full translation.

2.2.1 The Evidence Flow Graph

An evidence flow graph is a form of data flow graph. It is a directed graph where each node represents a computational or logical process. In applying the graph to expert systems, each node is usually a rule. Other node types exist to perform special functions as shown in Section V.

Why use a graphical representation? A graph is an intuitive representation for a reasoning process [12]. One might also envision a complete programming environment where an expert system is represented graphically, and all development and testing is done with the aid of a visual representation of the rule base.

```
    Given the rules,

         if A and B then
              C,
    and
         if B and D then
              E,
```

a simple EFG representation for this rule base is given in figure 2.3, where each node would be activated when all its inputs have arrived. If a boolean value for the inputs to node A and node B were present, node C would be activated, and a value for C would then be output. #4.

However, suppose both nodes received all their inputs at the same time. If resources were limited, which one would we choose to execute first? Also, how are we to represent confidence in the rules' conclusions? What if a node needed only some of its input to reach a conclusion? It is apparent that a much richer representation is needed. This is described in section 2.2.1.1.

The EFG representation has currently been found to have two main uses. One is in the validation and verification (V&V) of expert systems. The other is in the parallelization of expert systems to improve performance.

Validation and verification is the process by which an expert system is analyzed for desired behavior. The faults that may be found are indications of errors in the system. However, a system may still function correctly when these faults exist given that there are redundant sections to carry out the work of the faulty portion. A number of specific conditions may be tested for. Both these conditions and the methods which uncover them are described in section 5 of this Section.

The second EFG use is in parallelization. Most expert systems contain portions which may execute simultaneously. If multiple processors are available, the expert system will run faster if these portions may be distributed. This topic will not, for the most part, be discussed any further than it has been already in section

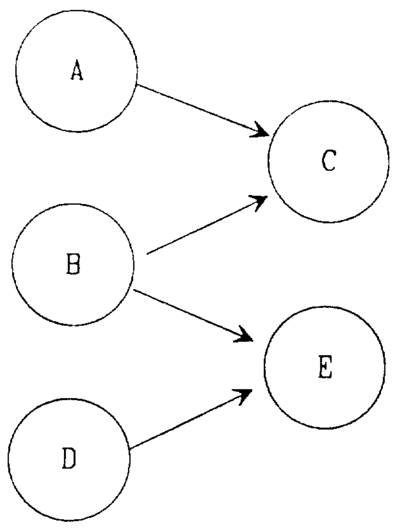


Figure 2.3 Simple EFG Representation

2.1.2.5. It is, however, a very promising area for further research.

There is an important set tradeoffs involved in defining a common expert system representation. The first is that of generality versus expressiveness. The more general and flexible the system, the more work that is required to express a given set of specific functions. Representations may be optimized for a given domain. Because of the intent of the design undertaken here, expressiveness is sacrificed to an extent. One design objective is to create a representation that allows simple systems to be expressed succinctly while allowing sufficient richness to express more complex systems if necessary.

A second tradeoff is that of clarity versus brevity. The popular programming language C allows for very brief, compact code that is unintelligible to the novice. It also allows the same code to be expressed in a longer and hopefully more readable format. Languages such as Ada do not allow compact yet unclear code. To an extent, the approach taken in this EFG research has been more in line with the philosophy of C.

A last consideration is that of fast execution versus human readability. Assembly language is very fast but nearly unreadable. Rule based system shells are very readable but also very slow. The solution to this problem is to follow the example of the modern programming language - provide a readable language interface to the user and compile the language into machine language. Here the readable language is the original rule base language and the target language is Ada which is then compiled into assembly language.

2.2.1.1 Node Information

The information in an evidence flow graph representation consists of a number of different conceptual partitions. This is illustrated in figure 2.4. The most basic is the physical portion. This section lists the nodes and the arcs that connect them. The next portion is the description of the rule that the node represents. This includes both the predicates or left hand side of the rule, and the conclusion or right hand side. Together with this portion may be grouped the variables and constants that the node needs, and the type of the node. The most abstract portion consists of descriptions of the procedures which control the execution of the nodes. It is this highest part which is the most complex.

There are four different fields in the highest conceptual portion of a node. Each field contains the name of a particular method for controlling one aspect of a node's execution. They are: token consuming, priming function, importance function, and confidence function.

Token consuming refers to the method by which information is used and renewed at a node's inputs. Two different methods have currently been identified. In some systems when a rule receives all the necessary inputs, it fires or reaches a conclusion, and the information that it received is discarded. In other systems, old information is retained. When new information arrives the rule may fire again using both the old and new

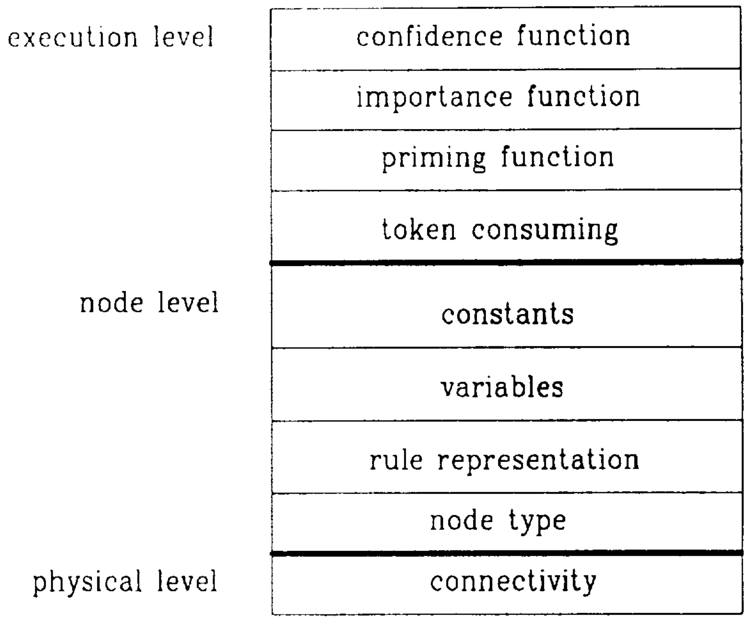


Figure 2.4 Node Information

information, instead of waiting for a completely new set of inputs. For example consider the rule:

```
if A>50 and B>30 then
    C
```

In a non-consuming system, if the rule receives A=72 and B=40, it concludes C. When the information A=60 arrives, it concludes C again. In a consuming system when the new A arrives it must wait for a new B value to arrive because the old value for B has been discarded.

The priming function refers to the condition under which a node is considered to have enough information to evaluate its expression. Two different methods have currently been identified. In the first type, a rule will be considered ready to fire if sufficient evidence is present to form a logical conclusion. In the second type, a rule must have all its inputs present to fire. For example, in the first type of system, the rule

```
if (A and B) or (C and D) then
    E
```

must have only A and B or C and D to fire, because both portions of the disjunction are not necessary to reach a conclusion. In the second type of system, all the inputs must be present for the rule to fire.

The importance function determines which primed node fires next. This is usually static in a forward chaining system. However, in a backward chaining system, a complex, dynamic scheme is necessary, combined with extensive graph augmentation. This will be described in section 2.2.3.2.

This function is very closely related to the control metaknowledge of the inference engine. It is not, however, identical. There is a certain amount of freedom that may exist in execution order. In this case, optimization for rule firing order may take place without changing the functionality of the system. Importance functions vary widely from system to system. Only the functions for the system examined in section 2.3 will be explained in detail.

The confidence function is the algorithm that calculates confidence in the conclusions that the expert system reaches. Several of these schemes are explained in section 2.1.3. Because most of these systems are simple, they are best implemented as functions of the EFG. However, in a system as complex as the causal net, the process must be achieved by creating a separate network to calculate confidence.

2.2.2 Expert Systems

2.2.2.1 Mycin

Although Mycin is a relatively complex system in total, it provided some general design examples which were adopted by many later systems. In this section and the following sections, the features to be examined are the confidence and control strategies. Many of Mycin's peculiarities and special cases have been omitted in this section. Only the general confidence and control strategies are examined.

In Mycin both rules and facts have an associated confidence value. The confidence in a new fact is a function of the confidence in the inputs and the rule that combines them. When facts are combined using an "and" function, the confidence in the conclusion is the product of the rule confidence and the minimum confidence of the inputs. When facts are combined using an "or" function, the confidence in the conclusion is the product of the maximum confidence of the inputs and the rule confidence. For example, consider two rules:

```
Rule #1                       Rule #2

if A and B then               if A or B then
   C                             G
```

If rule #1 has a confidence of 0.90, A has confidence 0.75 and B 0.80, then C will have confidence 0.75*0.90=0.675. If rule #2 has confidence of 0.90 then G will have confidence 0.80*0.90=0.72.

There is also one more complicated calculation that is used when the same fact is concluded by two different rules. When there is no explicit relation stated, an equation is used to pool the evidence. Given that S1 is the belief in conclusion 1, and S2 is the belief in conclusion 2 the equation is S_total=S1+S2*(1-S1).

For example, if conclusion A is reached from two sources, one with a confidence of 0.80 and one with a confidence of 0.90 the resulting confidence is 0.80+0.90*(1-0.80)=0.98.

Mycin is a backward chaining system. The goal with the highest confidence is the one evaluated first. Subsequent rules are evaluated in a depth first manner. The rule with the highest confidence that supports the current goal or subgoal is evaluated first. In this way, there is a train of thought in the questions the system asks. The information that it gathers is increasingly specific. If an hypothesis is refuted, it backs up and tries the closest related goal which has not been tried.

2.2.2.2 Insight 2+

Insight 2+ is a simple commercial expert system shell. It is designed for a wide variety of applications. It has no facility for pooling evidence. Only confidence given explicit relations may be calculated. Conjunctions or "and" functions take the minimum of the input confidences, and disjunctions or "or" functions take the maximum. In other respects it has the same confidence system as Mycin.

This package has a very simple backward chaining control strategy. The rule with the highest rule confidence that supports the current goal or sub- goal is evaluated first. In other respects it has the same control strategy as Mycin.

2.2.2.3 Exsys

The confidence system in EXSYS has been geared for flexibility. It has five confidence systems from which one may be selected. They are, for the most part, similar to those discussed above. In the first, facts may be true or false only; in the second, values in the range of zero to ten are allowed. In the third, values from zero to one hundred are allowed, and three choices are provided for combining identical conclusions that lack a specific relation. These are: average all results, multiply all results, and pooling given by the equation S3=1-((1-S1)*(1-S2)). The fourth option is an increment/decrement system where evidence levels are only raised or lowered by addition or subtraction. The last option is really the absence of a confidence system. The user may write his own at an arbitrary level of complexity.

This package has a straightforward control strategy. There are three options which control the extent to which the rule base is examined. Otherwise, it is essentially a forward chaining system in which the rules are evaluated in the order that they were entered in the rule base.

The first option is to examine all rules that have enough information to reach a conclusion. The second is to stop after one result is found. That is, to stop after a rule that has reached a conclusion that does not form the input to another rule. The last option is to fire only those rules which do not reach a conclusion that has been reached previously by another rule.

2.2.2.4 Ops 5

OPS5 is a rule-based (production-system based) programming language for intelligent systems which was developed at Carnegie-Mellon University. It is forward chaining and uses the very efficient RETE matching algorithm for finding applicable rules[15,16]. The design of the CLIPS programming language is based to a significant degree on that of OPS5.

2.2.2.5 Clips

Clips is an expert system shell developed by NASA and is available for use on government contracts and can be purchased commercially through COSMIC. Clips is a relatively simple system that also contains a number of potentially complex extensions. This makes it suitable for both the novice and serious developer.

Clips has no confidence strategy. Its control strategy is similar in philosophy to that of Ops 5. The most recent conclusions are those targeted for execution. It is a forward chaining system. Rules which refer to the most recently asserted fact are those which may be executed. Among those rules, the one executed first is the one which appears first in the rule base. Facts are not consumed once used by a rule, but remain to be used again should more recent rules be exhausted.

One added complexity is that Clips handles a Prolog style unification. Facts which are asserted may have not only a predicate and a value, but a whole series of parameters each of which may be the index for the others. For example, the fact

    (married bob jane)

may be matched with

    (married bob ?X),

or

    (?relation bob jane).

This complex syntactical structure introduces may problems for translation into a common expert system representation.

2.2.3 Special Considerations of the Evidence Flow Graph

2.2.3.1 Node Typing

The node may be one of a number of conceptual types. These are: rule, value, and procedure. The rule type is the most common. This node simply represents a rule. A value node contains a variable of arbitrary type. This type of node must have a higher importance function value than all rule nodes to ensure that nodes do not work with old or inconsistent data. This type of node is desirable when more than one node shares a given variable.

The last type of node is the procedure node. This is used only in very special cases. Normally, a library of named procedures is maintained. This library may contain user procedures in addition to a standard set. During conversion of the expert system into Ada code, these procedures are identified by name and added to the code for the EFG. For purposes of conceptual clarity, or as an aid to optimal parallelization, the procedure may be represented as a node.

A procedure node is desirable when the procedure it represents is large and complex and is accessed by a number of rule nodes. It does not function like a regular rule node. The information that a procedure node generates is only valuable for the node that sent the procedure node its input. Therefore, the output of a procedure node must only be sent to the node that provided its input. Normally, a node sends its results to all the other nodes that reference these results. A procedure node, however, must send its results only to the nodes that ask for them by sending the parameters the procedure needs.

2.2.3.2 Backward chaining

Converting a backward chaining expert system (BCES) to an EFG is a somewhat difficult task. This is due to the fact that an EFG is essentially a feedforward system, and a BCES is a feedbackward system. The conversion process involves extensive graph augmentation. This conversion works basically as follows: first, the rule base is converted as if it were a forward chaining system (see figure 2.5). Next, another graph is created identical to the first in all but one respect, the direction of the arcs is reversed. Finally, all the nodes in the second graph are connected to their counterparts in the first (see figure 2.6).

if A and B then
    C
if S and T then
    C
if A and D and E then
    F
if X and Y then
    A

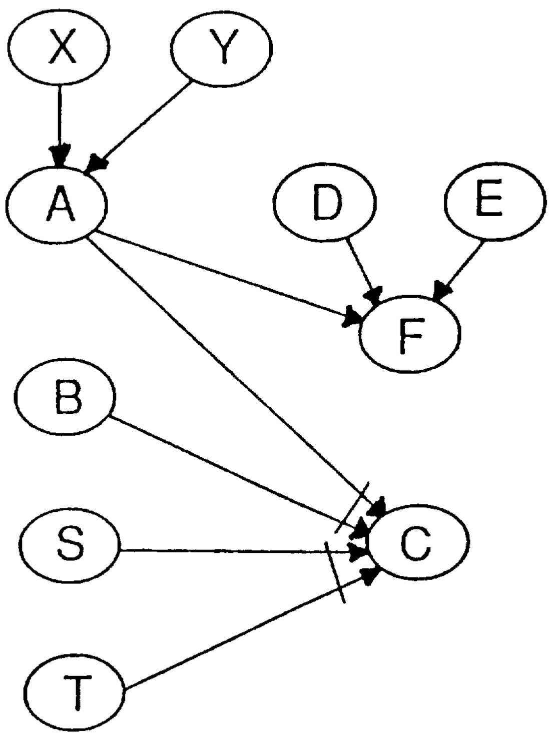


Figure 2.5 Forward Chaining System.

2.3 An Application of the Evidence Flow Graph

2.3.1 Causal Network

2.3.1.1 Explanation

An implementation of a causal tree in reality consists of two related trees. One explicitly controls the calculation of confidence, and the other controls the acquisition of evidence. While the first tree is the subject of the current research that deals with causal trees, the second is needed in any case where the acquisition of evidence is software controlled. The acquisition tree is in this case a simple rule base.

The confidence tree is a tree structured graph where the root contains the entire possible set of hypotheses. Each successive level going down the tree breaks this set into smaller and smaller groups. The leaf nodes contain individual hypotheses. Initially, the confidence tree contains no evidence. The designer supplies an initial guess as to the likelihood of each hypothesis being true. In addition, the designer must supply a matrix

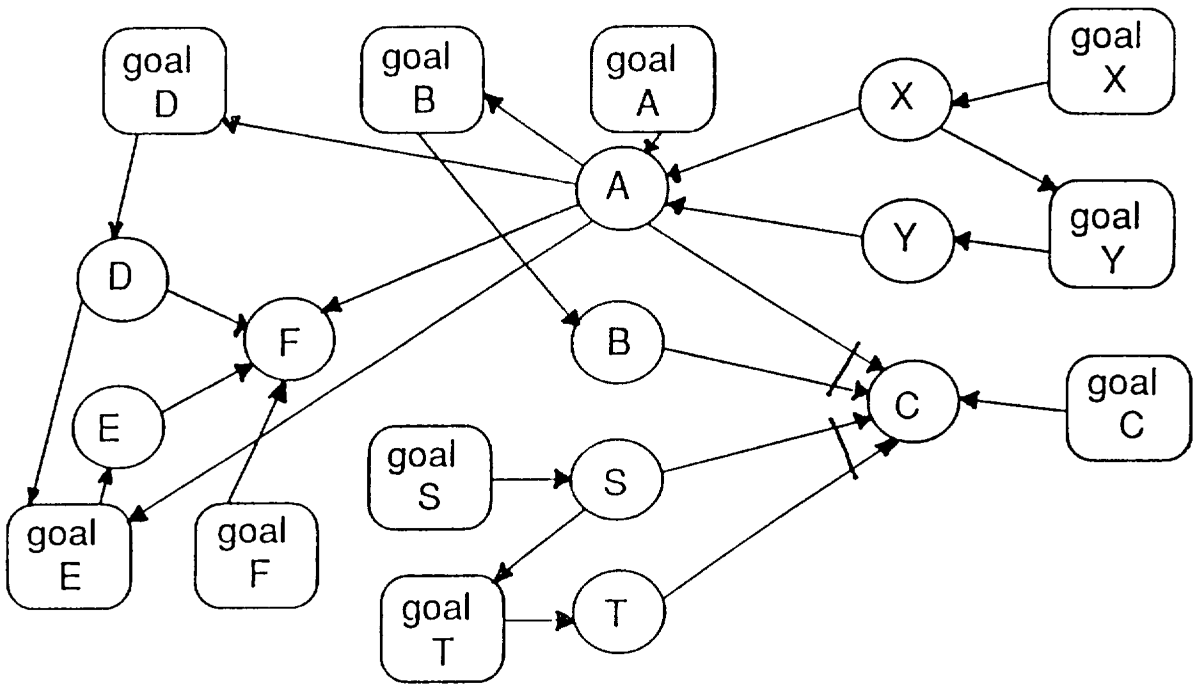


Figure 2.6 Backward Chaining System.

of probabilities for each link in the graph. This matrix contains the conditional probabilities of the child nodes given the parent's truth.

A preparatory stage occurs as follows. The initial guess is supplied to the root node. This information propagates down the tree in a breadth first fashion. Calculations occur at each node. When the terminal nodes are reached, they may send information back up the tree which propagates up to the root. At this point, the tree is ready to accept evidence.

When an hypothesis is found to be either true or false, this is entered in one of the node's variables. The new information must propagate up and then back down the tree for equilibrium to be reached. See section 2.3.1.3 for more information.

2.3.1.2 TASC system

The example causal tree that will be used is part of a system that evaluates the health of a navigational system in a future fighter aircraft. The navigation works by triangulating electronic signals from several satellites (called GPS) and by comparing this position against an inertial navigation system (INS). The causal tree has eight inputs, and five parent nodes (see figure 2.7), which are briefly explained below. An asterisk denotes an input node.

The expert system that controls that acquisition of evidence is shown in figure 2.8. The conclusions that may be reached are signals to acquire evidence. The evidence to acquire forms the input to the causal tree. The names for the conclusions correspond to the names of the terminal nodes of the causal tree. There are four system inputs to the evidence expert system. They are: equip health, jtids, busy, and health. Equip health

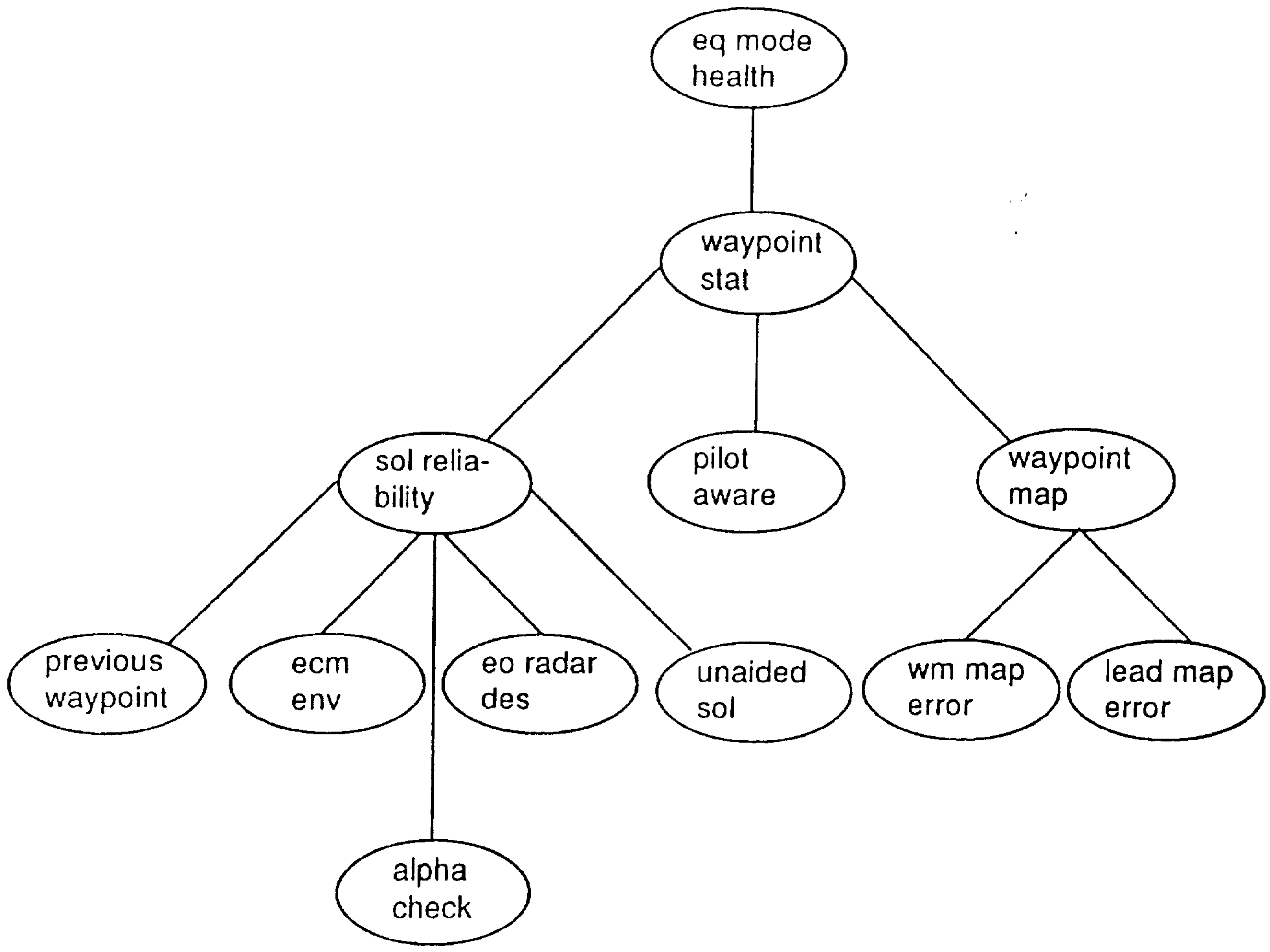


Figure 2.7 TASC System - Confidence Tree

Term	Explanation
equip health	health of current navigational system
waypoint stat	missed waypoint?
sol reliability	solution reliability
pilot aware *	was the pilot aware?
waypoint map	was map in error?
previous waypoint *	was last waypoint seen?
ecm env *	electronic counter measure
alpha check *	check with wingman
eo radar des *	electro-optical radar designation
unaided sol *	unaided solution (INS only)
wm map error *	wingman map error
lead map error *	lead pilot map error

signifies that the equipment health has been asked for. Jtids signifies that an electronic transmission of information has been completed from another aircraft. Busy signifies that the pilot is not busy and may be asked a question. This is a function of the pilot's workload. Health signifies that the health of the equipment is doubtful or good.

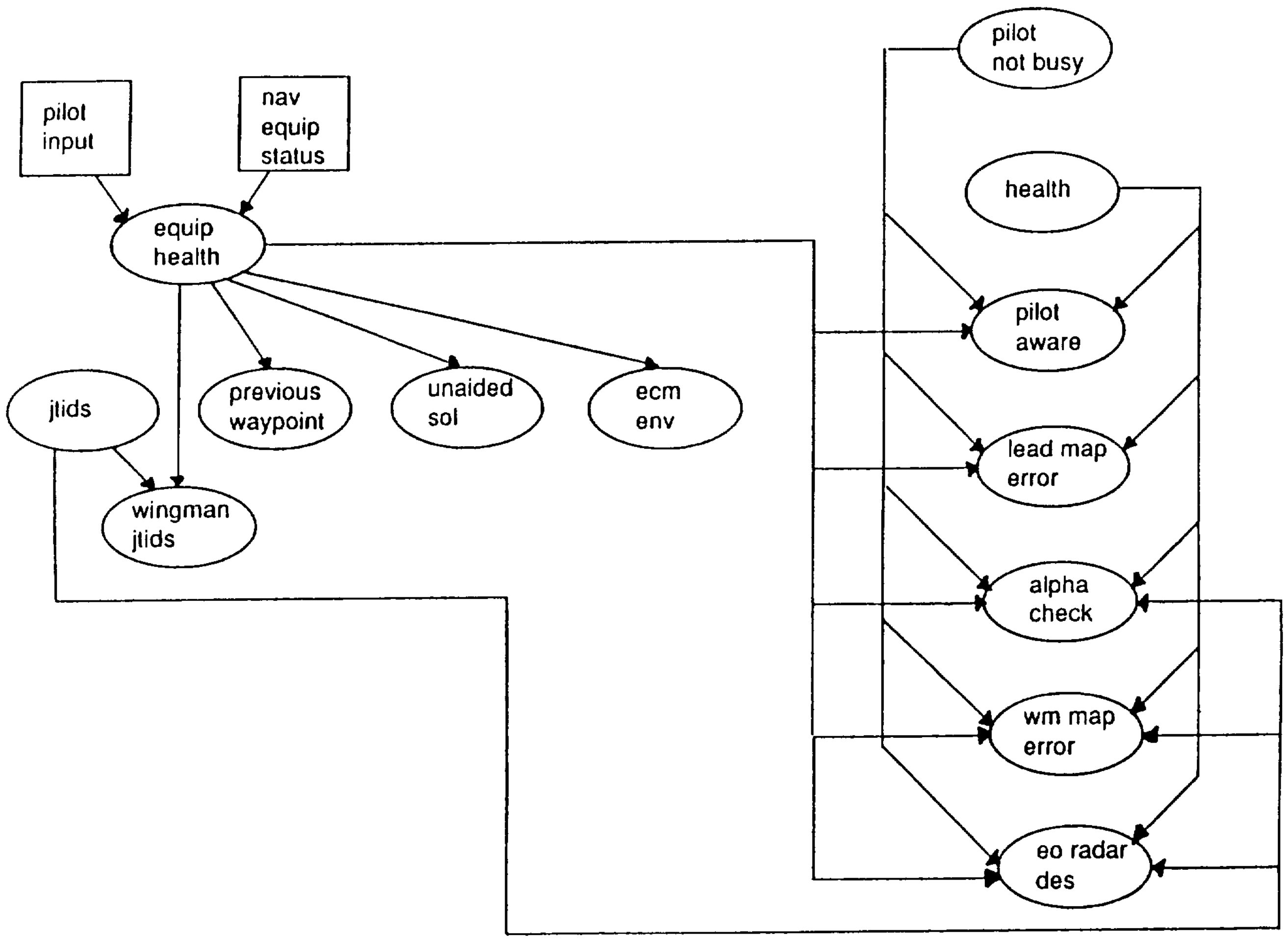


Figure 2.8 TASC System - Acquisition Network

While most of these nodes may have only two values (good/bad, yes/no), others have three including an intermediate value. These values are listed in the table below for each node in the tree.

2.3.1.3 Translation and Details

Each node must contain five variables: P, L (usually called lambda), Q, R and S[12]. These variables are all column vectors with a size equal to the number of values of the variable represented by the node. In addition, the node must contain a constant matrix with a size equal to the number of possible input values to the node by the number of its possible output values. R's and L's are sent up the tree, and Q's and S's are sent down the tree. P may be sent to an output node for printing or other use.

At a terminal node, R represents the state of its possible values. At first, this variable contains only the probability of truth of each of these values. As information is obtained, these values become locked at either completely true or completely false.

Node	Values
equip health	good, bad
waypoint stat	mode bad, pilot unaware, waypoint off
sol reliability	good, bad
previous waypoint	nominal, doubtful, missed
ecm env	jammers, clear
eo radar des	agree, disagree
unaided sol	agree, disagree
pilot aware	yes, no
waypoint map	agree, disagree
wm map error	yes, no
lead map error	yes, no
alpha check	agree, disagree

Q and R are the only variable inputs to the system. Changes in R's and L's are not affected by changes in Q's and S's. When new information is obtained and R changes as a result, L's change and are communicated upwards with the R's. This causes changes in the Q's and S's which are communicated downwards. Because R's and L's are not affected by Q's and S's, the cycle of changes is complete with one pass up the tree and one pass down.

A node may be considered primed when it receives any input.

The control strategy for the acquisition tree may be a simple in-order scheme. The causal tree's control strategy is inherent in its function. As soon as any node has new input it may fire without restriction - a simple in-order scheme applies here also. There is however one important consideration. The evidence found in P will not be valid unless both the upward and downward passes through the evidence tree are complete. If the acquisition tree relies on this knowledge, then it must wait for the passes to be complete before providing new input.

All nodes use these equations:

1) L = pi(Rj)

2) Si = Q * pi(Rj) where j<>i

3) P = LQ (normalized)

4) R = ML

5) Q = MtSi

All of the variables on the left hand side are vectors whose dimensions are equal to the number of values that the node may represent. For example, in the waypoint map node, they would all have a dimension of two. "pi" denotes the successive, component-wise multiplication of the particular variable. Normalized means that all the components of the resultant vector must sum to one. M is the conditional probability matrix for a node, and Mt is the transpose of that matrix.

Equation #1 means that for all the children of the given node, multiply their R's together.

Equation #2 means that to send an S to every child node of the given node, multiply vector Q by the pi of the R's of each child, except where the S is output to the node from which the given R is an input.

Equation #3 means to multiply vector L by vector Q and normalize the result. It is not necessary to normalize any other vector, but P must be normalized since it is output.

Equation #4 means to find the R going to the parent node, multiply the matrix of the link between the node and its parent by L.

Equation #5 means to find the Q going to a given child node of the current node, to multiply the inverse of the matrix on the link between the node and its child by the S that is sent to that child.

It is important to note that for a terminal node, there is only one R as input and so equation #1 reduces to L = R.

Figure 2.9 shows the flow of data between nodes in the confidence tree.

