NASA Contractor Report 179441

Expert Systems for Real-Time Monitoring and Fault Diagnosis

S.J. Edwards and A.K. Caglayan

Contract NAS 2-12725
April 1989

National Aeronautics and
Space Administration

NASA Contractor Report 179441

Expert Systems for Real-Time Monitoring and Fault Diagnosis

S.J. Edwards and A.K. Caglayan
Charles River Analytics Inc., 55 Wheeler Street, Cambridge, Massachusetts 02138

Prepared for
Ames Research Center
Dryden Flight Research Facility
Edwards, California
Under Contract NAS2-12725

1989

National Aeronautics and
Space Administration
Ames Research Center
Dryden Flight Research Facility
Edwards, California 93523-5000

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

This report summarizes the research and development results of the SBIR Phase I study entitled "Expert Systems for Real-Time Monitoring and Fault Diagnosis" supported by NASA Dryden under Contract No. NAS2-12725. The major aim of this study is the definition, design and prototype demonstration of a knowledge compiler concept which retains the desirable attributes of expert systems during the development stage while producing an efficient conventional embedded code for real-time onboard expert system applications. In this study, we generalize this concept into a Rule Set Processor (RSP) method allowing the specification of topological and procedural application knowledge for time-critical applications, the interactive development of an expert system based on this specification, and the integration of a compiled version of this knowledge into conventional time-critical application software. For physical systems composed of interconnected elemental dynamic objects, RSP provides a knowledge representation facility which allows the specification of topological information about the physical interconnection among these elemental dynamic objects and procedural information about the functional dynamic behavior of the dynamic elements. Moreover, RSP provides a hierarchical dynamic representation mechanism allowing multiple representations of a subsystem at several levels of abstraction. Under the Phase I effort, a preliminary specification of the RSP design has been completed and a prototype RSP implementation has been developed in Ada.

1.1 Real-Time Monitoring and Fault Diagnosis

Real-time fault monitoring and diagnosis algorithms are crucial in building highly reliable systems. These software-implemented hardware fault tolerance algorithms have been used to increase system reliability for a given redundant hardware configuration, or to reduce hardware redundancy for a given reliability figure. Such algorithms have been applied to the detection, isolation, and compensation of failures in various components (sensors, actuators, valves, linkage, circuitry, etc.) in electromagnetic, electronic, mechanical, and hydraulic systems. For instance, BIT (Built-In-Test) for on-line diagnosis of Line Replaceable Unit's (LRU's) and failure detection and isolation (FDI) algorithms for on-line diagnosis of sensor and actuator failures are commonly used on current generation aircraft.

The major problem in current real-time fault monitoring systems is the high rate of false alarms, i.e. "retest OK" and "cannot duplicate" conditions (Malcolm and Highland 1981). One of the major reasons for these deficiencies is the limitations in the model assumptions that the monitoring algorithm is based upon. These limitations are , in turn, due to the inaccuracy of the analytic representation on which the BIT/FDI algorithm is based on.

Faced with the high rate of false alarms in a conventional approach, a monitor designer has only one choice: increase the accuracy of the numerical model on which the monitoring algorithm design is based. This approach necessarily increases the complexity of the monitoring algorithm. Expert systems offer an alternative approach to this problem: model the deficiencies of the monitoring algorithms using a rule-based approach.

Another major reason for these deficiencies is that most BIT/FDI implementations do not make use of the information about the unit's operational environment (e.g., status of other interconnected units,

temperature, RF interference, power supply conditions) and the operating conditions for the vehicle that the unit is located in (e.g., aircraft maneuver, vibration).

The conventional approach to fault diagnosis system design does not provide a representation capability allowing the easy integration of such high level knowledge (e.g., topological structure, maintenance history). Here again, expert systems offer an alternative approach allowing the easy integration of such knowledge into the monitoring system design. For instance, for non-real-time fault diagnosis applications, expert system based approaches have been demonstrated in the maintenance diagnostics area (Davis 1988).

Qualitative reasoning based on symbolic representation of domain knowledge applied by expert systems can enhance the performance of fault/event monitoring systems. For instance, when used as a supervisory decision maker, such an expert system can ignore the failure indication of the underlying algorithm when deemed to be a false alarm and declare a failure indication when deemed to be a missed detection as dictated by the Rule Base. Furthermore, such research would lay down the rules of integrating expert systems to the design of new monitoring systems.

The successful application of expert systems technology to onboard fault diagnosis problems in the aerospace domain requires the development of expert systems that operate in real-time. Problem solving in a real-time environment is different from that faced in conventional applications of expert systems which presuppose a high degree of human interaction during the problem solving process. Further, the powerful explanation feature of the inference mechanism in an expert system is likely to be neither required nor desirable during on-line execution of fault diagnosis algorithms. We thus see a number of unique attributes of real-time expert systems-based problem solving, in the fault diagnosis area, including:

- Data and the associated facts deduced from the data are not static but dynamic. Hence, problem solving requires temporal reasoning to handle the implications of event sequencing and temporal interdependence.

- An expert system for real-time fault diagnosis should interface with onboard sensor measurement data and other conventional real-time software programmed using a procedural language.

- Problem solving in a real-time fault diagnosis environment requires both numeric and symbolic reasoning. Hence, an integrated numeric and symbolic knowledge representation is needed in any model of the domain knowledge.

- In a real-time fault diagnosis domain, problem solving requires the handling of unscheduled events on an interrupt basis according to their importance. Hence, a time-varying attention allocation strategy is needed in solving a problem for such a real-time environment.

- Finally, a guaranteed response time is required for problem solving in in a real-time fault diagnosis environment. Therefore, a decision strategy producing the best possible answer within the deadline constraints is required.

1.2 Expert Systems Overview

An expert system is a computer program that can perform a task normally requiring the reasoning ability of a human expert (Stefik et al. 1982). Expert systems are highly specialized according to their application domains. Recent interest in artificial intelligence is mainly due to the success of expert systems in various applications: SOPHIE in computer assisted instruction (Brown et al. 1974), MYCIN in medical diagnosis (Shortliffe 1976), PROSPECTOR in oil exploration (Duda et al. 1978), and DENDRAL in biology (Buchanan and Feigenbaum 1978). Current expert systems research for time-critical aerospace applications include Experimental Expert Flight Status Monitor (EEFSM) in flight control systems monitoring (Regenie and Duke 1985), (Duke and Regenie 1985), (Disbrow et al. 1986), Faultfinder for building diagnostic expert systems in onboard aircraft applications (Palmer et al. 1987); (Schutte et al. 1987), (Abbott et al. 1987), and FIXER for automatic failure management in Space Station applications (Malin 1987).

Although any program solving a particular problem may be considered to exhibit expert behavior, expert systems are differentiated from other programs according to the manner in which the application domain specific knowledge is structured within a program. In particular, expert system programs partition their knowledge into the following three blocks: Data Base, Rule Base, and Inference Engine.

In other words, the knowledge about the application domain is compartmentalized rather than distributed throughout the program. The Data Base contains the facts about the application domain. The Rule Base contains the set of rules specifying how facts in the Data Base can be combined to generate new facts and form conclusions. The Inference Engine determines the construct of reasoning in the application of the rules. For instance, the diagnostic system MYCIN starts from the symptom facts in order to find the conditions causing the symptom. This manner of reasoning is called "backward chaining." In contrast, "forward chaining" inference starts with the established facts to find a set of consistent conclusions.

The partitioning of application domain knowledge in expert systems allow the incremental addition of rules to the Rule Base without major revisions to the program. Moreover, the expert system can explain the reasoning chain by recording the rules as they are applied.

While expert systems have been traditionally built using collections of rules based on empirical associations, interest has grown recently in knowledge-based expert systems which perform reasoning from first principles such as those based on representations of structure and function knowledge. For instance, an expert system for digital electronic systems troubleshooting is being developed by using a structural and behavioral description of digital circuits (Davis et al. 1982), (Davis 1983, 1984, 1987). Qualitative Process (QP) theory (Forbus 1982, 1984, 1987, 1988) is another approach which allows reasoning from first principles using the representation of causal behavior based on a qualitative representation of numerical knowledge using predicate calculus. QP theory is a first order predicate calculus defined on objects parameterized by a quantity consisting of two parts: an amount and a derivative, each represented by a sign and magnitude. In Qualitative Process theory, physical systems are described in terms of a collection of objects, their properties, and the relationships among them within the framework of a first order predicate calculus.

In applying QP theory to physical dynamic systems such as aircraft fault diagnosis problems, the bottoms-up approach in getting the qualitative rules from low levels of elemental descriptions can possibly yield erroneous results at higher levels (Govindaraj 1987). In contrast, finding qualitative rules at high levels using a complete knowledge of the system via reduced order modelling would not be susceptible to such problems. Consider the following example for a QP description (where the magnitudes have been omitted for clarity) of altitude hold and autothrottle subsystems on an aircraft:

If altitude drops, then the altitude hold system pitches the aircraft up.
If the aircraft pitches up, then the aircraft speed decreases.
If the aircraft speed decreases, then the auto throttle increases thrust after a time interval.
If the thrust goes up, then the aircraft speed increases.

Note that the sequence of qualitative rules are the kind of explanations that a knowledge engineer is likely to get when questioning an FCS designer. Similarly, these rules can be obtained by a system theoretic approach with reduced order modelling. In this example, Qualitative Process theory can explain, through symbolic reasoning, the oscillations in sink rate and thrust if the altitude-hold and auto throttle subsystems are not properly designed. In such a case, when the aircraft reaches its desired altitude, the thrust would be higher than the required trim value with resultant overshoots in both altitude and speed. This event would fire another set of rules, and the cycle would continue. However, it is not clear that one can get the same rules at this level of abstraction if we apply QP approximation to an elemental level such as flight control law, aircraft dynamics, actuators, etc.

For fault diagnosis in digital circuit applications, Davis advocates reasoning from first principles starting with simple hypotheses, keeping track of simplifying assumptions made, and using multiple representations (e.g., both physical and functional representation of a digital circuit) (Davis et al. 1982) and (Davis 1983, 1984, 1987). Multiple representation approach is analogous to Rasmussen's hierarchical knowledge representation at several levels of abstraction (Rasmussen 1985) used in modelling human problem solving strategies for complex systems.

Rasmussen introduces an abstraction hierarchy in modelling human fault diagnostic strategies. This hierarchy is two dimensional. The first is the functional layers of abstraction for the physical system: functional purpose, abstract function, generalized function, physical function, and physical form. The second is the structural layers of abstraction for the physical system: system, subsystem, module, submodule, component. Using a qualitative approximation method based on a simplified version of such a functional hierarchy, a training system for marine engineers of a steam power plant has been developed by Govindaraj (1987).

Current commercially available expert system building tools (shells) are not generally applicable to building expert systems for onboard fault diagnosis applications due to the following reasons (Laffey et al. 1988): 1) the shells are not fast enough; 2) the shells have insufficient facilities for temporal reasoning; 3) the shells are not easily embeddable into conventional high level programming languages and most cannot run on numeric microprocessors used for embedded applications; 4) the shells have insufficient facilities for devoting attention to significant events; 5) the shells are not designed to accept onboard sensor data; 5) the shells have no

integration with a real-time clock and do not handle hardware interrupts; and 7) the shells cannot provide guaranteed response times.

As discussed in (Gupta 1985), most interpretive expert system shells spend 90% of their time in matching the current facts against the antecedent of rules in their rule base. Hence, an expert system development approach where the interpretive processing is performed off-line would offer a substantial execution time improvement. Similarly, the execution efficiency is a strong function of the knowledge representation facilities employed in the expert system shell. For instance, an approach based on multiple hierarchical representations of a physical system and using forward chaining would have a linear execution time complexity as compared to a rule based system with forward chaining having exponential time complexity.

For ease of integration into conventional high level programs, programming language of the expert system shell is an important choice. For instance, the choice of a programming language commonly used for embedded application such as Ada or C would be advantageous from an integration viewpoint. Moreover, such an expert system would be easily portable to microprocessors commonly used for embedded applications (e.g., 1750A, 80386, 68020). Moreover, the language constructs for handling real-time issues (tasking, interrupt servicing, exception handling) would be available to such an expert system development tool.

As discussed by (Laffey et al. 1988), there are two formal definitions for real-time expert systems: the expert system is said to exhibit real-time performance of a) it is predictably fast enough for the process being served, or b) if it can provide a response within a given time limit.

For real-time fault diagnosis applications, even if an expert system satisfies both of these premises, it would still not be sufficient for inclusion in an embedded application since the quality of the response would determine its inclusion in an onboard time-critical system. We believe that the following is an appropriate criterion for real-time expert systems for embedded applications: an expert system is said to exhibit real-time performance if the execution speed of a standalone compiled version of the expert system for a fixed application is comparable to the speed of a real-time conventional program written to solve the specific application at hand.

The integration of expert systems technology into time-critical applications presents new challenges due to the unique attributes of these applications. For instance, most expert systems have usually been implemented as standalone computer programs that presuppose a high degree of human interaction will be available during the problem solving process. While this approach is quite satisfactory for many naturally interactive applications, immediate human interaction is neither available nor desirable in time-critical avionics monitoring applications. Similarly, the powerful explanation feature of the inference mechanism in expert systems is also neither required nor desirable during on-line execution in these applications.

1.3 Summary of Approach and Results

As discussed by Duke et al. (1986), what is needed for real-time onboard expert systems development is a knowledge compiler for converting the developed knowledge base into a conventional program, thus retaining the

desirable attributes of the expert systems during the development stage while producing an efficient conventional code for a target embedded microprocessor. In this study, we generalize this concept into a Rule Set Processor (RSP) method allowing the specification of topological and procedural application knowledge for time-critical applications, the interactive development of an expert system based on this specification, and the integration of a compiled version of this knowledge into a conventional time-critical application.

The target application domain of our expert system development tool is real-time fault diagnosis applications for physical systems composed of interconnected elemental dynamic objects. Our expert system shell allows multiple hierarchical representations of such dynamic systems at several levels of abstraction. Our work generalizes Davis' approach to real-time dynamic systems in that the RSP allows the definition of elemental dynamic objects such as an integrator, actuator, sensor, etc. and specification of both the physical interconnection among these elements and their functional dynamic behavior. Hence, our physical system model generalizes the conventional functional dynamic system representation (state-space, transfer function, etc.) to include topological representation (physical interconnections, structural description). In addition, RSP includes an inheritance mechanism such that a dynamic object can be defined in terms of previously defined dynamic elements. Moreover, RSP provides a hierarchical dynamic representation mechanism allowing multiple representations of a subsystem at several levels of abstraction associated with each element.

Under the Phase I effort, a preliminary specification of the RSP design has been completed. In addition, a prototype implementation of the RSP has been programmed using Ada. The prototype RSP includes the BNF specification of the User System Description Language (USDL), a parser, and an interpreter. The following is a brief functional summary of these major RSP components:

- User System Description Language (USDL) supports both topological knowledge (i.e., system, blocks, paths, externals) and procedural knowledge (i.e., ruleset, rules, expressions) about dynamic systems.

- RSP I parser translates the USDL source after a lexical, syntactic, and semantic check into an Internal System Description (ISD) suitable for interpretation.

- RSP I interpreter interactively interprets the ruleset determined by the selected user command (e.g., simulate, diagnose) using the error-free ISD.

The proposed integration of expert systems technology with existing time-critical applications would provide a significant upgrade potential of existing monitoring systems. Many time-critical applications today have been solved by use of a conventional programming language with the human supplied expertise translated by a programming team. Often, the solution for a time-critical problem is also based on earlier conventional language treatments of a similar problem - a method employed in part due to the high cost of original development and testing with new systems, new human experts, and new programming staff. Another goal of our investigation is to demonstrate an alternative approach to the implementation of an expert system - an approach that allows for the integration of current expert system technology with existing time-critical application solutions.

1.4 Outline of the Report

In Chapter 2, we outline the desirable attribution of a real-time expert system development environment in the context of an actuator failure detection and isolation system example. The RSP knowledge representation approach and the USDL semantics are discussed in Chapter 3. In Chapter 4, we discuss the expert system development cycle using the RSP and provide an example. Chapter 5 contains the architecture and software components of RSP. The report ends with Chapter 6 providing the conclusions and recommendations.

2. DESIRED ATTRIBUTES OF RSP FOR DYNAMIC SYSTEMS

In this chapter, we discuss the desirable attributes of a Rule Set Processor architecture in the context of an actuator failure detection and isolation (FDI) system example. The selected sample application is from a Self-Repairing Flight Control System (Caglayan et al. 1987). The objective of the actuator FDI system is to detect, isolate and classify actuator failures on the Control Reconfigurable Combat Aircraft. The actuator FDI decisions are used by a reconfiguration strategy to reconfigure the aircraft flight control law after impairment to provide safety of flight and to recover maximum performance. Here, we discuss the various implementations of such an FDI system: a procedural language implementation, a rule-based expert system implementation and a hybrid implementation.

2.1 Actuator FDI System - Two Implementations

The actuator FDI system for the CRCA application has been implemented as two standalone applications: one as a single FORTRAN program driven by the CRCA simulation through a file interface, and second as a rule-based expert system using CLIPS (Giarratano 1987) driven by the CRCA simulation through a similar file interface.

CLIPS - C Language Integrated Production System - is a tool for the development of rule-based expert systems. CLIPS provides a powerful rule syntax and an inference engine based on the Rete match algorithm (Forgy 1982). We have selected CLIPS for this example since it is written in C, embeddable to other programs written in different languages (C, Ada, FORTRAN), and portable across various hardware platforms. Here, we summarize and compare the two implementations.

The implemented actuator FDI algorithm functions as follows. At each sampling instant, the actuator command and surface position measurements are read in. Using a fixed length moving window of the surface position measurements, estimates for the actuator command and rate of change are computed. If the computed rate is greater that the actuator rate limit, or if the actuator position measurement exceeds the maximum or minimum position limits, or if the difference between the actual and estimated command is greater than a specified threshold, then the actuator is declared as failed. In case of a failed sensor, the actuator failure is classified as either runaway or floating or stuck through further tests.

Table 2.1 Attributes of Actuator FDI Program (FORTRAN Version)

	Source Code (No. of Lines)	Executable Image (Kb)
Initialization/ run-time I/O	40	8
Estimation/FDI	198	13
Total	238	21

Table 2.2 Attributes of Actuator FDI Expert System (CLIPS Version)

	Source Code (No. of Rules)	Executable Image (Kb)
Initialization	13	--
Run-time I/O	8	--
Estimation	8	--
FDI	19	--
Total	48	352

Tables 2.1 and 2.2 show a comparison of two implementations. As seen from the tables, the FORTRAN version consists of 238 lines of code. In contrast, the CLIPS version contains 40 rules. The executable image of the FORTRAN version is 21 Kb whereas the rule-based expert system equivalent is 352 Kb. Since the standalone run-time CLIPS is 278 Kb, the rule base introduces an additional 74 Kb. Since increasing software size imposes additional weight requirements on an aircraft (more memory, wires, power, etc.), the comparison underscores the importance of generating a tight expert system code for embedded applications.

Figure 2.1 shows the rule for asserting an actuator failure based on the difference between actual and estimated actuator commands. Figure 2.2 shows the rule for asserting a locked actuator failure based on whether the actuator is following the local angle of attack or not.

```
(defrule check-for-failure
   (umave ?umave-val)
   (ym ?ym-val)
   (comdif ?comdif-val)
   (cmdthr ?cmdthr-val)
   (loop ?count)
   (surface ?count ?name)
   ?actuator <- (actuator surface    ?name
                          status     ok
                          tick       1
                          time       ?val
                          sp-sensor  ?sps
                          sp-command ?spc
                          pos-lim-mx ?plmx
                          pos-lim-mn ?plmn
                          rate-limit ?rl)
   (test (>= (* ?comdif-val ?comdif-val) (* ?cmdthr-val ?cmdthr-val)))
   =>
   (retract ?actuator)
   (fprintout t "Failure detected: " ?name crlf)
   (assert (actuator surface    ?name
                     status     failure
                     tick       1
                     time       ?val
                     sp-sensor  ?sps
                     sp-command ?spc
                     pos-lim-mx ?plmx
                     pos-lim-mn ?plmn
                     rate-limit ?rl))
   (assert (sfapos ?name ?umave-val))
   (bind ?sfalfa-val (* ?ym-val 57.29578018))
   (assert (sfalfa ?name ?sfalfa-val)))
```

Figure 2.1: Rule for Asserting an Acuator Failure

```
(defrule if-failure-check-locked
   (loop ?count)
   ?check <- (check-isolation)
   (surface ?count ?name)
   ?actuator <- (actuator surface     ?name
                          status      failure
                          tick        1
                          time        ?val
                          sp-sensor   ?sps
                          sp-command  ?spc
                          pos-lim-mx  ?plmx
                          pos-lim-mn  ?plmn
                          rate-limit  ?rl)
   (alferr ?alferr-val)
   (alfthr ?alfthr-val)
   (test (> (* ?alferr-val ?alferr-val) (* ?alfthr-val ?alfthr-val)))
   =>
   (retract ?actuator)
   (retract ?check)
   (assert (actuator surface     ?name
                     status      locked
                     tick        1
                     time        ?val
                     sp-sensor   ?sps
                     sp-command  ?spc
                     pos-lim-mx  ?plmx
                     pos-lim-mn  ?plmn
                     rate-limit  ?rl))
   (fprintout t "Actuator locked: " ?name crlf)
   (assert (fail-cont)))
```

Figure 2.1: Rule for Asserting an Actuator Failure

In terms of execution speed, the CLIPS version was about 25 times slower than the FORTRAN version. Since the efficiency of the CLIPS code was not optimized, the execution speed performance can be further improved using the standard iterative techniques for improving the efficiency of production systems (Braunston et al. 1986). We suspect that any improvement beyond a 10 to 1 execution speed ratio would be hard to accomplish due to the overhead associated with pattern matching, and fact assertion and retraction.

Although not implemented, there are at least two logical hybrid implementations of this actuator FDI example. The first one would be using the FORTRAN code for reading the measurements and computing the various estimation parameters (sequential algorithmic tasks performed at every sampling instant) as user defined functions in CLIPS. This would reduce the number of rules by about 50%, increase execution speed over the standalone CLIPS version with an accompanying slight decrease in program size. The other alternative would be the replacement of FDI code in the FORTRAN version with a CLIPS call for performing the FDI decision. We suspect that the program size and execution speed of this hybrid implementation would be comparable to the first one.

2.2 External Environment Interface

Real-time fault monitoring systems have to read in data at a fixed rate from a set of sensors (e.g., FCS outputs, surface position RVDT's, differential pressure transducers, BIT results, accelerometers, rate gyros, etc.). Hence, a real-time expert system for onboard applications should support efficient input and display data functions. Since conventional expert systems development presuppose an interactive environment, an efficient repetitive data read-in and assignment facility is not available in most expert system shells.

2.3 Knowledge Representation Issues

As typified by the simple actuator FDI example, real-time onboard fault diagnosis systems require a hybrid knowledge representation allowing both structural declarative knowledge and sequential procedural knowledge. For instance, in the actuator FDI example, the description of the physical interconnection between the FCS, actuator and surface position measurement sensor requires a topological knowledge representation capability. Such a symbolic representation is ideally suited for an expert system implementation. In contrast, sequential command estimation algorithm performed at each sampling interval requires a procedural knowledge representation facility. Ideally, a real-time expert system shell should support both of these knowledge representation facilities.

Since it was originally developed using a procedural programming language, there is an important knowledge representation construct missing from the actuator FDI example: namely, the hierarchical representation of a physical system at several levels of abstraction. Such a structural knowledge representation facility is usually available in hybrid expert system shells which allow object definitions with inheritance relationships. In this example, such a hierarchical representation of an actuator can take the following form: At the highest level, the system can be described by three objects: actuator command input post, actuator subsystem and surface measurement RVDT. At this level, actuator subsystem model can, for example, be a first order model. At the next lower level of hierarchy, actuator susbystem can be further decomposed into input limiter, mode selector, mechanical bias, position limiter, and a first order dynamic system with rate limiting.

Apart from the evidence of similar diagnosis strategies employed by humans, such a hierarchical representation would enable a faster reasoning mechanism than a flat description where all elemental dynamic objects have to be tested at each iteration. Moreover, in such an inference tree, a failure declaration at a higher level may deemed to be a false alarm at a lower level based on a more accurate physical system model. Ideally, a real-time expert system shell for onboard applications should support hierarchical knowledge representation at various levels of abstraction so that both top-down diagnosis, bottoms-up simulation or hybrid failure diagnosis strategies can be employed.

2.4 Temporal Reasoning

In real-time systems, an expert system has to reason about past, present, and future events. Moreover, the temporal sequence of events has to be accounted for as well. In the theory of temporal reasoning, a number of formulations have been developed (Shoham 1988). The two most important ones are based on, first, assertions about time intervals and, second, assertions about time points. For instance, in an interval based formalism, one deals between interval relations such as before, after, overlaps, starts, finishes, etc. Such a temporal logic propagates constraints about intervals by transitivity.

Most expert system shells do not support such temporal reasoning. Only in hybrid expert systems supporting dynamic objects, objects and their links to classes can be modified at runtime. In the actuator FDI example, the temporal reasoning is implicitly contained in the length of the moving window over which the masurements are saved. In general, every physical model of a physical dynamic system would dictate a different time interval for which the input and output measurements have to be saved. For instance, for an auto regressive moving average description, such a choice is explicitly stated. Ideally, an expert system shell should support the specification of the memory attribute of a dynamic object (the time interval over which the reasoning about a fault has to be performed).

2.5 Integration into Conventional Software

Since most current embedded applications dictate either C or Ada, an expert system shell written in one of these languages would allow an easy integration into conventional software. CLIPS is an example of such a shell written in C. It is completely embeddable in other applications written in FORTRAN, Ada, and C by building an appropriate interface package. In a real-time onboard expert system, such an interface should be accomplished without incurring any significant computational overhead.

2.6 Symbolic and Numeric Reasoning

In the actuator FDI example, the reasoning about the interconnections between the actuators and surface position sensors need a topological knowledge representation which requires symbolic reasoning. Other higher level information such as hydraulic system test results, maintenance history about a specific unit can be easily incorporated into such a representation, thus allowing additional reasoning power for asserting malfunctions. In contrast, the expressions on the left hand side of if-then-else rule in the actuator FDI system involve extensive mathematical computations (e.g., computation of the command estimate). This example is a fairly simple application; in most systems, more elaborate mathematical computations (involving, for instance, operations with matrices vectors, etc.) would be needed. Hence, an ideal expert system shell for onboard real-time expert system applications should support extensive domain algebra in rule expressions.

2.7 Real-Time Response

An actuator FDI system such as the example described, has to exhibit strict real-time performance. For instance, in an unstable aircraft such as the X-29, such a system has to produce the correct answer in at most two sampling instants. Therefore, just being predictably fast enough most of the time or just providing an answer within a time limit are not satisfactory criteria for real-time performance. Hence, the worst case execution time performance of an expert system has to be determinable before embedding into a time-critical application. Therefore, a real-time expert system shell should support user defined search strategies so that the fault diagnosis strategy of the domain expert can be incorporated into the expert system design.

3. RULE SET PROCESSOR KNOWLEDGE REPRESENTATION

The primary goal of the entire project is the discovery and exploration of novel ways for implementing expert system style programming techniques for use in real time applications. The central approach towards the fulfillment of this goal is the specification of a new form of programming language along with the means to translate and interpret this new language. The new aspect of this language, the USDL (User System Design Language), is the incorporation of additional representational facilities for handling topological system information in the same way as conventional programming languages handle procedural information. The design of the USDL itself and the implementation of a program that translates and interprets the USDL are the two joint tasks, executed in parallel, that formed the main research effort.

The syntactical specification of the User System Description Language is presented as a document appendix. Goals for USDL design include: 1) to manipulate variables and expressions in a comprehensive fashion similar to that employed by conventional programming languages (e.g., Ada and FORTRAN); 2) to define and access subprograms in a block structured fashion supportive of structured programming methodology; 3) to support the usage of if-then-else rules in a forward chaining manner; 4) to allow rule clustering according to user indicated functional contexts; 5) to specify topological information regarding the interconnection of components in a general dynamic system; 6) to provide mechanisms to support multiple representation of components at various levels of abstraction - either as single objects or as entire subsystems; 7) to supply complete user control of flow of control throughout the entire system model - a combination of transversing any one system level and also moving down to expand embedded systems as required; 8) to specify topological types (aggregates of components and connections) and to bind procedural information to these types; 9) to allow rapid software prototyping by using powerful and quickly implemented compilation techniques; 10) to support interactive testing of a user system description; 11) and to support a translation of the user system model into a language suitable for implementation in a real time environment for time critical applications (a feature not readily available using conventional expert system development tools).

These were the goals for the parallel RSP prototype effort: 1) to provide a simple but functional user interface to explore usage of the User System Description Language; 2) to support the parsing of the USDL; 3) to detect and diagnose user errors in USDL programs; 4) to design and implement an Internal System Description (ISD) data structure used to construct an internal (to the RSP) version of the system model generated from the USDL program; 5) to provide a platform for the interactively directed interpretation of the ISD resulting from the translation of the user system model; 6) to show the ability to support retranslation of the ISD into a target language suitable for use in embedded computer systems; 7) and to demonstrate the feasibility of using Ada to fulfill the above mentioned goals.

3.1 A Hybrid Approach to Knowledge Representation

The User System Description Language (USDL) processed by the Rule Set Processor (RSP) is the means used to represent knowledge about a system. The USDL is a well defined language, with standards for both the representation of meaning (language semantics) and structure (language syntax). This section of

this document describes the language semantics of the USDL. The USDL BNF (syntax specification) appears as an appendix to this document.

Much of the specification of the USDL comes from commonly used block structured programming languages. For example, the expression syntax and semantics used by the USDL are roughly a subset of Ada, while scoping rules for most names are taken from Pascal. However, a most important feature of the language, a merger of specification of the topological relationships among systems and system hierarchies with procedural resources, is entirely new and so represents a novel approach to solving simulation and diagnosis requirements of general dynamic systems.