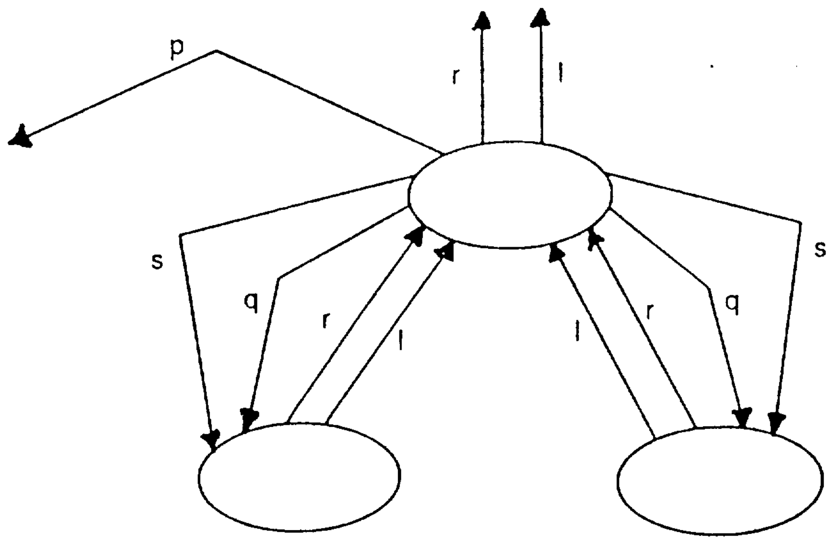


Figure 2.9 Confidence Tree - Node Diagram

3. EVIDENCE FLOW GRAPH TRANSLATORS

3.1 Design and Implementation of the Translators

Two translators were created. One converts an acquisition tree rule base to an EFG. The other converts an EFG into Ada AFOs. In addition, a hand translation of the confidence tree into an EFG and of the EFG into Ada AFOs was performed.

3.1.1 Rule to EFG Translator

This translator has two parts, a parser and a generator. The parser reads the file containing the rule base into an internal list representation. The list is passed to the generator. The generator first determines which rules are connected to each other and numbers all the connections. Rules which have inputs that are not the outputs from another rule are labeled as external inputs. Correspondingly, rules which have outputs that are not the inputs from another rule are labeled as external outputs. At this point, EFG nodes are generated and sent to a file. The generation of rule nodes is relatively straightforward. It essentially consists of listing inputs and outputs with their associated connectivity information, and reproducing the rule. Lastly, input/output nodes are generated. Input nodes simply receive input from a node called external which is generated later as an AFO. A single output node just prints its input.

3.1.2 EFG to Ada AFO Translator

This translator also has two parts, a parser and a generator. The parser reads the file containing the EFG into an internal list representation. The list is passed to the generator. For each EFG node the generator creates a file which will be an AFO. It creates a priming function which tests to see if messages are present on every input port. It also creates a transfer function which receives input, executes a rule, and forwards the output. The same format is generated for input nodes except that instead of executing a rule they merely route messages. An output AFO is created which prints the conclusions of the rule nodes. A frame file is also created which specifies to AF what AFOs exist and what the names of their respective priming and transfer functions are.

3.1.3 Execution of the AFOs

AF is executed with the frame file and AFOs from the EFG to Ada translator. It creates a compilation batch file which when executed links the AFOs with the AF execution environment. The result of this process is a single executable file.

3.2 Evidence Flow Graph Syntax

The syntactical representation of the nodes in an EFG is simply a concise listing of the fields illustrated in section 2.2.1.1. The representation is listed in Backus-Naur form. Keywords that appear literally in the syntax are shown in boldface. ::= means "is defined as". | means "or". {} enclose items which are to be repeated zero or more times. [] enclose optional items. Characters which are part of BNF syntax that appear literally in the EFG syntax are enclosed in single quotes. Comments may appear anywhere in the node when surrounded by curly braces.

```
node ::=  node integer string is
               input
               output
               statvar
               dynamvar
               constant
               transfer
               token
               priming
               truth
               importance
               confidence
          end node string;
```

```
input ::= [inputs {(inputnum) vardef;}]
output ::= [outputs {(outputnum) vardef;}]
statvar ::= [statvars {vardef;}]
dynamvar ::= [dynamvars {vardef;}]
constant ::= [constants {constdef;}]
transfer ::= transfer transfer_body; {transfer_body;} end transfer;
token ::= token token_name;
priming ::= priming logical;
truth ::= [truth expression;]
importance ::= [importance import_name import_type real;]
confidence ::= [confidence certainty conf_name;]


transfer_body ::= horn | statement
inputnum ::= inport:n/outport
outputnum ::= outport:n/inport
vardef ::= string:ddl
constdef ::= string : ddl = value
horn ::= if expression then {statement;} endif
function ::= string({parameter})
procedure ::= string({parameter})
token_name ::= consuming | non_consuming | string
import_name ::= constant_equal | string
import_type ::= static | dynamic
conf_name ::= null | mycin | string
logical ::= sublogical {logic_operator sublogical}
certainty ::= integer {real}
comment ::= '{'string'}'


inport ::= i# | string
outport ::= o# | string
n ::= n# | string
ddl ::= {ddl_element}
value ::= constant | record_constant | array_constant
expression ::= subexpression {logic_operator subexpression}
statement ::= assignment | procedure
parameter ::= value | variable
sublogical ::= sublogical | (sublogical) | inport
logic_operator ::= and | or | not
parameter ::= value | variable

ddl_element ::= C | I | F | array_def | string_def | list_def
constant ::= string | integer
record_constant ::= (record_constant_element {;record_constant_element})
array_constant ::= (value {value})
assignment ::= variable := rhs
subexpression ::= (subexpression) | comparison
port ::= inport | outport
variable ::= port | string

array_def ::= A(ddl_element)[integer {,integer}]
string_def ::= S[integer]
list_def ::= L(ddl_element)
comparison ::= variable comp_operator rhs
record_constant_element ::= value

rhs ::= variable | value | function
comp_operator ::= < | > | = | =< | =>
```

3.3 Causal Network Syntax

In section 3.3.1 the syntax definition for a node on the confidence tree is given. Section 3.3.2 provides a listing of the node definitions for the twelve nodes in the confidence tree for the TASC system which was depicted in figure 2.7. In section 3.3.3 the syntax definition for the acquisition tree is given, and section 3.3.4 provides the rules for the acquisition tree.

3.3.1 Confidence Tree Syntax

Besides the header information, there are four sections in a node. The states refer to the set of values which a node may have. The parent and children identify the parent and children of a node and the matrices give the conditional probability matrix for each node. The matrices consist of an integer followed by a list of two conditional probabilities for each state followed by the vectors of conditional probabilities of the child nodes given the parent's truth.

The following is the syntax definition for a node on the confidence tree.

```
node ::= (node integer string
           (states integer (string))
           (parent [n#])
           (children integer (n# ))
           (matrices integer ((((%)))))

# ::= integer
% ::= real
```

3.3.2 Confidence Tree Listing

The following is a listing of the node definitions for the twelve nodes in the confidence tree which was depicted in figure 2.7.

```
(node 1 eq_mode_health
     (states 2 good bad)
     (parent)
     (children 1 n2)
     (matrices 1     ((.1 .45 .45)
               (.9 .05 .05)) ))


(node 2 waypoint_stat
     (states 3 mode_bad pilot_unaware waypoint_off)
     (parent n1)
     (children n3 n4 n5)
     (matrices 4     ((.1 .45 .45)
               (.9 .05 .05))

               ((0 1)
               (.9 .1)
               (.9 .1))

               ((.9 .1)
               (0 1)
               (.9 .1))

               ((.9 .1)
               (.9 .1)
               (0 1)) ))


(node 3 sol_reliability
     (states 2 good bad)
     (parent n2)
     (children n6 n7 n8 n9 n10)
     (matrices 6     ((0 1)
               (.9 .1)
               (.1 .1))
```

```
                        ((.7 .2 .1)
                        (.3 .3 .4))

                        ((.1 .9)
                        (.8 .2))

                        ((.8 .2)
                        (.2 .8))

                        ((.8 .2)
                        (.2 .8))

                        ((.8 .2)
                        (.2 .8)) ))


(node 4 pilot_aware
      (states 2 yes no)
      (parent n2)
      (children)
      (matrices 1      ((.9 .1)
                        (0 1)
                        (.9 .1)) ))


(node 5 waypoint_map
      (states 2 agree disagree)
      (parent n2)
      (children n11 n12)
      (matrices 3      ((.9 .1)
                        (.9 .1)
                        (0 1))

                        ((.2 .8)
                        (.8 .2))

                        ((.2 .8)
                        (.8 .2)) ))


(node 6 previous_waypoint
      (states 3 nominal doubtful missed)
      (parent n3)
      (children)
      (matrices 1      ((.7 .2 .1)
                        (.3 .3 .4)) ))


(node 7 ecm_env
      (states 2 jammers clear)
      (parent n3)
      (children)
      (matrices 1      ((.1 .9)
                        (.8 .2)) ))


(node 8 alpha_check
      (states 2 agree disagree)
      (parent n3)
      (children)
      (matrices 1      ((.8 .2)
                        (.2 .8)) ))
```

```
(node 9 eo_radar_des
     (states 2 agree disagree)
     (parent n3)
     (children)
     (matrices 1     ((.8 .2)
               (.2 .8)) ))

(node 10 unaided_sol
     (states 2 agree disagree)
     (parent n3)
     (children)
     (matrices 1     ((.8 .2)
               (.2 .8)) ))

(node 11 wm_map_error
     (states 2 yes no)
     (parent n5)
     (children)
     (matrices 1     ((.2 .8)
               (.8 .2)) ))

(node 12 lead_map_error
     (states 2 yes no)
     (parent n5)
     (children)
     (matrices 1     ((.2 .8)
               (.8 .2)) ))
```

3.3.3 Acquisition Tree Syntax

The following section shows the syntax definitions for the acquisition tree.

```
node ::=   (rule integer string
           (input {input-vardef})
           (output {output-vardef})
           (horn))

input ::= inputnum | string
output ::= outputnum | string
vardef ::= string ddl
horn ::= if expression then statement

inputnum ::= i#/n#/o#
outputnum ::= o#/n#/i#
ddl ::= {ddl_element}
expression ::= subexpression {logic_operator subexpression}
statement ::= assignment | function

ddl_element ::= C | I | F | array_def | string_def | list_def
logic_operator ::= and | or | not
assignment ::= variable = rhs
subexpression ::= (subexpression) | comparison
function ::= string({parameter})

array_def ::= A(ddl_element)[integer {,integer}]
string_def ::= S[integer]
list_def ::= L(ddl_element)
variable ::= port | string
comparison ::= variable comp_operator rhs
rhs ::= variable | value
comp_operator ::= < | > | = | =< | =>
parameter ::= value | variable

port ::= inport | outport
```

```
inport ::= i#integer
outport ::= o#integer
```

3.3.4 Acquisition Tree Listing

This section lists the rules for the acquisition tree.

```
(rule 1 a_previous_waypoint
     (inputs eq_mode_health)
     (outputs previous_waypoint)
     (if eq_mode_health=true then
           get_previous_waypoint=true))

(rule 2 a_unaided_sol
     (inputs eq_mode_health)
     (outputs unaided_sol)
     (if eq_mode_health=true then
          get_unaided_sol=true))

(rule 3 a_ecm_env
     (inputs eq_mode_health)
     (outputs ecm_env)
     (if eq_mode_health=true then
          get_ecm_env=true))

(rule 4 a_pilot_aware
     (inputs eq_mode_health pilot_not_busy health)
     (outputs pilot_aware)
     (if eq_mode_health=true and pilot_not_busy=true and
          health=true then
          get_pilot_aware=true))

(rule 5 a_lead_map_error
     (inputs eq_mode_health pilot_not_busy health)
     (outputs lead_map_error)
     (if eq_mode_health=true and pilot_not_busy=true and
          health=true then
          get_lead_map_error=true))

(rule 6 a_alpha_check
     (inputs eq_mode_health pilot_not_busy health jtids)
     (outputs alpha_check)
     (if eq_mode_health=true and pilot_not_busy=true and
          health=true and jtids=true then
               get_alpha_check=true))

(rule 7 a_wm_map_error
     (inputs eq_mode_health pilot_not_busy health jtids)
     (outputs wm_map_error)
     (if eq_mode_health=true and pilot_not_busy=true and
          health=true and jtids=true then
               get_wm_map_error=true))

(rule 8 a_eo_radar_des
     (inputs eq_mode_health pilot_not_busy health jtids)
     (outputs eo_radar_des)
     (if eq_mode_health=true and pilot_not_busy=true and
          health=true and jtids=true then
          get_eo_radar_des=true))
```

3.4 TASC Causal Network Translation

3.4.1 Confidence Tree Pattern

Each group of three nodes is identical to any other except for the numbering of the nodes at the destination ports, additional ports for connections to multiple child nodes, and that the input ports for the root and terminal nodes are connections to external input/output nodes. See figures 3.1 and 3.2for node representations.

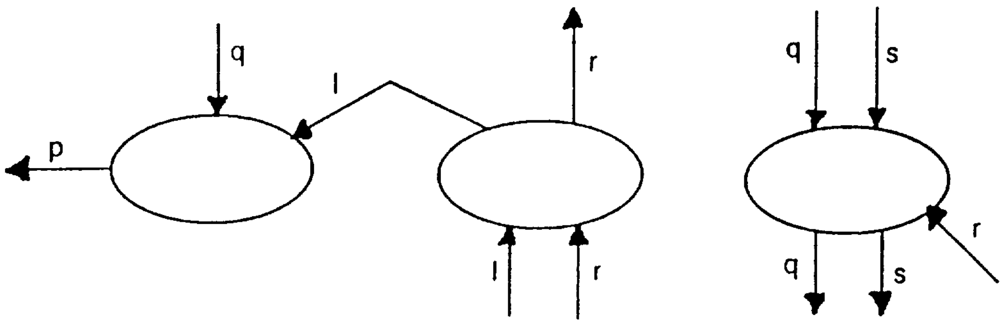


Figure 3.1 Confidence Tree - Node Subdivision

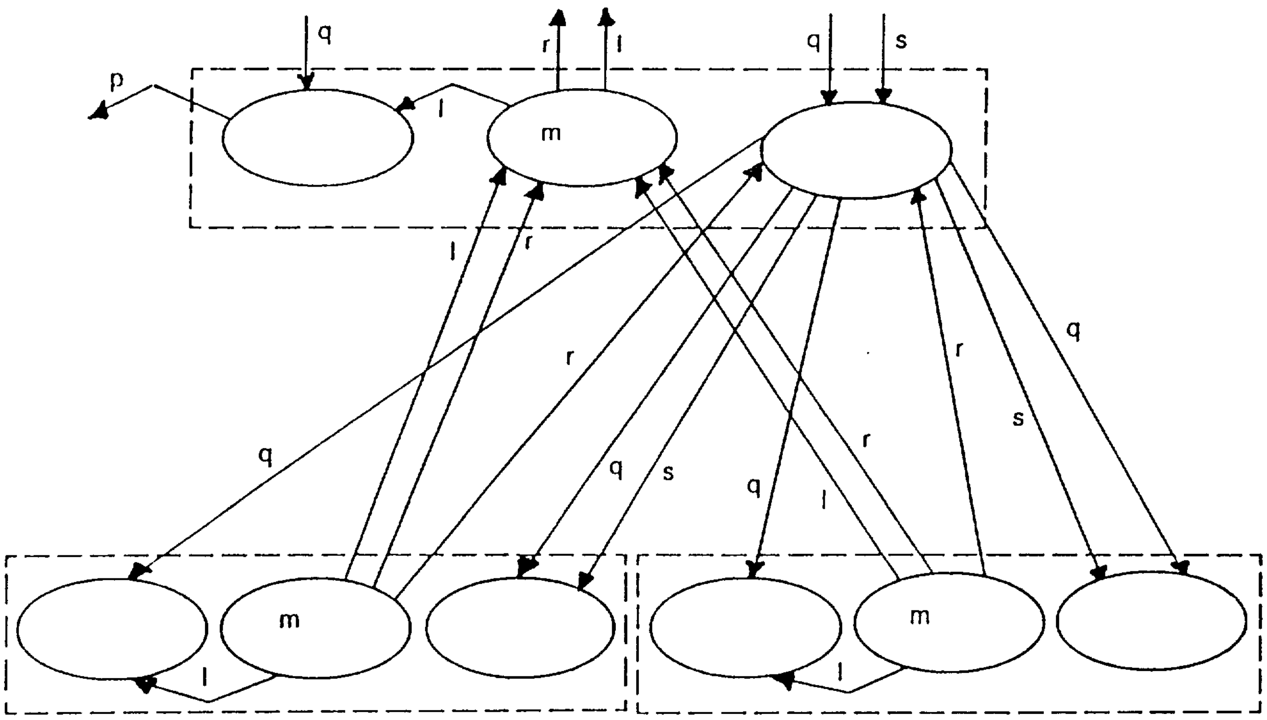


Figure 3.2 Confidence Tree - Parallelization of Nodes

The pattern for the triple is as follows:

```
node 4 left is
     inputs     (il:same_center/ol)  l:A(F)[2];
          (i2:above_right/ol)  q:A(F)[2];
     outputs    (ol:same_out/il)      p:A(F)[2];
     transfer
          get_msg (i1,l);
          get_msg (i2,q);
          causal_left (l,q,p);
          send_msg (p,o1);
     end transfer;

     token non_consuming;
     priming (i1 and i2);
end node left;


node 5 center is
     inputs    (i1:below_center/o3) r1:A(F)[2]; { Repeated for # children }
     outputs   (o1:same_left/i1)    l:A(F)[2];
          (o2:above_center/i1) r2:A(F)[2];
          (o3:above_right/i2)  r3:A(F)[2];
     constants m1:A(F)[2,2] = ((.9 .1) (0 1));
               m2:A(F)[2,2] = ((.8 .2) (.2 .8));

     dynamvars rlist:L(A(F)[2]);
               mlist:L(A(F)[2,2]);
     transfer
          get_msg (i1,r1); { These 3 lines are repeated for # children }
          makenull (rlist);
          append (rlist,r1);
          makenull (mlist);
          append (mlist,m1);
          append (mlist,m2);
          causal_center (rlist,l,r2,r3,mlist);
          send_msg (l,o1);
          send_msg (r2,o2);
          send_msg (r3,o3);
     end transfer;

     token non_consuming;
     priming all_ready (i1); { Repeated for # children }
end node center;


node 6 right is
     inputs    (i1:above_right/o2)  q1:A(F)[2];
          (i2:below_center/o3) r:A(F)[2]; { Repeated for # children }
     outputs   (o1:below_left/i2)   q2:A(F)[2];{ Repeated for # children }
          (o2:below_right/i1)  q3:A(F)[2];{ Repeated for # children }
     dynamvars s:A(F)[2];
               rlist:L(A(F)[2]);
               q1list:L(A(F)[2]);
               q2list:L(A(F)[2]);

     transfer
          get_msg (i1,q1);
          makenull (q1list);
          append (q1list,q1);
          get_msg (i2,r);  { Next 3 lines are repeated for # children }
          makenull (rlist);
          append (rlist,r);
          causal_right (q1,rlist,q1list,q2list);
          q2 := car(q1list); { Next 2 lines are repeated for # children }
          q3 := car(q2list);
          send_msg (q2,o1);  { Next 2 lines are repeated for # children }
          send_msg (q3,o2);
     end transfer;

     token non_consuming;
```

```
          priming all_ready (i1 and i2); { More if more children }
     end node right;
```

3.4.2 Transfer Functions for the Confidence Tree

```
     causal_left (l : in vector; q : in vector; p : out vector)

       p := normalize(matmul (l,q))


     causal_center   (rlist : in vector_list; l : out vector;
                     r : out vector; r2 : out vector; mlist : in matrix_list)

       l := pi (list_len(rlist),rlist)
       r := matmul (mlist(1),l)
       r2 := r


     causal_right    (q : in vector; rlist : in vector_list; ql : out vector_list;
               q2 : out vector_list)

       for i:=1 to list_len(rlist)
         s(i) := matmul (q,except_pi(list_len(rlist),rlist,i))
         ql(i) := matmul (transpose(mlist(i+1)),s(i))
```

3.4.3 Input/Output Nodes for the Causal Network EFG

In the case that a node in the confidence tree is a root node or a leaf node, the EFG representation differs slightly from the pattern shown in section 2.4.4.1. A root node triple is as follows.

3.4.3.1 Root node triple

```
     node 1 left is
          inputs     (i1:same_center/o1)  l:A(F)[2];
               (i2:root/o1)     q:A(F)[2];
          outputs    (o1:same_out/i1)     p:A(F)[2];

          transfer
               get_msg (i1,l);
               get_msg (i2,q);
               causal_left (l,q,p);
               send_msg (p,o1);
          end transfer;

          token non_consuming;
          priming (i1 and i2);
     end node left;

     node 2 center is
          inputs     (i1:below_center/o3) r1:A(F)[2]; { Repeated for # children }
          outputs    (o1:same_left/i1)    l:A(F)[2];

          constants m1:A(F)[2,2] = ((.9 .1) (0 1));
                    m2:A(F)[2,2] = ((.8 .2) (.2 .8));

          dynamvars rlist:L(A(F)[2]);
                    mlist:L(A(F)[2,2]);
          transfer
               get_msg (i1,r1); { These 3 lines are repeated for # children }
               makenull (rlist);
               append (rlist,r1);
               makenull (mlist);
          append (mlist,m1);
          append (mlist,m2);
          causal_center (rlist,l,r2,r3,mlist);
          send_msg (l,o1);
     end transfer;

     token non_consuming;
     priming all_ready (i1); { Repeated for # children }
end node center;


node 3 right is
     inputs     (i1:root/o2)    q1:A(F)[2];
          (i2:below_center/o3) r:A(F)[2]; { Repeated for # children }
     outputs    (o1:below_left/i2)    q2:A(F)[2];{ Repeated for # children }
          (o2:below_right/i1)  q3:A(F)[2];{ Repeated for # children }
     dynamvars  s:A(F)[2];
                rlist:L(A(F)[2]);
                q1list:L(A(F)[2]);
                q2list:L(A(F)[2]);

     transfer
          get_msg (i1,q1);
          makenull (q1list);
          append (q1list,q1);
          get_msg (i2,r);  { Next 3 lines are repeated for # children }
          makenull (rlist);
          append (rlist,r);
          causal_right (q1,rlist,q1list,q2list);
          q2 := car(q1list); { Next 2 lines are repeated for # children }
          q3 := car(q2list);
          send_msg (q2,o1);  { Next 2 lines are repeated for # children }
          send_msg (q3,o2);
     end transfer;

     token non_consuming;
     priming all_ready (i1 and i2); { More if more children }
end node right;
```

3.4.3.2 Terminal node triple

```
node 22 left is
     inputs     (i1:same_center/o1)  l:A(F)[2];
          (i2:above_right/o1)  q:A(F)[2];
     outputs    (o1:same_out/i1)     p:A(F)[2];

     transfer
          get_msg (i1,l);
          get_msg (i2,q);
          causal_left (l,q,p);
          send_msg (p,o1);
     end transfer;

     token non_consuming;
     priming (i1 and i2);
end node left;


node 23 center is
     inputs     (i1:terminal/o3)          r1:A(F)[2];
     outputs    (o1:same_left/i1)    l:A(F)[2];
          (o2:above_center/i1) r2:A(F)[2];
          (o3:above_right/i2)  r3:A(F)[2];
          constants  m1:A(F)[2,2] = ((.9 .1) (0 1));
                     m2:A(F)[2,2] = ((.8 .2) (.2 .8));

          dynamvars  rlist:L(A(F)[2]);
                     mlist:L(A(F)[2,2]);
          transfer
               get_msg (i1,r1);
               makenull (rlist);
               append (rlist,r1);
               makenull (mlist);
               append (mlist,m1);
               append (mlist,m2);
               causal_center (rlist,1,r2,r3,mlist);
               send_msg (r2,o2);
               send_msg (r3,o3);
          end transfer;

          token non_consuming;
          priming all_ready (i1);
     end node center;


     node 24 right is
          inputs     (i1:above_right/o2)  q1:A(F)[2];
               (i2:terminal_center/o3)    r:A(F)[2];

          dynamvars  s:A(F)[2];
                     rlist:L(A(F)[2]);
                     q1list:L(A(F)[2]);
                     q2list:L(A(F)[2]);

          transfer
               get_msg (i1,q1);
               makenull (q1list);
               append (q1list,q1);
               get_msg (i2,r);
               makenull (rlist);
               append (rlist,r);
               causal_right (q1,rlist,q1list,q2list);
          end transfer;

          token non_consuming;
          priming all_ready (i1 and i2);
     end node right;
```

3.4.3.3 Root input node

```
     node 1 root is
          outputs    (o1:node_left/i2)    q1:A(F)[2];
               (o2:node_right/i1)   q2:A(F)[2];
          transfer
               write_string ("variable 1 : ");
               read_real (q1(1));
               write_string ("variable 2 : ");
               read_real (q1(2));
               set_equal (q2,q1);
               send_msg (q1,o1);
               send_msg (q2,o2);
          end transfer;

          token non_consuming;
          priming ();
     end node root;
```

3.4.3.4 Terminal input node

```
node 50 terminal is
     outputs   (o1:node_center/i1)  r1:A(F)[2];
          (o2:node_right/i3)   r2:A(F)[2];
     transfer
          write_string ("variable 1 : ");
          read_real (r1(1));
          write_string ("variable 2 : ");
          read_real (r1(2));
          set_equal (r2,r1);
          send_msg (r1,o1);
          send_msg (r2,o2);
     end transfer;

     token non_consuming;
     priming ();
end node terminal;
```

3.4.3.5 Output node

```
node 60 output1 is
     inputs    (i1:node_left/o1)    p:A(F)[2];

     transfer
          get_msg (i1,p);
          write_real (p(1));
          write_real (p(2));
     end transfer;

     token non_consuming;
     priming (i1);
end node output1;
```

3.4.4 Acquisition Network EFG

```
node 1 a_previous_waypoint is
     inputs  (i1:eq_mode_health/o1)     eq_mode_health:B;
     outputs (o1:system_output/i1)  out:I;

transfer
     get_msg (i1,eq_mode_health);
     if eq_mode_health=true then
          send_msg (1,o1);
     end if;
end transfer;

priming (i1);
importance static (1);
token consuming;
end node a_previous_waypoint;


node 2 a_unaided_sol is
     inputs  (i1:eq_mode_health/o2)     eq_mode_health:B;
     outputs (o1:system_output/i1)  out:I;

transfer
     get_msg (i1,eq_mode_health);
     if eq_mode_health=true then
          send_msg (2,o1);
     end if;
end transfer;

priming (i1);
importance static (2);
token consuming;
end node a_unaided_sol;


node 3 a_ecm_env is
     inputs  (i1:eq_mode_health/o3)     eq_mode_health:B;
     outputs (o1:system_output/i1)  out:I;

transfer
     get_msg (i1,eq_mode_health);
     if eq_mode_health=true then
          send_msg (3,o1);
     end if;
end transfer;

priming (i1);
importance static (3);
token consuming;
end node a_ecm_env;


node 4 a_pilot_aware is
     inputs  (i1:eq_mode_health/o4)     eq_mode_health:B;
             (i2:pilot_not_busy/o1)     pilot_not_busy:B;
             (i3:health/o1)     health:B;
     outputs (o1:system_output/i1)  out:I;

transfer
     get_msg (i1,eq_mode_health);
     get_msg (i2,pilot_not_busy);
     get_msg (i3,health);
     if eq_mode_health=true and
        pilot_not_busy=true and
        health=true then
          send_msg (4,o1);
     end if;
end transfer;

priming (i1 and i2 and i3);
importance static (4);
token consuming;
end node a_pilot_aware;


node 5 a_lead_map_error is
     inputs  (i1:eq_mode_health/o5)     eq_mode_health:B;
             (i2:pilot_not_busy/o2)     pilot_not_busy:B;
             (i3:health/o2)     health:B;
     outputs (o1:system_output/i1)  out:I;

transfer
     get_msg (i1,eq_mode_health);
     get_msg (i2,pilot_not_busy);
     get_msg (i3,health);
     if eq_mode_health=true and
        pilot_not_busy=true and
        health=true then
          send_msg (5,o1);
     end if;
end transfer;
```

```
priming (i1 and i2 and i3);
importance static (5);
token consuming;
end node a_lead_map_error;


node 6 a_alpha_check is
     inputs  (i1:eq_mode_health/o6)      eq_mode_health:B;
             (i2:pilot_not_busy/o3)      pilot_not_busy:B;
             (i3:health/o3)     health:B;
             (i4:jtids/o1)     jtids:B;
     outputs (o1:system_output/i1)  out:I;

transfer
     get_msg (i1,eq_mode_health);
     get_msg (i2,pilot_not_busy);
     get_msg (i3,health);
     get_msg (i4,jtids);
     if eq_mode_health=true and
        pilot_not_busy=true and
        health=true and
        jtids=true then
          send_msg (6,o1);
     end if;
end transfer;

priming (i1 and i2 and i3 and i4);
importance static (6);
token consuming;
end node a_alpha_check;


node 7 a_wm_map_error is
     inputs  (i1:eq_mode_health/o7)      eq_mode_health:B;
             (i2:pilot_not_busy/o4)      pilot_not_busy:B;
             (i3:health/o4)     health:B;
             (i4:jtids/o2)     jtids:B;
     outputs (o1:system_output/i1)  out:I;

transfer
     get_msg (i1,eq_mode_health);
     get_msg (i2,pilot_not_busy);
     get_msg (i3,health);
     get_msg (i4,jtids);
     if eq_mode_health=true and
        pilot_not_busy=true and
        health=true and
        jtids=true then
          send_msg (7,o1);
     end if;
end transfer;


priming (i1 and i2 and i3 and i4);
importance static (7);
token consuming;
end node a_wm_map_error;


node 8 a_eo_radar_des is
     inputs  (i1:eq_mode_health/o8)      eq_mode_health:B;
             (i2:pilot_not_busy/o5)      pilot_not_busy:B;
             (i3:health/o5)     health:B;
             (i4:jtids/o3)     jtids:B;
     outputs (o1:system_output/i1)  out:I;
```

```
transfer
     get_msg (i1,eq_mode_health);
     get_msg (i2,pilot_not_busy);
     get_msg (i3,health);
     get_msg (i4,jtids);
     if eq_mode_health=true and
        pilot_not_busy=true and
        health=true and
        jtids=true then
          send_msg (8,o1);
     end if;
end transfer;

priming (i1 and i2 and i3 and i4);
importance static (8);
token consuming;
end node a_eo_radar_des;


node 9 eq_mode_health is
     inputs  (i1:external/std_port)     iv1:B;
     outputs (o1:a_previous_waypoint/i1)     ov1:B;
             (o2:a_unaided_sol/i1)     ov2:B;
             (o3:a_ecm_env/i1)     ov3:B;
             (o4:a_pilot_aware/i1)     ov4:B;
             (o5:a_lead_map_error/i1)     ov5:B;
             (o6:a_alpha_check/i1)     ov6:B;
             (o7:a_wm_map_error/i1)     ov7:B;
             (o8:a_eo_radar_des/i1)     ov8:B;
transfer
     get_msg (i1,iv1);
     send_msg (iv1,ov1);
     send_msg (iv1,ov2);
     send_msg (iv1,ov3);
     send_msg (iv1,ov4);
     send_msg (iv1,ov5);
     send_msg (iv1,ov6);
     send_msg (iv1,ov7);
     send_msg (iv1,ov8);
end transfer;

priming (i1);
importance static (9);
token consuming;
end node eq_mode_health;


node 13 pilot_not_busy is
     inputs  (i1:external/std_port)     iv1:B;
     outputs (o1:a_pilot_aware/i2)     ov1:B;
             (o2:a_lead_map_error/i2)     ov2:B;
             (o3:a_alpha_check/i2)     ov3:B;
             (o4:a_wm_map_error/i2)     ov4:B;
             (o5:a_eo_radar_des/i2)     ov5:B;
transfer
     get_msg (i1,iv1);
     send_msg (iv1,ov1);
     send_msg (iv1,ov2);
     send_msg (iv1,ov3);
     send_msg (iv1,ov4);
     send_msg (iv1,ov5);
end transfer;

priming (i1);
importance static (13);
```

```
token consuming;
end node pilot_not_busy;


node 14 health is
     inputs  (i1:external/std_port)     iv1:B;
     outputs (o1:a_pilot_aware/i3)     ov1:B;
             (o2:a_lead_map_error/i3)     ov2:B;
             (o3:a_alpha_check/i3)     ov3:B;
             (o4:a_wm_map_error/i3)     ov4:B;
             (o5:a_eo_radar_des/i3)     ov5:B;
transfer
     get_msg (i1,iv1);
     send_msg (iv1,ov1);
     send_msg (iv1,ov2);
     send_msg (iv1,ov3);
     send_msg (iv1,ov4);
     send_msg (iv1,ov5);
end transfer;

priming (i1);
importance static (14);
token consuming;
end node health;


node 17 jtids is
     inputs  (i1:external/std_port)     iv1:B;
     outputs (o1:a_alpha_check/i4)     ov1:B;
             (o2:a_wm_map_error/i4)     ov2:B;
             (o3:a_eo_radar_des/i4)     ov3:B;
transfer
     get_msg (i1,iv1);
     send_msg (iv1,ov1);
     send_msg (iv1,ov2);
     send_msg (iv1,ov3);
end transfer;


priming (i1);
importance static (17);
token consuming;
end node jtids;


node 20 system_output is
     inputs  (i1:a_previous_waypoint)     iv1:I;
             (i1:a_unaided_sol)     iv1:I;
             (i1:a_ecm_env)     iv1:I;
             (i1:a_pilot_aware)     iv1:I;
             (i1:a_lead_map_error)     iv1:I;
             (i1:a_alpha_check)     iv1:I;
             (i1:a_wm_map_error)     iv1:I;
             (i1:a_eo_radar_des)     iv1:I;

transfer
     get_msg (i1,iv1);
     put_line ("ask_question_#");
     put (iv1);
     new_line (2);
end transfer;

priming (i1);
importance static (20);
token consuming;
```

```
end node system_output..
```