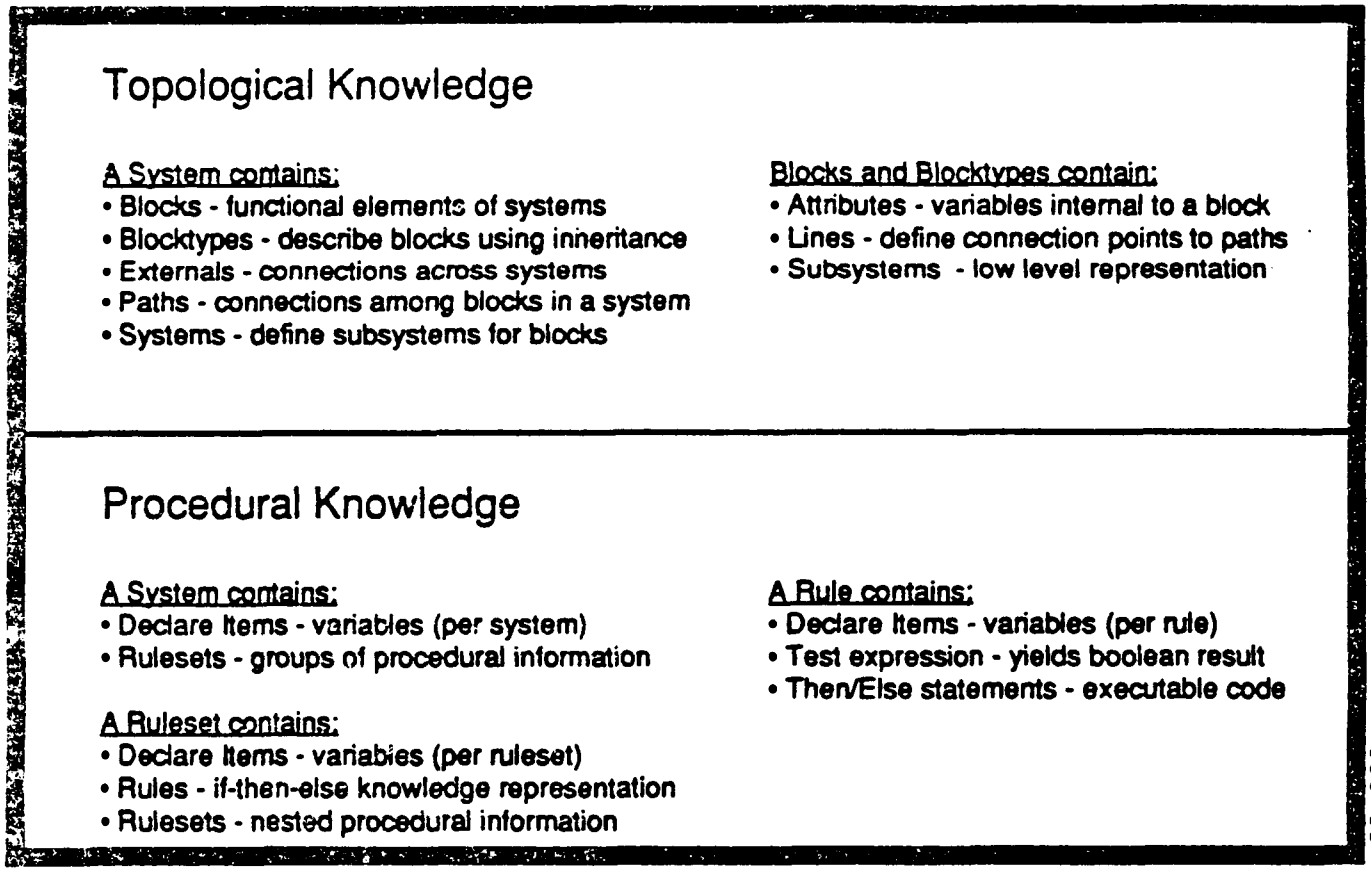
The USD Language described here is the result of a preliminary investigation into the requirements of a general purpose system description language, and as such may probably undergo revision under a Phase II development. It is unlikely that any of the current features may be deleted - although some may be changed - and it is expected that new language features will be added as necessary to improve performance and increase productivity. The USDL referred to in this document is officially known as "USDL 1.1"; future versions will be assigned new version numbers as appropriate.

The kind of knowledge commonly used to represent sequences of actions and algorithms is referred to as procedural information. Procedural information is usually realized as mostly linear arrangements of instructions. Traditional computer languages are examples of knowledge representation tools using primarially procedural information strategies. These languages work well for those problems that can be treated in an easily reducible, one step at a time, sequential approach.

The kind of knowledge commonly used to represent the relationships among multiple entities in a fixed but arbitrary arrangement is referred to as topological information. Topological information is usually realized as a graph structure composed of a set of nodes with arcs forming the various interconnections of the nodes. Traditional representations of topological knowledge include both graphical approaches (box and arrow diagrams) and computer generated network data structures. Graphic hardcopy diagrams can be easily made, but are not easily represented by traditional computer languages. It is possible to dynamically produce data structures to represent topological information, but the design and development needed for this task is often time consuming.

The problem of the computer modeling of a general system requires both types of knowledge for a complete representation. For each component (block) of the system, procedural knowledge is required to model the actions of that component for certain inputs and outputs. From the viewpoint of the component, it is unimportant to know from where the input values arrive or to where the output values are sent; it is only necessary to create (using procedural knowledge) the correct output results from the given current and past input data. For the entire arrangement of the components that make up the general system, topological information is required to accurately move data throughout the system model. From the viewpoint of the network that connects the components, it is unimportant to know how the various values transmitted are generated or used; it is only necessary to correctly transmit (using topological knowledge) these values among the components and to and from the world outside of the model.

The USDL is designed for the realization of both procedural and topological information about general systems. Figure 3.1 illustrates the elements of a system model using the USDL.



Elements of a System using the User System Description Language

Figure 3.1

3.2 USDL Semantics: Systems

A system is represented by the USDL as a collection of resources bracketed by a system description header and tail. An example:

```
-- ***************
-- Start of example.
-- Here is an example system description.
-- The system name is "example_system_1".
-- Note that comments are always preceded by a double dash.
--

system example_system_1 is
 begin

  -- Various system resources appear here.

 end example_system_1;

--
-- End of example.
-- ***************
```

The reserved word "system" introduces a system description. (Reserved words have special meaning for the RSP and are unavailable for use as user identifiers.) It is followed by the user defined name to identify the system. The reserved words "is" and "begin" follow the system identifier. After the reserved word "begin", an arbitrary number of system resources (defined below) may appear, all of which are thereafter associated with the enclosing system. A system description is concluded with the reserved word "end" and a semicolon.

The system name may be optionally repeated immediately prior to the semicolon for the sake of clarity. Note that all reserved words in the USDL must appear using only lower case letters. User identifiers may use a mixture of upper and lower case letters along with digits and the underscore character. All identifiers must begin with a letter. Identifiers may be arbitrarily long and all characters are considered significant.

Each system description defines an enclosing name scope. This enclosing scope is used to control access to the names defined as a result of resource definitions within the system description. All names, except for a few classes of identifiers described later, defined at a given scope level are available only within that scope and only after their defining descriptions. Only the name of a system along with certain identifiers (external labels and ruleset names) defined at that system level can be referenced outside of the system description. This feature, also found in all modern block structured languages, helps to limit unnecessary complexity by restricting identifier access to only those regions that require such access.

When a system description is given, it actually describes a system type - a template that can be used as a resource of a larger, enclosing system one or more times. For example, a particular system may consist of ten subsystems, all of which are identical except for their arrangement within the enclosing system. The USDL facilitates this usage by allowing the one time definition

of the subsystem (as a type) and then referencing this system as required as a building block in an enclosing system.

A USDL main program source file, a User System Description (USD), is just a single system description. This single system type, because it appears at the outermost possible level, is interpreted as the actual system being modeled. One and only one such system description may appear at the outermost level of a USDL source file. (Important note: the outermost system present acts as a root to the system model and so is commonly referred to as the "root system".)

System descriptions in the USDL may include various resources that define both the topological and procedural information required to completely specify the system model. Each of these resources are defined below; Table 3.1 lists all the available resources and their functions:

Table 3.1: USD Resources and Functions

Resource	Purpose
Block	A node in a system graph; has inputs/outputs
Blocktype	Definitions used to help create blocks
Declare	Defines and allocates a user defined scalar
External	Connects inputs/outputs with enclosing system
Path	An arc in a system graph; connects nodes
Ruleset	Contains rules and other procedural information
System	A subsystem type; same format as enclosing system

The block, blocktype, external, and path resources are primarily concerned with the representation of topological knowledge. Ruleset resources, which include rules with executable statements, are closely connected with the representation of procedural knowledge. Systems combine all resources (including subsystems) to merge topological and procedural knowledge.

3.3 USDL Semantics: Block

A block (a.k.a. component) may be considered as a node of the graph that makes up the immediately enclosing system. For example, if a given system describes a simple logic circuit, each gate could be represented as a separate block. For another example, a complex electro-mechanical control system may have blocks that represent entire systems (computers, actuators, sensors, filters, and mechanical linkages) - such blocks themselves could be represented by subsystems.

Here are some examples of block descriptions:

```
-- **********
-- Start of example.
-- There are three blocks here, "block_52", "switch_18", and "stick_1".
--

system example_system_1 is
 begin

  -- Various system resources appear here.
```

```
   -- Here is a block that uses a general type.

   block block_52 is general
    begin

     -- Various block resources appear here.

    end block_52;

   -- Here is a block that uses a specific named type.

   block switch_18 is type switch_block_type
    begin

     -- Various block resources appear here.

    end switch_18;

   -- Here is another block that uses a specific named type and has
   -- no other resources:

   block stick_1 is type stick_block_type;

 end example_system_1;

-- End of example.
-- ***************
```

The reserved word "block" introduces a block description. It is followed by the user defined name to identify the block. The reserved word "is" follows the block identifier. Blocks may inherit types from blocktype descriptions (described below) or may have general types. If a given block has no inherited resources, the reserved word "general" appears after the word "is" in the block header. If the block does inherit some resources, the blocktype that contains those resources is represented by the reserved word "type" followed by the appropriate blocktype identifier. In either case of type reference, the reserved word "begin" immediately follows. After the reserved word "begin", an arbitrary number of block resources (defined below) may appear, all of which are thereafter associated with the enclosing block. A block description is concluded with the reserved word "end" and a semicolon. The block name may be optionally repeated immediately prior to the semicolon for the sake of clarity.

In the case where no additional block resources are defined, then the "begin end <block-id>" may be deleted. For example:

```
block no_extra_resources is type complete_block_type;
```

is a complete block description.

Blocks and blocktypes contain an arbitrary number of block resources. There are three kinds of block resources: attributes, lines, and subsystems. These resources are described below. These attributes resources are given in Table 3.2.

Table 3.2: Elements of USD Resources

Resource	Elements Resources
Block	Attributes, lines and subsystems
Blocktype	Attributes, lines and subsystems
Declare	Identifier, basetype indicator
External	Identifier
Path	Block and line identifiers
Ruleset	Declarations, rules, nested rulesets
System	Blocks, blocktypes, declarations, externals, paths, rulesets

3.3.1 USDL Semantics: Block - Block Attributes

An attribute resource acts as a statically allocated variable bound to a given block. Attributes allow for the parameterization of block function by associating scalar values with particular blocks or blocktypes. For example, a block representing a simple switch with a single input and a single output would have an attribute with a boolean value indicating whether or not the input-output path was currently conducting. A much more complicated block that describes a multipole bandpass filter may require many attributes with floating point values used to describe the filter transfer function.

Attributes have identifiers and basetypes. There are three basetypes used in the USDL. These basetypes are: boolean, float, and integer. (These three words are reserved by the USDL.) A boolean basetype value may be either true or false, a float basetype value takes on floating point values, and an integer basetype indicates integral values. (The words "true" and "false" are also reserved.)

An attribute may be given an initial value as part of their definition. The value of an attribute may be changed later unless it is declared to be constant.

Here are some examples of attribute descriptions:

```
-- ***************
-- Start of example.

system attribute_example_system is
 begin

  -- Various system resources appear here.

  block simple_switch_1 is general
   begin
```

```
  -- Here is a simple attribute that gives
  -- the switch conducting status.
  -- The words "attribute", "is", "basetype",
  -- and "boolean" are reserved.

  attribute is_closed is basetype boolean;

 end simple_switch_1;

block simple_switch_2 is general
 begin

  -- Here is another switch example with an initial value (closed).
  -- The words "default" and "true" are reserved.

  attribute is_closed is basetype boolean default true;

 end simple_switch_2;

block simple_switch_3 is general
 begin

  -- Here is another switch example with
  -- a constant initial value (open).
  -- The words "constant" and "false" are reserved.

  attribute is_closed is constant basetype boolean default false;

 end simple_switch_3;

block several_attributes_block is
 begin

  -- Note that identifiers used within a block description must be
  -- unique within that description.

  attribute flag_1 is basetype boolean;
  attribute flag_2 is basetype boolean default true;
  attribute flag_3 is basetype boolean default false;
  attribute flag_4 is constant basetype boolean default true;
  attribute flag_5 is constant basetype boolean default false;

  attribute x_1 is basetype float;
  attribute x_2 is basetype float default 2.71828;
  attribute x_3 is basetype float default -3.14159;
  attribute x_4 is constant basetype float default 0.0;
  attribute x_5 is constant basetype float default 2.59e+06;

  attribute ival_1 is basetype integer;
  attribute ival_2 is basetype integer default 0;
  attribute ival_3 is basetype integer default -312;
  attribute ival_4 is constant basetype integer default 32767;
  attribute ival_5 is constant basetype integer default -1;

 end several_attributes_block;
```

```
end attribute_example_system;

-- End of example.
-- ***************
```

The reserved word "attribute" introduces an attribute resource. It is followed by the user defined name that identifies the attribute. The reserved word "is" follows the attribute identifier. If the attribute value is to remain constant, the reserved word "constant" follows immediately. The attribute basetype is then defined by the reserved word "basetype" followed by one of the three available basetype indicators. An initial value can then be specified by the reserved word "default" and then a literal value of the appropriate basetype. A semicolon concludes an attribute resource.

Note that the attribute identifier is always required but the constant and default value clauses are optional. The basetype clause is required if no basetype information for that attribute has been inherited from a previously appearing blocktype description. Attribute identifiers must be unique within a block.

Attributes can be referenced as parts of expressions (described below) in the enclosing system description. An attribute reference is of the form:

component_id.attribute_id

where presence of the component identifier in an attribute reference allows for disambiguation where two or more components have attributes that happen to have the same name. The symbol between the component and attribute identifiers is a period and is referred to as a "selector" operator.

3.3.2 USDL Semantics: Block - Block Lines

In order that blocks may receive and transmit data (in this case, discrete scalar values), a mechanism defining the inputs and outputs of a given block is necessary. The USDL uses the "line" block (and blocktype) resource. Each line resource within a block describes a single data channel either in or out of the block. A complete line resource includes an identifier, a mode indicator (flow data direction) clause, a basetype clause, and an optional history (recent value record) clause.

Each line of a block has an associated value. This value represents some scalar value at a particular moment in time corresponding to some potentially measurable quantity at that line. The basetype of this value may be boolean (binary values, single pole switches, etc.), float (voltages, pressures, currents, forces, etc.), and integer (multipole switches, discrete positioned mechanisms, etc.) as required by the application. Because the modeling of dynamic systems requires not only a current value at a given point, but also recent values at the same point, the USDL allows for automatic recording of recent measurements in addition to the current measurement of any line value in the system. The count of recorded history (default one) values at a line is specified using a history clause. History recording is particularly useful in modeling systems with components with behavior dependent not only on current inputs but also past recent input and output values.

Here are some examples of line resources:

```
-- ***************
-- Start of example.
--

system line_system_example is
 begin

  -- Various system resources appear here.

  block low_pass_filter_53 is general
   begin

    -- Here are attributes for the filter:

    attribute cut_off is basetype float default 20.0;
    attribute gain is basetype float 0.95;

    -- Here is the input line:

    line filter_input is mode input basetype float;

    -- And here is the output line:

    line filter_output is mode output basetype float;

   end low_pass_filter_53;


  block push_button_8 is general
   begin

    -- This is a model of a SPST momentary contact push button.

    -- Here is the mechanical input:

    line button is mode input basetype boolean;

    -- Here are the electrical connections:

    line current_in is mode input basetype float;
    line current_out is mode output basetype float;

   end push_button_8;

  -- Here is a block that keeps the last four inputs and
  -- the last two outputs:

  block debouncer_0 is general
   begin

    line db_in is mode input basetype integer history 4;
    line db_out is mode output basetype integer history 2;

   end debouncer_0;
```

```
 end line_system_example;

-- End of example.
-- ***************
```

The reserved word "line" starts a line resource. It is followed by the user defined name that identifies the line within the block. The reserved word "is" appears immediately after the line identifier. The next part of the line definition is the information flow indicator for the channel; this is referred to as the flow mode of a line and this mode clause consists of the reserved word "mode" followed by either the reserved word "input" or the reserved word "output". An "input" mode indicates that data is flowing into the block; an "output" mode indicates that data is flowing out of the block. After the mode clause, the basetype clause appears. The basetype clause defines which of the available basetypes is used for the representation of the value of the line. As with basetype clauses for attributes, a basetype for a line starts with the reserved word "basetype" followed by one of the available basetype indicators. Following the basetype information, an optional history clause appears if records of multiple recent values are required. A line history clause is the reserved word "history" followed by a positive integer constant that defines the number of records of a line's value. If no history clause is present, a default value of one is assumed. A line resource is concluded by a semicolon.