3.4.5 Acquisition Network AFOs

```
separate (af_start)

package body a_alpha_check is

  p1,  p2,  p3,  p4 : integer;
  s1 : string (1..16) := "a_alpha_check/i1";
  s2 : string (1..16) := "a_alpha_check/i2";
  s3 : string (1..16) := "a_alpha_check/i3";
  s4 : string (1..16) := "a_alpha_check/i4";
  obj : string (1..3) := "obj";
  destp1 : integer;
  dests1 : string (1..16) := "system_output/i1";

function prim6 (afo : af_struct) return boolean is

begin
  p1 := port_num(s1);
  p2 := port_num(s2);
  p3 := port_num(s3);
  p4 := port_num(s4);
  return msg_chk (p1) and
    msg_chk (p2) and
    msg_chk (p3) and
    msg_chk (p4);
end prim6;

procedure trans6 is

  eq_mode_health : boolean;
  pilot_not_busy : boolean;
  health : boolean;
  jtids : boolean;
  outv : short_integer;

  eq_mode_health_ad : address := eq_mode_health'address;
  pilot_not_busy_ad : address := pilot_not_busy'address;
  health_ad : address := health'address;
  jtids_ad : address := jtids'address;
  outv_ad : address := outv'address;

begin
  p1 := port_num(s1);
  p2 := port_num(s2);
  p3 := port_num(s3);
  p4 := port_num(s4);
  get_obj (p1,obj,B,eq_mode_health_ad);
  get_obj (p2,obj,B,pilot_not_busy_ad);
  get_obj (p3,obj,B,health_ad);
  get_obj (p4,obj,B,jtids_ad);
  if eq_mode_health = true and
    pilot_not_busy = true and
    health = true and
    jtids = true then
    destp1 := port_num(dests1);
    outv := 6;
    snd_obj (destp1,obj,SI,outv_ad,0,0.0);
  end if;
  ret_afo;
end trans6;
end a_alpha_check;


separate (af_start)

package body a_ecm_env is

  pl : integer;
  sl : string (1..12) := "a_ecm_env/il";
  obj : string (1..3) := "obj";
  destpl : integer;
  destsl : string (1..16) := "system_output/il";

function prim3 (afo : af_struct) return boolean is

begin
  pl := port_num(sl);
  return msg_chk (pl);
end prim3;

procedure trans3 is

  eq_mode_health : boolean;
  outv : short_integer;

  eq_mode_health_ad : address := eq_mode_health'address;
  outv_ad : address := outv'address;

begin
  pl := port_num(sl);
  get_obj (pl,obj,B,eq_mode_health_ad);
  if eq_mode_health = true then
    destpl := port_num(destsl);
    outv := 3;
    snd_obj (destpl,obj,SI,outv_ad,0,0.0);
  end if;
  ret_afo;
end trans3;

end a_ecm_env;


separate (af_start)

package body a_eo_radar_des is

  pl,  p2,  p3,  p4 : integer;
  sl : string (1..17) := "a_eo_radar_des/il";
  s2 : string (1..17) := "a_eo_radar_des/i2";
  s3 : string (1..17) := "a_eo_radar_des/i3";
  s4 : string (1..17) := "a_eo_radar_des/i4";
  obj : string (1..3) := "obj";
  destpl : integer;
  destsl : string (1..16) := "system_output/il";

function prim8 (afo : af_struct) return boolean is

begin
  pl := port_num(sl);
  p2 := port_num(s2);
  p3 := port_num(s3);
  p4 := port_num(s4);
  return msg_chk (pl) and
    msg_chk (p2) and
    msg_chk (p3) and
    msg_chk (p4);
end prim8;

procedure trans8 is

  eq_mode_health : boolean;
  pilot_not_busy : boolean;
  health : boolean;
  jtids : boolean;
  outv : short_integer;

  eq_mode_health_ad : address := eq_mode_health'address;
  pilot_not_busy_ad : address := pilot_not_busy'address;
  health_ad : address := health'address;
  jtids_ad : address := jtids'address;
  outv_ad : address := outv'address;

begin
  p1 := port_num(s1);
  p2 := port_num(s2);
  p3 := port_num(s3);
  p4 := port_num(s4);
  get_obj (p1,obj,B,eq_mode_health_ad);
  get_obj (p2,obj,B,pilot_not_busy_ad);
  get_obj (p3,obj,B,health_ad);
  get_obj (p4,obj,B,jtids_ad);
  if eq_mode_health = true and
    pilot_not_busy = true and
    health = true and
    jtids = true then
    destp1 := port_num(dests1);
    outv := 8;
    snd_obj (destp1,obj,SI,outv_ad,0,0.0);
  end if;
  ret_afo;
end trans8;

end a_eo_radar_des;


separate (af_start)

package body a_lead_map_error is

  p1,  p2,  p3 : integer;
  s1 : string (1..19) := "a_lead_map_error/i1";
  s2 : string (1..19) := "a_lead_map_error/i2";
  s3 : string (1..19) := "a_lead_map_error/i3";
  obj : string (1..3) := "obj";
  destp1 : integer;
  dests1 : string (1..16) := "system_output/i1";

function prim5 (afo : af_struct) return boolean is

begin
  p1 := port_num(s1);
  p2 := port_num(s2);
  p3 := port_num(s3);
  return msg_chk (p1) and
    msg_chk (p2) and
    msg_chk (p3);
end prim5;

procedure trans5 is
```

```
  eq_mode_health : boolean;
  pilot_not_busy : boolean;
  health : boolean;
  outv : short_integer;

  eq_mode_health_ad : address := eq_mode_health'address;
  pilot_not_busy_ad : address := pilot_not_busy'address;
  health_ad : address := health'address;
  outv_ad : address := outv'address;

begin
  p1 := port_num(s1);
  p2 := port_num(s2);
  p3 := port_num(s3);
  get_obj (p1,obj,B,eq_mode_health_ad);
  get_obj (p2,obj,B,pilot_not_busy_ad);
  get_obj (p3,obj,B,health_ad);
  if eq_mode_health = true and
    pilot_not_busy = true and
    health = true then
    destp1 := port_num(dests1);
    outv := 5;
    snd_obj (destp1,obj,SI,outv_ad,0,0.0);
  end if;
  ret_afo;
end trans5;

end a_lead_map_error;


separate (af_start)

package body a_pilot_aware is

  p1,  p2,  p3 : integer;
  s1 : string (1..16) := "a_pilot_aware/i1";
  s2 : string (1..16) := "a_pilot_aware/i2";
  s3 : string (1..16) := "a_pilot_aware/i3";
  obj : string (1..3) := "obj";
  destp1 : integer;
  dests1 : string (1..16) := "system_output/i1";

function prim4 (afo : af_struct) return boolean is

begin
  p1 := port_num(s1);
  p2 := port_num(s2);
  p3 := port_num(s3);
  return msg_chk (p1) and
    msg_chk (p2) and
    msg_chk (p3);
end prim4;

procedure trans4 is

  eq_mode_health : boolean;
  pilot_not_busy : boolean;
  health : boolean;
  outv : short_integer;

  eq_mode_health_ad : address := eq_mode_health'address;
  pilot_not_busy_ad : address := pilot_not_busy'address;
  health_ad : address := health'address;
  outv_ad : address := outv'address;
```

```
begin
  pl := port_num(sl);
  p2 := port_num(s2);
  p3 := port_num(s3);
  get_obj (pl,obj,B,eq_mode_health_ad);
  get_obj (p2,obj,B,pilot_not_busy_ad);
  get_obj (p3,obj,B,health_ad);
  if eq_mode_health = true and
    pilot_not_busy = true and
    health = true then
    destpl := port_num(destsl);
    outv := 4;
    snd_obj (destpl,obj,SI,outv_ad,0,0.0);
  end if;
  ret_afo;
end trans4;

end a_pilot_aware;


separate (af_start)

package body a_previous_waypoint is

  pl : integer;
  sl : string (1..22) := "a_previous_waypoint/il";
  obj : string (1..3) := "obj";
  destpl : integer;
  destsl : string (1..16) := "system_output/il";

function priml (afo : af_struct) return boolean is

begin
  pl := port_num(sl);
  return msg_chk (pl);
end priml;

procedure transl is

  eq_mode_health : boolean;
  outv : short_integer;

  eq_mode_health_ad : address := eq_mode_health'address;
  outv_ad : address := outv'address;

begin
  pl := port_num(sl);
  get_obj (pl,obj,B,eq_mode_health_ad);
  if eq_mode_health = true then
    destpl := port_num(destsl);
    outv := 1;
    snd_obj (destpl,obj,SI,outv_ad,0,0.0);
  end if;
  ret_afo;
end transl;

end a_previous_waypoint;


separate (af_start)

package body a_unaided_sol is

  pl : integer;
  sl : string (1..16) := "a_unaided_sol/il";
  obj : string (1..3) := "obj";
  destp1 : integer;
  dests1 : string (1..16) := "system_output/i1";

function prim2 (afo : af_struct) return boolean is

begin
  p1 := port_num(s1);
  return msg_chk (p1);
end prim2;

procedure trans2 is

  eq_mode_health : boolean;
  outv : short_integer;

  eq_mode_health_ad : address := eq_mode_health'address;
  outv_ad : address := outv'address;

begin
  p1 := port_num(s1);
  get_obj (p1,obj,B,eq_mode_health_ad);
  if eq_mode_health = true then
    destp1 := port_num(dests1);
    outv := 2;
    snd_obj (destp1,obj,SI,outv_ad,0,0.0);
  end if;
  ret_afo;
end trans2;

end a_unaided_sol;


separate (af_start)

package body a_wm_map_error is

  p1,  p2,  p3,  p4 : integer;
  s1 : string (1..17) := "a_wm_map_error/i1";
  s2 : string (1..17) := "a_wm_map_error/i2";
  s3 : string (1..17) := "a_wm_map_error/i3";
  s4 : string (1..17) := "a_wm_map_error/i4";
  obj : string (1..3) := "obj";
  destp1 : integer;
  dests1 : string (1..16) := "system_output/i1";

function prim7 (afo : af_struct) return boolean is

begin
  p1 := port_num(s1);
  p2 := port_num(s2);
  p3 := port_num(s3);
  p4 := port_num(s4);
  return msg_chk (p1) and
    msg_chk (p2) and
    msg_chk (p3) and
    msg_chk (p4);
end prim7;

procedure trans7 is

  eq_mode_health : boolean;
  pilot_not_busy : boolean;
  health : boolean;
  jtids : boolean;
  outv : short_integer;

  eq_mode_health_ad : address := eq_mode_health'address;
  pilot_not_busy_ad : address := pilot_not_busy'address;
  health_ad : address := health'address;
  jtids_ad : address := jtids'address;
  outv_ad : address := outv'address;

begin
  p1 := port_num(s1);
  p2 := port_num(s2);
  p3 := port_num(s3);
  p4 := port_num(s4);
  get_obj (p1,obj,B,eq_mode_health_ad);
  get_obj (p2,obj,B,pilot_not_busy_ad);
  get_obj (p3,obj,B,health_ad);
  get_obj (p4,obj,B,jtids_ad);
  if eq_mode_health = true and
    pilot_not_busy = true and
    health = true and
    jtids = true then
    destp1 := port_num(dests1);
    outv := 7;
    snd_obj (destp1,obj,SI,outv_ad,0,0.0);
  end if;
  ret_afo;
end trans7;

end a_wm_map_error;


separate (af_start)

package body eq_mode_health is

  p1 : integer;
  s1 : string (1..17) := "eq_mode_health/i1";
  obj : string (1..3) := "obj";
  destp1, destp2, destp3, destp4, destp5, destp6, destp7, destp8 : integer;
  dests1 : string (1..22) := "a_previous_waypoint/i1";
  dests2 : string (1..16) := "a_unaided_sol/i1";
  dests3 : string (1..12) := "a_ecm_env/i1";
  dests4 : string (1..16) := "a_pilot_aware/i1";
  dests5 : string (1..19) := "a_lead_map_error/i1";
  dests6 : string (1..16) := "a_alpha_check/i1";
  dests7 : string (1..17) := "a_wm_map_error/i1";
  dests8 : string (1..17) := "a_eo_radar_des/i1";

function prim9 (afo : af_struct) return boolean is

begin
  p1 := port_num(s1);
  return msg_chk (p1);
end prim9;

procedure trans9 is

  iv1 : boolean;
  ov1, ov2, ov3, ov4, ov5, ov6, ov7, ov8 : boolean;

  iv1_ad : address := iv1'address;
  ov1_ad : address := ov1'address;
ov2_ad : address := ov2'address;
ov3_ad : address := ov3'address;
ov4_ad : address := ov4'address;
```

```
ov5_ad : address := ov5'address;
ov6_ad : address := ov6'address;
ov7_ad : address := ov7'address;
ov8_ad : address := ov8'address;

begin
  pl := port_num(sl);
  get_obj (pl,obj,B,ivl_ad);
  ovl := ivl;
  ov2 := ivl;
  ov3 := ivl;
  ov4 := ivl;
  ov5 := ivl;
  ov6 := ivl;
  ov7 := ivl;
  ov8 := ivl;
  destpl := port_num(destsl);
  destp2 := port_num(dests2);
  destp3 := port_num(dests3);
  destp4 := port_num(dests4);
  destp5 := port_num(dests5);
  destp6 := port_num(dests6);
  destp7 := port_num(dests7);
  destp8 := port_num(dests8);
  snd_obj (destpl,obj,B,ovl_ad,0,0.0);
  snd_obj (destp2,obj,B,ov2_ad,0,0.0);
  snd_obj (destp3,obj,B,ov3_ad,0,0.0);
  snd_obj (destp4,obj,B,ov4_ad,0,0.0);
  snd_obj (destp5,obj,B,ov5_ad,0,0.0);
  snd_obj (destp6,obj,B,ov6_ad,0,0.0);
  snd_obj (destp7,obj,B,ov7_ad,0,0.0);
  snd_obj (destp8,obj,B,ov8_ad,0,0.0);
  ret_afo;
end trans9;

end eq_mode_health;


separate (af_start)

package body health is

  pl : integer;
  sl : string (1..9) := "health/il";
  obj : string (1..3) := "obj";
  destpl,  destp2,  destp3,  destp4,  destp5 : integer;
  destsl : string (1..16) := "a_pilot_aware/i3";
  dests2 : string (1..19) := "a_lead_map_error/i3";
  dests3 : string (1..16) := "a_alpha_check/i3";
  dests4 : string (1..17) := "a_wm_map_error/i3";
  dests5 : string (1..17) := "a_eo_radar_des/i3";

function priml4 (afo : af_struct) return boolean is

begin
  pl := port_num(sl);
  return msg_chk (pl);
end priml4;

procedure transl4 is

  ivl : boolean;
  ovl,  ov2,  ov3,  ov4,  ov5 : boolean;

  ivl_ad : address := ivl'address;
  ov1_ad : address := ov1'address;
ov2_ad : address := ov2'address;
ov3_ad : address := ov3'address;
ov4_ad : address := ov4'address;
ov5_ad : address := ov5'address;

begin
  p1 := port_num(s1);
  get_obj (p1,obj,B,iv1_ad);
  ov1 := iv1;
  ov2 := iv1;
  ov3 := iv1;
  ov4 := iv1;
  ov5 := iv1;
  destp1 := port_num(dests1);
  destp2 := port_num(dests2);
  destp3 := port_num(dests3);
  destp4 := port_num(dests4);
  destp5 := port_num(dests5);
  snd_obj (destp1,obj,B,ov1_ad,0,0.0);
  snd_obj (destp2,obj,B,ov2_ad,0,0.0);
  snd_obj (destp3,obj,B,ov3_ad,0,0.0);
  snd_obj (destp4,obj,B,ov4_ad,0,0.0);
  snd_obj (destp5,obj,B,ov5_ad,0,0.0);
  ret_afo;
end transl4;

end health;


separate (af_start)

with text_io, word_io; use text_io, word_io;

package body inputafo is

  p1 : integer;
  s1 : string (1..11) := "inputafo/i1";
  destp1, destp2, destp3, destp4 : integer;
  dests1 : string (1..17) := "eq_mode_health/i1";
  dests2 : string (1..17) := "pilot_not_busy/i1";
  dests3 : string (1..9) := "health/i1";
  dests4 : string (1..8) := "jtids/i1";
  done,val : boolean;
  val_ad : address := val'address;
  item,value,str : string80;
  last,length : integer;
  infile : file_type;

function inpprim (afo : af_struct) return boolean is

begin
  p1 := port_opn (s1);
  return not msg_chk (p1);
end inpprim;

procedure getinput is

begin
  destp1 := port_num (dests1);
  destp2 := port_num (dests2);
  destp3 := port_num (dests3);
  destp4 := port_num (dests4);
  put ("Input file : ");
  get_line (item,last);
  open (infile,in_file,item (1..last));
  reset (infile,in_file);
  while not end_of_file (infile) loop
    get_word (infile,str,length,done);
    if not end_of_file (infile) and not done then
      get_word (infile,value,length,done);
      if value (1..4) = "TRUE" then
        val := true;
      else
        val := false;
      end if;
      case str (1..length) is
        when "EQ_MODE_HEALTH" => snd_obj (destp1,obj,B,val_ad,0,0.0);
        when "PILOT_NOT_BUSY" => snd_obj (destp2,obj,B,val_ad,0,0.0);
        when "HEALTH"         => snd_obj (destp3,obj,B,val_ad,0,0.0);
        when "JTIDS"          => snd_obj (destp4,obj,B,val_ad,0,0.0);
        when others           => null;
      end case;
    end if;
  end loop;
  close (infile);

  p1 := port_num (s1);
  snd_obj (p1,obj,B,val_ad,0,0.0);
  return afo;
end getinput;

end inputafo;


separate (af_start)

package body jtids is

  p1 : integer;
  s1 : string (1..8) := "jtids/i1";
  obj : string (1..3) := "obj";
  destp1,  destp2,  destp3 : integer;
  dests1 : string (1..16) := "a_alpha_check/i4";
  dests2 : string (1..17) := "a_wm_map_error/i4";
  dests3 : string (1..17) := "a_eo_radar_des/i4";

function prim17 (afo : af_struct) return boolean is

begin
  p1 := port_num(s1);
  return msg_chk (p1);
end prim17;

procedure trans17 is

  iv1 : boolean;
  ov1,  ov2,  ov3 : boolean;

  iv1_ad : address := iv1'address;
  ov1_ad : address := ov1'address;
ov2_ad : address := ov2'address;
ov3_ad : address := ov3'address;

begin
  p1 := port_num(s1);
  get_obj (p1,obj,B,iv1_ad);
  ov1 := iv1;
  ov2 := iv1;
  ov3 := iv1;
  destp1 := port_num(dests1);
  destp2 := port_num(dests2);
  destp3 := port_num(dests3);
  snd_obj (destp1,obj,B,ov1_ad,0,0.0);
  snd_obj (destp2,obj,B,ov2_ad,0,0.0);
  snd_obj (destp3,obj,B,ov3_ad,0,0.0);
  ret_afo;
end trans17;

end jtids;


separate (af_start)

package body pilot_not_busy is

  p1 : integer;
  s1 : string (1..17) := "pilot_not_busy/i1";
  obj : string (1..3) := "obj";
  destp1,  destp2,  destp3,  destp4,  destp5 : integer;
  dests1 : string (1..16) := "a_pilot_aware/i2";
  dests2 : string (1..19) := "a_lead_map_error/i2";
  dests3 : string (1..16) := "a_alpha_check/i2";
  dests4 : string (1..17) := "a_wm_map_error/i2";
  dests5 : string (1..17) := "a_eo_radar_des/i2";

function prim13 (afo : af_struct) return boolean is

begin
  p1 := port_num(s1);
  return msg_chk (p1);
end prim13;

procedure trans13 is

  iv1 : boolean;
  ov1,  ov2,  ov3,  ov4,  ov5 : boolean;

  iv1_ad : address := iv1'address;
  ov1_ad : address := ov1'address;
ov2_ad : address := ov2'address;
ov3_ad : address := ov3'address;
ov4_ad : address := ov4'address;
ov5_ad : address := ov5'address;

begin
  p1 := port_num(s1);
  get_obj (p1,obj,B,iv1_ad);
  ov1 := iv1;
  ov2 := iv1;
  ov3 := iv1;
  ov4 := iv1;
  ov5 := iv1;
  destp1 := port_num(dests1);
  destp2 := port_num(dests2);
  destp3 := port_num(dests3);
  destp4 := port_num(dests4);
  destp5 := port_num(dests5);
  snd_obj (destp1,obj,B,ov1_ad,0,0.0);
  snd_obj (destp2,obj,B,ov2_ad,0,0.0);
  snd_obj (destp3,obj,B,ov3_ad,0,0.0);
  snd_obj (destp4,obj,B,ov4_ad,0,0.0);
  snd_obj (destp5,obj,B,ov5_ad,0,0.0);
  ret_afo;
end trans13;
end pilot_not_busy;


separate (af_start)

package body system_output is

  p1 : integer;
  s1 : string (1..16) := "system_output/i1";
  obj : string (1..3) := "obj";

function prim20 (afo : af_struct) return boolean is

begin
  p1 := port_num(s1);
  return msg_chk (p1);
end prim20;

procedure trans20 is

  iv1 : short_integer;

  iv1_ad : address := iv1'address;

  msg : string (1..14) := "ask_question_#";

begin
  p1 := port_num(s1);
  get_obj (p1,obj,SI,iv1_ad);
  put_line(msg);
  put(  iv1);
  new_line(  2);
  ret_afo;
end trans20;

end system_output;
```

4. A Description of the AFA Activation Framework Software

4.1 Introduction

This section describes the concepts behind AFA, the Ada version of the Activation Framework software. This section describes the implementation of AFA and details procedure calls used at the applications level. It also describes the Ada version of the LMS list management system.

4.2 AFA Concepts

AFA is a software tool for use in developing intelligent real-time systems such as that shown in figure 4.1. Such systems capture and process data from sensors and use this data as the basis for intelligent control of actuating devices. These systems can interpret data from multiple sensors, combine this with operator input, and build a model, the so-called "world model", of the system's external environment. This world model is used for planning the actions of the system and as the basis for decisions about advice to give to operators or how to adapt automatic control strategies.

Not all intelligent real-time systems have all the functions shown in figure 4.1, but all have more than just the signal processing and control algorithms characteristic of conventional real-time systems. These systems are called intelligent real-time systems because they have many of the event driven characteristics of real-time systems and yet they contain higher level intelligence which permits the systems to perform in a more flexible manner than conventional real-time systems.

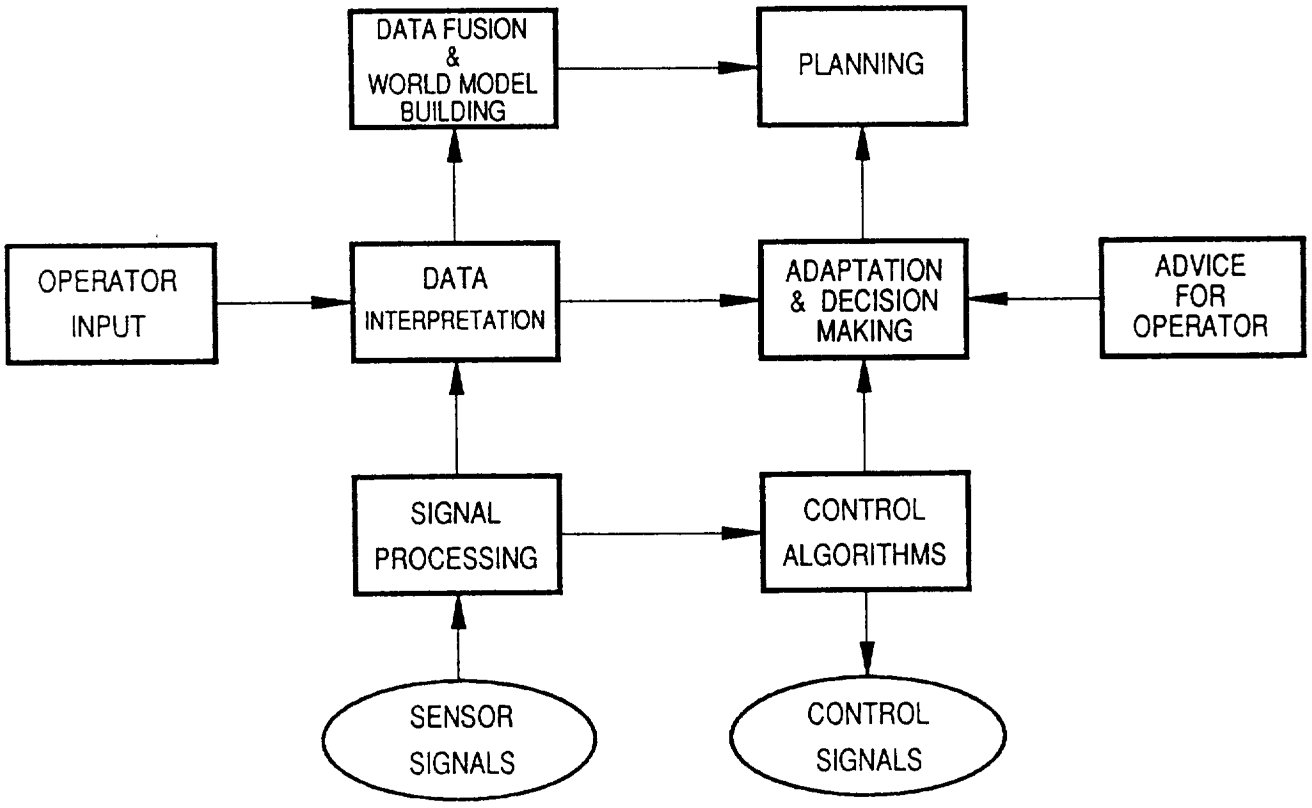


Figure 4.1 An Intelligent Real Time System

AFA provides a structured framework for building real-time intelligent systems. This framework includes an execution environment which provides support for those actions which are common to most real-time intelligent systems. These actions include the pre-emption, scheduling, and execution of independent code modules on a heterogeneous network of computers; the routing, prioritization, and delivering of messages between code modules; the dynamic management of memory and management of lists of data objects; the automatic encoding and decoding of data objects as messages; and the provision of mechanisms to allow for rapid changes in the focus of attention of the system.

Conceptually AFA provides a uniform interface for most applications code modules irrespective of the computer hardware or operating system in use at that node in the network of computers. As shown in figure 4.2, AFA is intended to provide a uniform execution environment so that the overall application system can make effective use of all the processors for cooperative parallel processing. AFA currently works in conjunction with MSDOS but Unix and VMS versions are expected to be developed in the future as are stand-alone versions which will work in environments such as a 680x0 based shared memory multiprocessor. AFA has been designed to make it easy to port to other environments by recoding about a hundred lines of assembly language code which saves and restores the processor's register state.