Note that the line identifier is always required and the history clause is optional. Each of the mode and basetype clauses are required if such information for that line has not been inherited from a previously appearing blocktype description. Line identifiers must be unique within a block.

Line values (current and past) can be referenced as parts of expressions (described below) in the enclosing system description. A reference of the current value of a line can be of two forms:

```
component_id.line_id
```

and also:

```
component_id.line_id.history[0]
```

where the zero inside of the brackets of the second reference indicates the current value ("time zero" or "current time plus zero"). The value inside the brackets can be an arbitrary integral expression but should, when evaluated, fall in the range of zero to (minus N), where N equals history reservation minus one. Here is an example of a reference at time "current time minus three":

```
filter_18.voltage_in.history[-3]
```

Note that the reserved word history always precedes the bracketed expression, and that the expression is never positive. History values are kept only for lines and not for attributes.

The USDL implements the passage of time in the system description as a sequence of equal interval discrete periods of duration. The system developer's selection of actual units of time in use (seconds, microseconds, etc.) is not important to the model; the RSP only requires that each interval

is equal in length to the next, and that all portions of the system model are synchonized. The advancement of model time is under user control and the RSP automatically shifts values along the history storage lists.

Of course, most real world systems function using continuous time, which is somewhat difficult to simulate with discrete digital computers. The quantized treatment of time by the USDL can be thought of as a periodic sampling of continuous time with each time dependent value in the system model being updated simultaneously.

3.3.3 USDL Semantics: Block - Block Subsystems

For a very low level system model, the individual blocks directly enclosed by the system description will represent the lowest level, undivisible components of the system. For systems descriptions at higher levels, blocks may represent entire subsystems. These subsystems in turn may have blocks that also represent lower level subsystems, and so forth all the way down to systems composed of only atomic components.

The USDL allows for the association of subsystems as a block (and blocktype) resource. Note that a given block (or blocktype) may have both a subsystem representation and a simple representation composed of only lines and attributes. The motivation for this is to provide greater developer flexibility in modeling systems: some simulation strategies may require in depth subsystem representation while many diagnostic approaches would employ top-down methods that first use a simple view of a component and investigating a block's subsystem representation only when necessary.

A block subsystem representation is the simplest of the block resources. A subsystem resource starts with the reserved word "subsystem" to indicate the presence of a subsystem representation. The name (system identifier) that identifies the subsystem type appears after the subsystem keyword, and the resource is concluded by a semicolon. Here are some examples of block subsystem resources:

```
subsystem and_gate_system;

subsystem high_pass_filter_system;
```

Note that the system identifier must correspond to a system type already defined. This is an example of the rule that each identifier in the USDL must be defined before it is used.

A maximum of one subsystem resource is permitted per block or blocktype.

Here are some examples of subsystem resource usage:

```
-- ***************
-- Start of example.
--

system subsystem_usage_example is
 begin

  -- Various system resources appear here.
```

```
-- For a block to use a subsystem resource, the indicated resource
-- must first be defined.  The following are two system descriptions
-- usable as subsystems:

system simple_switch_system is
 begin

  -- Various system resources appear here.  These resources define
  -- the characteristics of the system type "simple_switch_system",
  -- a subsystem that can be referenced by blocks in the enclosing
  -- system "subsystem_usage_example".

 end simple_switch_system;

system tricky_op_amp_system is
 begin

  -- Various system resources appearing here define a system to
  -- model tricky operational amplifiers.

 end tricky_op_amp_system;

-- With the appropriate subsystems defined, block descriptions can
-- now reference them as block subsystem resources.

block switch_1 is general
 begin

  -- Various block resources appear here.

  -- Here is the subsystem resource:

  subsystem simple_switch_system;

 end switch_1;

block op_amp_33 is general
 begin

  -- Here is an instance of the use of "tricky_op_amp_system".

  subsystem tricky_op_amp_system;

 end op_amp_33;

block op_amp_34 is general
 begin

  -- Here is another use of the "tricky_op_amp" subsystem.  Note that
  -- each such instantiation of a subsystem refers to a different
  -- copy of the subsystem; the subsystem system description actually
  -- defines a system type that can be used repeatedly among different
  -- blocks.

  subsystem tricky_op_amp_system;
```

```
   end op_amp_34;

 end subsystem_usage_example;

-- End of example.
-- ***************
```

3.4 USDL Semantics: Blocktype

Some systems described by the USDL many contain many blocks that are made up of only a few types. For example, an actively controlled beam composed of three hundred components could be modeled using only a few types of blocks (lateral and longitudinal struts, piezoelectric strain sensors, and integrated servo actuators) repeated throughout the overall system. To aid in the development of models for such kinds of systems, the USDL supports a language construct called a blocktype. A blocktype is another system resource (like a block) and appears in system descriptions in the same places a block description appears. A blocktype description is quite similar to a block description and its only use is to help with the definition of blocks (or other blocktypes).

Blocktype descriptions have the same three kinds of resources as do blocks: attributes, lines, and subsystems.

An attribute or a line may be either partially or completely defined in a blocktype. A subsystem resource, if present, must be completely defined in a blocktype description. The basic idea for the function of blocktypes is to copy the collection of block resource information from a type parent (if any) and combine it with explicit resource information in the blocktype description and then construct a new set of block resource definitions (a type) to made available for usage by blocks and other blocktypes.

Here are some examples of block and blocktype usage:

```
-- ***************
-- Start of example.

system blocktype_example is
 begin

  -- Various system resources appear here.

  -- Here is a simple blocktype declaration that defines the presence
  -- of a single attribute.

  blocktype single_scalar_blocktype is
   begin
    attribute factor is basetype float;
   end single_scalar_blocktype;

  -- Here are two usages of the above blocktype:

  block multiplier_32 is type single_scalar_blocktype
   begin
```

```
    -- Because of the type inheritance, an implicit resource of
    -- "attribute factor is basetype float;" exists for this block.

    line voltage_in is mode input basetype float;
    line voltage_out is mode output basetype float;

  end multiplier_32;

 block divider_91 is type single_scalar_blocktype
  begin

    -- Because of the type inheritance, an implicit resource of
    -- "attribute factor is basetype float;" exists for this block.

    line pressure_in is mode input basetype float;
    line pressure_out is mode output basetype float;

  end divider_91;

 -- Here is a blocktype example sequence that includes attribute,
 -- line, and subsystem resources.

 blocktype one_from_two_type is general
  begin
   line q_in_1 is mode input;
   line q_in_2 is mode input;
   line q_out is mode output;
  end one_from_two_type;

 blocktype boolean_gate_type is type one_from_two_type
  begin
   line q_in_1 is basetype boolean;
   line q_in_2 is basetype boolean;
   line q_out is basetype boolean;
  end boolean_gate_type;

 blocktype adder_type is type one_from_two_type
  begin
   line q_in_1 is basetype float;
   line q_in_2 is basetype float;
   line q_out is basetype float;
  end adder_type;

 blocktype and_nand_gate_type is type boolean_gate_type
  begin
   attribute inversion_flag is boolean;
   subsystem and_nand_system;
  end and_nand_gate_type;

 blocktype and_gate_type is type and_nand_gate_type
  begin
   attribute inversion_flag is constant false;
  end and_gate_type;

 blocktype nand_gate_type is type and_nand_gate_type
  begin
```

```
    attribute inversion_flag is constant true;
   end nand_gate_type;

  block and_gate_1 is type and_gate_type;
  block and_gate_2 is type and_gate_type;
  block and_gate_3 is type and_gate_type;

  block nand_gate_1 is type nand_gate_type;
  block nand_gate_2 is type nand_gate_type;
  block nand_gate_3 is type nand_gate_type;

 end blocktype_example;

-- End of example.
-- ***************
```

The important point to remember about blocktype usage is: when the ultimate inheritor of blocktype information (a block) is described, each attribute and line resource must be minimally defined. An attribute is minimally defined by its basetype (constant and default information is supplimental). A line is minimally defined by its mode and its basetype.

3.5 USDL Semantics: Declare

A "declare" description acts as a resource to USDL system descriptions to indicate a user declared scalar associated with that system type. USDL rule and ruleset descriptions (described below) also use declare descriptions as resources in a similar fashion.

A declare description indicates the binding of a user defined identifier with storage for a scalar of one of the available basetypes. Here are some examples of declare descriptions:

```
declare active_mode: boolean;

declare standby_voltage: float;

declare control_lever_detent_position: integer;
```

A declare description is started by the reserved word "declare". This is followed by a user supplied identifier, a colon, the desired basetype indicator, and finally a semicolon. Identifiers used in declare identifiers should be unique at the scope level of their declaration.

Items defined as a result of declare descriptions can be referenced as parts of expressions or as variables (both are described below).

Here is an example of declare descriptions appearing in system descriptions:

```
-- ***************
-- Start of example.

system declare_example_system is
 begin
```

```
  declare temporary_sum_1: float;
  declare temporary_sum_2: float;

  -- Various system resources appear here that may use the variables
  -- "temporary_sum_1" and "temporary_sum_2".

  system another_system is
   begin

    declare delta_x: integer;
    declare delta_y: integer;

    -- Various system resources appear here that may use "delta_x"
    -- and "delta_y" along with "temporary_sum_1" and "temporary_sum_2".

   end another_system;

 end declare_example_system;

-- End of example.
-- ***************
```

Items defined as a result of declare descriptions obey scoping access rules. This means that an identifier declared at one system level are accessible to interior system descriptions, unless another object with the same name is declared at an interior level. An identifier declared at one system level is not accessible at enclosing (exterior) levels.

3.6 USDL Semantics: External

Most systems have an interface to an exterior environment; this environment is either the "outside world" or an immediately enclosing system. To facilitate the transmission of values into and out of systems, the USDL has a system resource for associating connection points (a particular line of a particular component) with labels (identifiers) external to the system. Here are some examples of external descriptions:

```
external voltage_in is terminal_strip.connector_4;

external torque_xy_out is actuator_2.shaft_output;
```

There are two different interpretations of external descriptions. If an external description appears at the outermost system level, the external identifier corresponds to a connection to the world outside the system model. If an external description appears at any level interior to the outermost system level, the external identifier corresponds to a line (with the same identifier) within a block that uses the subsystem with the external description.

An example USD illustrates both cases:

```
-- ***************
-- Start of example.

system external_sample_outermost_system is
```

```
begin

 -- Here is a system type that describes a simple switch:

 system simple_switch_system is
  begin

   block sss_b1 is general
    begin
     line sss_b1_in is mode input basetype float;
     line sss_b1_out is mode output basetype float;
    end sss_b1;

   block sss_b2 is general
    begin
     line sss_b2_in is mode input basetype float;
     line sss_b2_out is mode output basetype float;
    end sss_b2;

   block sss_b3 is general
    begin
     line sss_b3_in is mode input basetype float;
     line sss_b3_out is mode output basetype float;
    end sss_b3;

   external sw_in is sss_b1.sss_b1_in;
   external sw_out is sss_b3.sss_b3_out;

  end simple_switch_system;

 -- Here are two blocks that use the above subsystem:

 block switch_in is general
  begin
   line sw_in is mode input basetype float;
   line sw_out is mode output basetype float;
   subsystem simple_switch_system;
  end switch_in;

 block switch_out is general
  begin
   line sw_in is mode input basetype float;
   line sw_out is mode output basetype float;
   subsystem simple_switch_system;
  end switch_in;

 -- Here are two external descriptions that are system resources to the
 -- outermost system:

 external current_in is switch_1.sw_in;
 external current_out is switch_2.sw_out;

end external_sample_outermost_system;

-- End of example.
-- ***************
```

3.7 USDL Semantics: Path

A path description is a system resource used to provide a connection between an output line of one block and the input line of another block. (Actually, a path may also connect an output line of a given block to an input line of the same block.) The restrictions on path definitions are that: an output must always be connected to an input, the basetypes of the input and output connections must be identical, and paths may only exist between components at the same system level.

Paths represent physical connections such as wires, struts, beams, hydraulic lines, etc. The USDL provides for mechanisms to propagate information along these paths using the "pulse" statement (detailed below). The information moved along a path consists of a single scalar value of the basetype of the input and output lines.

Here are some examples of path descriptions:

```
path alpha is from block_23.line_4 to block_38.line_3;

path p_23 is from piston_2.flow_out to valve_2.flow_in;

path from beam_support.corner_2 to brace_3.west;
```

A path description begins with the reserved word "path". An identifier may be supplied for a path, but is not required; if present, it is immediately followed by the reserved word "is". The source connection is introduced by the reserved word "from" and is specified by giving the corresponding block and line identifiers separated by a period. The destination connection point then appears with the reserved word "to" and then the corresponding block and line identifiers also separated by a period. Note that the line referenced as part of the source connection point must be declared with a flow mode of output and the line associated with the destination connection point must be declared with an input flow mode. The blocks referenced in the path description must be declared prior to the appearance of the path description.

Here is an example of a system with paths:

```
-- ***************
-- Start of example.

system example_path_system is
 begin

   blocktype simple_block_type is general
    begin
     line input_terminal in mode input basetype float;
     line output_terminal in mode output basetype float;
    end simple_block_type;

   block blk_1 is type simple_block_type;
   block blk_2 is type simple_block_type;
   block blk_3 is type simple_block_type;

   -- Paths are used to make a circular linkage:
```

```
   path alpha is from blk_1.output_terminal to blk_2.input_terminal;
   path beta is from blk_2.output_terminal to blk_3.input_terminal;
   path floyd is from blk_3.output_terminal to blk_1.input_terminal;

 end example_path_system;

-- End of example.
-- ***************
```

The USDL allows for an arbitrary number of paths in a system, and a connection point may have an arbitrary number of paths connected as long as the above directional mode and basetype matching rules are obeyed. It is not necessary for all connection points to have paths, but such a condition may be commented upon by the RSP to indicate a possibly incomplete system specification.

3.8 USDL Semantics: Rulesets

A ruleset is a language construct, analogous to a subroutine, that acts as the carrier of procedural information in the USDL. Specifically, a ruleset is composed of three kinds of ruleset resources: declarations ("declare" items, same as system level declare items), rules (containing declarations, tests and executable statements), and nested rulesets (analogous to nested systems).

A system may have zero or more associated rulesets declared as system resources. Those rulesets declared at the system level (and not those declared in other rulesets) are the only rulesets accessible as a higher system level, either by use of an "elaborate" statement (described below), or by interactively specified elaboration.

A reminder: system descriptions may contain nested systems (used as templates for block subsystems); ruleset descriptions may contain nested rulesets (these can be called from rules as described below); and systems may also have zero or more rulesets (these are the only rulesets accessible from the immediately enclosing system to the system of declaration).

Here is an example including ruleset descriptions:

```
-- ***************
-- Start of example.

system ruleset_example_system is
 begin

  system interior_system is
   begin

    -- The following ruleset is embedded in the
    -- system "interior_system":

    ruleset interior_ruleset_19 is
     begin

      -- Various ruleset resources appear here.
```

```
     end interior_ruleset_19;

    -- The following ruleset is also embedded in the
    -- system "interior_system":

    ruleset interior_ruleset_42 is
     begin

      -- Various ruleset resources appear here.

      -- The following ruleset is embedded in the
      -- ruleset "interior_ruleset_42":

      ruleset used_only_by_ruleset_42 is
       begin

        -- Various ruleset resources appear here.

       end used_only_by_ruleset_42;

     end interior_ruleset_42;

   end interior_system;

  -- Here is a ruleset embedded in the outermost system:

  ruleset main_ruleset_1 is
   begin

    -- Various ruleset resources appear here.

   end main_ruleset_1;

 end ruleset_example_system;

-- End of example.
-- ***************
```

The reserved word "ruleset" introduces a ruleset description. It is followed by the user defined name used to identify the ruleset. The reserved words "is" and "begin" follow the ruleset identifier. After the word "begin", an arbitrary number of ruleset resources (declare items, rules, and interior rulesets) may appear. After the last (if any) ruleset resource, the ruleset description is concluded by the reserved word "end" followed by a semicolon. The ruleset name may be optionally repeated immediately prior to the closing semicolon.