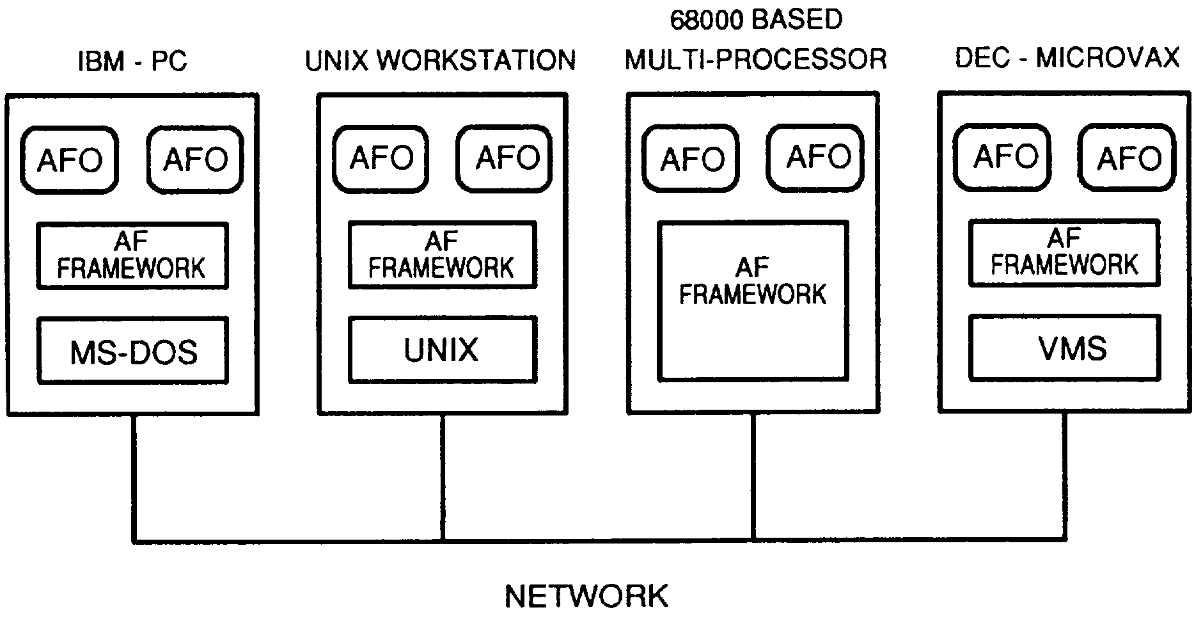


Figure 4.2 AFA Provides a Uniform User Interface for Applications code modules Activation Framework Object

AFA uses an object-oriented message-based paradigm. Applications are coded in the form of Activation Framework Objects (AFOs) which are Ada language subroutines that communicate by sending and receiving messages as shown in figure 4.3. AFA provides the mechanisms necessary for the scheduling of the execution of the AFOs and the transmission of messages between them. AFA also provides other necessary functions such as list management and message encoding and decoding. AFA is capable of supporting various communications mechanisms between the processors including shared memory, local area networks, serial data links, and satellite links. In the future, AFA will be capable of execution on a group of widely distributed processors.

Most AFOs are programmed in a template-like format:

```
get_message()

perform algorithm

send_message()
```

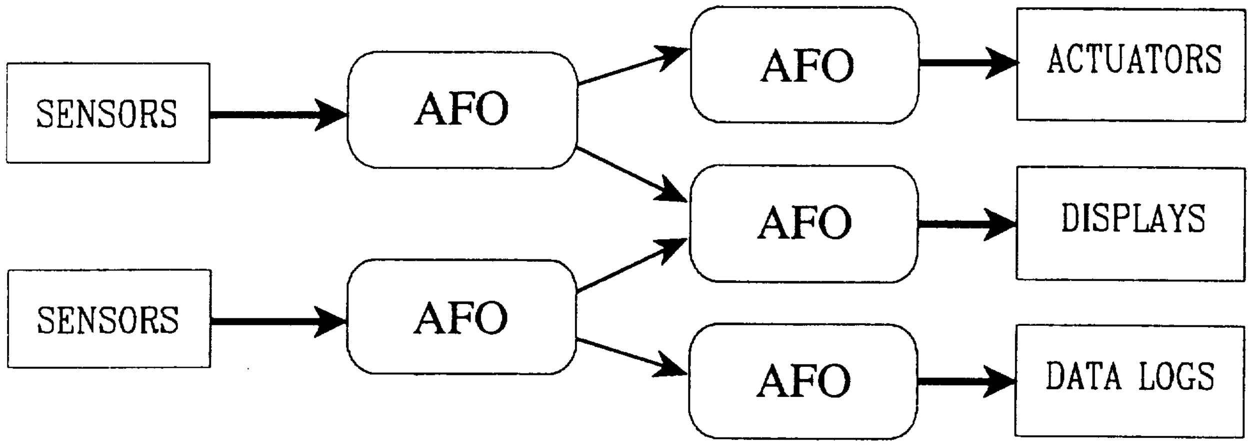


Figure 4.3 An Intelligent Real Time System coded using Activation Framework Objects (AFOs)

This makes the AFOs easy to code. This also makes it simple to develop translators from other representations, such as evidence flow graphs to a set of AFOs.

While AFOs are written as Ada procedures, they are executed as parallel code threads which have their own register state and their own stacks. As events such as message delivery occur within the system the importance of each AFO changes dynamically, and the currently executing AFO can be pre-empted by another AFO if the other AFO becomes more important to execute.

The benefits of using AFA for developing intelligent systems are:

1) AFA minimizes software development time by providing the routines common to all intelligent real-time systems within one integrated framework.

2) AFA's object oriented approach allows programming team members to develop and test AFOs independently thus minimizing elapsed time for systems development.

3) The standard interface between the AFOs and the AFA run time environment allows the development and simulation of a system of AFOs on a workstation before being transferred to embedded hardware.

4) AFA's focus-of-attention scheduling mechanism makes quasi-optimal use of processor and communications resources, thereby reducing the cost of the computer hardware required relative to a conventional real-time system.

5) AFA's standard template format for programming permits rapid development of applications code modules and makes the development of automatic translators from evidence flow graphs a straightforward process.

AFA is linked with applications AFOs to form a run-time module for each processor which contains a framework as shown in figure 4.4. The framework is responsible for providing a number of support services:

a) Scheduling of the AFOs within a framework according to the dynamic estimation of their importance, which is based on the overall system's current state and goals.

b) Delivering messages between local AFOs and between local AFOs and those on remote frameworks, dynamically prioritizing message delivery according to the importance of the messages.

c) Providing dynamic memory management through the list management subsystem.

d) Performing automatic encoding and decoding of data in messages to maximize delivery efficiency and to minimize the amount of user coding required.

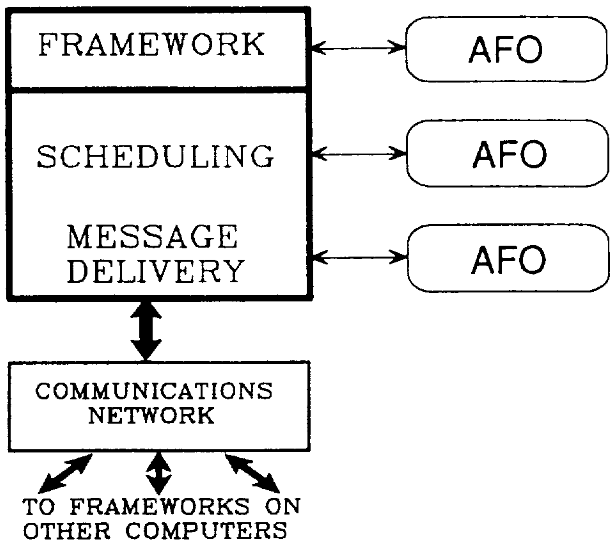


Figure 4.4 An Activation Framework

AFA is designed to support real-time systems which are demanding of resources and yet must function within strict time constraints. The philosophy behind the design of AFA is to build intelligent systems which are able to perform the best they can within the constraints of available cpu, memory, and communications resources.

Within an intelligent real-time system, it is desirable to execute many pieces of code at the same time. As can be seen from figure 4.1, planning can take place in parallel with data interpretation, adaptation, and control. It is not possible to determine ahead of time which is the most important action to carry out when there is contention for processor resources. In some cases it may be very important to collect data while in others it may be critical to devote all the resources into planning how to deal with an emergency.

In conventional real-time systems, the problem of allocating processor time has traditionally been solved by allocating a fixed time slice of each processor to each function the system has to perform. A fast enough processor is then used to make sure that the algorithms can process their input data within their time slice. Intelligent systems typically evaluate alternatives. In doing this, they build and search trees and lists whose size depends on the input data. Thus the processor and memory requirements of an expert module can vary widely for different inputs. If fixed time slice scheduling were used then it would be necessary to curtail the searches of some modules while there was available time unused by other modules. AFA solves this problem by dynamically varying the priority of the modules according to their current importance levels. Thus data collection or planning might get the most processor time at one instant while background processor diagnostics might be run at another when there is no other action required.

In conventional real-time systems operation, the processing of input data is given high priority to ensure that this data is processed prior to the arrival of new data. This is normally achieved by performing the input data processing in the interrupt handlers used for data input. As interrupts have high priority they preempt execution of higher levels of the system such as planning. This can be very detrimental in emergency situations when it may be desirable to cease processing new input data and instead focus on planning how to deal with the emergency which has already been detected.

In a conventional scheme, valuable processor time is taken up processing data which may not be needed. In AFA, an interrupt handler sends its data in the form of a message to a processing AFO. The action of sending the data message raises the importance of the AFO. Normally this will cause the AFO to be executed and process the data. If, however, the data processing AFO is not the most important function at that time then it will not be executed, allowing, for example, a planning AFO to take precedence. This may result in data being delayed before it is processed but that may be preferable to having the system fail to deal with some catastrophic emergency.

Conventional time sharing system schedulers determine which task to execute next based on the past history of tasks. Tasks which are compute bound get low priority whereas tasks which are doing terminal I/O get high priority. This may be just the opposite of what is required in an intelligent real-time system. In an intelligent real-time system, knowledge about what is the most important thing to do next is contained in the application modules. A data interpretation module may correlate two pieces of input data to discover that a robot is about to collide with an obstacle. It is essential to raise the priority of the brake control module to a high level as soon as it receives the message from the data interpretation module. This knowledge is contained in the data interpretation module at the time it sends the message.

For an intelligent real-time system, the problem is how to dynamically determine the importance of executing code modules when the knowledge is contained in modules executing on different machines. The mechanism used must offer quick response and yet consume only a small amount of resources for its execution. The approach used in AFA is to have the AFO sending a message place an activation level on the message which is a measure of the importance of the message. These message activation levels are then used as the basis for computing how important it is to execute the recipient AFO. An AFO with a number of important messages on its input queue is more important to execute than one with few low activation messages. Since this information is carried along with the data between AFOs very little overhead is required to determine which AFO is most important to execute on each processor. This distributed approach is in contrast to schemes in which the importances are all collected at some central location, the modules to run are selected, and then the scheduling information is broadcast to all processors. Such centralized schemes are slow to respond and have high overhead relative to the distributed importance estimation scheme used in AFA.

AFA supports a number of refinements on this basic scheme. AFOs can also be assigned a global importance. Their importance estimation from message activation levels is then multiplied by their global importance. This keeps the importance of messages from an unimportant AFO being given too much significance in the scheduling decision. It also stops background AFOs, such as data-loggers, from being raised to high importance by the arrival of a message, which although important in a global sense, is not different in importance from any other message for the purposes of data logging. Messages can also be given deadlines after which the data in them is no longer valid. Messages closest to their deadline are delivered first if they are of equal importance, otherwise the most important message is delivered first. Messages beyond their deadline time are eliminated from the system. In this way AFA optimizes the use of communications resources according to applications knowledge encoded in the messages sent between AFOs.

A major problem for real-time AI systems is dynamic memory management. In classical real-time systems memory space is pre-allocated for every object which the system can possibly access. Real-time intelligent systems build lists and trees whose size depends on the input data. These represent its memory and its evaluation of alternatives. If the space for each type of object must be pre-allocated, then some modules will not be able to carry out their function properly because they do not have enough memory, while there is excess memory unused by other functions. In AFA this problem is addressed through the use of a common memory heap which can be accessed through the list management system (LMS).

LMS enables users to write data objects onto lists contained in the heap and to release the memory used by objects when they are no longer needed. This scheme enables users to make use of the total amount of memory available to be shared amongst the AFOs according to their needs. By requiring the user to explicitly release memory space occupied by modules, the user is given control of when to expend processor time in recovering memory. Usually this results in breaking up the "garbage collection" time into small increments during execution of the system. This is in contrast to many LISP systems which are infamous for taking tens of minutes to garbage collect memory after the last free memory cell has been used up. A real-time system can afford frequent execution of a routine taking a few hundred microseconds for explicit memory recovery whereas taking time-out to garbage collect memory for tens of minutes can result in undesirable side effects. It should be remembered however, that the total amount of time spent in garbage collection is no smaller.

A major problem for programmers developing a message-based object-oriented system is the encoding and decoding of messages. Using conventional techniques, if the user wished to send data which, for example, consisted of a list of lists of objects consisting of two floating point numbers and a string, he would have to develop a substantial body of code to serially encode this into a message and then to decode the message back into a list of lists of objects upon receipt. This encoding and decoding take considerable CPU time which can be avoided by simply copying the data structure if the message is to be delivered within the same shared memory, for example between AFOs running on the same processor.

Conceptually, AFA overcomes this problem by using Generalized Objects (GOs). A GO consists of a name, a Data Definition Language (DDL) description, and a data structure. The name is to allow dynamic binding by name to a GO and the DDL describes the data structure. The DDL for the list of lists example would be (L(L(ffs))) indicating a list of objects whose sole entry is a list consisting of two floating point numbers and a string. The memory space for GOs is obtained using LMS. Currently in AFA only simple objects consisting of a single integer or a single boolean have been implemented.

When a user wishes to send a data object in a message, he codes it as a GO using AFA support routines. Then the user calls the msg_send() routine which determines whether the destination AFO is local or on another processor. If the AFO is local, then the DDL is used to make a copy of the existing structure which is passed to the recipient AFO as part of the message. If the AFO is in another processor, then the DDL is used to serialize the data structure in a standard format for inclusion in the message. When a serialized message is received, the DDL is used to decode the serial data stream and convert it back into its original form for the recipient AFO. The AFA message delivery mechanism does not encode and decode messages if the sending and receiving AFOs share memory. This is performed without programmer supplied directions and results in a reduction of processing resources required for message delivery in the shared memory case.

Potential problems arise when processors with different characteristics are used in a system. For example some processors have different byte orders in strings and words, some processors have different representations for integer and floating point formats, and some processors use different representations for characters. This will be overcome in future versions AFA by using a standardized representation for the serial form of messages. Messages are converted from local formats into the standard format for transmission and converted again upon reception. In this way AFA can provide the functions of the presentation layer of the ISO network standard.

4.3 AFA Internals

In an AFA system, AFOs are pre-assigned to processors and do not migrate. This is because in most real-time systems there is not time to migrate processes in the event of a fault; rather the system must be designed to continue to operate with the resources that are left. However, redundant AFOs may be included in AFA load modules for processors and will not consume any processor resources (other than memory) until they are activated by the arrival of messages.

Figure 4.5 shows how an AFA load module is created for a processor. The user specifies which AFOs are to be included in the load module by listing them in a frame file. This frame file is translated by the AFA translator to create a main program for the load module. This is compiled, along with user-coded AFOs, to produce relocatable binary object modules. These modules are then linked with the AFA libraries to produce a run-time executable object module which contains all the functions of an AFA framework.

When first executed, the framework initializes all the AFOs by creating a data structure for each AFO which contains all the information about the AFO including its register state, allocating stack space for each AFO, and creating the input ports for each AFO . Once this is done, the framework selects the AFO with the highest initial importance (as specified in the frame file) and executes it.

When an AFO sends a message to another AFO, the execution of the sending AFO is pre-empted and the importance of the recipient AFO is increased due to the delivery of the message, as shown in figure 4.6. If the sending AFO is still the most important AFO to run then the AFO execution is continued by a return from the call to send the message. If, as a result of message delivery, the recipient AFO has become the most important, then the state of the previously executing AFO is saved and execution of the most important AFO is started. If the AFO was previously pre-empted then execution is continued from the state the AFO was in when it was pre-empted. Otherwise, execution starts from the beginning of the AFO.

AFOs specify importance values for messages when they send them. These importances are specified on a 1-to-10 scale and represent the relative importance of the messages from the local viewpoint of the sending AFO. These importances are multiplied by the global importance of the sending AFO, again on a 1-to-10 scale, to obtain the activation level of the message being sent. This activation level is used to prioritize the transmission of messages and is also used in the computation of the importance of executing the recipient AFO.

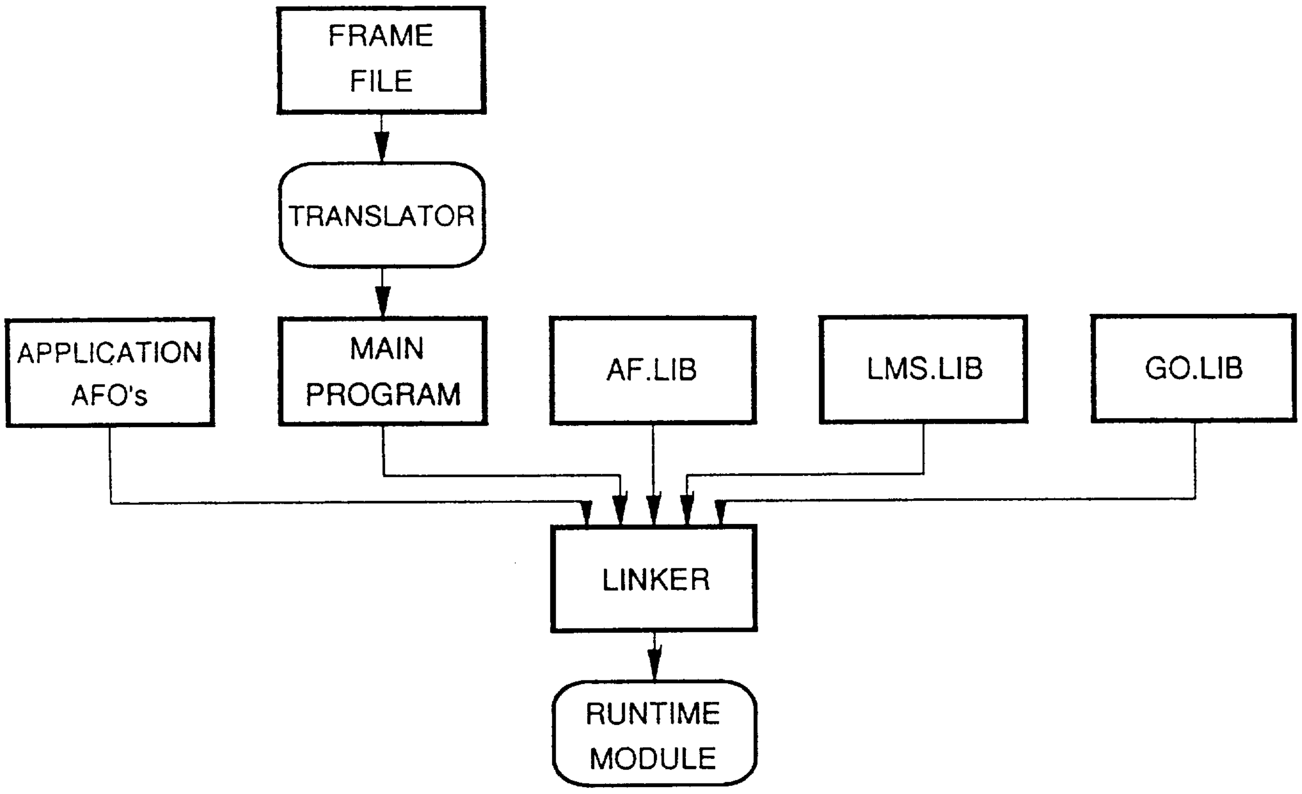


Figure 4.5 Creating an AF Run-Time Module

By default the importance of the receiving AFO, when a message is delivered, is computed by adding up the importances of all the messages on its input ports and multiplying by the global importance of the recipient AFO. Other ways of calculating the importance can be specified by providing an importance function calculation procedure for the AFO.

The global importance of an AFO is based on the importance of the AFO in the overall system. Thus a data logging module might have a low importance whereas an emergency response module might have a high importance. This allows the modules to be written, and to prioritize messages, independent of their importance in the final system. These global importances can be set a priori in the frame file. They can also be changed dynamically by sending a message to the input port of the framework to which the AFO, whose priority is to be changed, is attached.

AFOs can have multiple input ports as shown in figure 4.7. Each port on an AFO has a name. The default port with which every AFO is created is called "std_port". Other ports can be created dynamically as needed. Messages sent within a framework are sent to a specified AFO and port by opening a port by name with the form "afo/port". The open routine returns an index into a framework port table which is then used as the destination identification for messages. In future versions, when messages are sent between frameworks, the port id (index into the port table) will be converted into a full destination address of the form "frame/afo/port" which becomes the destination field of the message. This destination name will be converted into a destination port id upon arrival of the message at the destination framework.

In future versions, messages destined for AFOs on other processors will be sent to surrogate AFOs on the local processor which are responsible for their delivery using interrupt or DMA handlers or using network servers if AFA is running on top of an operating system such as UNIX. Messages arriving from other processors will be received by AFOs designed for this purpose. Execution of these AFOs will be initiated by the delivery of messages onto their input queues by the input device interrupt handlers, or they may wake

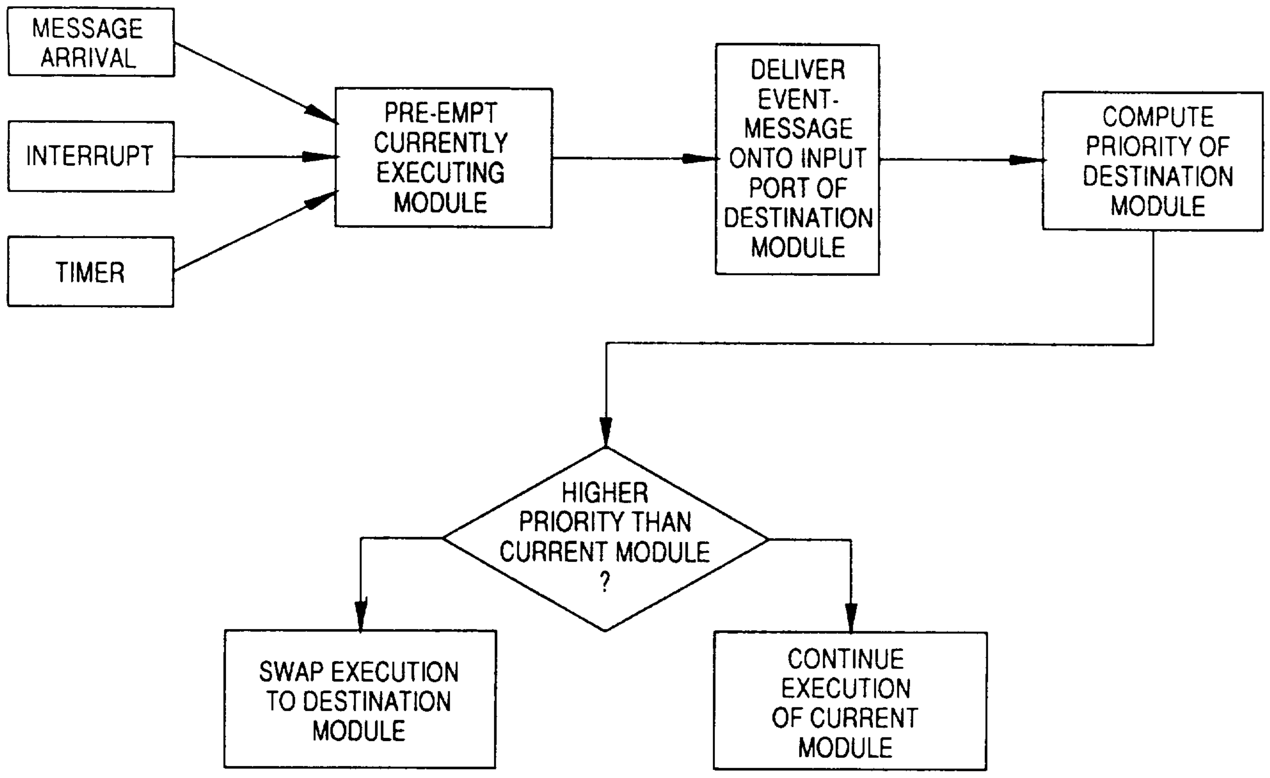

MESSAGE ARRIVAL
INTERRUPT
TIMER
PRE-EMPT CURRENTLY EXECUTING MODULE
DELIVER EVENT-MESSAGE ONTO INPUT PORT OF DESTINATION MODULE
COMPUTE PRIORITY OF DESTINATION MODULE
HIGHER PRIORITY THAN CURRENT MODULE ?
SWAP EXECUTION TO DESTINATION MODULE
CONTINUE EXECUTION OF CURRENT MODULE

Figure 4.6 Dynamic Prioritization of Code Module Execution

themselves up periodically to collect data from a network server.

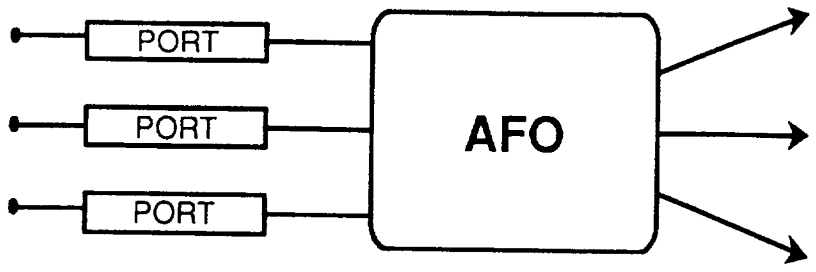
AFA provides, and uses, dynamic memory allocation by using the List Management System (LMS) library. Lists are implemented as doubly linked lists as shown in figure 4.8 to enable rapid traversal of lists in either direction. A full set of library function calls is provided which allow data objects to be written and retrieved from lists. Memory is allocated automatically, as needed, when lists are created or new elements are written into lists. This memory is released automatically when elements are deleted from a list.

The LMS is also used to manage the Lists used internally within AFA. Information about the AFOs on a processor is kept in a list of AFO object structures. Message queues are also built using LMS.

Messages are queued on a pending queue on each AFO input port. The pending queue is a list of messages managed using LMS. When messages are delivered onto the AFO's port they are placed on the pending queue in order of their importance and closeness to deadline. When the AFO wishes to access a message it calls get_msg() which causes the message to be removed from the pending queue and placed in the AFO's active message pool. AFOs can access data from messages in their active pool using the pool entry index returned from the call to get_msg().

The importances of messages on all of an AFO's pending queues plus the importance of all messages in its active message pool are used in determining the importance of an AFO. In this way the importance of the AFO is not reduced simply because it has accessed a message. An AFO can automatically release all messages from its active pool by calling return() after execution.

The format of messages in memory is shown in figure 4.9. Each has a TO_AFO_ID and a LOCAL_TO_AF_ID. The IDs are indexes into a table of ports which is constructed by the translator from the frame file. This table contains the full string names of each of the ports and the addresses of the port structures for each of the AFOs on the local processor. The TO_LOCAL_AF_ID is used to determine to which port on the local processor the message should be delivered. The TO_AFO_ID is used to encode the port to which the



PORTS ARE NAMED

AFO's SEND MESSAGES TO PORTS BY NAME
AFO_NAME/PORT_NAME

NETWORK USES FULL ADDRESSES
/FRAME_NAME/AFO_NAME/PORT_NAME

Figure 4.7 AFOs can have multiple input ports on which messages or events can be queued.

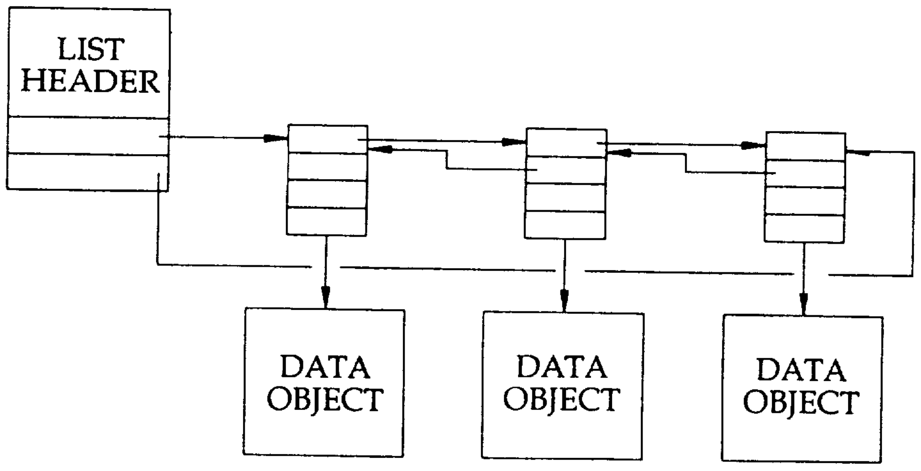


Figure 4.8 List Management System Structures.

message is to be delivered as an ascii string if the message has to be serialized for delivery to another processor. The FROM_AFO_ID is included to allow a reply to be sent, if required, by the recipient AFO. This is translated into a string if the message is encoded serially for transmission to another processor.

Messages contain an event type field, an activation level field (which is a measure of their importance), and a deadline by which they must be processed or dropped from the system. The data part of messages is in the form of a Generalized Object (GO).

A generalized object, figure 4.10, contains pointers to three elements: a name string, a DDL string, and the Data Object. The Data Object may in future versions be a List or an object which contains a pointer to a

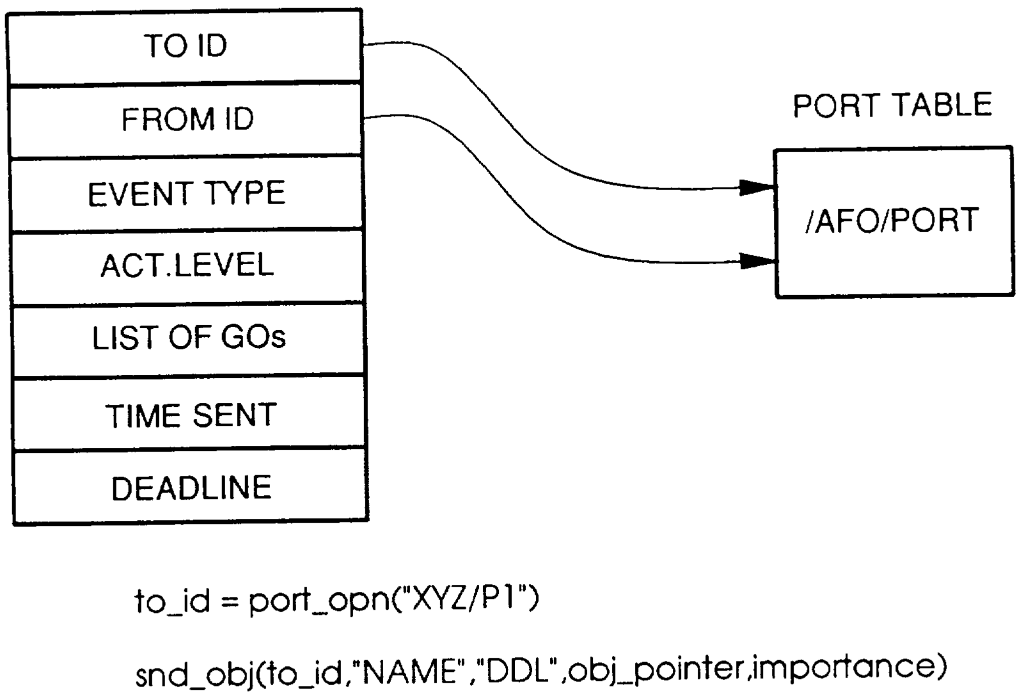


Figure 4.9 Format of AF messages in memory.

List. This list of lists of lists recursion is only limited by the amount of available memory. It will be useful for building and passing tree structures. The name is used for dynamic binding to a data object when the message is received. The DDL is a complete concise description of the data structure so that it can be copied or serialized for transmission.