A ruleset description defines a name scope in a manner similar to a system description. Names defined in a ruleset description (except for the ruleset name itself) are accessible only within the ruleset description.

Rulesets embedded in the highest level system are accessible from the outside environment. The RSP interpreter treats three ruleset identifiers at this level in a special manner; these three are directly executable by using interactive interpreter commands:

Interpreter Command	Ruleset Identifier	Intended Purpose
preset	preset	Initialize external values.
simulate	simulate	Step simulated time and simulate.
diagnose	diagnose	Diagnose and report system performance.

More such specially cased interpreter commands may be added during further development.

3.8.1 USDL Semantics: Rulesets - Declarations

Ruleset descriptions may contain an arbitrary number of declare items in a fashion similar to system descriptions. A declare item in a ruleset description has the same syntax as a system description declare item (described above), and the identifier used for the declare item storage is accessible only within the enclosing ruleset and only after the appearance (definition) of the declare item itself.

Here is an example of declare items in a ruleset description:

```
-- ***************
-- Start of example.

system example_for_declare_items_in_ruleset_system is
 begin

  ruleset a_ruleset_with_some_declare_items is
   begin

     declare x: float;
     declare i: integer;
     declare a_flag_variable_with_a_long_name: boolean;

   end a_ruleset_with_some_declare_items;

 end example_for_declare_items_in_ruleset_system;

-- End of example.
-- ***************
```

For the sake of clarity, all declare items in a ruleset description should be grouped together at the start of sequence of ruleset resources that make up the body of the ruleset description.

The storage associated with declare items in a ruleset is statically allocated. This means that, although rulesets can be recursively referenced (by use of the call statement - described below), each invocation of a ruleset uses the same storage for the declare items (i.e, only a single unit of storage for a declare item is allocated). This static based style of allocation was chosen because of its potentially greater speed and simplicity for time critical embedded applications. A future revision of the USDL may offer a way to specify automatic (per invocation) storage in place of the default static allocation.

3.8.2 USDL Semantics: Rulesets - Rules

Rule descriptions appear as ruleset resources. Taken collectively, rules contain all of the procedural information about the system model. Rules also provide for the static and dynamic ordering of the execution of procedural information.

An arbitrary number of rules may appear in any ruleset. When present, rule descriptions appear serially (never recursively), in a ruleset. Rules usually appear after any declare item descriptions and after any embedded (interior) ruleset definitions. Unlike the other, topological portions of the USDL, the order in which rules appear in a ruleset is very important, as the rules embody procedural information and so the rules in a given ruleset are (nominally) executed in sequential order. Rule execution starts with the first rule in a ruleset and continues with subsequence rules in the ruleset unless redirected by certain executable statements. The purpose behind the described rule/ruleset organization is to group rules in semantically related groups and to allow user greater user control of rule search and execution by use of such context sharing. The common alternative to such rule placement is to have all the rules in a single, linear database and so require extensive search and evaluation overhead. While the latter approach is conceptually simpler, it is regrettably unacceptable for those applications that cannot tolerate conventional rule based expert systems because of their excessive time requirements.

Rules have two main parts: the declaration part and the conditional part. The declaration part appears first and consists of zero or more declare items with the same characteristics as ruleset declare items. The conditional part consists of an expression test followed by a affirmative statement (the "then" part) optionally followed by an alternative statement (the "else" part. When the rule is executed, the expression test is evaluated and, if evaluated to be true (or nonzero), the affirmative statement is executed. If the expression test evaluates to false (or zero), the alternative statement is executed (if present).

A useful convention in the current USDL is to simply use the boolean constant "true" in the rule's expression test to unconditional execute the affirmative statement. It is possible that a future version of the USDL may allow a condensed version of the above and just allow a compound statement instead.

3.8.2.1 USDL Semantics: Rulesets - Rules - Declarations

A rule may have zero or more declare items. A rule declare item associates a unit of storage with a user specified identifier. The declare item is available for reference from the point of its definition to the end of the rule description in which it is defined.

Here are some rule declare items:

```
declare q_flag: boolean;
declare counter: integer;
declare scale_factor: float;
```

Note that the declare syntax for rule declare items is identical to that of rulesets and that of systems. The reason for allowing declare item

descriptions to appear inside of these three forms is to allow for the user controlled association of names (variable identifiers) with areas of reference (systems, rulesets, and rules). Because the USDL enforces name scopes for the above forms (i.e, names may not be referenced outside of their scopes), it is easier for the system developer to enforce good programming style by reducing opportunities for inadvertent object references.

3.8.2.2 USDL Semantics: Rulesets - Rules - Expressions

Expressions are groups of identifiers and symbols, assembled according to specific syntactical rules, that provide for the manipulation of arithmetic quantities using common mathematical operations. USDL expressions appear in rules, both as rule test expressions and as general expressions in many of the available statement kinds. Expression forms in the USDL have been designed to be very similar to those in conventional programming languages (e.g., Ada and FORTRAN) to minimize system developer learning requirements.

3.8.2.2.1 USDL Semantics: Rulesets - Rules - Expressions - Literals

The USDL allows for the representation of different types of constant values. Such values are usually referred to as literals. There are three type of literals available: boolean constants, float constants, and integer constants. Here are some examples of these scalra literals:

```
boolean:   false   true
float:     0.0   1.0   389.334   1.0e+6   42.5e-11
integer:   0   1   412   12442
```

Note that constant negative literal values are disallowed. However, a literal integer or literal float, when preceeded by a minus sign, can be correctly processed as an expression with a unary minus (negation operator).

String literals are also allowed by the USDL. Although string literals are not permitted as parts of expressions, they are used in other contexts in certain statement kinds. A string literal is a sequence of zero or more nonquote chracters delimited by quotes. Here are some examples of string literals:

```
"x"
"hello there folks"
""
"a long string literal value is okay to use; no length limit"
```

3.8.2.2.2 USDL Semantics: Rulesets - Rules - Expressions - Variables

A variable in the USDL is an object that is associated with a value such that the value can be changed. Each variable is of exactly one of three scalar basetypes: boolean. float, and integer. Variables come in several classes:

Class	Declared in:
declare item	systems, rulesets, rules
block attribute	blocks, blocktypes
block line	blocks, blocktypes

A declare item is referenced by use of the declare item identifier.

A block attribute is referenced by use of an attribute indication - a three part construct made up of: the block identifer corresponding to the block that includes the attribute, a period, and the attribute identifier. Examples:

```
block_52.threshold_limit
battery_2.electrolyte_level
```

A block line can be referenced by use of a block line current value indication - the block identifier corresponding to the block that includes the line, a period, and the line identifier. This manner of line value indication always refers to the present value of a line ("current time minus zero"). Examples:

```
block_52.output_displacement
battery_2.output_voltage
```

A block line can also be referenced by use of a block line history value indication. This mode of line value indication is similar to the block line current value mode, except that a history buffer selection suffix is appended. A history suffix consists of a period, the reserved word "history", and a nonpositive integer expression enclosed in brackets. The value of the integer expression is the number of time periods in the past that corresponds to a stored history value. A bracketed expression that evaluates to zero indicates the current line value. A bracketed expression with a value of minus one indicates the immediately (chronological) preceding line value, and successively negative indices indicate earlier and earlier line values. Examples:

```
block_52.output_displacement.history[0]          -- redundant suffix
block_52.output_displacement.history[-2]         -- two ticks ago
battery_2.output_voltage.history[i + (j / 2)]    -- expression index
```

3.8.2.2.3 USDL Semantics: Rulesets - Rules - Expressions - Operators

The USDL supports a wide variety of expression operators that manipluate boolean, float, and integer objects (literals and variables). A USDL operator is either monadic (one operand, prefix format) or dyadic (two operands, infix format) and always returns a single value as a result of evaluation.

Table 3.3: USD Language Operators

Monadic Operators:

```
+   arithmetic affirmation
-   arithmetic negation
not boolean negation
```

Dyadic Operators:

```
+    arithmetic sum
-    arithmetic difference
*    arithmetic product
/    arithmetic quotient
**   arithmetic exponentiation
>    relational: greater than
>=   relational: greater than or equal
<    relational: less than
<=   relational: less than or equal
=    relational: equal
/=   relational: not equal
and  boolean product
or   boolean inclusive disjunction
xor  boolean exclusive disjunction
cand boolean product, conditional second operand evaluation
cor  boolean inclusive disjuction, conditional 2nd op evaluation
```

The order of evaluation follows the usual conventions, and for those operators that are also present in Ada, the precedence ranking is the same as in Ada. Parentheses may be used to group expressions for both readability and to overide default precedence.

The exact rules for expression formation are described in the BNF specification appendix of this document.

3.8.2.3 USDL Semantics: Rulesets - Rules - Statements

Statements in the USDL represent various kinds of actions that can be performed during system modeling. Each statement kind has its own syntax and semantics. Table 3.4 presents a list of available ruleset statements in the USD language. Each statement kind (except assignment statements) starts with a reserved word for that kind. A statement is terminated by a semicolon. Multiple statements may be grouped together for (nominally) sequential execution by using a block statement.

Table 3.4: USD Language Ruleset Statements

Statement	Function
Accept	Bring external data into system model
Advance	Advance history buffers by one step
Assignment	Evaluate expression and assign to a variable
Call	Transfer control to a descendent ruleset
Compound	Associate an arbitrary number of statements
Display	Export data outside the model
Elaborate	Transfer control to subsystem ruleset
Exit	Terminate user system description
If-then-else	Control conditional execution
Null	Perform no action
Pulse	Copy output line value to connected input lines
Read	Read a value from an external file
Reset	Reset all scalar values in system model
Return	Terminate ruleset execution and return to caller
Write	Write a value to an external file

3.8.2.3.1 USDL Semantics: Rulesets - Rules - Accept Statement

An "accept" statement is used for bringing data values into the system model from a source outside of the model. During the interactive execution of the RSP, this external source is the standard input (console) of the environment; when running in an embedded system, the accept statement is ignored.

An accept statement takes one of the two following forms:

```
accept <variable> ;
accept <prompt-string> <variable> ;
```

The action of an accept statement (when running interactively) is to pause USD interpretation and request a value of the user. In the first form, a prompt character ">" is printed on the console and the RSP waits for the user to type in a scalar value of a basetype appropriate for the <variable>. After a valid scalar literal is entered, the RSP assigns the value to the <variable>. In the second form, the <prompt-string> is printed on the console on its own line before the user is prompted for a value for the <variable>.

Here are some examples of the accept statement:

```
accept "Enter a value for voltage: " voltage_2;
accept "Retry count: " k_val;
accept connect_status;
```

3.8.2.3.2 USDL Semantics: Rulesets - Rules - Advance Statement

The "advance" statement provides control over the advancement of values along the history buffers associated with the lines of blocks. There is only one form of the advance statement:

```
advance ;
```

When executed, the advance statement causes all history buffers in the entire system being modeled to be advanced by one step. This action is intended for the writing of "simulate" rulesets to model time advancement. The advancement of a history buffer associated with a line causes each value to be moved towards the "past" by one time unit; a value at position P will be moved to position (P - 1). Values that are at the least recent position in the buffer are lost. The value at the most recent position of the buffer is cleared (becomes zero or false as appropriate).

3.8.2.3.3 USDL Semantics: Rulesets - Rules - Assignment Statement

An assignment statement is used to calculate a value from an expression and to assign the result to a variable. There is a single form:

```
<variable> := <expression> ;
```

When executed, the assignment statement evaluates the <expression> on the right side of the assignment symbol ":=" and stores the result in the storage indicated by by the <variable> on the left side of the assignment symbol. Note that allowable variables include declare item identifiers, line values, and block attribute values. Here are some examples of assignment statements:

```
x0 := 0.0;
delta_y := y1 - y0;
block_19.voltage_out := block_19.voltage_in * block_19.scaling;
```

3.8.2.3.4 USDL Semantics: Rulesets - Rules - Call Statement

The "call" statement is used to transfer control to a descendent ruleset in the same system. When the descendent ruleset terminates, control is returned to the point following the call. Execution of the call statement does not change the current system level being modeled (compare with the "elaborate" statement). There is a single form of the call statement:

```
call <ruleset-id> ;
```

Here are some examples:

```
call load_inputs;
call calculate_results;
call store_outputs;
```

3.8.2.3.5 USDL Semantics: Rulesets - Rules - Compound Statement

The compound (or block) statement is used to associate an arbitrary number of statements for (nominally) sequential execution. A compound statement has the form:

```
begin
  <statement> ...
end ;
```

Here are some examples:

```
begin
  accept "Enter scale: " scale;
  s := (y1 - y0) / (x1 - x0) * scale;
  call put_slope;
end;
begin
  call rs1;
  begin
    call rsx_a;
    call rsx_b;
  end;
end;
```

3.8.2.3.6 USDL Semantics: Rulesets - Rules - Display Statement

The "display" statement is used for bringing data values from the system model to a destination outside of the model. During the interactive execution of the RSP, this external source is the standard output (console) of the environment; when running in an embedded system, the display statement is ignored.

A display statement takes one of the three following forms:

```
display <label-string> ;
display <expression> ;
display <label-string> <expression> ;
```

The action of the display statement, when running interactively, is to print the <label-string> (when present), followed by the result of evaluating the <expression> (when present). Here are some examples of the display statement:

```
display "Now entering phase 3.";
display block_14.line_3.history[-5];
display "Subsystem W fault indication: " flag_a or flag_b;
```

3.8.2.3.7 USDL Semantics: Rulesets - Rules - Elaborate Statement

The "elaborate" statement provides the only means by which the subsystem representation of blocks, when present, can be expanded for modeling. There is a single form of the elaborate statement:

```
elaborate <block-id> using <ruleset-id> ;
```

The action of an elaborate statement is to transfer control to the ruleset designated by <ruleset-id> in the system associated by the subsystem resource in the block designated by <block-id>. Of course, to work properly, the <block-id> block must have a subsystem representation and that subsystem must include a directly enclosed ruleset with the name of <ruleset-id>. When the indicated ruleset concludes execution, control is returned to the statement following the elaborate statement. In this manner, the elaborate statement is similar to a call statement. However, the elaborate statement

also changes (for the duration of its indicated ruleset execution) the system associated with the "current system scope".

Here are some examples of the elaborate statement:

```
elaborate block_49 using diagnose;
elaborate part_4_b using simulate;
elaborate converter_24 using dump_variables;
```

The action of elaborate statement execution also provides for moving data between the enclosing system (the one containing the elaborate statement) and the enclosed system (the one used as the subsystem representation by the block being elaborated). When an elaborate statement executes, all of the scalar values associated with the current values of the input lines of the elaborated block are copied into the associated external items in the enclosed system, and then the indicated ruleset is called. When the indicated ruleset concludes execution, all of the scalar values associated with the current values of the output lines of the elaborated block are copied from the associated external items in the enclosed system.

This one-to-one corresondance of lines in the elaborated block and external items in the subsystem is the basic formal/actual parameter mechanism for data transmission betreen system levels.

3.8.2.3.8 USDL Semantics: Rulesets - Rules - Exit Statement

The "exit" statement terminates the modeling of the user system description and control exits to the enclosing environment. When run interactively, such termination is indicated to the user via the console. The modeling is also terminated when the first ruleset elaborated terminates exception. There is only one form of the exit statement:

```
exit ;
```

An exit statement may appear in any ruleset and still have the same action regradless of placement. The intended use of the exit statement is to provide a means of model termination when exceptional conditions occur.