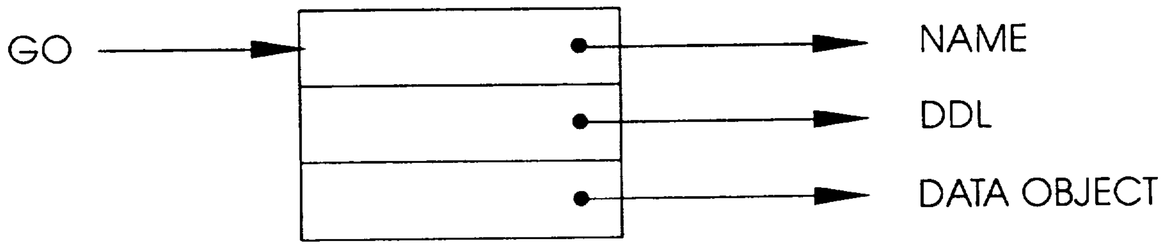


Figure 4.10 Generalized Object.

As shown in figure 4.11, users can have control over system level functions by providing initialization, priming, and importance functions as well as the transfer function which must be provided for every AFO. These procedures can be specified in the frame file and allow for flexibility in use of AFA. The initialization function for each AFO is called as soon as the data structures for the AFO have been created. This procedure can be used to perform such functions as opening additional ports and forming linkages with interrupt handlers for data I/O.

An AFO is not allowed to run unless it is primed. By default an AFO is considered to be primed if it has any messages queued on any of its ports. This can be over-ridden by a priming function which may require that messages be present on all ports before the AFO is allowed to execute. Users can also provide their own importance function so that difference evidence computation mathematics can be used other than the default incremental evidence method of adding up the activation levels of all the messages.

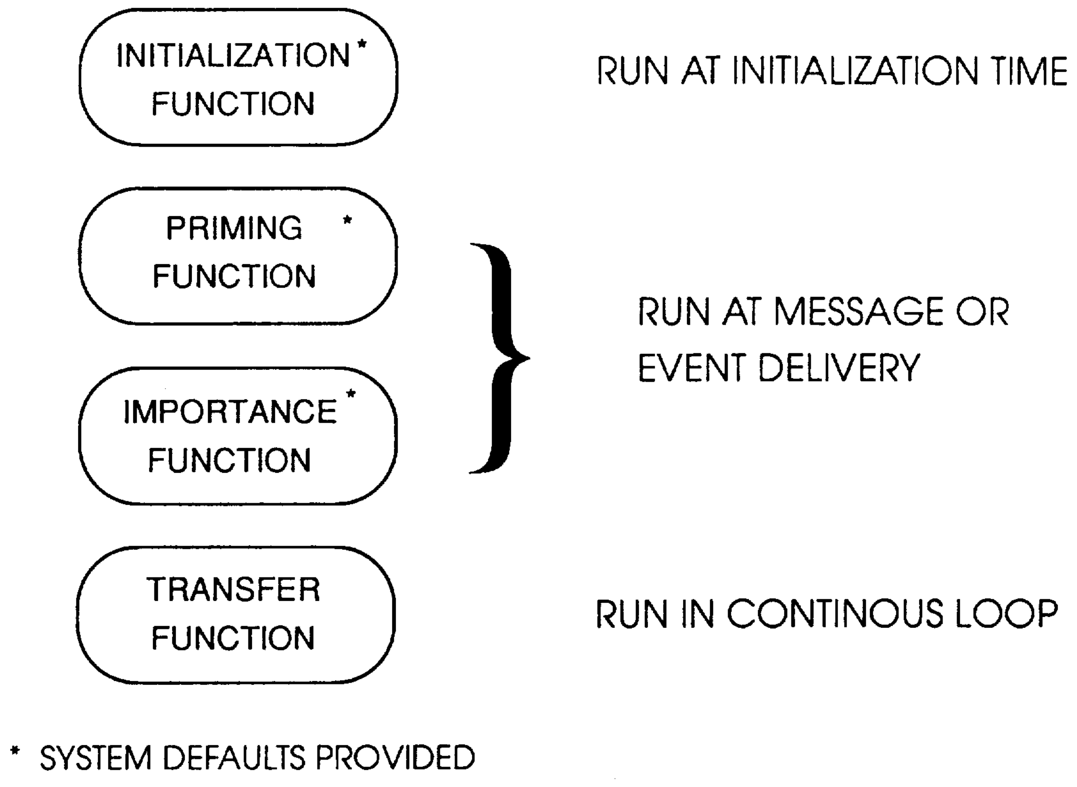

INITIALIZATION * FUNCTION
RUN AT INITIALIZATION TIME
PRIMING * FUNCTION
IMPORTANCE * FUNCTION
RUN AT MESSAGE OR EVENT DELIVERY
TRANSFER FUNCTION
RUN IN CONTINOUS LOOP
* SYSTEM DEFAULTS PROVIDED

Figure 4.11 User Control Over System Functions.

4.4 Coding AFOs

Users of AFA decompose their systems into AFOs which they code as Ada language subroutines. An example of an AFO is shown below:

```
procedure temp_decis (afo_st : in out afo_struct) is

  old_switch : integer;
  err : integer;
  port_id : integer;
  temp : float;
  temp_ad : address := temp'address;
  in_ddl : char := 'F';
  switch : integer;
  switch_ad : address := switch'address;
  out_ddl : char := 'I';
  t : string(1..11) := "temperature";
  t2 : string(1..9) := "temp_cont";
  s : string(1..6) := "switch";
  mid : message_handle;

begin
  mid := get_msg;
  err := bind (mid,t,in_ddl,temp_ad);
  if err > 0 then
    null;
  end if;
  if temp > 75 and old_switch = 1 then
    switch = 0;
    end if;
    if temp < 65 and old_switch = 0 then
      switch = 1;
    end if;
    port_id := port_open (t2);
    send_obj (port_id,1,s,out_ddl,switch_ad,10,rel_time(0.5));
    old_switch := switch;
  end temp_decis;
```

A summary of salient points about AFOs is given below.

1) AFOs are self contained entities. Some, such as the example here, simply receive and send messages. Others interact with sensor or display hardware, either directly or through associated interrupt routines. They can be developed and debugged separately before being integrated into an overall system.

2) AFOs do not share memory variables. This is so they can be run on any processor and can be pre-empted at any time in their execution with no danger of deadlock.

3) While AFOs are written as Ada language procedures, they are executed as asynchronous tasks. Each has its own stack which is re-initialized every time the AFO is executed from its entry point. The AFO can be suspended, that is have its state saved, every time it calls get_msg() or send_obj(), if it is not the most important AFO to be executed at that time. If it is suspended, it will be resumed where it left off once it becomes the most important AFO in its local processor once more. When an AFO performs a return to the framework, it is then executed from its initial entry point the next time it becomes the most important AFO to run.

4) Extensive checking is done on the content and format of messages. In a distributed message-based system, one of the biggest problems in software development is in assuring that two modules agree on the format of messages to be sent between them. Detection of problems in which programmers change a message format in a sending AFO but not in a receiving AFO are handled easily in AFA.

5) All AFOs have names, and messages are sent to AFOs by name. This allows the development of AFOs without concern for which processor they will ultimately be executed. This allows the simulated execution of a whole system on a workstation prior to porting it to some multi-processor embedded environment.

6) Users do not have to be concerned with the format or encoding of messages. In future versions of AFA messages will exist in a number of forms. They will consist of a message header with a pointer to its constituent parts, they will exist as a compact serialized bit stream for transmission over high-speed bit-serial data links, or they will exist as a sequence of bytes for transmission over low speed asynchronous serial data links such as RS232c. These transformations will take place transparently to the user.

7) Messages can be typed. The type number allows the recipient AFO to switch between different sections of code to process received messages of different types.

8) Each AFO has multiple input ports on which messages are queued. For each input port, the messages are queued in order of importance and then messages of equal importance are queued in order of their closeness to their deadline. AFOs can access and process their input queues in any manner the programmer chooses. Get_obj() gets the message from the top of the queue. More specifically, each port has a pending and an active queue. Arriving messages are queued on the pending queue. A call to get_obj() removes the top message from the pending queue and transfers it to the active pool. The message id value returned by get_msg() is a handle by which the message can be accessed by the programmer for further manipulations on the message.

9) Messages are delivered by the framework as soon as they are sent by a call to send_msg(). If the recipient AFO is local, the message is delivered onto its input port's pending queue. If the recipient AFO is on another processor, then the message is delivered to the AFO designated to transmit the message to its remote destination.

10) The delivery of a message may cause the recipient of the message to become more important than the sender, and, if they are executing on the same processor, a switch in which AFO is being executed is

made. This is a highly desirable property as it allows emergency actions to rapidly ripple through a system without special coding being needed to accomplish this effect.

11) Messages are given an importance by their sender. This importance is an integer number which is multiplied by the global importance of an AFO (another integer) to obtain the activation level of a message. When there is contention for a communications link, messages with the highest activation level are given priority, and within that, messages closest to their deadline are given priority.

12) When writing AFOs, the framework to which they will be attached and their global importances are not specified. These are set at system configuration time using frame files which specify the AFOs attached to a framework and their initial global importances. AFOs can modify their global importances. This can also be coded so that a "manager" AFO sends messages to other AFOs telling them to modify their global importances which then results in a change in systems behavior. This can even be encoded hierarchically to give a similar effect to a military or management command structure.

13) The importance of an AFO, for scheduling purposes, is computed by adding up the activation levels of all messages on its input queue, adding the activation levels of all messages in its active pool, and then multiplying by the global importance of the AFO. The importance of the AFOs is evaluated every time a call to the framework is made (such as by send_obj() and get_obj()) and the most important AFO is executed. If the AFO was previously executing when it was suspended, it continues from where it left off. If it completed its execution with a return or has never been run then it is started executing at the beginning of the AFO with an empty stack.

14) Messages are dropped from the system once they pass their deadlines. This saves the system from becoming cluttered with messages that the system does not have time to process. This also places some fault tolerance constraints on the coding of AFOs. They should not be coded to send a message and then to wait for a reply. Rather they are coded to send a message requesting information and have a separate code section (triggered by a specific message type) to respond to the reply if it ever comes. This is particularly appropriate to distributed intelligent real-time systems in which message delivery within a finite time can never be guaranteed.

4.5 The Frame File

The frame file is used to specify which AFOs run on a particular processor. An example frame file is shown below:

```
FRAME PCAT ; sets the name of the frame

AFO  get_temp 10 0    ; declares an AFO whose name is get_temp with a global
                      ; importance of 10 units and no initial internal
                      ; importance

AFO  temp_decis 20 0  ; declares an AFO whose name is temp_decis with a global
                      ; importance of 20 units and no initial internal
                      ; importance

AFO  serial_IO 30 10  ; declares an AFO whose name is serial_IO with a global
                      ; importance of 30 units and an initial importance
                      ; of 10 units

EXTERN temp_cont serial_IO ; declares that temp_cont is an AFO on another
                           ; framework to which
                           ; messages may be sent and to send messages
                           ; to the serial_IO AFO for delivery to this AFO
```

There are a number of points to be noted here:

1) Frame files have one entry per line. Anything following a semicolon is a comment.
2) AFA has no limit on the length of AFO names but, as these are subroutine names, the Ada compiler used may impose some limitation on name length.
3) Frames have names which must be globally unique.
4) The decision of which AFOs are assigned to which processor is left to the user as this depends on the performance characteristics desired. These decisions are easily changed by simply modifying the frame files to move AFOs from one frame to another.
5) If messages are to be sent to an AFO on another processor, then this must be declared as EXTERN in the frame file and the surrogate AFO to which messages are to be sent for transmission designated. This allows for the checking of message destinations during message transmission.

4.6 Message Transmission

Messages sent within the same framework are created as soon as send_obj() is called and then a pointer to the created message is placed in the recipient AFO's pending queue. There is no copying of any data except the name, DDL, and data object supplied by the user. The reason these are copied is that the AFO can modify these as soon as a return is made from send_msg which may be before the message has been processed by its recipient.

In future versions of AFA, messages to be sent externally will be queued on the input port of the surrogate AFO designated in an EXTERN statement in the frame file. This AFO will then be responsible for delivering the message to a corresponding AFO on the recipient frame over some communications network or data link which it interfaces to. The user will be responsible for ensuring that the communications AFOs are set-up in such a way as to obtain correct routing of messages. Messages may be routed through intermediate nodes if their frame files contain appropriate EXTERN statements.

A message arriving at a framework over an external data link will have a destination port address, encoded as an ascii string, inside the message. This destination port address will be checked against the port table and, if the port exists on the local processor either directly or through a surrogate, then the message will decoded and delivered onto the destination AFO port. Decoding is done by the initial network recipient AFO to convert the message from its bit serial form into the memory data structure form of a message. The message is hten delivered, and AFO execution is swapped if the change in importances warrants this action.

In future versions, if a message arrives at a framework and the destination port does not exist then a port will be created, given an ID in the port table, and the message will be queued on the port until the port is opened. The reason for this action is that future versions of AFA will provide the capability to create ports dynamically, and messages may arrive for a port before it is created.

Messages arriving from another framework may have a return port address which is not in the local port table. In this case an entry will be created in the port table with a surrogate AFO specified by the network receiving AFO (usually itself as these AFOs will usually handle both the transmission and reception of data.

4.7 Application AFO Calls

This section describes procedure calls which are commonly used by applications AFOs. Initialization of AFOs is usually done automatically by the main program generated from the frame file by the translator program.

The user is required to provide a transfer procedure for each AFO. The user may provide an initialization function which is called with an argument which is the AFO structure - any changes can be made to the AFO. This routine is called at initialization time by afo_init. This initialization routine may establish linkage with an interrupt routine and may send messages to ports which have already been created. Such message delivery will not result in any task swapping until all of the AFO's have been initialized. If this routine is specified as NULL in the frame file (fwork.afa) then it will be ignored.

Users can provide a priming function which is called with the AFO's structure. This is called when a message is delivered and is expected to return true/false meaning the AFO's priming status was updated or not. If the priming function is specified as NULL, then the AFO is marked as primed as soon as a message arrives. By default, an AFO is marked as not primed once it has no messages on any input port or in its active pool. If a priming function is provided, it will be called whenever messages are released or removed because their deadlines pass, so that priming can be recalculated.

Note that the importance of an AFO is updated whenever a message is delivered irrespective of whether it is primed on a message. This importance is only used to schedule an AFO. This allows the importance level to be computed incrementally rather than having to traverse all the messages in an AFO's ports and message pool in order to determine its current importance.

The following are routines which can be called from the transfer function of the AFO:

```
function port_num(name : string) return integer;
-- name is the string containing local port name in the form "afo/port"

Returns local port id corresponding to name. Port must previously have been
created. "std_port" is port 0 which is defined by STDPORT.


procedure get_msg(n : integer;objname,ddlname : string; obj_address : address);
-- n is the local input port id for AFO -> by convention port 0 is the default
-- port named "std_port"
-- objname = name of data object
-- ddlname = ddl of data ( only short integer and boolean supported now)
--          SI = short integer and B = boolean
-- obj_address = address of data object which should receive data

      Gets the most important message which is closest to its
      deadline from the specified input port. Overlays the data object in this
      message with obj_address. Messages are checked for deadlines and will
      not be transferred if they are past their deadlines. Instead these
      messages will be discarded and the next message on the pending queue
      selected.


procedure rels_msg(afonum : integer; port : pt_struct; msgnum : integer);
-- afonum = number of afo in afolist
-- port   = port in afo
-- msgnum = number of message in portlist pending queue

      Releases a message from the AFO's queue, freeing up its memory
      space and reducing the importance of the AFO by removing its activation
      level from the importance calculation. All messages are released from
      the active pool when the AFO's transfer function performs a return to
      its execution loop.


pointer bind_msg(msg : integer;go_name, ddl : string);
-- msg is the message event id returned from evnt_get()
-- go_name is the pointer to string containing name of object to be bound to
-- ddl is the pointer to string containing DDL expected for data object


      Performs name binding action on GO contained within message. This
      function will scan the list of GOs in a message and attempt to find one
      whose name matches the go_name argument. It then checks the that the
      DDL of the GO in the message is compatible with the ddl string given as
      an argument to bind_msg. If both these tests are passed, then
      bind_msg() returns a pointer to the data object. If bind_msg() cannot
      find the named object, then a value of NOTFOUND is returned and if the
      DDL does not match then a value of DDLBAD is returned. DDL is not
      checked if DDL call argument is NULL.


procedure snd_obj(port_id : integer;obj_name,ddl : string;
      data : address;importance : integer;deadline : duration);
```

```
-- port_id is the id of port to send message to
-- obj_name is the pointer to name string for object
-- ddl is the pointer to ddl string for object
-- data is the pointer to data object
-- importance is the importance of object
-- deadline is the relative amount of time from the current time

       This copies the object name string, ddl string, and data object and
       then builds a memory format message containing a list which has a
       single GO with pointers to these objects. Snd_obj sets the to_port_id
       in the message to the port_id specified in the call and sets the reply
       port id to the default port for the AFO. Snd_obj multiplies the
       importance of the message by the global importance of the AFO to obtain
       the activation level and inserts this into the message body along with
       the deadline and time of origination. Snd_obj then calls af_deliver() to
       deliver the message.

       Messages have a deadline specified in relative Ada durations. Messages
       older than their deadlines are discarded. If the deadline is
       specified as zero, then the message has an infinite lifetime.


function chk_msg(port_id : integer)return integer;
-- port_id is the local port id of an afo.

       Checks to see if there are any messages on the pending queue of the port.


procedure ret_afo;
-- Terminates the transfer procedure for an afo.
       Causes a transfer from one afo to the frame afo.
```

4.8 AFO Initialization Procedures

Calls to these procedures are normally generated by the translator program when it translates the frame file into a main program for an AFA load module.

```
procedure fr_init(name : string;stk_adr,sav_addr : address;);
-- Creates frame structure and port table and the "frame" afo whose id = 0.
-- The stack address is store the address of the stack into the afo structure.
-- name is the pointer to name of framework.
-- The save address is the address of the afo's machine state save area.


procedure afo_init(fptr : access;name : string;class,init, prime, import : integer;
       trans : address;globimp,stacksz: integer;
       stk_adr,save_adr : address)
-- Creates the AFO's af_struct, puts it on the list of af_structs,
            allocates its stack and initializes the register state
-- fptr is the pointer to frame structure returned from framinit()
-- name is the name of AFO
-- class is the class of the afo.
-- init is the number of the initialization routine - if NULL then there is no
              initialization routine. Init is called with pointer to af_struct
              for AFO after structure has been created by afoinit
-- prime is the number of the priming function - will be called after message
              delivery to determine if AFO can run. If specified as NULL
              then AFO will be considered primed whenever there are any
              messages queued on any of its ports. Prime is called with a
              pointer to the port structure, which contains the list on
              which the message was placed, as its argument
-- import number of the importance function for AFO - will be called
              whenever a message is delivered to a port. It is called
```

```
                whenever a new message is delivered to a port with a pointer
                to af_struct structure for the recipient AFO and the message
                activation level as its arguments. If import is NULL then the
                message activation level of the arriving message is multiplied
                by the global importance of the recipient AFO and added to the
                current importance of the AFO
-- trans is the address of the transfer function for AFO which is called with
                the address of the af_struct for the AFO
-- globimp is the global importance of AFO
-- stacksz is the stack size to be allocated for AFO. If zero then no stack is
--              to be allocated - this feature is used by the framework to set
--              up its own af_struct while using its existing stack
-- stk_ad is the address of the stack space allocated in the higher level routine
-- save_ad is the address of the save area that is allocated for the machine
state.

      Creates the afo under frame fr_ptr and also creates the std_port for the afo.
Also creates the stack for the afo.


procedure port_cr(afonum : integer;name : string)return integer;
-- afonum = afo id
-- name is the name of the port in the form "afo/port"

      Creates an input port on the AFO and places the port name in the form
      afo/port into the framework port table. Returns the id into the
      port table.  This is usually called by the main program during the
      creation of the AFOs.


function port_sur(remote,frame,local:string)return integer;
-- not fully debugged yet so not implemented.
-- remote is the name of remote port in form afo/port
-- frame is the name of remote framework
-- local is the name of local port in form afo/port

      Creates an entry in the port table indicating that all messages
      destined for the remote port are to be sent to the local port.
      If entries for either the local or remote port are not in the port
      table they are entered in this table. This routine returns the id of
      the remote port. This routine is usually called by main() during AFO
      initialization.


procedure af_run;
-- is called to start the process of running the AFOs. It looks
--           to see which AFO has the highest importance and runs that AFO */
-- fptr is the pointer returned from framinit
```

4.9 Framework Systems Procedures

```
function af_deliver(msg:msg_struct)return integer;
-- delivers messages and causes a task swap if recipient AFO
--                is more important than the executing AFO if flag is set
-- msg is the message


tsk_init(afo : access;trans : address; stack : integer)
-- initializes AFO task state - calls assembly
--        language routine af_swinit()
-- afo is the pointer to AFO's af_struct
-- trans is the pointer to transfer function
-- stack is the stack size desired
```

```
af_swap(), af_swinit(), and af_swsize() are assembly routines which have to be
recoded for each new machine.


procedure af_swap(old,new:address);
-- saves register state of current process and starts
--       execution of new process
-- old is the area to save state of current process
-- new is the location of state of new process


procedure af_swinit(savarea, pc, stackptr:address);
-- initializes save area for AFO
-- savarea is the address of the save area for register state
-- pc is the address of the transfer function
-- stackptr is the address of the bottom of AFO's stack


function af_swsize return integer;
-- returns amount of space needed for save area


procedure af_collect;
--is a routine which is called every time a call is made to the
--           framework from an AFO and within the afo execution loops. It is
--           usually NULL but is used in some installations to cause the
--           collection of messages from the operating system and their
--           delivery. Notyet implemented


function fr_ptr return access;
-- get pointer to frame structure

               Returns NIL if frame structure not yet created.


procedure afi_off;
-- assembly language routine to turn interrupts off. Not yet implemented


procedure afi_on;
-- assembly language routine to turn interrupts on. Not yet implemented.
```

4.10 List Management Calls

4.10.1 Introduction

The List Management System is a set of subroutines which provide list functions. They are specified here in terms of the Ada language. The system is designed for simplicity and fast execution.

4.10.2 The Structure

Essentially, the structure consists of a header node pointing to a doubly linked list of nodes which contain user data objects as well as other information. Each list of nodes can contain multiple types of user data objects. Below is an illustration of the LMS structure.

```
        Header Node  ==>        Node0  <==>  Node1  <==>  Node2 ...
```

4.10.3 Using Access/Task Types as Data Objects

If an access/task type is passed as a parameter to an LMS function then the contents of the access/task type and not the contents of what it is pointing to, will be stored on the list. To store what is pointed to by an access type, use the ptr.all format where ptr is the access type.

4.10.4 LMS Routines

The routines described here are only those available to the user. Please note that the type headtype is a pointer to the header node.

```
lcreat          : headtype
```

This function creates a header node that points to the list of nodes that contain the data objects. This header must be created in order to perform any list manipulations. So this function returns a pointer that will eventually point to the list of nodes that contain the data objects.

```
lwrite(header,n,addr,size) header : headtype;addr : address;n,size : integer
```

This procedure allocates space for the user-defined object and stores it in node n of the list pointed to by the header. By using n as 0, the insertion of the object is made at the beginning of the list (this can also be done by using AT_START). To insert at the end of the list pointed to by the header, set n to -2 or to the length of the list (or you can use AT_END). For example, if there are 5 nodes in the list then setting n to 2 causes insertion of a node with the object between the second and third nodes (the nodes are numbered from zero and so the length of a non-empty list is one greater than the number of the last node). So, in this example, the insertion will be made just before node number 2 (the third node). Note that the address (addr) passed is the address of the object and the size passed is the size of the object. The size can be easily passed by using the size attribute but the address of the object cannot be passed by using the address attribute in standard Ada. Consequently, an address variable must be set equal to the object'address and then that variable can be passed. See the example at the end of this manual.

```
lread(header,n,addr,size)        (same parameters as lwrite)
```

This procedure returns the object stored in the nth node of the list pointed to by the header. Here, the exact node number must be used (i.e between 0 and the length - 1 of the list). For example, if thje header contained 5 nodes (numbered 0 to 4) then only the node numbers 0 - 4 could be used; otherwise an exception would be raised (see LMS Errors). The address (addr) of the object to be filled is passed in a variable form (see lwrite), and the size of the same object is put in the attribute format (i.e 'size).

```
lrewrite(header,n,addr,size)     (same parameters as lwrite)
```

This procedure replaces the nth node of the list pointed to by the header with the new node whose address is addr. Note that both the concerned nodes must have the same size. This is alterable - see source code.

```
lgetaddr(header,n)               header : headtype; n : integer;
```

This function returns the address of the DATA in the nth node in the list header. This is useful for accessing the data itself and not just a copy of it.

```
llength(header) : integer        header : headtype
```

This integer function returns the number of nodes in the list pointed to by the header. For example, if the header pointed to a list with the nodes numbered 0 to 6 then llength would return 7.

```
ldel(header,n)            header : headtype, n : integer
```

This procedure deletes the nth node from the list pointed to by the header. Again, only the node numbers (numbered from 0 to llength - 1) can be used otherwise errors will be raised.

```
lempty(header)          header : headtype
```

This procedure removes every node from the list pointed to by the header. As a result, all the data objects are freed and henceforth inaccessible.

```
lkill(header)                 header : headtype
```

This procedure removes every node pointed to by the header AND removes header too. So, this list then becomes inaccessible and header = null (header is de-allocated).

```
lcopy(head1,n1,head2,n2)        head1,head2 : headtype, n1,n2:integer
```

This procedure copies the n1th node of the list pointed to by head1 to the n2th node in the list pointed to by head2. Again, the correct node numbers must be used. Note that an insertion into the list head1 is made and no deletions occur.

```
leq(head1,n1,head2,n2)          (same parameters as lcopy)
```

This boolean function compares the contents of the data object contained in the n1th node of the list pointed to by head1 with the contents of the data object stored in the n2th node which, itself, is contained in the list pointed to by head2. If the two data objects are identical in every respect then the value true is returned; otherwise, false is returned.

4.10.5 Setting Vital Global Constants in LMS

There are two constants that can be set in the LMS source code : MAX_SIZE and DEF_SIZE. MAX_SIZE is the maximum size (in BYTES) of the largest data object that the user will be using. At present it is set to 1Kb (1024 Bytes). To change it, just enter the source code and modify its value in the beginning of the package specifications. The other constant, DEF_SIZE, is the default size (in bytes)of the data object stored on the list nodes. At present, it is set at 100 bytes and can be changed in the same way as MAX_SIZE. Note that DEF_SIZE should ALWAYS be less than or equal to MAX_SIZE.

4.10.6 LMS-Defined Errors

Apart from the standard Ada error exceptions (such as STORAGE_ERROR), there are five others that are generated by LMS:

ERROR1 :	List node index out of bounds. Occurs when a non-existent node number is accessed.
ERROR2 :	Deleting from a null list. Occurs when a node is deleted from an empty list.
ERROR3 :	Reference count overflow. Occurs when the reference count of a node becomes too large and causes an arithmetic overflow.
ERROR4 :	Comparison of lists with different lengths. Occurs when a comparison is made between two lists of different lengths.
ERROR5:	Comparison or assignment of objects with different sizes Occurs when a comparison/assignment is made between two data objects of different sizes.

Note that LMS does relatively little consistency checking.

4.10.7 Purpose of LMS

LMS (Ver 1.0) was a generic package and therefore required multiple instantiations which, in cases with several hundreds of data objects, can become very tiresome. Another side effect of this is the almost exponential increase in compilation time due to the multiple instantiations. Consequently, LMS (Ver 3.0) was created by popular demand. Version 3.0 requires only one instantiation because it is a package and also takes MUCH less compilation time but is limited by the two constants described earlier.

4.10.8 Example Program

The following program illustrates the use of LMS by computing a few numbers in the Fibonacci sequence. As a background, the Fibonacci sequence is defined with the following recurrence relation:

```
        Fib(n) = Fib(n-1) + Fib(n-2)    where Fib(0) = Fib(1) = 1
So, the next three Fibonacci numbers would be 2, 3 and 5.
```

Here is the sample program that computes a few Fibonacci numbers and their corresponding squares:

```
with lms,system,text_io; use text_io,system,lms;
-- Fibonacci example
procedure ex is

        type node is record
                number : integer;
                numsqr : integer;
        end record;

        package pos_io is new integer_io(integer); use pos_io;

        ndmnd2 : node;
        ad : address := nd'address;
        fib : headtype
        vall, val2, i, newval, newsqr : integer;

        begin
                fib := lcreat;
                vall := 1;
                val2 := 1;
                nd.number := vall;
                nd.numsqr := val2;
                lwrite(fib,AT_START,ad,nd'size)
                lwrite(fib,AT_END,ad,nd'size)

                for i in 2..9
                loop
                        newval := vall + val2;
                        newsqr := newval * newval;
                        nd.number := newval;
                        nd.numsqr := newsqr;
                        lwrite(fib,i,ad,nd'size);
                        vall := val2;
                        val2 := newval;
                end loop;

                for i in 0..9
                loop
                        lread(fib,0,ad2,nd2'size);
                        put(nd2.number);
                        put(nd2.numsqr);
                        put_line(" <--");
                        ldel(fib,0);
                end loop;
```

```
      end ex;
```

Here is the output of the Fibonacci example:

```
 1      1  <--
 1      1  <--
 2      4  <--
 3      9  <--
 5     25  <--
 8     64  <--
13    169  <--
21    441  <--
34   1156  <--
55   3025  <--
```

LMS provides two new structures to the user: dynamically allocated "list elements" and indefinite-sized "lists". Lists can contain only list elements, and list elements exist only as parts of lists. When a list element is no longer contained on any list it will be deallocated. A list element may be a member of several lists simultaneously.

Variables referring to lists should be of type "headtype", where headtype is defined in the package specifications of LMS.ADA. The headtype refers to the type of header node associated with the list. For a more detailed discussion, please see the LMS manual.

4.11 Procedures for Handling Generalized Objects

Generalized Objects (GOs) provide a uniform and homogeneous representation of complex data structures for the specific purpose of transmitting objects between AFOs. These have not yet been implemented in AFA.

4.12 Structures

This section of the manual describes the structure of objects such as messages and lists used in AFA.

4.12.1 Lists

A list is a pointer to a data structure which is a list header whose fields are:

name	type	description
size	integer	size in bytes of list element
length	integer	number of elements in list
start	access	pointer to first node header
endd	access	pointer to last node header
lastref	integer	node number last referenced
lastptr	access	pointer to the node header last referenced.

The list is actually a list of node headers which point to the data objects themselves. These node headers contain the forward and backward chaining pointers for a list. The structure of the node header is:

name	type	description
link	access	pointer to next node header in list - NULL if last entry
plink	access	pointer to previous header in list - NULL if first entry
data	datatype	pointer to data object.
size	integer	size of the object data in bytes.

Lists have forward and backward pointers for rapid access to list elements. Access to a specific list element is made from the beginning, end, or last referenced point in the list, whichever is closest.