3.8.2.3.9 USDL Semantics: Rulesets - Rules - If Statement

The "if" statement in the USDL is like a rule within a rule and is used for controlling conditional execution. An if statement has two forms:

```
if <expression>
  then <affirmative-statement>
end if;
if <expression>
  then <affirmative-statement>
  else <alternative-statement>
end if;
```

When an if statement is executed, the <expression> is evaluated first. If the result of the evaluation is nonzero (or true), then the <affirmative-statement> is executed. If the result is zero (or false) and the

<alternative-statement> is present (second form), then the <alternative-statement> is executed. Compound statements can be used to group multiple statements in either the <affirmative-statement> or the <alternative-statement>.

Here are some examples of the if statement:

```
if flag
  then call rs_4;
end if;
if (delta_s / delta_t) = 0
  then display "Speed zero" ;
  else
    begin
      speed := (delta_s / delta_t) * fudge_factor - adjustment;
      display "Speed: " speed;
    end;
end if;
```

3.8.2.3.10 USDL Semantics: Rulesets - Rules - Null Statement

The "null" statement performs no action. It is intended to be used as a placeholder for system models under development to indicate as yet unwritten executable code. It has a single form:

```
null ;
```

3.8.2.3.11 USDL Semantics: Rulesets - Rules - Pulse Statement

The pulse statement is used to help automate the simulation of a system model by copying all the current values of the output lines of a designated block to the appropriate input lines of connected blocks. The pulse statement has a single form:

```
pulse <block-id> ;
```

When the pulse statement is executed, the value of each output line of the indicated block is copied into the corresponding inputs of connected blocks. Note that because more than one path may be connected to an output line, more than one copy of the output value is made. Here are some examples:

```
pulse input_pads;
pulse block_18;
```

3.8.2.3.12 USDL Semantics: Rulesets - Rules - Read Statement

The "read" statement is used to read a value from an external file to the system model. All read statements read from the same file (named "dfr"), and each read statement reads a single value in text format reading a single text line in the input file. There are two forms of the read statement:

```
read ;
read <variable> ;
```

The first form actually transfers no value but is used to read in a single text line (which is ignored). The intent here is to provide a means of skipping over commentary lines in the input file. The second form also reads in a single text line and causes the value of the scalar literal on that line to be assigned to the indicated <variable>.

Here are some examples:

```
read;
read delta_t;
read block_4196.inversion_attribute;
```

3.8.2.3.13 USDL Semantics: Rulesets - Rules - Reset Statement

The "reset" statement provides a way to reset all of the scalar values in the system model. There is a single form:

```
reset ;
```

When a reset statement is executed, all model values are reset: all declare items, block line values, and block line history buffer values are all cleared. All block attribute variable values are cleared, unless a default clause is present; if so, the attribute variable value is reset to the default value. A boolean variable is cleared by setting it to the value false; integer and float variables are cleared by setting them to zero.

3.8.2.3.14 USDL Semantics: Rulesets - Rules - Return Statement

The "return" statement causes a ruleset to terminate execution and return to its caller. If the ruleset was invoked using a call statement, execution is returned to the point following the call statement in the calling ruleset, and the current level of system modeling is unchanged - the system level remains the same. If the ruleset was invoked using an elaborate statement, execution is returned to the point following the elaborate statement in the elaborating ruleset in the enclosing system, and the current level of system access is moved one step closer to the top level; a return to an elaboration while already at the top level (implied by a ruleset elaborated as a direct command to the RSP), terminates the system model.

There is a single form for a return statement:

```
return ;
```

Rulesets also have implicit returns present. Each ruleset, upon execution of its last available statement, will return to its invoker. Explicit return statements are provided so that a ruleset may return early, and so not execute all of its rules, in order to same processing resources when appropriate.

3.8.2.3.15 USDL Semantics: Rulesets - Rules - Write Statement

The "write" statement is used to write a value from the system model to an external file. All write statements write to the same file (named "dfw"),

and each write statement writes a single value in text format on a single text line in the output file. There are two forms of the write statement:

```
write ;
write <expression> ;
```

The first form actually transfers no value but is used to write out a single empty text line. The intent here is to provide a means of inserting blank lines in the output file to enhance readability. The second form also writes out a single text line and causes the value of the indicated <expression> to be written out on the single output text line.

Here are some examples:

```
write;
write mass * velocity;
write block_4196.inversion_attribute;
```

4. RULE SET PROCESSOR USAGE

One goal of the RSP I (Rule Set Processor Prototype) effort has been the relative ease of use of the program. The motivation here is to enhance productivity by making the software tool solution simpler than the system model problem so that the developer may willingly spend more time on the problem than on the solution. Since an over-elaborate tool solution may require extensive training for a tool such as the RSP to be useful, it should be simple enough to learn to use.

The RSP prototype has been implemented in the style of a conventional programming language interpreter/compiler. Operation of the RSP itself is therefore quite simple (and is similar to operation of commonly available language processors). The result of this decision is to emphasize the role of the USDL (User System Description Language) itself along with the developer's ability to express a system model in terms of the USDL. Because the USDL was carefully designed to represent conventionally specified systems (blocks + interconnections + subsystem abstraction), a developer has a relatively simple task of transferring the topological aspects of a system model into the USDL. Although the procedural information portion of a system model actually does require some programming using rulesets (structures analogous to procedures), this programming is not too much different from that of frequently used block structured programming languages (e.g., C, Pascal, Ada).

In addition to ease of use, another important reason for implementing the RSP in the style of a conventional language processor is that the USDL formal language specification, required for system description, also enforces a useful formalism upon the expression of system models. As system models require a formal description that is realized external to the RSP software, the future modeling effort is not necessarily tied to the fate of a particular RSP implementation. For example, it would be possible to develop other software modeling tools to work in conjunction with the RSP (e.g., graphical interfaces, alternative debugging tools) without having to re-specify the system model itself. Because the USDL supports (and encourages) usage of component libraries (resuable system models), there is a potentially high return in investing time in using a formal language description of dynamic systems.

4.1 System Model Development Cycle

The first step in the system model development cycle is to study the system to be described. It is not important at this stage of the cycle to fully specify the entire system, even if such knowledge is available as would be the case in an already existing system. What is important is to organize a complex system description in terms of a nested hierarchy where much of the lower level details are (temporarily) hidden by a high level description. An appropriate high level description may, for instance, consist of less than a dozen modules along with their interconnections. Those blocks at one level in the hierarchical arrangement can later be represented as entire subsystems at the immediately lower level; these subsystem descriptions may be supplied at a later time. The goal here is to use the USDL's power of nested representation to hide low level details so as to avoid having such details overwhelm the entire modeling effort.

The next step in modeling a system is to examine any available component libraries for resuable subsystem descriptions. The USDL allows any USDL system model to be used as a component in a larger and more complex model. For example, an aerospace engineer may have a library composed of limiters, multipliers, filters, notch filters, actuators and sensors; a mechanical engineer may use a component library stocked with various models of controls, motors, linkages, and power supplies; a structures engineer may have a custom library built up from previous work filled with system models of struts, beams, and strain gauges. A proper set of component libraries may go far in relieving the system developer of repetitive and error prone work.

The third step of the system modeling task is to write (using the USDL) a first attempt at a USD (User System Description). This first try should use only the highest layer of the modeled system along with any usable library subsystems. After the USD is written, it can be run through the RSP prototype to detect and report various errors even though not enough information may be present for simulation or diagnostic activities.

Once this high level topographical description is proved syntactically correct, the fourth step of incorporating procedural information in the USD. As stated elsewhere in this document, there are three designated rulesets (procedures) available for direct interpretation at the highest system level: "simulate" (intended for simulation), "preset" (intended for reading in a system state from a data file), and "diagnose" (intended for diagnostic activities). At this stage in the model cycle, it would be prudent to first write the procedural information required for system state presetting and simulation and insure its proper functioning before implementing diagnostic knowledge.

The fifth step is to try interpreting the system model simulation using the RSP, and continuing refinements in the model based upon observations of its behavior. As more confidence in the correctness of the model is gained, the fidelity of the model can be improved with the expansion/substitution of various components throughout the model with lower level subsystem representations. Ultimately, every component in the model's topological knowledge is either atomic (undivisible) or represented by a subsystem; additionally, the entire system has simulation code present and tested.

Once a system model is established with a complete topography and complete simulation procedural knowledge, the sixth stage of model development is to provide the system model with diagnostic procedural knowledge. For each system and component in the model, diagnostic ruleset code in written to perform tests upon functions for that part of the model. In order to test the diagnostic code, the developer can (purposely) introduce faults in the system simulation. Such introduction can be performed by various techniques: reading in a faulty system state via a preset operation, writing deliberate (and temporary!) faults in the system topography or simulation information, or by providing for the interactive prompting for critical information during simulation. (An obvious extension to the RSP is to have it, upon command, introduce errors automatically throughout the simulation. This error injection would be either random or uniform dependent upon user command, and the results of the diagnostic information performance would be tabulated automatically.)

Now armed with a well-tested system model, the developer should now review the model for any potentially reusable components, and to take such components and add them to the system library for future application.

The final step in the USD development cycle (not yet supported by the RSP prototype), is to use the RSP to translate the USD into a conventional programming language in a manner suitable for porting the model to an embedded computer environment. Under this stage of the development, the standalone interpreted/compiled version of the RSP would be tested, first, using a real-time simulation generated sensor data, and fourth, using the embedded program in a flight test.

4.2 System Model Interpretation

The RSP prototype user interface includes three commands used to activate system model interpretation. Each of these commands, "simulate", "preset", and "diagnose", initiate interpretation of the ruleset of the same name in the outermost level of the system model. Other than starting the indicated ruleset interpretation, the RSP treats each of the above user commands in the same fashion and so it not "aware" as to the particular function of the ruleset. The three ruleset names chosen were picked to establish a programming convention and have no other significance.

During the interpretation of these rulesets, interaction with the user is allowed, but is not necessary. The USDL has statements that perform input and output of data; to and from both the console and external files. These statements can be incorporated into any ruleset throughout the system model according to the desire of the developer.

The intented purpose of the "simulate" ruleset is to perform a one step simulation of the entire system. The interpretation of the "simulate" ruleset should: take the values of system's inputs, simulate each component, propagate results in the direction from internal input connections to internal output connections, and finally assign values to the system model external outputs. In multilevel system models, each subsystem should have its own "simulate" ruleset. When the system model is translated and ported to an embedded environment, the "simulate" ruleset code is no longer needed since the physical environment now supplies actual values.

The intended purpose of the "preset" ruleset is provide an alternative means of initializing the scalar values of a system model. The idea is to use the "preset" ruleset to read values from an external file instead of producing them via simulation. Use of the "preset" method can then help test out the diagnostic ruleset code by providing consistent values for development purposes. In multilevel system models, each subsystem that requires such initialization should have its own "preset" ruleset. The "preset" ruleset, like the "simulate" ruleset, would be removed upon porting of the system model to an embedded environment.

The intended purpose of the "diagnose" ruleset is to provide "intelligent" diagnosis. For a multilevel system model, a "diagnose" ruleset is required for each subsystem within the model for which diagnosis is to be performed. Unlike the "simulate" and "preset" rulesets, the "diagnose" ruleset should be preserved upon porting to the application environment so as to provide real time diagnostic ability.

4.3 USD Simulation Strategy

The basic technique employed for designing simulation rulesets is the input-to-output, bottom-up approach. Each system in the system model should have a "simulate" ruleset, and this ruleset should only be concerned with activities for that system and no other system. Fortunately, once a simulation ruleset is written for a given system (and then incorporated into that system), that system can then be used repeatedly as a subsystem in more complex systems.

Here is the basic algorithm for system simulation (using a psuedocode listing):

```
procedure simulate

 /* This routine is called first for the root system and later for
    each subsystem in a depth first manner.  This depth-first
    search insures that the simulation of the components is
    performed in the correct order so that all information
    generated at the lower levels is made available to the higher
    levels of the simulation. */

 begin

  /* Handle system inputs */

  for each external carrying a value into the system do
   begin

    propagate input value
     from external connection
     to input line of of the appropriate component;

   end;


  /* Handle components (blocks) */

  while unsimulated blocks remain do
   begin

    for each unsimulated block do
     begin

      /* process only blocks with complete inputs */

      if the block has valid data for all of inputs then
       begin

        /* check for recursion */

        if the block has a subsystem representation then

         /* recurse and process lower level */

         elaborate the block using the simulate ruleset;
```

```
            else

             /* no further recursion */

             hand-simulate the block;

            end if;   /* subsystem exists test */

            /* propagate outputs */

            for each output line of the block
             propagate the line value to the next block;

            /* done with this block */

            mark the block as simulated;

           end;   /* block simulation */
          end if;   /* all inputs valid */

        end;   /* for loop */
      end;   /* while loop */


     /* Handle system outputs */

     for each external carrying a value out of the system do
      begin

       propagate output value
        from output line of of the appropriate component
        to external connection;

      end;

    end;   /* simulate */
```

4.4 USD Diagnosis Strategy

As to supplying of such intelligence, the major responsibility resides with the developer. The USDL and its RSP interpreter provide a considerable level of support for its system model implementation (topological/procedure knowledge fusion, built-in block structured programming language, forward chaining rules, etc.), but it is up to the user to employ these tools properly.

RSP I leaves the writing of the specific diagnostic ruleset code to the developer. That is, RSP I provides the environment so that the application domain expert can select the appropriate procedural rules from the wealth of algorithms developed for BIT and fault diagnosis in dynamic systems (Pau 1981), (Mozgalevskii 1978), (Willsky 1980), and (Basserville 1981). However, as is the case with simulation ruleset code, diagnostic code for a given system only has to be written once and can then be duplicated and reused for other system models. Also, both diagnostic and simulation procedural information for subsystems can be written by specialists and then later used

by generalists without requiring the generalists to be fully familiar with the lower level details.

Diagnosis of a system begins with the "diagnose" command at the user interactive interface. Diagnostic interpretation should be performed only when the system model has a full set of valid values as will be the case after a complete simulation or preset. The complete interpretation of the "diagnose" ruleset is assumed to take place between successive time sample instants using a "frozen" set of values that do not vary during the diagnosis period.

A useful concept in writing diagnostic code is idea of the "consistency relation" check. A consistency relation is some set of arithmetic operations performed on the inputs and outputs of a block that results in a determination of fault for that block (e.g., a wrap-around BIT, a parity check or a statistical hypothesis test). If a consistency relation fails for a block, it can be assumed that there is a failure of that block (or its subsystem representation, if any). If a consistency relation succeeds for a block, it can be assumed that the subsystem representation for that block (if any) is functioning correctly. In the case where it is not feasible to construct such a consistency check at a high level it may be necessary to instead write multiple consistency checks at a lower level and then combine the results of these checks for a higher level decision.

The goal of achieving high speed diagnostic capability can be met by designing consistency relation checks for the higher level components; when these checks are passed, it allieviates time consuming examination of checks at lower levels.

It may not be possible to write concise fault/no-fault consistency check ruleset code for each component type in a system model. For many real world components, usual consistency check methods produce only probabilistic results, and it is the responsibility of the developer to combine these results. The USDL supports a full set of operations upon probability values (using floating point variables) thus allowing the user to perform customized conditional analysis.

For certain time critical applications, circumstances may occur so that it may not be possible to run all of the desired diagnostic code in the limited time available. For these applications, a family of consistency checks may be written such that the quicker running (more general) checks are used first and slower running (more specific) checks are used should time remain available. This graded strategy helps ensure that at least some diagnostic results are generated even if the interval allowed for diagnosis is insufficient for the circumstances for a particular cycle. The reasoning here is that it is better to derive a general, partially useful result instead of no result at all.

The exact details of a diagnostic ruleset will vary among differing systems. However, for system models with multiple levels, a top-down selective approach would be appropriate for fixed time interval diagnosis. This approach, unlike the bottom-up full evaluation approach used for simulation, will spend time working on only those subsystems where problems are suspected.