4.12.2 Activation Framework Objects

All the information pertaining to an AFO is contained in a structure known as an af_struct. The format of an af_struct is as follows:

name	type	description
af_name	string	name of AFO
af_idnum	integer	identifying number for AFO
af_class	integer	used for differentiating uset from system AFOs
af_gimp	integer	global importance
af_cimp	integer	current importance
af_primd	integer	0 = not ready to execute; 1 = ready to execute
af_portl	headtype	pointer to port list - see section 7.3
af_init	integer	pointer to initialization routine
af_prime	integer	pointer to priming function routine
af_impor	integer	pointer to importance calculation routine
af_trans	address	pointer to transfer function routine
af_stack	address	pointer to stack
af_save	address	address of area in which to save last executing register status of the AFO.

Af_structs are contained on a list which is pointed to by a pointer contained in the structure frm_struct, whose format is:

name	type	description
frm_name	string	name of framework
frm_afl	headtype	pointer to list of af_structs
pt_tbl	headtype	pointer to the port table (see section 7.5)
frm_cafo	integer	id of current afo

4.12.3 Ports

AFO ports are kept on a list which is pointed to by the portlist entry in the afo_struct. Each port has the following structure:

name	type	description
pt_name	string	name of port
pt_pend	headtype	pointer to list of pending messages (see section 7.4)
pt_num	integer	local port id number
pt_afo	integer	id of its afo.

4.12.4 Messages

Messages in memory (i.e. prior to serialization) are kept in the following format:

name	type	description
msg_toid	integer	index of the addressee in the port table
msg_frid	integer	index of the reply port in the port table
msg_type	integer	message type
msg_gol	headtype	pointer to a GO.
msg_act	integer	activation level of message
msg_dead	time	time at which message is to be deleted from system
msg_sent	time	time message was sent

Messages are encoded in a serial format when they are transmitted between processors as described in section 7.7.

4.12.5 Port Table

For speed of access all ports are referenced by an ID number, rather than by name. The ID is an index into a port table, each entry in which has the format:

name	type	description
ptb_name	string	name of port
ptb_ptr	integer	id of port
ptb_afo	integer	id of owning afo

5. VALIDATION AND VERIFICATION USING MONTE CARLO SIMULATION

There are three basic parts to the validation and verification system developed for this project, a test generator, an EFG "simulator", and a test evaluator. Along with these parts a test generation metaknowledge language was developed to drive this system. Another type of metaknowledge was identified but has not yet been explored, this is test configuration knowledge.

5.1 Test Generation

A test generator was designed and implemented to take the metaknowledge syntax and create a given number of test cases. This system was implemented in common LISP and is running on Gold Hill Common LISP for a PC, and in Ibuki Common LISP on an Encore. The basic interaction with the generation system is the creation of the metaknowledge file. This file contains the information needed in the generation of test cases. Once this file is created the user then enters common LISP and loads the generation program. The command (start input-file num output-file) can be issued. The system takes the file input-file and reads the entire file in. It then parses the information into an internal representation. The system then attempts to generate num test cases. It also generates any partial and full test cases and merges these into the list of test cases. These cases are then written to the output-file.

The system will randomly generate test cases within the given conditional relations and in a proper order. The strategy the system uses to generate a test is fairly simple. First pick an input name for which no input has been generated. Randomly choose a value for this input given its ddl. Check the conditional relationships to make sure that this value is consistent with the current inputs generated so far. If the case is compatible pick a new name to work on, otherwise try to fix the current case to match the conditional relations. If no fix can be found abandon this value and try again. This process continues until an entire test case has been generated. The current system tries to make test cases using all the inputs. It also only supports two ddl types, string and boolean.

5.1.1 Test Generation Metaknowledge Syntax

```
Input/Output/Intermediate Node Value Metaknowledge:

     (name ddl i (x..y %)(a..b %)...|(c..d %)(e..f %)...)

     name contains no spaces
     ddl is description of the data using the ddl syntax described as
     in the EFG syntax
     i is the flag for type of node Info
            I:    input
            O:    output
            none: intermnediate
     x,y,a,b,c,d,e, and f are integers, real numbers, or characters
     or can be replaced by a single discrete value
     % (optional) argument to specify the probability of
     the particular range or discrete value
     | is used to separate the different parts of the ddl

     Examples:

          (digits I I (0..9))
          (color S I (red green blue yellow))
          (temp R I (25.0..35.0 0.90))
          (step I I (1..5 0.10)(6..10 0.75)(11..20 0.10)(21..30 0.05))
          (size S I (large 0.55)(medium small 0.20))
          (letters C I (a..d 0.10)(e..q 0.90))
          (flag B O (True 0.70))
          (on B)
          (jtids FF I (0.0 0.5)(1.0 0.5)|(0.0 0.7)(1.0 0.3))


Input Co-occurrence Metaknowledge:
```

```
(If x then_never y)
(If x then y)
```

Where x and y can be any combination of inputs using a name specified by the format for individual values, any mathematical operator (=,>,<,etc.), and any combination of these forms using boolean operators (and,or,not). Presence of a value is denoted by not using any mathematical operator.

Examples:

```
(If (color = red) then_never (temp > 15.0))
(If color then_never digit)
(If ((color <> red) and (temp < 15.0)) then_never (step > 5))
(If flat then_never gloss)
(If gloss then_never flat)
```

Input Ordering Metaknowledge:

```
(x before y)
(x after y)
(x not_before y)
(x not_after y)
```

Where x and y can be any names specified by the individual input syntax

Examples:

```
(color before step)
(gloss not_after flat)
(flat before color)
```

Conditional Relation and Test Case Metaknowledge:

```
(if x then y)
(if x then_never y)
```

Where x and y can be any combination of inputs, outputs, or intermediate value names and any logical (and, or, not) or mathematic (=, <, >, etc.) operators applied to them. A full test case specifying all inputs as x and the expected output values as y. A partial test case being where only some of the inputs are specified.

Examples:

```
(if ((color = red) and (step = 10)) then (temp = 100))
(if ((((color = red) or (temp > 100)) and (step = 1..10)) and
    not(digits = 5)) then ((flat = 5) and (not (gloss > 7))))
```

Definitions:

Full Test Case:
A full test case has no inputs left unspecified. It has no boolean conjunctions and has only inputs in the If clause. No inputs are allowed in the Then clause.

Partial Test Case:
A partial test case allows some inputs to be left unknown. Only inputs are allowed in the If clause but no inputs are allowed in the Then clause.

```
Full example:

     Inputs:

          (a S i (high med low))
          (b B i)
          (c S i (summer spring winter fall))

     Outputs:

          (advance B o)
          (retreat B o)

     Input Co-occurrence:

          (if (a = high) then ((c = spring) or (c = fall)))
          (if (b = true) then ((a = med) or (a = low)))

     Input Ordering:

          (a before b)
          (c not_before a)

     Full Test Cases:

          (if (((a = high) and (b = false)) and (c = spring)) then
              (retreat = true))
          (if (((a = med) and (b = true)) and (c = summer)) then
              (advance = true))

     Partial Test Cases:

          (if ((a = low) and (c = summer)) then (retreat = true))
```

5.1.2 Test Configuration Metaknowledge

This is a new form of metaknowledge which was uncovered in the design of the test generator. What this knowledge specifies is exactly how to use the test generation metaknowledge. The test generation metaknowledge specifies what the test space looks like, but the configuration knowledge tells you how to test that space.

This knowledge is currently restricted to the number of test cases which are to be generated. There are many other possibilities which could be allowed. There are three broad categories of constraint knowledge which have been identified, types of tests, number of tests, and input timing. The types of tests to be generated includes the description of how to use the ranges and probability distributions. These could be fully random within the given bounds, make sure each part of the probability distribution has at least one value generated for it, just use the average values, etcetera. The number of tests to be run could include a user input number or a number of test cases to obtain a certain probability of coverage. The timing knowledge has information about how long of a delay exists between inputs, which could be an average number or a random delay.

This knowledge could be very important to testing a system. Just the knowledge about the test space is not sufficient to "know" how to test the space accurately. It is important to find out where most of the testing time should be spent in order to effectively test a system.

5.2 EFG Simulation

When this project started out a separate simulation module was going to be designed and built using Simscript. After some initial work with the design of this system it was determined that this was not the best way to go about the creation of an EFG simulator. There were several factors which led to the abandoning of Simscript. The first reason was the complexity involved with the data structures needed to simulate an EFG. What is needed to capture the richness of the EFG syntax is a list data structure. This data structure would be difficult to duplicate using Simscript. The second factor involved was the fact that AFA is very close to the system which would be needed as a simulator. Using AFA saved on time since a majority of the environment would already be designed. Another advantage to using AFA is that the tests would be performed on the actual code and not a simulation of the code. A possible future advantage would be the testing of the code in a parallel environment. This would allow for the parallelism to be tested and would also increase the speed at which tests could be conducted. All these reasons outweighed any reasons for staying with a Simscript simulation of an EFG.

The simulation environment is currently identical to AFA. This environment will be expanded to include many other enhancements which would keep track of the message flow. This would allow for the examination of internal as well as external nodes and the values which are associated with these nodes. Overall this approach has the benefit of being flexible enough to handle many more cases without the need to duplicate the code being written for AFA in Simscript.

5.3 Test Evaluator

A test evaluator was designed and implemented to take a single test case and the output generated and determine if the results were within the rules specified by the metaknowledge syntax. This system was implemented in common LISP and is running in Ibuki Common LISP on an Encore. To use this system first LISP must be entered and the evaluator must be loaded into memory. The command (begin input-file output-file) can be issued. The system takes the file input-file as the input of the test case and output file as the the results of running the test case. A file called SYS.OUT is used to pass information between the generator and the evaluator. The evaluator currently runs two different checks on the test case. The first is a ddl check on the results of the run. This will return any inconsistencies in the data. The second is to check any rules which are applicable to make sure that the values are within specifications.

5.4 Sample Run

This is a script file which shows the test generator and test evaluator in operation. This file was created on an Encore which was running Ibuki Common LISP.

First a file (NEW.DAT) with the testing metaknowledge is entered. Then, file OUT2.DAT is generated which contains a set of test case inputs. Finally, the test evaluation is carried out by comparing the results of the runs on the generated test cases with the testing metaknowledge and any discrepancies are reported.

Testing Metaknowledge

```
Script started on Fri Jan  5 11:33:23 1990
Dave>> cat NEW.DAT
(eq_mode_health b i)
(pilot_not_busy b i)
(health b i)
(jtids b i)
(get_previous_waypoint b)
(get_unaided_sol b)
(get_ecm_env b)
(get_lead_map_error b)
(get_alpha_check b)
(get_wm_map_error b)
(get_eo_radar_des b)
(q s o ((1 2 3 4 5 6 7 8) (1 2 3 4 5 6) (1 2 3 4)()))
(eq_mode_health before pilot_not_busy)
(eq_mode_health before health)
(pilot_not_busy before jtids)
(health before jtids)
(if ((eq_mode_health = true) and (pilot_not_busy = true) and
     (health = true) and (jtids = true))
 then (q = (1 2 3 4 5 6 7 8)) )
(if ((eq_mode_health = true) and (pilot_not_busy = true) and
     (health = false) and (jtids = false))
 then (q = (1 2 3 4 5 6)) )
(if ((eq_mode_health = true) and (pilot_not_busy = true) and
     (health = false) and (jtids = true))
 then (q = (1 2 3 4 5 6)) )
(if ((eq_mode_health = true) and (pilot_not_busy = true) and
     (health = true) and (jtids = false))
 then (q = (1 2 3 4 5 6)) )
(if ((eq_mode_health = true) and (pilot_not_busy = false))
 then (q = (1 2 3 4)) )
(if (eq_mode_health = false)
 then (q = ()) )
```

Test Case Generation

```
Dave>> ibcl
IBUKI Common Lisp  release 01/01  October 15, 1987

This software is provided by IBUKI pursuant to a written license agreement
and may be used, copied, transmitted and stored only in accordance with the
terms of such license.

;; Copyright (c) 1987, 1986 IBUKI  All rights reserved.
;; Copyright (c) 1986, 1985, 1984 T. Yuasa and M. Hagiya  All rights reserved.

For more information: (describe 'copyright) or (describe 'acknowledgements)
Loading init.lsp
Warning:
ED is being redefined.
Loading parse.lsp
Finished loading parse.lsp
Loading before.lsp
Finished loading before.lsp
Loading gen.lsp
Finished loading gen.lsp
Loading evl.lsp
Finished loading evl.lsp
Finished loading init.lsp

>(start "NEW.DAT" 12 "OUT2.DAT")
T

>(bye)
Bye.

Dave>> cat OUT2.DAT
#BegTest
EQ_MODE_HEALTH FALSE
HEALTH FALSE
PILOT_NOT_BUSY FALSE
JTIDS FALSE
#BegTest
EQ_MODE_HEALTH TRUE
HEALTH TRUE
PILOT_NOT_BUSY TRUE
JTIDS TRUE
#BegTest
EQ_MODE_HEALTH TRUE
HEALTH FALSE
PILOT_NOT_BUSY FALSE
JTIDS FALSE
#BegTest
```

```
EQ_MODE_HEALTH TRUE
HEALTH TRUE
PILOT_NOT_BUSY FALSE
JTIDS TRUE
#BegTest
EQ_MODE_HEALTH FALSE
HEALTH TRUE
PILOT_NOT_BUSY FALSE
JTIDS TRUE
#BegTest
EQ_MODE_HEALTH FALSE
HEALTH TRUE
PILOT_NOT_BUSY FALSE
JTIDS FALSE
#BegTest
EQ_MODE_HEALTH TRUE
HEALTH TRUE
PILOT_NOT_BUSY FALSE
JTIDS FALSE
#BegTest
EQ_MODE_HEALTH FALSE
HEALTH TRUE
PILOT_NOT_BUSY TRUE
JTIDS TRUE
#BegTest
EQ_MODE_HEALTH FALSE
HEALTH TRUE
PILOT_NOT_BUSY TRUE
JTIDS FALSE
#BegTest
EQ_MODE_HEALTH TRUE
HEALTH FALSE
PILOT_NOT_BUSY TRUE
JTIDS FALSE
#BegTest
EQ_MODE_HEALTH TRUE
HEALTH TRUE
PILOT_NOT_BUSY TRUE
JTIDS FALSE
#BegTest
EQ_MODE_HEALTH TRUE
HEALTH FALSE
PILOT_NOT_BUSY TRUE
JTIDS TRUE
###
```

Test Case Evaluation

```
Dave>> ibcl
IBUKI Common Lisp  release 01/01  October 15, 1987

This software is provided by IBUKI pursuant to a written license agreement
and may be used, copied, transmitted and stored only in accordance with the
terms of such license.

;; Copyright (c) 1987, 1986 IBUKI  All rights reserved.
;; Copyright (c) 1986, 1985, 1984 T. Yuasa and M. Hagiya  All rights reserved.

For more information: (describe 'copyright) or (describe 'acknowledgments)
Loading init.lsp
Warning:
```

```
ED is being redefined.
Loading parse.lsp
Finished loading parse.lsp
Loading before.lsp
Finished loading before.lsp
Loading gen.lsp
Finished loading gen.lsp
Loading evl.lsp
Finished loading evl.lsp
Finished loading init.lsp


>(begin t1 o2)
the test case to evaluate
((EQ_MODE_HEALTH TRUE) (HEALTH FALSE) (PILOT_NOT_BUSY TRUE)
 (JTIDS FALSE))
the results of the test case
((Q (1 2 3 4 5 6)))
T

>(begin t1 o3)
the test case to evaluate
((EQ_MODE_HEALTH TRUE) (HEALTH FALSE) (PILOT_NOT_BUSY TRUE)
 (JTIDS FALSE))
the results of the test case
((Q (1 2 3 4)))
The results of this test case do not conform to the rules
T

>(begin t1 o5)
the test case to evaluate
((EQ_MODE_HEALTH TRUE) (HEALTH FALSE) (PILOT_NOT_BUSY TRUE)
 (JTIDS FALSE))
the results of the test case
((Q XYZ) (RETREAT SPRING) (GUMBO FALSE) (ADVANCE TRUE))
Q has a value inconsistent with its definition
No definition for RETREAT
No definition for GUMBO
No definition for ADVANCE
The results of this test case do not conform to the rules
T

>(begin t2 o4)
the test case to evaluate
((EQ_MODE_HEALTH FALSE) (HEALTH FALSE) (PILOT_NOT_BUSY TRUE)
 (JTIDS FALSE))
the results of the test case
((Q NIL))
T

>(begin t3 o2)
the test case to evaluate
((EQ_MODE_HEALTH TRUE) (HEALTH TRUE) (PILOT_NOT_BUSY TRUE)
 (JTIDS FALSE))
the results of the test case
((Q (1 2 3 4 5 6)))
T

>(begin t4 o1)
the test case to evaluate
((EQ_MODE_HEALTH TRUE) (HEALTH TRUE) (PILOT_NOT_BUSY TRUE)
 (JTIDS TRUE))
the results of the test case
((Q (1 2 3 4 5 6 7 8)))
T
```

```
>(begin t5 o3)
the test case to evaluate
((EQ_MODE_HEALTH TRUE) (HEALTH TRUE) (PILOT_NOT_BUSY FALSE)
 (JTIDS TRUE))
the results of the test case
((Q (1 2 3 4)))
T

>(bye)
Bye.
Dave>> exit
Dave>>
script done on Fri Jan  5 11:39:04 1990
```

6. VALIDATION AND VERIFICATION OF EXPERT SYSTEMS USING PETRI NETS

This investigation centers on determining how the body of theory which currently exists for the analysis of Petri Nets can be applied to the verification and validation of expert systems. Previous research has shown that it is possible to express expert systems applications using a highly parallel representation called an evidence flow graph. Further, it has been shown that this evidence flow graph representation can be expressed as an equivalent Petri Net. Since techniques exist to analytically quantify the behavior of Petri Nets, it seems obvious that these techniques should be applicable to analyzing the behavior of evidence flow graphs, and thus, should be applicable to analyzing the behavior of expert systems.

In order to explore this idea, the methodology shown in figure 6.1 was adopted. First, an expert system was selected for analysis and was translated into its evidence flow graph representation. Second, the evidence flow graph nodes and arcs were translated into places and transitions in such a manner that a decision-free Petri Net was formed. Next, this Petri Net was subjected to analysis in order to determine if the resultant network exhibited any characteristics which would be undesirable. In particular, the characteristics of concern were:

Boundedness.	If a system is not bounded, a condition may exist in which demands for system resources (memory, queues, I/O) become unlimited.
Reachability.	If a system contains states which are not reachable during normal processing, and thus untestable, then system behavior may be unpredictable if one of these states is entered as a consequence of a fault.
Safeness.	Certain conditions in a system may be considered undesirable or "Unsafe" if, when the system enters such a state, there is the possibility for catastrophic failure.

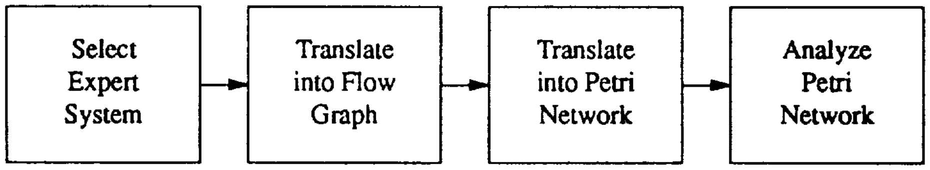


Figure 6.1 Methodology of Investigation

In the following sections, the details of translation and analysis are described. The next section discusses the manner in which an evidence flow graph can be translated into a decision free Petri Net. Section three describes the algorithm used to analyze the behavior of the resultant Petri Net. Section four presents a summary of the findings of this investigation.

6.1 Evidence Flow Graph Translation

An evidence flow graph, shown in figure 6.2, is a directed graph which is used to explicitly model the control structure which underlies the expert system being represented. If, for example, the expert system being represented was a production system, each node would represent a rule and each arc would represent the effect of one rule on another.

Using an evidence flow graph representation, a node (rule) is considered enabled if the logical relation specified by the node's preconditions is satisfied. Once enabled, an evidence flow graph node fires by consuming data items presented on its input arcs, generating a result according to the algorithm associated with the node, and passing the result along one or more output arcs to nodes which need the result as input.

While the translation between an evidence flow graph and a Petri Net may seem quite straightforward based on the above description, there two problems which prevent a direct translation. These problems can be categorized as the multiple output problem and the multiple input problem.

The multiple output problem is illustrated in figure 6.3. The evidence flow graph shown on the left side of the figure is intended to operate by placing a token into both nodes 2 and 3 once a token is received and processed by node 1. If each node is replaced directly by a transition and place of a Petri Net the intended result is not achieved. Rather, a token arrives at the input transition causing it to deposit a token into place 1. At this

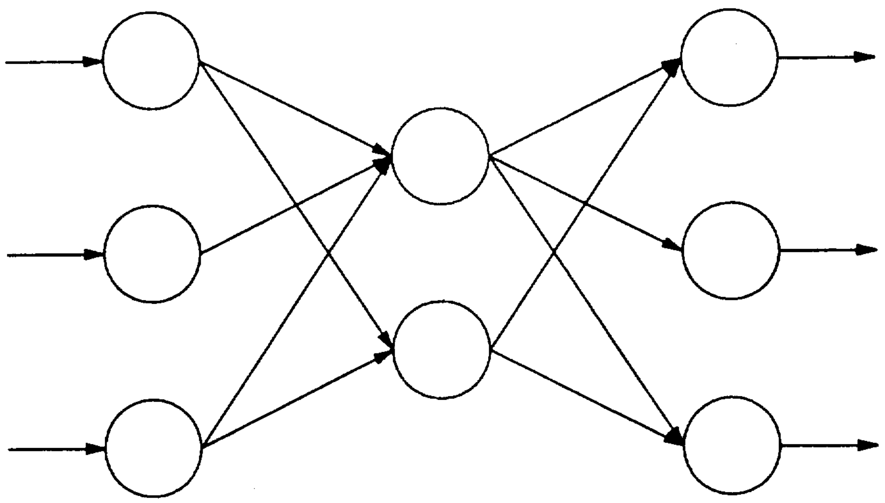


Figure 6.2 Example of an Evidence Flow Graph

point, place 1 only contains a single token and thus, it can only enable one of the two transitions on its output. The Petri Net is forced to "decide" which place should receive the token (when, in fact, both places 2 and 3 should get one).

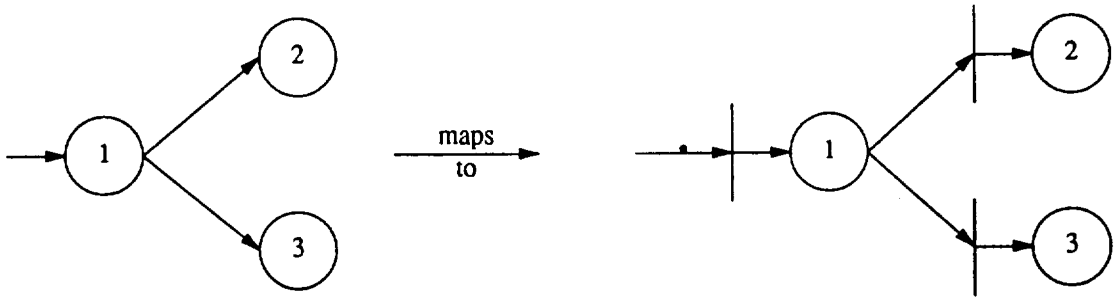


Figure 6.3 The Multiple Output Problem

The multiple output problem can be solved by simply reversing the order of the places and transitions of the Petri Net. As shown in figure 6.4, each evidence flow graph node gets translated into a Petri Net place followed by a transition. Since transitions can generate multiple tokens when they fire, this property can be used to distribute tokens to an arbitrary number of output places. Thus, under this translation, the ability of an evidence flow graph node to broadcast to several other nodes is preserved.

The translation proposed for solving the multiple output problem, however, is incomplete since it does not provide a mechanism for implementing arbitrary logical preconditions on the inputs of a node. In fact, the only precondition possible for the translation proposed in figure 6.4 is an OR precondition since any token deposited in the input place from any source will cause it to fire. In an evidence flow graph each node may have a precondition which is a arbitrary logical function of the input arcs of that node. Thus, the translation from an evidence flow graph to a Petri Net must correctly preserve the AND, OR, and AND/OR relationships which exist between the multiple inputs of an evidence flow graph node.

As was the case with the multiple output problem, the multiple input problem is also solved by adopting an appropriate mapping of evidence flow graph constructs into Petri Net components. In order to allow a Petri Net model to faithfully model the manner in which an Evidence Flow Graph handles its inputs, a more complex Petri Net node model must be used. This more complicated model consists of two elements: a conjunct element and an action element. Figure 6.5 shows how these elements are used to create a faithful Petri Net

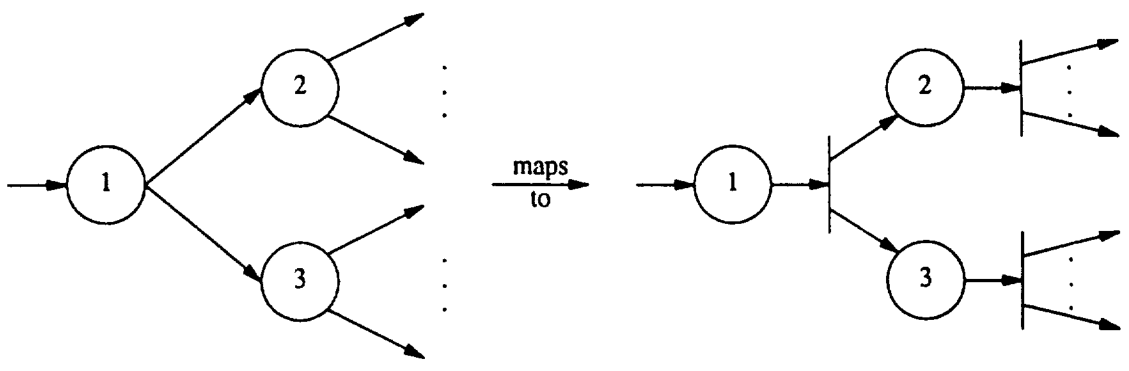


Figure 6.4 Solving the Multiple Output Problem

Model on an evidence flow graph. Preconditions of a rule, as can be seen in the figure, are interpreted as being either "OR" conditions (no interconnecting semicircle) or "AND" conditions (interconnected by a semicircle).

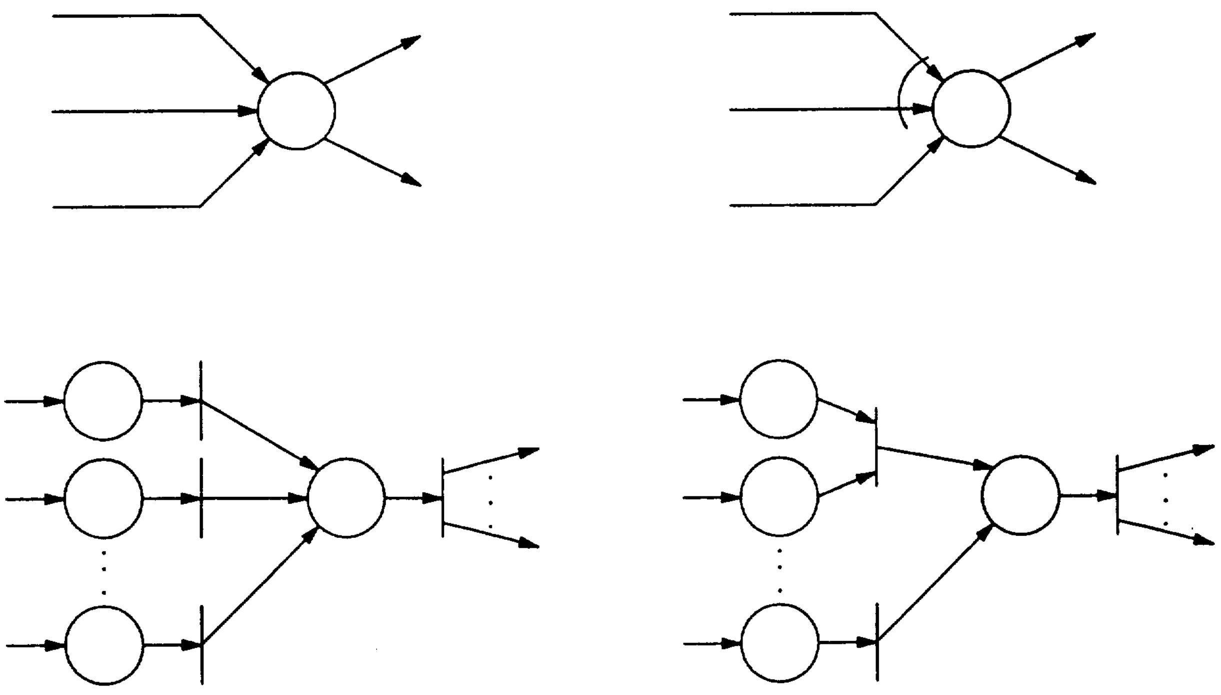


Figure 6.5 Representation of OR, and AND/OR Conjuncts

6.2 Analysis Algorithm

Once an evidence flow graph has been translated into its Petri Net representation, analysis of the translated system may begin. Currently, first-order analysis is done by performing a reachability analysis on the Petri Net under investigation. This analysis can be used to determine boundedness by determining if there are any Petri Net nodes which have token counts which continually grow, it can be used to determine reachability by identifying nodes which are never activated, and it can locate states which have *a priori* been identified as unsafe.

The basic algorithm used for performing reachability analysis is Dugan's algorithm which is given in figure 6.6. Given some initial Petri Net and marking, the algorithm inspects each transition. If a selected transition is enabled, it is fired and a potentially new marking is created. This potentially new marking is compared to all previously encountered markings and, if it is unique, it is added to the reachability tree of the Petri Net. The process continues until all possible markings which are reachable from the initial marking have been processed.

```
Step 1:   Get marking c for analysis
Step 2:   For j = 1...t do
Step 3:   if transition j is enabled by marking c
              then generate new marking $M_{temp}(k)$
              and for each k do
Step 4:   if $M_{temp}(k)$ is not already in the tree,
              then m = m + 1
              $M(m) = M_{temp}(k)$
              c = c + 1
```

Figure 6.6 Dugan's Algorithm

Figure 6.7 illustrates a situation in which translation of an evidence flow graph yields an unbounded Petri Net. In this case, the first place contains a token resulting in its output transition being enabled. When this transition fires, it will place tokens both in the first and second place thus enabling both transitions to fire. After firing, the first place will now contain two tokens and the second place will contain one. Thus, both transitions continue to be enabled and will continue to fire, each time adding one token to the first place. The result is that the number of tokens resident in the first place will grow without bound indicating a potential problem area.

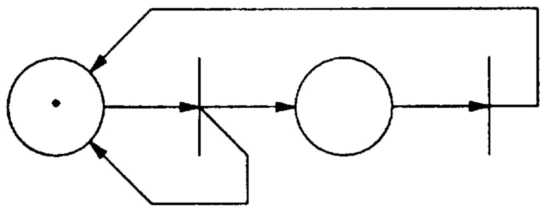


Figure 6.7 Unbounded Network

Figure 6.8 shows a Petri Net and marking which results in unreachable states. In this figure, only places 2 and 3 contain tokens and thus, only transition 2 is enabled. Once transition 2 fires, it a token is put into place 5 which, in turn, enables transition 4. When transition 4 fires, a token is put into place 6 and Petri Net execution is complete. Since there is no mechanism in this Petri Net to put tokens into places 1 or 4, transitions 1 and 3 will never be able to fire. Thus, the nodes and transitions which lie on the path from p1 to p6 are all unreachable.

6.3 Summary of Results

The techniques described in the previous sections have defined how an evidence flow graph can be translated into a Petri Net and how a first-order analysis of the resultant Petri Net may be performed. This first-order analysis is based on the extraction of the reachability tree from a Petri Net with a given marking. Based on this reachability tree, it can be determined if there are states which exist in the Petri Net but are unreachable from a given marking, whether a given marking results in one or more places receiving an unbounded number of tokens, or if state which has been determined *a priori* to be unsafe can evolve from a marking.

While this type of first-order analysis can provide useful insight into the behavior of a system, there are some important limitations of the technique presented. First, the NP-complete nature of the algorithm limits the utility of this technique to Petri Networks with relatively small numbers of places and transitions. For networks of a few dozen nodes, a few hours of runtime on an IBM PC AT class machine are required. For larger systems, the expected limit on network size would be a few hundred places and transitions. Since simple

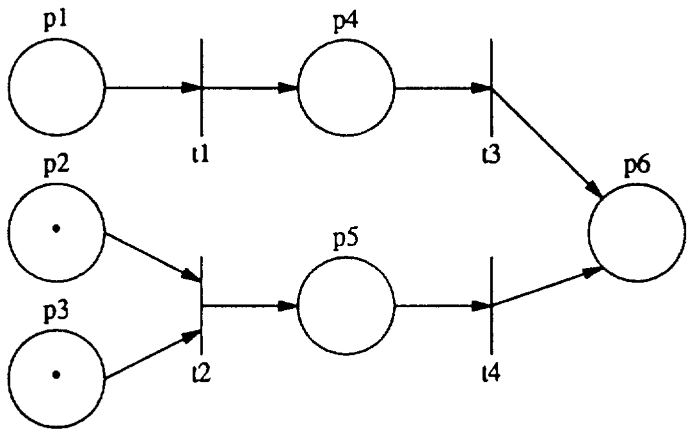


Figure 6.8 Network with Unreachable Nodes

evidence flow graphs translate into moderately complex Petri Nets, evaluating complicated evidence flow graphs would quickly become impractical.

A second problem with the first-order analysis presented is that the models used for transitions and places are rather simplistic. There has been no effort taken to account for common generalizations to Petri Networks such as stochastic transitions or non-decision free places. While these extensions are possible, they add significant complexity to the resultant model and therefore exacerbate the computational burden of network analysis.

6.4 Conclusion

The analysis performed suggests that the ability to analyze evidence flow graphs using Petri Net techniques is severely limited by the computational complexity of the analysis process. While this conclusion is justified based on the experimental results, it may be that the results were biased by the methodology more than any inherent limitation of the idea of using Petri Nets to analyze evidence flow graphs.

The methodology presented in section 6.2 was based on the premise that the evidence flow graph had to be translated into a lower level Petri Net representation prior to analysis. This translation was required to cast the flow graph into a format for which analysis techniques currently exist. It is possible that with an appropriate generalized Petri Net model the evidence flow graph can be analyzed directly, thus substantially reducing the computational burden of the analysis. For such a model to be analyzed, however, new tools and techniques for analyzing this new class of Petri Net would have to be developed. Such an effort may provide additional analysis capability such as a means for detecting race conditions or for multiple processor system validation.

7. CONVERTING CLIPS TO AN EFG

In this section we will discuss the conversion of CLIPS rule bases into Evidence Flow Graphs. The expert system written for the MMU application does not use the full range of CLIPS constructs. In this section, we will first discuss converting the subset of CLIPS that is found in the MMU system and then consider the difficulties in converting unrestricted CLIPS.

7.1 Converting MMU CLIPS to an EFG

We divide the process of converting the MMU CLIPS into an EFG into two stages. This process is illustrated in Figure 7.1. First, the MMU CLIPS rules together with metaknowledge about 'collapsing' are used to 'decollapse' the rules. This process expands the number of conjuncts in the rules, and possibly the number of rules itself. This decollapsing process produces what we refer to as 'single-argument rules'. Single-argument rules are rules that only refer to single-argument or single-valued facts. Such facts consist of a predicate-name and a single value-field. Of course, it is possible to have an expert system in CLIPS that is originally written using only rules which refer to single-argument facts; in this case no decollapsing is required, and the rules can be translated into an EFG directly. In this subsection we will first discuss the principles for the translation of 'single-argument' CLIPS rules to an EFG, then the motivation for the 'single-argument rules' as an intermediate level of representation, and finally the method for decollapsing.

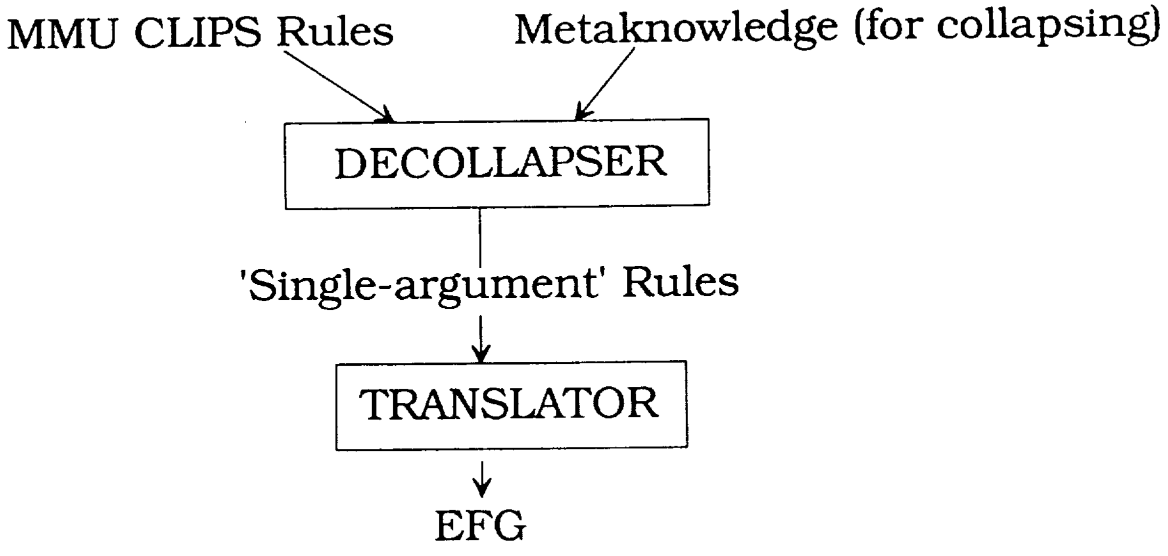


Figure 7.1 Generating an EFG from MMU Clips Rules.

There are four principles for the translation of 'single-argument' CLIPS rules into an EFG:

1) For each rule, there is a RULE NODE.
2) For each value referred to in the antecedent(lhs) or consequent(rhs) of any rule there is a VALUE NODE.
3) There is a link FROM the value node of a value referred to in an antecedent TO that rule node.
4) There is a link FROM a rule node TO to the value node of each of the values referred to in its consequent.

Below are eight rules written in terms of single-valued facts. The nodes of the EFG created by following principles 1 and 2 above are depicted in Figure 7.2. Due to the large number of links, they have not been included in this figure. There would be one link from each rule node to the FAILURE node and one link from each rule node to the SUSPECT node. There would be a link to each of the rule nodes from each of the following input nodes: AAH, GYRO, GYRO-MOVEMENT, SIDE-A, SIDE-B, RHC-ROLL, RHC-PITCH, RHC-YAW, THC-X, THC-Y, THC-Z. In addition, there would be a link from each of the following input nodes, VDA-A-L1, VDA-A-R2, VDA-A-L3, VDA-A-R4, to each of the odd-numbered rule nodes, R1, R3, R5, R7<, and from each of VDA-B-L1, VDA-B-R2, VDA-B-L3, VDA-B-R4 to each of the even-number rule nodes, R2, R4, R6, R8.

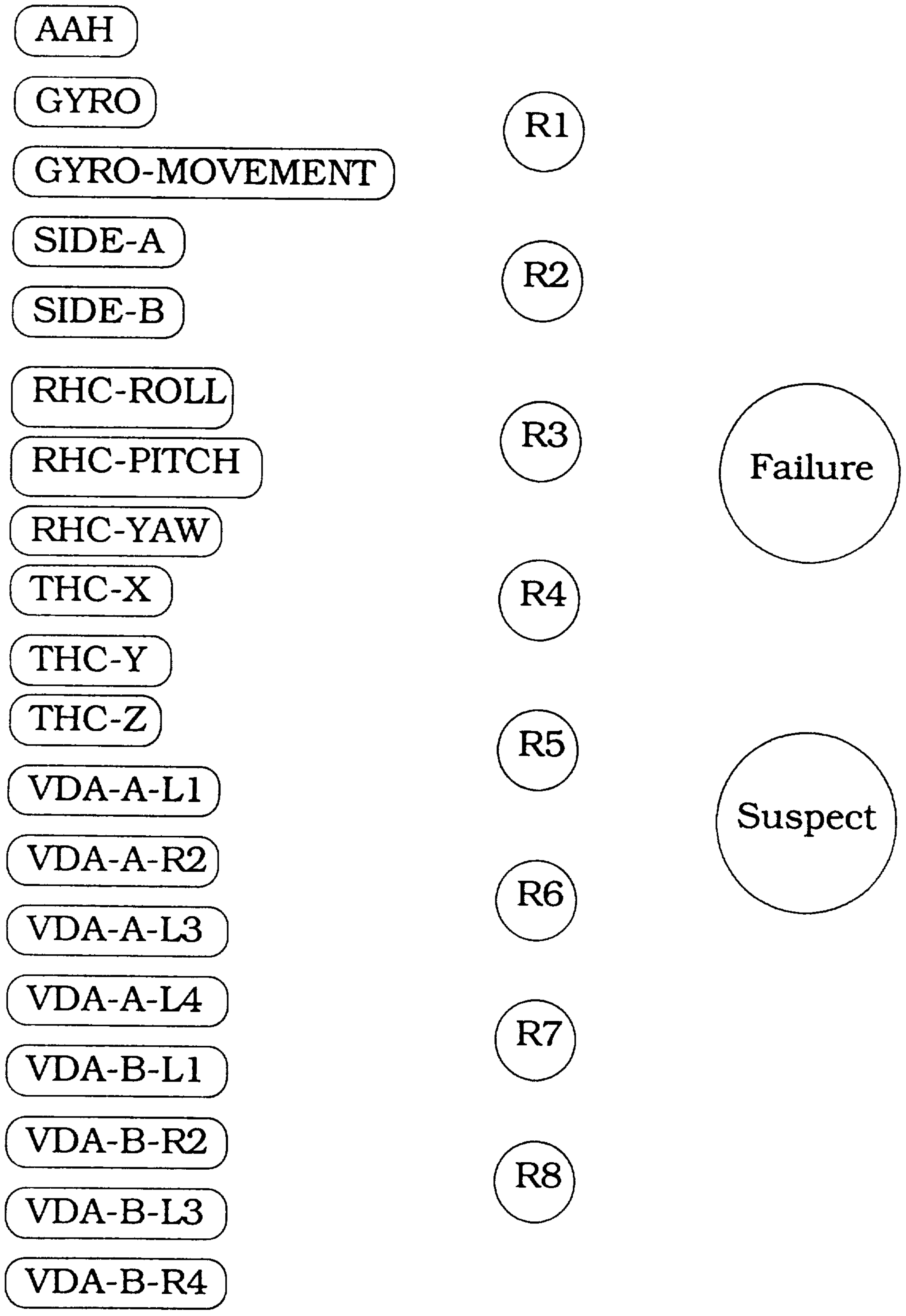


Figure 7.2 Nodes in the Translation of the Eight Rules.

```
RULE1:
(AND
     (OR aah=off (AND gyro=on gyro-movement=none)
     (side-a=on)
     (side-b=on)
     (rhc-roll=none)
     (rhc-pitch=none)
     (rhc-yaw=none)
     (thc-x=none)
     (thc-y=pos)
     (thc-z=none)
     (OR vda-a-r2=off vda-a-r4=off vda-a-l1=on vda-a-l3=on)
)
->
(AND
     (failure=cea)
     (suspect=a)
)


RULE2:
(AND
     (OR aah=off (AND gyro=on gyro-movement=none)
     (side-a=on)
     (side-b=on)
     (rhc-roll=none)
     (rhc-pitch=none)
     (rhc-yaw=none)
     (rhc-yaw=none)
     (thc-x=none)
     (thc-y=pos)
     (thc-z=none)
     (OR vda-b-r2=off vda-b-r4=off vda-b-l1=on vda-b-l3=on)
)
->
(AND
     (failure=cea)
     (suspect=b)
)


RULE3:
(AND
     (OR aah=off (AND gyro=on gyro-movement=none)
     (side-a=on)
     (side-b=on)
     (rhc-roll=none)
     (rhc-pitch=none)
     (rhc-yaw=none)
     (thc-x=none)
     (thc-y=neg)
     (thc-z=none)
     (OR vda-a-l1=off vda-a-l3=off vda-a-r2=on vda-a-r4=on)
)
->
(AND
     (failure=cea)
     (suspect=a)
)


RULE4:
(AND
     (OR aah=off (AND gyro=on gyro-movement=none)
     (side-a=on)
```

```
    (side-b=on)
    (rhc-roll=none)
    (rhc-pitch=none)
    (rhc-yaw=none)
    (thc-x=none)
    (thc-y=neg)
    (thc-z=none)
    (OR vda-b-l1=off vda-b-l3=off vda-b-r2=on vda-b-r4=on)
)
->
(AND
    (failure=cea)
    (suspect=b)
)


RULE5:
(AND
    (OR aah=off (AND gyro=on gyro-movement=none)
    (side-a=on)
    (side-b=on)
    (rhc-roll=pos)
    (rhc-pitch=none)
    (rhc-yaw=none)
    (thc-x=none)
    (thc-y=none)
    (thc-z=none)
    (OR vda-a-r2=off vda-a-l3=off vda-a-l1=on vda-a-r4=on)
)
->
(AND
    (failure=cea)
    (suspect=a)
)


RULE6:
(AND
    (OR aah=off (AND gyro=on gyro-movement=none)
    (side-a=on)
    (side-b=on)
    (rhc-roll=pos)
    (rhc-pitch=none)
    (rhc-yaw=none)
    (thc-x=none)
    (thc-y=none)
    (thc-z=none)
    (OR vda-b-r2=on vda-b-r4=on vda-b-l1=on vda-b-l3=on)
)
->
(AND
    (failure=cea)
    (suspect=b)
)


RULE7:
(AND
    (OR aah=off (AND gyro=on gyro-movement=none)
    (side-a=on)
    (side-b=on)
    (rhc-roll=neg)
    (rhc-pitch=none)
    (rhc-yaw=none)
    (thc-x=none)
    (thc-y=none)
    (thc-z=none)
    (OR vda-a-r2=on vda-a-r4=on vda-a-l1=on vda-a-l3=on)
)
->
(AND
    (failure=cea)
    (suspect=a)
)


RULE8:
(AND
    (OR aah=off (AND gyro=on gyro-movement=none)
    (side-a=on)
    (side-b=on)
    (rhc-roll=neg)
    (rhc-pitch=none)
    (rhc-yaw=none)
    (thc-x=none)
    (thc-y=none)
    (thc-z=none)
    (OR vda-b-r4=off vda-b-l1=off vda-b-r2=on vda-b-l3=on)
)
->
(AND
    (failure=cea)
    (suspect=b)
)
```

The motivations for the intermediate level of representation are the desire for parallelism, the reduction of the routing effort for messages, and the restriction of unnecessary potential access. In addition, the translation of this intermediate level into an EFG is a very straightforward process, as indicated by the principles described above.

One of the primary thrusts of our work is the parallelization of processing. We want to use multiple processors to speed up reasoning. The actual applications delivery architectures and numbers of processors are currently unknown, but we have would like to have the potential of maximizing parallelism. Therefore we would like initially to create an EFG with the FINEST GRAIN that is possible. We will then RECLUSTER, when that is desirable or required, based on architecture of system, i.e. numbers and characteristics of processors and pathways. It is also possible to do some reclustering based on a kind of data access metaknowledge that will be discussed below.

Creating an EFG with the finest grain possible applies not only to the rule nodes or decision processes, but also to the value nodes. In other words, each value should be located in a separate node. With respect to values, using the largest grain would correspond to a single central memory. Every access or update of any value would have to be done by communication with this central memory. When a value was changed in this central memory, messages would have to be sent to those nodes which need to access that value. This could involve a substantial routing effort. If, on the other hand, a node only held a single value, it would be connected with outgoing links to all the rule nodes which need that value. Every time the value would be changed by a message sent to that value node, the same message would go out on each outgoing link from that node. In some sense, the value nodes serve as 'routers' for values. In general, the greater the number of values which reside in a node, the greater the computational routing effort will be. Restricting the number of values in a node to one, also limits potential access to that value to those nodes that need to know that value. If on the hand, a node held three values, it would need to be connected with an outgoing link to all the nodes which needed any of those values; a node might then potentially get access to one of the two values it did not need to know.

On the following pages are shown the eight CLIPS rules from which the eight single-argument rules above were generated by decollapsing. Although the number of rules has remained the same, the number of conjuncts has increased.

```
;;;;;;;;;;;;;;;;;;;;;;;;;

;pos y,
(defrule cea-a-test-input-posy-null-null-7
        (or (aah off) (and (gyro on)(gyro movement none none)))
        (side a on)
        (side b on)
        (rhc roll none pitch none yaw none)
        (thc x none y pos z none)
        (or
        (vda a r2 off)
        (vda a r4 off)
        (vda a ?n&~r2&~r4 on)
        )
=>
        (assert (failure cea))
        (assert (suspect a))
        (printout crlf "failure -during translational command ")
        (printout "in the pos y direction" crlf)
)

(defrule cea-b-test-input-posy-null-null-7
        (or (aah off) (and (gyro on)(gyro movement none none)))
        (side a on)
        (side b on)
        (rhc roll none pitch none yaw none)
        (thc x none y pos z none)
        (or
        (vda b r2 off)
        (vda b r4 off)
        (vda b ?n&~r2&~r4 on)
        )
=>
        (assert (failure cea))
        (assert (suspect b))
        (printout crlf "failure -during translational command ")
        (printout "in the pos y direction" crlf)
)
;
;neg y
(defrule cea-a-test-input-neg-null-null-8
        (or (aah off) (and (gyro on)(gyro movement none none)))
        (side a on)
        (side b on)
        (rhc roll none pitch none yaw none)
        (thc x none y neg z none)
        (or
        (vda a l1 off)
        (vda a l3 off)
        (vda a ?n&~l1&~l3 on)
        )
=>
        (assert (failure cea))
        (assert (suspect a))
        (printout crlf "failure -during translational command ")
        (printout "in the neg y direction" crlf)
)

(defrule cea-b-test-input-neg-null-null-8
        (or (aah off) (and (gyro on)(gyro movement none none)))
        (side a on)
        (side b on)
        (rhc roll none pitch none yaw none)
        (thc x none y neg z none)
        (or
        (vda b l1 off)
        (vda b l3 off)
        (vda b ?n&~l1&~l3 on)
        )
=>
        (assert (failure cea))
        (assert (suspect b))
        (printout crlf "failure -during translational command ")
        (printout "in the neg y direction" crlf)
)
;
;pos roll
(defrule cea-a-test-input-null-pos-null-9
        (or (aah off) (and (gyro on)(gyro movement none none)))
        (side a on)
        (side b on)
        (rhc roll pos pitch none yaw none)
        (thc x none y none z none)
        (or
        (vda a r2 off)
        (vda a l3 off)
        (vda a ?n&~r2&~l3 on)
        )
=>
        (assert (failure cea))
        (assert (suspect a))
        (printout crlf "failure -during rotational command ")
        (printout "in the pos roll direction" crlf)
)

(defrule cea-b-test-input-null-pos-null-9
        (or (aah off) (and (gyro on)(gyro movement none none)))
        (side a on)
        (side b on)
        (rhc roll pos pitch none yaw none)
        (thc x none y none z none)
        (vda b ?m on)

=>
        (assert (failure cea))
        (assert (suspect b))
        (printout crlf "failure -during rotational command ")
        (printout "in the pos roll direction" crlf)
)
;
```

```
;neg roll
(defrule cea-a-test-input-null-neg-null-10
        (or (aah off) (and (gyro on)(gyro movement none none)))
        (side a on)
        (side b on)
        (rhc roll neg pitch none yaw none)
        (thc x none y none z none)
        (vda a ?m on)

=>
        (assert (failure cea))
        (assert (suspect a))
        (printout crlf "failure -during rotational command ")
        (printout "in the neg roll direction" crlf)
)

(defrule cea-b-test-input-null-neg-null-10
        (or (aah off) (and (gyro on)(gyro movement none none)))
        (side a on)
        (side b on)
        (rhc roll neg pitch none yaw none)
        (thc x none y none z none)
        (or
        (vda b r4 off)
        (vda b l1 off)
        (vda b ?n&~r4&~l1 on)
        )
=>
        (assert (failure cea))
        (assert (suspect b))
        (printout crlf "failure -during rotational command ")
        (printout "in the neg roll direction" crlf)
)
;
;;;;;;;;;;;;;;;;;;;;;;;;;;
;logic for z, roll, pitch
;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;pos z,
(defrule cea-a-test-input-posz-null-null-11
        (or (aah off) (and (gyro on)(gyro movement none none)))
        (side a on)
        (side b on)
        (rhc roll none pitch none yaw none)
        (thc x none y none z pos)
        (or
        (vda a d1 off)
        (vda a d2 off)
        (vda a ?n&~d1&~d2 on)
        )
=>
        (assert (failure cea))
        (assert (suspect a))
        (printout crlf "failure -during translational command ")
        (printout "in the pos z direction" crlf)
```

We will now compare the two versions of the first rule. These two versions are shown below in figure 7.3.

```
(defrule cea-a-test-input-posy-null-null-7
     (or (aah off) (and (gyro on)(gyro movement none none)))
     (side a on)
     (side b on)
     (rhc roll none pitch none yam none)
     (thc x none y pos z none)
     (thc x none y pos z none)
     (or
     (vda a r2 off)
     (vda a r4 off)
     (vda a ?n&-r2&-r4 on)
     )
=>
     (assert (failure cea))
     (assert (suspect a))
     (printout crlf "failure -during translational command ")
     (printout "in the pos y direction" crlf)
)

R1:
(AND
     (OR aah=off (AND gyro=on gyro-movement=none)
     (side-a=on)
     (side-b=on)
     (rhc-roll=none)
     (rhc-pitch=none)
     (rhc-yaw=none)
     (thc-x=none)
     (thc-y=pos)
     (thc-z=none)
     (OR vda-a-r2=off vda-a-r4=off vda-a-l1=on vda-a-l3=on)
)
->
(AND
     (failure=cea)
     (suspect=a)
)
```

Figure 7.3 Collapsed and Decollapsed Versions of a CLIPS rule.

Initially let us consider the second and third conjuncts. In the original rules these were (side a on) and (side b on), while in the version with single-argument rules these are (side-a on) and (side-b on). The original conjuncts referred to multi-valued facts, where the field SIDE remained the same but the values of each of the other fields could vary (A/B and ON/OFF). If we had a single SIDE value node, every access or update of either SIDE A or SIDE B would have to be done by communication with this node. When either value is changed, messages would have to be sent to just those nodes which need to access that value. This would involve computational routing effort. Having a single SIDE value node, instead of one each for SIDE-A and SIDE-B, could also limit potential parallelism.

The same issues are addressed in the change of (vda a r2 off) to (vda-a-r2=off). Only those nodes which need to access or update VDA-A-R2 would be connected to the node holding its value. On the other hand, if we had a single VDA node, any node that updated any VDA-__-__ would have to send messages to this single node, and any node that accessed any VDA-__-__ would need to have a communication connection from it.

Next consider the change of (rhc roll none pitch none yaw none) to the three conjuncts, (rhc-roll=none) (rhc-pitch=none) (rhc-yaw=none). Facts represented in the original manner could be thought of as RHC records with three fields, ROLL, PITCH, and YAW, each of which could have the values POS, NEG, and

NONE. Just as in the cases above, if we had a single RHC value node, we would have to computationally route messages to the nodes that need the values of the individual fields ROLL, PITCH and YAW.

It should be noted that in the MMU expert system, any rule which referred to any one the these values, say ROLL, also referred to the other two values, PITCH and YAW. Of course, it need not always be the case that what we have referred to as a multi-valued fact corresponds to a conceptual record. When it is the case, this metaknowledge about cooccurrence of references to data items could be used to recluster the EFG created from the single-argument rules.

It is important to point out that it takes certain knowledge to interpret the semantic relationships between the fields of the facts corresponding to the conjuncts discussed above, (side a on), (vda a r2 off), and (rhc roll none pitch none yaw none). Syntactically, these just consist of 3, 4, and 7 fields, respectively. All the metaknowledge that we used to make these interpretations, and thus change the rules into single argument ones, was surmised from either our own domain knowledge or from our analysis of the MMU rule base as a whole. Fortunately, these were sufficient for the rules at hand, but they may not always be so. THIS METAKNOWLEDGE SHOULD BE EXPLICITLY STATED as a part of the documentation/specification of the rule base itself. This metaknowledge is crucial for validation of the rule base. Since the facts referred to by these conjuncts are all inputs to the system, this metaknowledge is closely related to the input metaknowledge discussed in the section on generating data for test runs.

Consider now the change of the single conjunct (vda a ?n&~r2&~r4 on) to the disjunction (OR vda-a-l1=on vda-a-l3=on). This change involves decollapsing a conjunct, as opposed to the examples above, which involve decollapsing multi-argument facts to make them single-valued. The decollapsing requires metaknowledge about the collapsing or abbreviatory constructs in the representation language. The meaning of the original conjunct is "there exists any VDA-A except -R2 or -R4 which is ON". Obviously, order to change this to the disjunction requires the metaknowledge that the other VDA-A's, in addition to -R2 and -R4, are -L1 and -L3. Again this metaknowledge was acquired by examining the MMU rule base, although this may not always be sufficient. When it is not, this metaknowledge would have to come from the knowledge engineer or the domain expert.

Another example of decollapsing a conjunct can be found in the sixth rule, where the conjunct (vda b ?m on) is changed to the disjunction (OR vda-b-l1=on vda-b-r2=on vda-b-l3=on vda-b-r4=on). The original conjunct means "there exists any VDA-B which is ON." Here the metaknowledge required in order to decollapse was that the set of VDA-B included -L1, -R2, -L3, and -R4.

7.2 Dealing with Unification

With the approach discussed thus far the MMU CLIPS rule base can be translated into an EFG, but that rule base does not use the full range of capabilities of CLIPS. In particular, it does not require unification. For unification, it is necessary to find a set of variable bindings which which meet some conditions. The critical issue for translation to EFG or an any parallelization is 'where are the potential bindings to be found?' If we have a single central memory, the potential sets of bindings can all be compared within that memory. Obviously, this restricts the possibilities for parallelism.

Instead of a single central memory, it might be possible based on a given rule base to group into value memory nodes all those variables that would ever need to be unified. Each one of these value memory nodes would need to be connected to all the rule nodes that updated or accessed any of its values. This is counter to the methodological approach taken above; it would introduce the routing problem and restrict parallelism. Still if the rule base includes rules that require unification, then this may be the best solution. An alternative approach is having each variable in a different local node and doing unification by sending potential bindings to the rules to check. This could easily cause an unmanageable message explosion.

The existing version of our software cannot translate a rule base requiring unification into an EFG. We are currently investigating methods of dealing with unification for our EFG data flow paradigm. For the current version of our software, we restrict the form of the facts to which rules can refer. Each fact would have exactly two fields, the first a predicate the second a value that can vary. In addition, for each predicate there can only be one single fact i.e. there can not be facts with different values.

Let us consider briefly the ramifications of including these restrictions. They would disallow the two facts (male john) and (male bill). Such information might instead be represented as (john-male T) or (john-sex male). The conjunctive condition, (and (heat ?x) (weight ?y) (?x < ?y)), would still be allowed, but (and (heat ?a ?x) (weight ?b ?y) (?x < ?y)) would not. Here the restrictions successfully rule out the second conjunctive condition, which does require unification, as is evident from the following reformulation, (and (object ?a) (object ?b) (heat ?a ?x) (weight ?b ?y) (?x < ?y)). This reformulation is ruled out on the basis of each of the restrictions suggested above: the first two conjuncts include the same predicate with different values and the second two conjuncts contain three fields. A rule with the two conjuncts (person ?x) (age ?x ?y) would also be ruled out.

It should be noted that the first restriction can be relaxed while still allowing unification to be excluded. In particular, it would be possible to have facts that correspond to conceptual records. For example, one might have a 'record-type' fact (rhc roll rval pitch pval yaw yval), as was alluded to above in the discussion on decollapsing and reclustering. Similarly, while (married mary john) would be ruled out, this information might be represented not only as (john-wife/spouse mary) (mary-husband/spouse john), but also with the relaxation as (marriage wife mary husband john).

Finally, we reiterate that if a problem cannot be solved conveniently without unification, adhering to these restrictions, then we can create an EFG which has, in effect, a single central memory or multiple grouped value memory nodes. As was emphasized above, this would reduce possible parallelization and would add to computational routing effort.

8. APPLYING EFG METHODS TO THE MMU EXPERT SYSTEM

8.1 Introduction

This chapter discusses progress made relative to the Validation and Verification Experiment issued by Sally Johnson of NASA Langley in the Spring of 1989. The experiment called for a number of research teams to evaluate how applicable their techniques would be to a selected test case and to demonstrate, as far as possible, their techniques on this test case. The test case in question is an expert system for diagnosing problems with a Manned Maneuvering Unit (MMU) for extra vehicular activities in space developed for NASA by McDonnell Douglas[17].

The MMU test case rule set has been received and evaluated to determine how applicable our techniques are to detecting errors within its rule set. Work is now in progress to extend our technology such that it can be used to test the MMU rule set, subject to certain caveats which are discussed below.

This introduction summarizes the current status of the project relative to the MMU test case and is followed by sections discussing the technical issues in greater depth.

Examination of the MMU test case revealed several issues:

1) That the MMU rule set was written for off-line diagnosis. That is, it obtains its inputs from an input file of assertions and produces its conclusions in the form of printout statements. The WPI KRAM diagnostic techniques are strongly oriented towards embedded real-time systems which obtain their data from external sources and produce their results in the form of messages to activate other systems.

2) In the KRAM technique, inputs are injected at specified times into the system under test and then the outputs are examined to determine whether specified performance constraints have been violated. In order to apply KRAM to the MMU test case it will be necessary to replace the test files of assertions with test cases in which data values are sent to the rules which require data values for the test.

3) The MMU produces its results in the form of arbitrarily formatted print statements instead of making assertions about the truth of variables. It is not possible, in general, to automatically analyze output in the form of print statements. In order to use the KRAM automatic test evaluator it will be necessary to replace these printout statements with fact assertions which can be automatically verified.