Here is a suggested algorithm for system diagnosis (using a psuedocode listing):

```
procedure diagnose

 /* This procedure is called first at the root level of the system
    model, and may be called recursively at lower system levels as
    required.  The goal is to expend effort at the level of
    invocation first and to only elaborate subsystems for
    diagnosis when consistency relation checks at the level of
    invocation fail and further analysis is indicated. */

 /* The diagnosis here is a very simple one with an interest
    in only a certain fail/no-fail status.  Most real world
    applications would use the standard rules for combination of
    certainty factors to produce a more useful result. */

begin

 /* If called at the root system level, clear the failure site
    global variable.  This variable holds the name of the first
    component (if any) that fails a consistency check. */

 if (called at root level) then
  failure_site := nowhere;
 end if;

 /* Should diagnosis time exceed the allocated interval, we can
    assume an interrupt will occur.  The value of this
    variable will indicate whether or not enough time was
    available for a complete diagnosis. */

 if (called at root level) then
  diagnosis_completed := false;
 end if;

 /* Perform diagnosis at this level.  Elaborate subsystems only
    when necessary. */

 while (failure_site = nowhere) and (undiagnosed blocks remain) do
  begin

   /* Select the block among the undiagnosed blocks with the
      highest diagnostic figure-of-merit.  This figure is given
      by the quotient of the probability of failure divided by
      the expected amount of time required for consistency
      relation checking for that block.  This approach will
      minimize the overall diagnosis time; it will not affect
      the accuracy of the diagnosis if enough time is available
      for full diagnostics. */

   current_block := highest_merit(undiagnosed blocks);

   /* Mark component as diagnosed */

   set_diagnosed(current_block);
```

```
        /* Perform the consistency check on the current block. */

        fault_detected := consistency_check_block(current_block);

        /* Process according to detection status. */

        if (fault_detected) then

         /* Inconsistency - check for subsystem representation.  */

          if (subsystem_exists(current_block)) then

            /* Recursively activate a lower level diagnostic ruleset.
               This recursive scan should eventually detect a fault
               at a lower level; the location will be reported back
               in the global variable "failure_site". */

            elaborate subsystem of the current_block with diagnose;

            /* Check to see if detection was false alarm. */

            if (failure_site = nowhere) then
              report("Warning: consistency check fault");
              report("At location: ", current_block);
            end if;

          else

            /* Fault detected of a simple block. */

            failure_site := current_block;

          end if;   /* subsystem exists test */
        end if;   /* consistency check failure test */

      end;   /* block scan */

     /* Diagnosis completed, adjust global completion indicator. */

     if (called at root level) then
      diagnosis_completed := true;
     end if;

    end;   /* diagnose */
```

4.5 RSP Example System

```
system binary_adder_system is (see Figures 4.1 and 4.2)
 --
 -- This system is used to model a full binary adder.  A binary adder is a
 -- computational element that takes two input bits along with a carry-in
 -- bit and produces a single bit sum and a single bit excess (carry-out).
 --
 -- This binary adder system directly encloses four modules to perfores its
 -- function.  The value module indicates input values (using lights), the
```

```
-- sum generation module produces the single bit sum, the excess generation
-- module produces the single bit excess (carry out), and the result system
-- indicates (also using lights) the results of the addition.
--
-- Here is the truth table that describes this system:
--
--      adder_a adder_b adder_c ------> adder_s adder_x
--     ------- ------- -------          ------- -------
--      false   false   false            false   false
--      true    false   false            true    false
--      false   true    false            true    false
--      true    true    false            false   true
--      false   false   true             true    false
--      true    false   true             false   true
--      false   true    true             false   true
--      true    true    true             true    true
--
begin

 -- The following global varaible, "failure", is used to indicate a
 -- detected failure at any level.  This variable is initially cleared
 -- by the root level diagnose ruleset and is set only if a problem is
 -- detected.

 declare failure: boolean;

 system and_system is (see Figure 4.3)
  --
  -- This system is used to represent an AND gate.  An AND gate takes two
  -- binary inputs and produces a single binary output that represents the
  -- logical product of the inputs.
  --
  -- This system is independent of all other systems.
  --
  -- Here is the truth table for this system:
  --
  --    a_in1  a_in2  ------>  a_out
  --    -----  -----          -----
  --    false  false          false
  --    true   false          false
  --    false  true           false
  --    true   true           true
  --
  begin

   block anchor is general
    begin
     line op1 is mode input basetype boolean;
     line op2 is mode input basetype boolean;
     line result is mode output basetype boolean;
    end anchor;

   external a_in1 is anchor.op1;
   external a_in2 is anchor.op2;
   external a_out is anchor.result;
```

```
  ruleset simulate is
   begin
    rule s_1 is
     begin
      if true then
       anchor.result := anchor.op1 and anchor.op2;
      end if;
     end s_1;
   end simulate;

  ruleset diagnose is
   begin
    rule d_1 is
     begin
      if (anchor.result /= anchor.op1 and anchor.op2) then
       begin
        display "failure detected: and_system";
        failure := true;
        return;
       end;
      end if;
     end d_1;
   end diagnose;

 end and_system;

 blocktype and_module_type is general
  begin
   line a_in1 is mode input basetype boolean;
   line a_in2 is mode input basetype boolean;
   line a_out is mode output basetype boolean;
   subsystem and_system;
  end and_module_type;

system fork_system is (see Figure 4.4)
 --
 -- This system is used to represent a forking connection.  Each of the
 -- two boolean outputs is set to the value of the single boolean input.
 --
 -- This system is independent of all other systems.
 --
 -- Here is the truth table for this system:
 --
 --    f_in   ------>   f1     f2
 --    -----           -----  -----
 --    false           false  false
 --    true            true   true
 --
 begin

  block anchor is general
   begin
    line operand is mode input basetype boolean;
    line result1 is mode output basetype boolean;
    line result2 is mode output basetype boolean;
   end anchor;
```

```
   external f_in is anchor.operand;
   external f1 is anchor.result1;
   external f2 is anchor.result2;

   ruleset simulate is
    begin
     rule s_1 is
      begin
       if true then
        begin
         anchor.result1 := anchor.operand;
         anchor.result2 := anchor.operand;
        end;
       end if;
      end s_1;
    end simulate;

   ruleset diagnose is
    begin

     rule d_1 is
      begin
       if (anchor.result1 /= anchor.operand) then
        begin
         display "failure detected: fork_system (result1)";
         failure := true;
         return;
        end;
       end if;
      end d_1;

     rule d_2 is
      begin
       if (anchor.result2 /= anchor.operand) then
        begin
         display "failure detected: fork_system (result2)";
         failure := true;
         return;
        end;
       end if;
      end d_2;

    end diagnose;

 end fork_system;

blocktype fork_module_type is general
 begin
  line f_in is mode input basetype boolean;
  line f1 is mode output basetype boolean;
  line f2 is mode output basetype boolean;
  subsystem fork_system;
 end fork_module_type;

system indicator_system is (Figure 4.5)
 --
```

```
-- This system is used to represent an indicator lamp.  This lamp is lit
-- if and only if the single boolean input is true.  The single boolean
-- output is the same value as the input.
--
-- This system is independent of all other systems.
--
-- Here is the truth table for this system:
--
--    indicator_in  ------>  indicator_out
--    ------------           -------------
--        false                  false
--        true                   true
--
begin

 block anchor is general
  begin
   attribute light is basetype boolean;
   line operand is mode input basetype boolean;
   line result is mode output basetype boolean;
  end anchor;

 external indicator_in is anchor.operand;
 external indicator_out is anchor.result;

 ruleset simulate is
  begin
   rule s_1 is
    begin
     if true then
      begin
       anchor.result := anchor.operand;
       anchor.light := anchor.operand;
      end;
     end if;
    end s_1;
 end simulate;

ruleset diagnose is
 begin

  rule d_1 is
   begin
    if (anchor.result /= anchor.operand) then
     begin
      display "failure detected: indicator_system (result)";
      failure := true;
      return;
     end;
    end if;
   end d_1;

  rule d_2 is
   begin
    if (anchor.light /= anchor.operand) then
     begin
```

```
        display "failure detected: indicator_system (light)";
        failure := true;
        return;
       end;
      end if;
     end d_2;

   end diagnose;

 end indicator_system;

blocktype indicator_module_type is general
 begin
  line indicator_in is mode input basetype boolean;
  line indicator_out is mode output basetype boolean;
  subsystem indicator_system;
 end indicator_module_type;

system or_3_system is (see Figure 4.6)
 --
 -- This system is used to represent an OR gate with three inputs. An OR
 -- gate with three boolean inputs produces the logical sum of its inputs
 -- and sets the single boolean output to this value.
 --
 -- This system is independent of all other systems.
 --
 -- Here is the truth table for this system:
 --
 --    or_3_in1  or_3_in2  or_3_in3  ------>  or_3_out
 --    --------  --------  --------           --------
 --      false     false     false              false
 --      true      false     false              true
 --      false     true      false              true
 --      true      true      false              true
 --      false     false     true               true
 --      true      false     true               true
 --      false     true      true               true
 --      true      true      true               true
 --
 begin

  block anchor is general
   begin
    line op1 is mode input basetype boolean;
    line op2 is mode input basetype boolean;
    line op3 is mode input basetype boolean;
    line result is mode output basetype boolean;
   end anchor;

  external or_3_in1 is anchor.op1;
  external or_3_in2 is anchor.op2;
  external or_3_in3 is anchor.op3;
  external or_3_out is anchor.result;

  ruleset simulate is
   begin
```

```
    rule s_1 is
     begin
      if true then
       anchor.result := anchor.op1 or anchor.op2 or anchor.op3;
      end if;
     end s_1;
   end simulate;

  ruleset diagnose is
   begin
    rule d_1 is
     begin
      if (anchor.result /= anchor.op1 or anchor.op2 or anchor.op3) then
       begin
        display "failure detected: or_3_system";
        failure := true;
        return;
       end;
      end if;
     end d_1;
   end diagnose;

 end or_3_system;

blocktype or_3_module_type is general
 begin
  line or_3_in1 is mode input basetype boolean;
  line or_3_in2 is mode input basetype boolean;
  line or_3_in3 is mode input basetype boolean;
  line or_3_out is mode output basetype boolean;
  subsystem or_3_system;
 end or_3_module_type;

system xor_system is (see Figure 4.7)
 --
 -- This system is used to represent an XOR gate.  An XOR gate takes two
 -- binary inputs and produces a single binary output that represents the
 -- exclusive-or (either but no both) of the inputs.
 --
 -- This system is independent of all other systems.
 --
 -- Here is the truth table for this system:
 --
 --    xor_in1  xor_in2  ------>  xor_out
 --    -------  -------           -------
 --     false    false             false
 --     true     false             true
 --     false    true              true
 --     true     true              false
 --
 begin

  block anchor is general
   begin
    line op1 is mode input basetype boolean;
    line op2 is mode input basetype boolean;
```

```
    line result is mode output basetype boolean;
   end anchor;

  external xor_in1 is anchor.op1;
  external xor_in2 is anchor.op2;
  external xor_out is anchor.result;

  ruleset simulate is
   begin
    rule s_1 is
     begin
      if true then
       anchor.result := anchor.op1 xor anchor.op2;
      end if;
     end s_1;
   end simulate;

  ruleset diagnose is
   begin
    rule d_1 is
     begin
      if (anchor.result /= anchor.op1 xor anchor.op2) then
       begin
        display "failure detected: xor_system";
        failure := true;
        return;
       end;
      end if;
     end d_1;
   end diagnose;

 end xor_system;

blocktype xor_module_type is general
 begin
  line xor_in1 is mode input basetype boolean;
  line xor_in2 is mode input basetype boolean;
  line xor_in3 is mode input basetype boolean;
  line xor_out is mode output basetype boolean;
  subsystem xor_system;
 end xor_module_type;

system value_system is (see Figure 4.8)
 begin

  block indicator_module_a is type indicator_module_type;
  block indicator_module_b is type indicator_module_type;
  block indicator_module_c is type indicator_module_type;

  external val_in_a is indicator_module_a.indicator_in;
  external val_in_b is indicator_module_b.indicator_in;
  external val_in_c is indicator_module_c.indicator_in;
  external val_out_a is indicator_module_a.indicator_out;
  external val_out_b is indicator_module_b.indicator_out;
  external val_out_c is indicator_module_c.indicator_out;
```

```
 end value_system;

system result_system is (see Figure 4.9)
 begin

  block indicator_module_a is type indicator_module_type;
  block indicator_module_b is type indicator_module_type;
  block indicator_module_c is type indicator_module_type;

  external res_in_a is indicator_module_a.indicator_in;
  external res_in_b is indicator_module_b.indicator_in;
  external res_in_c is indicator_module_c.indicator_in;
  external res_out_a is indicator_module_a.indicator_out;
  external res_out_b is indicator_module_b.indicator_out;
  external res_out_c is indicator_module_c.indicator_out;

 end result_system;

system sum_generation_system is (see Figure 4.10)
 begin

  block xor_module_1 is type xor_module_type;
  block xor_module_2 is type xor_module_type;

  path from xor_module_1.xor_out to xor_module_2.xor_in1;

  external sg_in_a is xor_module_1.xor_in1;
  external sg_in_b is xor_module_1.xor_in2;
  external sg_in_c is xor_module_2.xor_in2;

 end sum_generation_system;

system excess_generation_system is (see Figure 4.11)
 begin

  block fork_module_1 is type fork_module_type;
  block fork_module_2 is type fork_module_type;
  block fork_module_3 is type fork_module_type;

  block and_module_1 is type and_module_type;
  block and_module_2 is type and_module_type;
  block and_module_3 is type and_module_type;

  block or_3_module is type or_3_module_type;

  path from fork_module_1.f1 to and_module_2.a_in1;
  path from fork_module_1.f2 to and_module_3.a_in1;

  path from fork_module_2.f1 to and_module_1.a_in1;
  path from fork_module_2.f2 to and_module_3.a_in2;

  path from fork_module_3.f1 to and_module_1.a_in2;
  path from fork_module_3.f2 to and_module_2.a_in2;

  external eg_in_a is fork_module_1.f_in;
  external eg_in_b is fork_module_2.f_in;
```

```
   external eg_in_c is fork_module_3.f_in;
   external eg_out is or_3_module.or_3_out;

  end excess_generation_system;

block value_module is general
 begin
  line val_in_a is mode input basetype boolean;
  line val_in_b is mode input basetype boolean;
  line val_in_c is mode input basetype boolean;
  line val_out_a is mode output basetype boolean;
  line val_out_b is mode output basetype boolean;
  line val_out_c is mode output basetype boolean;
  subsystem value_system;
 end value_module;

block result_module is general
 begin
  line res_in_s is mode input basetype boolean;
  line res_in_x is mode input basetype boolean;
  line res_out_s is mode output basetype boolean;
  line res_out_x is mode output basetype boolean;
  subsystem result_system;
 end result_module;

block sum_generation_module is general
 begin
  line sg_in_a is mode input basetype boolean;
  line sg_in_b is mode input basetype boolean;
  line sg_in_c is mode input basetype boolean;
  line sg_out is mode output basetype boolean;
  subsystem sum_generation_system;
 end sum_generation_module;

block excess_generation_module is general
 begin
  line eg_in_a is mode input basetype boolean;
  line eg_in_b is mode input basetype boolean;
  line eg_in_c is mode input basetype boolean;
  line eg_out is mode output basetype boolean;
  subsystem excess_generation_system;
 end excess_generation_module;

external adder_a is value_module.val_in_a;
external adder_b is value_module.val_in_b;
external adder_c is value_module.val_in_c;
external adder_s is result_module.res_out_s;
external adder_x is result_module.res_out_x;

--
-- The following ruleset handles simulation at the root level.
--
ruleset simulate is
 begin
  rule s_1 is
   begin
```

```
    if true then
     begin
      elaborate value_module using simulate;
      elaborate sum_generation_module using simulate;
      elaborate excess_generation_module using simulate;
      elaborate result_module using simulate;
     end;
    end if;
   end s_1;
 end simulate;

--
-- The following ruleset handles diagnosis at the root level.
--
ruleset diagnose is
 begin

  rule diagnose_setup is
   begin
    if true then
     failure := false;
    end if;
   end diagnose_setup;

  rule check_value_module is
   begin
    if (not failure) then
     begin
      elaborate value_module using diagnose;
      if (failure) then
       display "failure detected: adder (value_module)";
      end if;
     end;
    end if;
   end check_value_module;

  rule check_sum_generation_module is
   begin
    if (not failure) then
     begin
      elaborate sum_generation_module using diagnose;
      if (failure) then
       display "failure detected: adder (sum_generation_module)";
      end if;
     end;
    end if;
   end check_sum_generation_module;

  rule check_excess_generation_module is
   begin
    if (not failure) then
     begin
      elaborate excess_generation_module using diagnose;
      if (failure) then
       display "failure detected: adder (excess_generation_module)";
      end if;
```

```
        end;
       end if;
      end check_excess_generation_module;

     rule check_result_module is
      begin
       if (not failure) then
        begin
         elaborate result_module using diagnose;
         if (failure) then
          display "failure detected: adder (result_module)";
         end if;
        end;
       end if;
      end check_result_module;

   end diagnose;

end binary_adder_system;
```