4) The MMU made extensive use of multiple value facts such as (rhc roll none pitch none yaw none) when much simpler "object-property-value" facts could have been used. This use of multiple value facts will result in much less efficient real-time execution of the MMU and also makes testing much harder. Before the conclusion of the present phase of the KRAM project (KRAM2-1), the KRAM software will be capable of handling CLIPS "object-property-value" facts. Multiple value facts are planned to be implemented in KRAM 2 phase 2 (KRAM2-2) as part of the implementation of unification handling. The reason for this is that the primary use of multiple valued facts is in performing unification functions. In order to test the MMU, during KRAM2-1, it will be necessary to convert the MMU multiple value facts to simpler "object-property-value" facts. It is planned to test the MMU in its original multiple value fact form during KRAM2-2.

5) All of the faults we have found so far in the MMU have been detected, or are capable of being detected, by the KRAM rule to EFG/AFO translator. In trying to create a viable EFG, the translator is able to determine unconnected rule inputs and outputs. It is also able to detect those inputs and outputs which are inconsistent with the input and output value meta-knowledge obtained from the test specification meta-knowledge. It will, in the future, be possible for the translator to detect rules which can never be fired by examining their predicate clauses.

6) The fact that the translator has the information to detect many faults as a byproduct of trying to generate a viable evidence flow graph was an unexpected result but a very valuable one. We have observed that all the members of the team have gained tremendous insight into the workings (and potential problems with) various expert systems shells by looking at how to translate their knowledge representations into flow graph form. Now we are beginning to realize that it may be possible to automatically detect many problems by attempting a translation from a rule based control flow representation to a data flow representation.

6) There was no performance specification meta-knowledge supplied with the MMU. We believe that this meta-knowledge is essential to the detection of problems. If no one has specified what the system should or shouldn't do then anything it does is supposedly OK. In order to perform any testing or fault evaluation on the MMU it will be necessary for us to attempt to generate a performance specification. We plan to use the 5 test cases supplied as part of the MMU evaluation data, together with some reverse engineering and general knowledge, to generate a test constraint language specification for the MMU rule set. This will then be used by the test generator and evaluator to test the MMU rules.

7) There do not appear to be any numeric values, either in the test cases or in the rules, for which it would be necessary to evaluate performance sensitivity. Thus the sensitivity evaluation feature of KRAM does not appear to be useful in this case.

8) There does not appear to be any time or ordering constraints on the inputs or outputs. Thus the ability of KRAM to evaluate problems related to time ordering or time changes of results would not appear to be useful here.

There follows some detailed commentary on some of these issues:

8.2 The Acquisition of Metaknowledge

The process of applying our methods to the MMU (or any knowledge base) involves acquisition of the necessary metaknowledge. Were this metaknowledge provided explicitly in the form of the specification language we have designed, this acquisition would not be necessary. It is highly recommended that this metaknowledge be acquired or developed at the same time that the knowledge base is acquired or developed. Since there exists no explicit metaknowledge with the MMU expert system, reverse engineering will be required to acquire the metaknowledge. Ideally this should be accomplished by interviews with domain experts or the original knowledge engineers.

The metaknowledge that would be needed to be acquired could be divided into two types: constraints on values and the structure of facts. The metaknowledge about the structure of facts will be discussed in section 8.3. The remainder of this section deals with metaknowledge about constraints on values. Different kinds of constraints on values differ in the extent to which they can be acquired by reverse engineering or whether an expert is required. This is summarized in the following table:

Inputs, Outputs, Intermediate Values	- inadequate data available
Ranges & Distributions	- very limited data obtainable from reverse engineering
Input Cooccurrence Constraints	- need expert input
Ordering of Input Availability	- not applicable
Test Cases 5	
Input Output/Intermediate Constraints	- need expert input

TABLE 8.1: Acquisition of constraints on values by reverse engineering

For the MMU, some difficulties were encountered in the determination of which of the values referenced in the rules are inputs, outputs, and intermediate values. The five test cases supplied with the MMU FDIR Automation Task Final Report [1] were useful in determining the inputs and outputs. It appears that all the cases have the same set of inputs. The outputs must be regarded as not only the changes in setting that were made, i.e. the corrections or treatments, but also the print statements that reflect the conclusions. The primary difficulty is that diagnostic conclusions and treatments were expressed as a part of strings in print messages, not consistently as facts in the database. There is thus a need to distinguish between messages which express conclusions from those which serve only to describe or partially trace the line of reasoning of the system; such messages included "testing ...", "suspected ...", "turning ...", "setting ...", "recalling ...".

Comparing the print statement expressing the expected failure for each test case with the printed output of the test cases, indicated the kinds of print statements that needed to be examined throughout the rule base to determine the possible diagnoses. In all cases the treatments involved setting side-A ON or OFF and/or side-B ON or OFF. It is interesting to note the discrepancy in the descriptions of the expected failures and the print statements. For example, for test case O the expected failure was "Cea failure - a signal from the valve drive amp on side A was not sent to thrusters", while in the run, which presumably was thought of as correct, the only failure reported was "cea failure on side a". We would advise that the diagnostic conclusions be expressed as specific facts in the database. The primary reason for this suggestion is that it would allow the treatment of final conclusions as a class of objects on which a uniform set of generic operations, i.e. a uniform interface or protocol, could be defined. One such operation would be Print(conclusion). This would lead to consistency in the declarative properties of each conclusion. Also changes in the implementation of any of the generic operations would not require modification of the code of each rule which infers each conclusion. In addition, this would allow input assertions (or facts) and inferrable assertions to share properties and operations at a higher level of abstraction, i.e. be subclasses of the same class. A uniform treatment of these two kinds of assertions would facilitate the operations required for test generation and evaluation in our, or any other, system.

For each of the inputs, as well as final and intermediate node outputs, we must determine the range of values that they can assume. The five test cases are insufficient to do this. The values in these cases must be augmented by an examination of the rule base (and the text portions of the report). If a value was ever observed in an input to a test case or referred to in the lhs or rhs of a rule, clearly it must be a possible value. Other values might be presumed to exist based on the analyzers knowledge of symmetries in value descriptions. For example, for binary features, like {ON, OFF}, {OPEN,CLOSED}, the observation of one member of the set would allow the inference of the other. For a ternary feature, like {POS,NEG,NONE}, observation of either of the poles would allow the inference of the other; however, it would be difficult to predict the precise name for the neutral value. In many cases parallelism or analogy can also be used to infer possible values; for example, thc-x, thc-y, and thc-z might all be assumed to have the same set of values, as would rhc-roll, rhc-pitch, and rhc-yaw.

By examining the rules and test cases it is not possible to determine the range for inputs that had scalar values; this would require a domain expert. There are a large number of these inputs: fuel-used-a,fuel-used-b, tank- pressure-was-a, tank-pressure-was-b, tank-pressure-current-a, tank-pressure- current-b, gyro-thruster-time, and hc-thruster-time. An expert would also be required to acquire the cooccurrence restrictions on values of inputs, which are important to reduce the space of inputs, as well as to acquire the distributions of input values. Constraints on the occurrence of conclusions and/or more extensive test suite would also desirable.

8.3 The Translation of CLIPS MMU to EFGs and AFOs

The manners of dealing with both the inputs and the outputs in the MMU expert system are problematic. First, one needs to deal with the issue, mentioned above, of the conclusions which are embedded in some of the many print messages; these are to correspond to the output (node)s in the EFG representation. Second, the MMU expert system does not take in any inputs. Instead there is a separate rule, triggered by a distinct fact, to initialize the database for each of the five test cases and to print out the expected results. These rules were just included for the demonstration of the system. Obviously, there is a need to identify and delete these rules which are not a part of the knowledge base proper. Some additional rules would have to be included in order to actually read the inputs from a file for each of the very large number of test runs our method would require.

Metaknowledge about the structure of facts is also required to do the translation. This is the case because the current version of out translator works with 'single-valued facts.' Single-valued facts are restricted to consist of a predicate and a value; further multiple values cannot exist for the same predicate. The purpose of adopting these restrictions was to rule out the possibility of having rules which required unification. They also served to reduce computational routing effort and to limit access to values. This was discussed in last year's project report[18].

Consider the clause (vda a r2 off), which can be characterized as a multi-valued fact. The only field which always appears with the same value is the first, vda; thus it may be regarded as the predicate. The other fields may be regarded as the values. With the appropriate metaknowledge the clause (vda a r2 off) would be changed to (vda-a-r2 off), in which the last field is treated as the designated value. Another example would be the clause (vda a ?n&~r2&~r4 on), i.e. any vda-a except -r2 or -r4 which is on; this would be changed to (OR (vda-a-l1 on) (vda-a-l3 on)).

For the MMU expert system, it appears to be possible to acquire the metaknowledge about the structure of facts by analysis of the rule base. With this metaknowledge the clause (vda a r2 off) would be changed to (vda-a-r2 off). In addition, the clause (vda a ?n&~r2&~r4 on), i.e. any vda-a except -r2 or -r4 which is on, would be changed to (OR (vda-a-l1 on)(vda-a-l3 on)). Our current research involves developing a method to do unification with the dataflow architecture of the EFG. After this method is developed the restriction to single-valued facts can be dropped, and this transformation of the structure of facts will not be required.

8.4 Errors Discovered During Analysis of the MMU Rules

During the analysis to determine the required metaknowledge, several errors were discovered. Although this analysis was done by hand, we expect that it will be possible to detect these errors automatically, when our tools are completed. The first two errors fall in the category of unreferenced values. (gyro-thruster-time 2) appears as an input in all five test cases, but the predicate gyro-thruster-time never appears in the lhs of any rule. Also (hc- thruster-time 1) appears as an input in the first two rules, but never appears in an lhs. Errors in the category of unreferenced values would be discovered in the translation stage, because there would be no outgoing link from the corresponding (input) node.

One "stylistic" inconsistency in the labeling of facts should also be mentioned. The predicate which most directly indicates a conclusion in this system is (failure ...). It can assume the values cea, cea-a, cea-b, cea-ab, thruster-a, thruster-b, cea-coupled. One failure, however, was represented by a distinct predicate (failure-thrusters-with-xfeed); it would have been more consistent to represent this with the same predicate, as (failure thrusters- with-xfeed).

8.5 Conclusions

The conclusions thus far are:

1) That it is possible to detect faults in the MMU test case using the KRAM techniques providing that the MMU test case is re-structured so that it is suitable for automatic testing. This restructuring involves modifying the system so that it has clearly identifiable inputs whose values can be generated and outputs whose values can be tested.

2) That all the faults found so far would have been detected during the translation of the rules to an EFG without the need for monte-carlo simulation. The discovery of errors and consistency checking of rules by this translation process is parallel to the error discovery done by a compiler for a high level language, such as Ada. Some of the detectable faults appear as graph connectivity or data type inconsistencies.

3) A constraint specification for the performance of the MMU rules will need to be prepared if thorough testing of the correctness of their execution is to be performed using monte-carlo test methods.

Two final comments from related work:

a) It would appear that the most useful aspect of the monte-carlo testing is in determining whether the system will produce the correct outputs in the correct order. We have not found many places where sensitivity to values in rules or inputs is a major problem. It is very hard, however, to determine a priori all the possible outputs and output orderings which can occur for possible sets of real-time inputs. This is where a formal performance constraint description, used in conjunction with automatic monte-carlo testing, can find problems which cannot be detected by any other methods.

b) Running monte-carlo simulations over a large set of rules will take a lot of computer time to achieve a reasonable confidence in the correctness of the rules. It is recommended that any large rule set be broken down into smaller groups of rules each of which can be extensively tested using monte-carlo simulation. Then it is recommended that the rule groups be separately instrumented during overall system testing to ensure that their input-output relationships, as specified by their performance constraint description, are not violated. This is comparable with hardware testing in which black boxes are extensively tested before being integrated into a system. The black boxes are then monitored during systems testing to ensure that they still work as specified. This also makes it easier to develop test specifications as the specification only have to apply to a small group of rules as opposed to the whole system.

9. LESSONS LEARNED

This chapter contains some observations about expert systems learned as a result of trying to mimic their actions with data flow mechanisms.

1) Expert systems shells contain a large amount of implicit knowledge. This implicit knowledge minimizes the work that a programmer/knowledge engineer has to do to encode an application.

   There is a trade-off between the implicit and explicit knowledge components of an expert system, as shown by the pie chart in figure 9.1. Relatively, the more implicit knowledge that a shell contains, the less work that has to be done to encode an application. This implicit knowledge, however, also limits the flexibility of the shell and its domains of application.

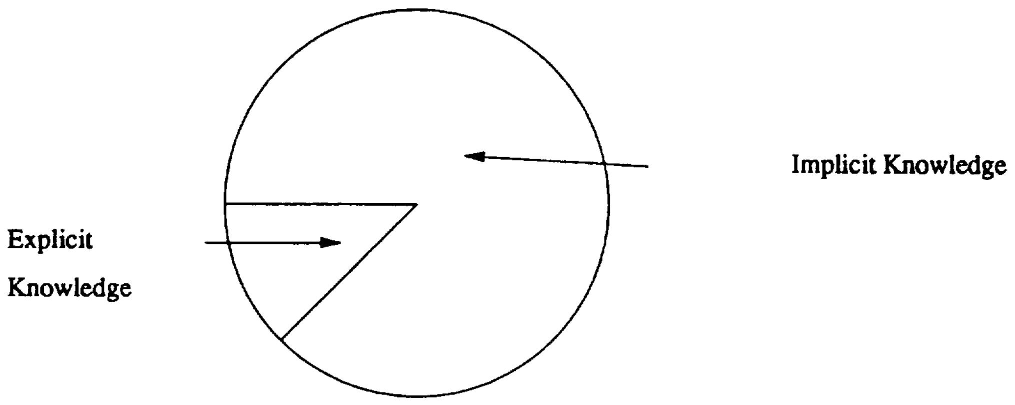


Figure 9.1 Implicit and Explicit Knowledge Components of an Expert System.

   This observation applies to all higher level languages, including fourth generation languages and their related CASE tools. To minimize the work in encoding an application we want the language to use as high a level of abstraction as possible. To maximize flexibility we want to allow users to code at as low level as possible. A good language provides some mix of these so that users can program at a high level of abstraction where possible and can use lower levels when not.
   In expert systems shells, such as CLIPS, users program in rules. Here the implicit knowledge contained within the shell is about how data is combined with rules to produce inferences such as through the use of the Rete algorithm and unification mechanisms.

2) The actions of an expert system shell are highly dependent on the embedded implicit knowledge. The same set of rules and data can draw different inferences when executed using different expert system shells. Thus any validation and verification procedure must also include the expert system shell as well as the rules.

3) All expert system shells contain conflict resolution mechanisms to decide which rule to fire next. For expert system shells these mechanisms are generally designed for single processor execution, although there has been considerable work on parallel and concurrent logic programming languages[19]. If it is desired to speed up execution of a set of rules by executing them on multiple processors, then the ordering constraints imposed by the conflict resolution mechanism must be relaxed. Otherwise the rules will execute sequentially in the same order they executed on a single processor. If these constraints are relaxed, however, then the inferences which result may be different from a single processor case.

4) If rule execution is to be independent of inference engine sequencing limitations, then the predicate clauses of the rules must be fully specified. That is, the outcome should not be dependent on rule order in the data base. Unfortunately this is in conflict with requirements for efficient execution of the Rete algorithm[20].

Conceptually, a rule base is a flat structure with rules being independent of their location in the rule base. In practice this is not true, with extensive use being made of rule ordering precedence to ensure correct operation of the rule base. We suggest that this is a bad practice from the viewpoint of maintainability as the inadvertent swapping of two rules could lead to a rule base no longer performing as required.

5) Flat rule structures are as bad as programs written as straight in-line code with no procedures from the viewpoint of maintenance. We believe that it is important to break rules down into related clusters which can be maintained as groups. This would be highly beneficial from the point of view of translation to run time code as all the rules in a cluster could be compiled into a single execution process, or for our system into a single AFO. At present each rule is translated into a separate AFO. This is not very efficient from the execution viewpoint as the overhead in message passing relative to the execution time for a single rule is relatively high. If multiple rules in a cluster were combined into one AFO then more efficient execution would be possible.

6) Present expert system building Shells essentially have control strategies which deparallelize the inherently parallelizable declarative rule representation. To the extent that the programmers encode knowledge in such a way as to take advantage of the control strategy it would not be possible to simply run their rules in parallel. The only potential hope for parallelism under these conditions is to run the rules in parallel and to precompile an ordering for the results received, so that they will correspond to those reached by the corresponding control strategy. The extent to which this precompilation can be done is not clear.

If programmers/knowledge engineers did not write rules whose correctness was dependent on the particular control strategy of their shell, then parallelization would be possible. To determine whether the knowledge engineers had written such rules, one could undertake a very large number of runs, first with the shell's control strategy and then with a parallel regime, and compare the results. It may also be possible to use analytic techniques to determine equivalence under a parallel execution strategy.

7) Present shells do not allow specification of dynamic conflict resolution schemes. The choice of which rule to fire from the set of rules that potentially could fire in current expert system shells is a function of the order in which the rules are included in the knowledge base, the specificity of the rules, or the recency of the new data which makes the rules eligible for firing, or a combination of these. The last is the most dynamic method, yet it still lacks flexibility. Alternatively, with current shells one can always explicitly include as a part of the rules variables whose purpose is to control the flow of execution, for example, goals or agenda modification statements. This approach does not make a clean and explicit division between the dynamic conflict resolution knowledge and the declarative problem-solving knowledge base. AF has the capability to support dynamic importance calculations for decision processes, but current expert system shells do not allow the explicit specification of this kind of (meta)knowledge. AF has the capability to support dynamic node firing ordering, but to take advantage of this, there must be the capability to specify dynamic conflict resolution knowledge.

8) Present shells also lack other types of meta-knowledge specification which are essential for translation as well as validation and verification testing. Some of the needed knowledge types are:

Metaknowledge for testing would be relevant for any dynamic testing; some of it would be relevant also for analytic techniques. This metaknowledge was discussed in the section on test generation and evaluation. It includes (1) constraints on the ranges of all inputs, outputs, and intermediate values, (2) cooccurrence constraints on input values, (3) constraints on the order of availability of inputs, and (4) conditional relations between inputs, outputs, and intermediate nodes. Fully or partially specified test cases would fall into the final category, with each fully specified test case being a conditional relation between a full set of input values and output values. The third category is only relevant in a system where inputs are not assumed to be all available at the same time. None of these four categories of metaknowledge can be acquired by analysis of the rule base. This metaknowledge must be a part of the specification of the expert system.

Metaknowledge about the inference engine and control strategy is required for generating the EFG nodes and is discussed above in that section. It includes metaknowledge about confidence calculations, conditions for firing and refiring rules, and conflict resolution strategies and/or priorities for choosing which eligible rules to fire. With the exception of priorities, which are sometimes represented explicitly in the rules themselves, all of these types of metaknowledge need to be specified independently. Normally they would be extractable from the (internal) documentation of the inference engine or shell, but it would be helpful if they were stated explicitly as a part of the specification of the expert system. This seems especially appropriate since the rule base should be expected to perform as specified only when used with an inference engine with these same characteristics.

Metaknowledge for decollapsing and reclustering the rule base is required so that the rule base can be translated into an EFG which has a very small grain. This will allow parallelization and reduce computational routing effort. This decollapsing will result in a rule base with simple conjunctive rules referring to single valued facts. Reclustering of the value nodes representing the individual facts may then be done based on the architecture of the delivery system as well as the constraints on the contemporaneousness of access to the facts.

The decollapsing metaknowledge is of two types. The first is based on the syntax of the representation language, in particular, its abbreviatory conventions. This metaknowledge is acquirable from the documentation for the representation language. The second type is based on the semantics of the domain. Best guesses about this metaknowledge can be made based on analysis of the rule base using general world knowledge and the limited domain knowledge of the analyst. The accuracy of these guesses would be a function of the quality of mnemonic symbols used in the rule base and the extent of the domain knowledge of the analyst; confirmation would still be required from the knowledge engineers who created the rule base. Again it would be desirable that this metaknowledge be explicitly stated as a part of the documentation, particularly, a description of the fact templates. With respect to the metaknowledge for reclustering, while the architecture of the delivery system is logically independent from the rule base, analysis of the co-occurrences of references can yield the constraints on the contemporaneousness of access to the facts.

9) Most present shells do not include methods of specifying time relationships or of reasoning about time in a real-time manner. They do not have ways of expressing the time period for which data or conclusions are valid or of expressing requisite precedence ordering of events. If a shell contained a means of expressing such knowledge then this could be incorporated into AF. The result would be a system in which the application could exert finer control over the execution of its component code threads.

10) At present AI shells are designed to run on single processors. It would be highly beneficial if they were extended to include knowledge representations to allow the user to specify which actions can be executed in parallel and which must be executed in a sequential manner. This is highly desirable from the viewpoint of translating automatically to code threads for execution on a parallel processor. There is a strong analogy here to the "dusty deck" problem of translating FORTRAN programs to run on a parallel processing system.

11) From the viewpoint of testing, it is essential to develop the test specification knowledge in a formal test language at the same time as the system is being developed. In this way the knowledge embedded in the test language evolves along with the rest of the system allowing the system to be thoroughly tested after each extension. The formal test description knowledge base contains all the knowledge against which prior system configurations have successfully run. Thus, when a change is made to the system, testing against the test language specification assures that the system still meets specifications in all areas, not just the area changed. This can help avoid the problem of introducing more bugs when fixing others.

In summary we have found that the object-oriented, message-based paradigm of AF is capable of providing far more capability for the execution of real-time AI applications than current expert system shells have the capability of expressing or simulating. There are three possible solutions to this problem:

a) Extend current expert system shells to be able to express the types of meta-knowledge needed for real-time AI applications.

b) Provide the meta-knowledge external to the shell, after the rules have been developed for single processor non-real-time execution. This is the current method except that the meta-knowledge is embedded in the translator. Ideally this metaknowledge would be stated explicitly in the form of rules or other "user friendly" representations.

c) Develop a shell interface to AF which will allow for rapid prototyping while embodying the knowledge needed for efficient testing and translation to a parallel real-time execution environment.

10. CONCLUSIONS

During the past year the following project objectives have been successfully achieved:

1. Development of a comprehensive BNF language specification for Evidence Flow Graphs which encompasses the majority of the features found in all the intelligent systems studied including many expert systems.
2. Acquisition of Adaptive Tactical Navigator system causal network and rule knowledge representations and their conversion into Evidence Flow Graphs.
3. Study of the CLIPS expert system shell and the development of techniques to convert forward and backward chaining rule representations into EFGS.
4. Development of an automatic translator to convert ATN rules in a CLIPS-like syntax to EFGs. This translator was written in Ada.
5. Development of a translator from EFGs to Ada code modules which, when linked with AFA, form an executable system. This translator is written in Ada.
6. Development of an Ada version of the Activation Framework called AFA.
7. Development of a test specification language.
8. Development of a test generator program written in Common Lisp.
9. Study of methods for test evaluation.
10. Study of Petri Network technology as a way of analyzing EFGs to detect problems.

The most important achievement was the demonstration of the viability of the concept of starting with rules and automatically converting these into executable Ada code. Not only was the resultant code executable in real-time but it was demonstrated that the system could be tested to ensure that it performed to required performance specifications for validation and verification purposes. Also, the associated KRAPP project has demonstrated that the code is executable with a high degree of parallelism on a fault tolerant parallel processor.

It was found that random monte-carlo testing was a viable approach to verifying performance, but this required large amounts of processor time. It was also found that a large volume of output data could be generated which indicated that it was highly desirable to use a much more sophisticated computer program to analyze the output.

Petri network technology was studied in the hope that it would lead to techniques for the analysis of EFGs thereby avoiding the processor time spent in monte-carlo testing. A technique for converting EFGs into Petri Nets was developed but the resultant networks were so complex as to make them intractable from an analytic viewpoint. It was concluded that it was desirable to extend Petri network techniques to be able to include EFGs so that graph analysis could be performed directly on the EFG to detect possible execution cycles resulting in livelock or deadlock.

An important byproduct of the research was an analytical understanding of the functioning of rule-based expert system shells. In the process of studying how to convert rules and their associated shell metaknowledge into EFGs, many potential problems that could cause incorrect execution became apparent. Intelligent systems are easy to develop using expert systems shells because of the metaknowledge contained within the shells. Users only have to provide the needed additional knowledge in the form of rules. The disadvantage is that users may be unaware of the constraints on or actions of the inference engine resulting in unexpected side effects under certain conditions.

As a result of this research we have come to believe that the meta-knowledge in expert systems shells should be explicitly stated and specified. Further, we have come to realize that minor changes to an inference engine could result in significant changes in the execution of a rule base. This implies that validation and verification procedures for rule-based expert systems must include the shells themselves as well as the rules they process.

We conclude that the proposed methodology for converting high level knowledge representations into Ada code capable of execution in real-time is feasible. We further conclude that an automatic test generator and evaluator is not only feasible but highly desirable. We believe that the results of this work have laid the foundation for the development of software tools which will considerably reduce the development and maintenance costs of software for embedded intelligent systems which need to function in real-time.

Our recommendations for follow on work are as follows:

1) Extension of the prototype translators developed under this project to include other knowledge representations such as those used by expert system development shells such as CLIPS and ART.

2) Extension of the prototype test generator and evaluator so that they become useful tools for performing test generation and evaluation.

3) Further study of the application of graph theoretic techniques to detect potential faults in knowledge based representations without extensive testing.

References

1. Final Technical Report for AF/WRDC under Contract No. F33615-86-C-1043, The Analytic Sciences Corporation, Reading, MA (February, 1990).
2. J. C. Giarratano, "CLIPS User's Guide," Artificial Intelligence Section, Lyndon B. Johnson Space Center (June, 1988).
3. P. E. Green and W. R. Michalson, "Real-Time Evidential Reasoning and Network Based Processing," *Proc. IEEE First Annual Conf. on Neural Networks*, San Diego, CA (June 1987).
4. J. R. Delaney, R. T. Lacoss, and P. E. Green, "Distributed Estimation in the MIT/LL DSN Testbed," *Proc. American Control Conference*, pp. 305-311, San Francisco, Ca. (June 22, 1983).
5. P. E. Green, "AF: A Framework for Real-Time Distributed Cooperative Problem Solving" in *Distributed Artificial Intelligence*, ed. M. N. Huhns, pp. 153-175, Pitman Publishing, London, England (1987).
6. P. E. Green, D. P. Glasson, J. M. Pomerade, and N. A. Acharya, "Real-Time Artificial Intelligence Issues in the Development of the Adaptive Tactical Navigator," *Proc. Space Operations-Automation and Robotics Workshop*, NASA Johnson Space Center, Houston, Texas (August 1987).
7. P. E. Green, "The Activation Framework Software Package AFC C-Language Version 2.1 Users Manual," The Real-Time Intelligent Systems Corp., Worcester, MA (February 1990).
8. T. Masotto, C. Babikyan, and R. Harper, "Knowledge Representation into Ada Parallel Processing," NASA Contractor Report 187451, The Charles Stark Draper Laboratory, Inc., Cambridge, MA (1990).
9. D. Rolston, "Principles of AI and Expert Systems Development" (1988).
10. R. Lindsay , B. Buchanan, E. Feigenbaum, and J. Lederberg, "Applications of AI for Organic Chemistry: The Dendral Project".," McGraw Hill, New York, NY (1980).
11. B. Buchanan, "Rule Based Expert Systems: The Mycin Experiment of the Stanford Heuristic Programming Project," Addison-Wesley, Reading, MA (1984).
12. J. Pearl, "Probabalistic Reasoning in Intelligent Systems: Networks of Plausible Inference," Morgan Kaufmann, San Mateo, CA (1988).
13. B. Buchanan and E. Shortliffe, "A Model for Inexact Reasoning in Medicine," *Mathematical Biosciences #23* (1975).
14. T. Greer, "Artificial Intelligence: A New Dimension in EW," *Defense Electronics* (October 1985).
15. Brownston, Lee, Farell, Robert, Kant, Elaine, Martin, and Elaine in *Programming Expert Systems in OPS5*, Addison Wesley, Reading, MA (1985).
16. Cooper, Thomas, Wogrin, and Nancy in *Rule-Based Programming with OPS5*, Morgan Kaufmann, San Mateo, CA (1988).
17. "MMU FDIR Automation Task Final Report (NASA Contract NAS9-17650)," Prepared by McDonnell Douglas Astronautics Engineering Services, NASA Crew and Thermal Systems Division, JSC Houston, TX (Feb 1988).
18. Lee Becker, Peter Green, and Jayant Bhatnagar, "Evidence Flow Graph Methods for Validation and Verification of Expert Systems," NASA Contractor Report 181810, NASA Langley Research Center, Hampton, Virginia 23665 (July 1989). Prepared by Worcester Polytechnic Institute, Worcester MA 01609.
19. E. Shapiro, "The Family of Concurrent Logic Programming Languages," *ACM Computing Surveys*, 21, 3, pp. 413-510 (Sept 1989).
20. J. Giarratano and G. Riley , "Expert Systems Principles and Programming," *Chapter 11, PWS-KENT Publishing Company*, Boston, MA (1989).

Report Documentation Page

1. Report No. NASA CR-187505	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Translating Expert System Rules into Ada Code with Validation and Verification		5. Report Date June 1991
		6. Performing Organization Code
7. Author(s) Lee Becker, R. James Duckworth, Peter Green, Bill Michalson, Dave Gosselin, Krishan Nainani, Adam Pease		8. Performing Organization Report No.
		10. Work Unit No. 549-03-31-50 488-80-04-01
9. Performing Organization Name and Address Worcester Polytechnic Institute 100 Institute Road Worcester, MA 01609		11. Contract or Grant No. NAG1-964
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225 **		13. Type of Report and Period Covered Contractor Report
		14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor: Sally C. Johnson
WRDC Technical Monitor: Victor Clark
Final Report

**US Air Force
Wright Research & Development Center
Wright-Patterson AFB, OH 45433

16. Abstract

The purpose of this ongoing research and development program is to develop software tools which enable the rapid development, upgrading, and maintenance of embedded real-time artificial intelligence systems. The goals of this phase of the research were to (a) investigate the feasibility of developing software tools which automatically translate expert system rules into Ada code and (b) develop methods for performing validation and verification testing of the resultant expert system. A prototype system was demonstrated which automatically translated rules from an Air Force expert system into executable Ada code which ran in real-time. Also, a prototype test and evaluation system was demonstrated which detected errors in the execution of the resultant system. This report described the method and prototype tools for converting AI representations into Ada code by converting the rules into Ada code modules and then linking them with an Activation Framework based run-time environment to form an executable load module. This method is based upon the use of Evidence Flow Graphs which are a data flow representation for intelligent systems. It also described the development of prototype test generation and evaluation software which was used to test the resultant code. This testing was performed automatically using Monte-Carlo techniques based upon a constraint based description of the required performance for the system.

17. Key Words (Suggested by Author(s)) Knowledge-Based Systems Expert Systems Software Validation	18. Distribution Statement Unclassified - Unlimited Star Category 62

19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of pages 115	22. Price A06

NASA FORM 1626 OCT 86