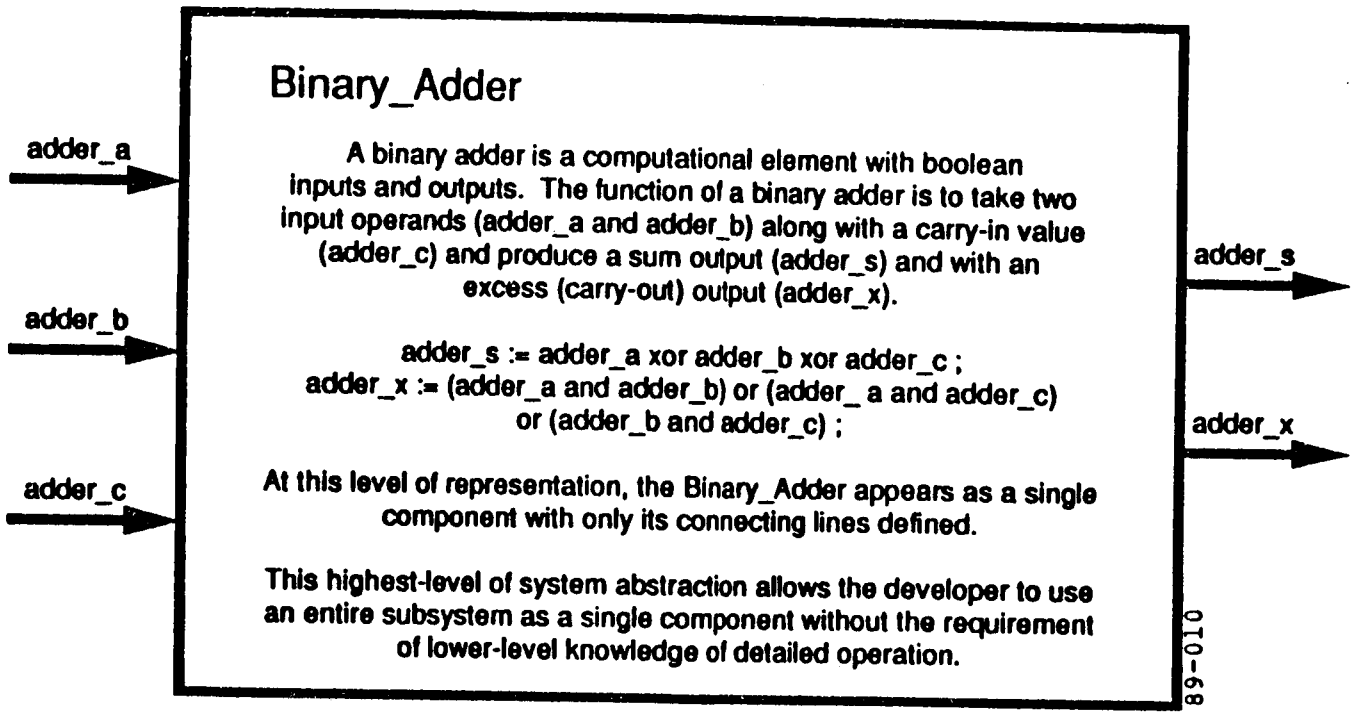


Figure 4.1

Example System Model: Binary Adder Highest Level Representation

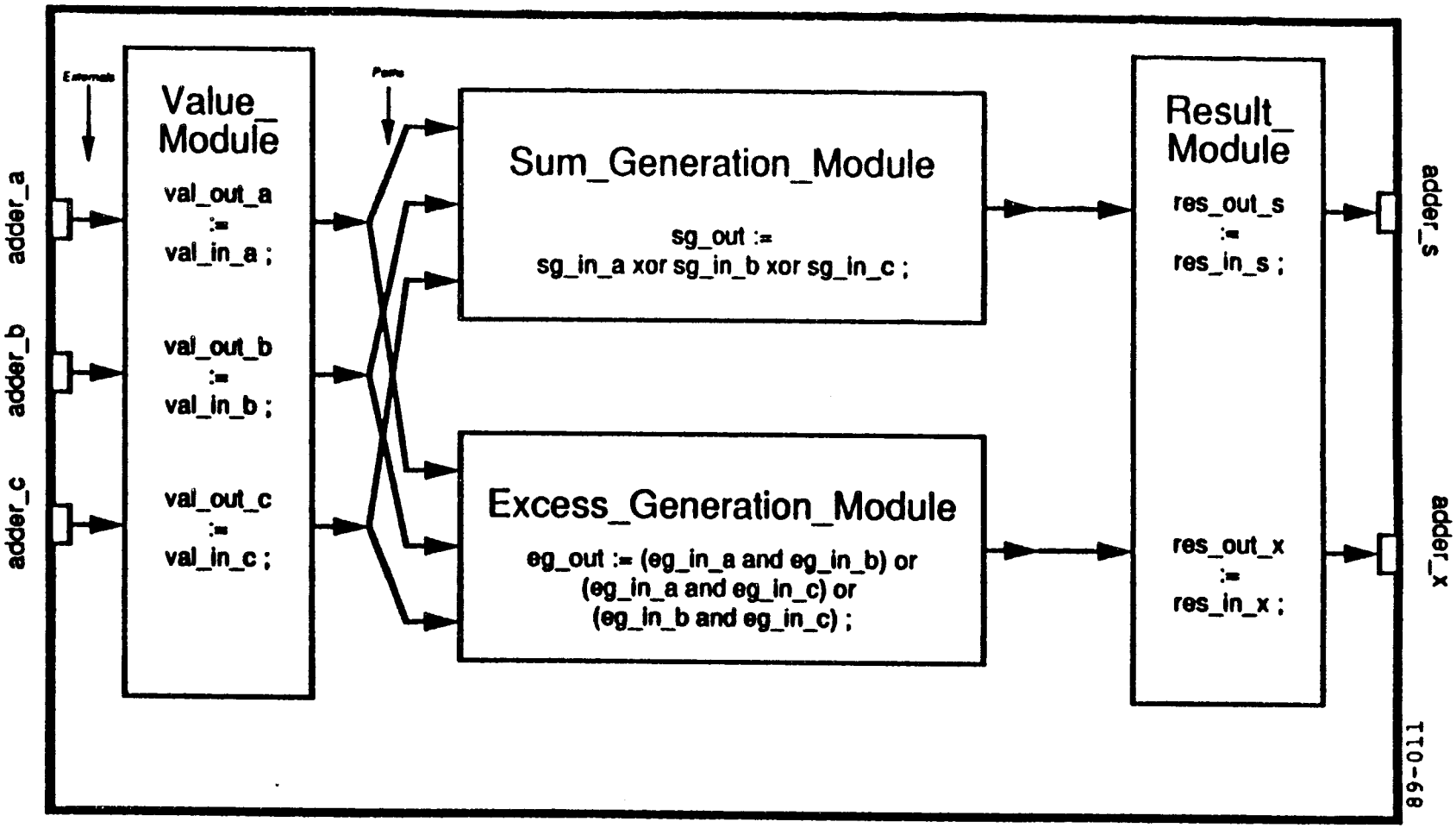


Figure 4.2

Example System Model: Binary Adder
Intermediate System Level Representation

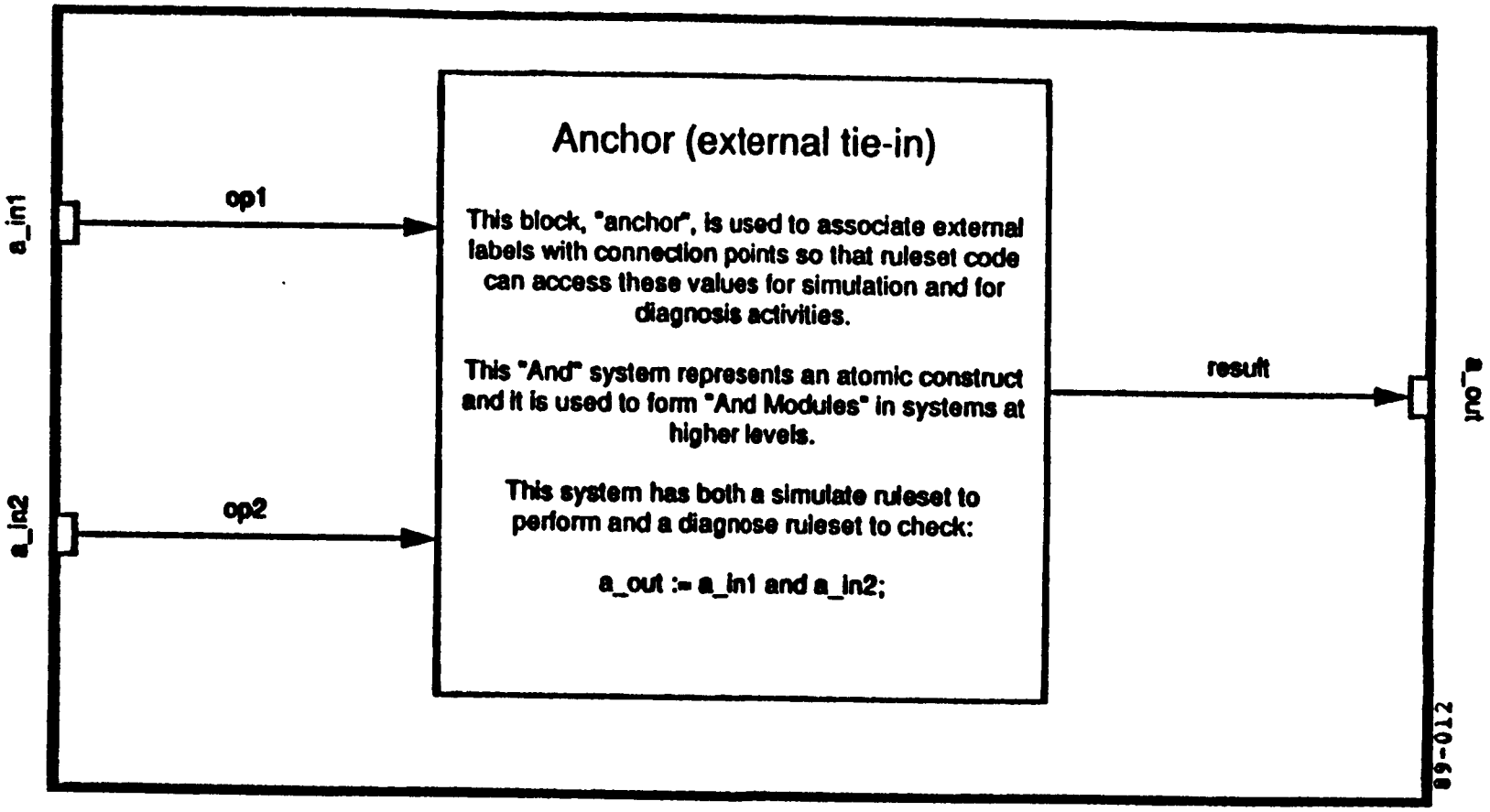


Figure 4.3

Example System Model: Binary Adder (And_System) Lowest System Level Representation

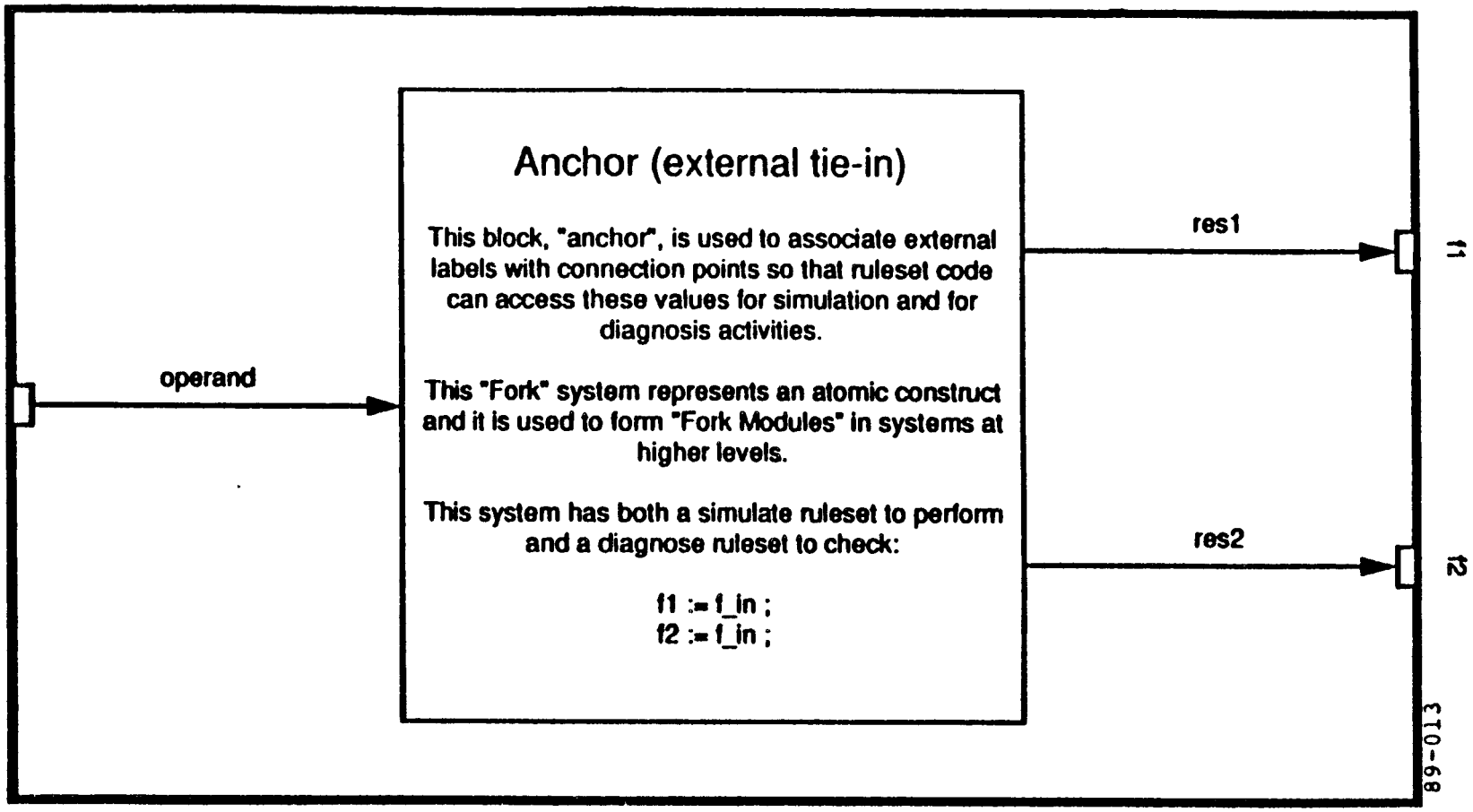


Figure 4.4

Example System Model: Binary Adder (Fork_System) Lowest System Level Representation

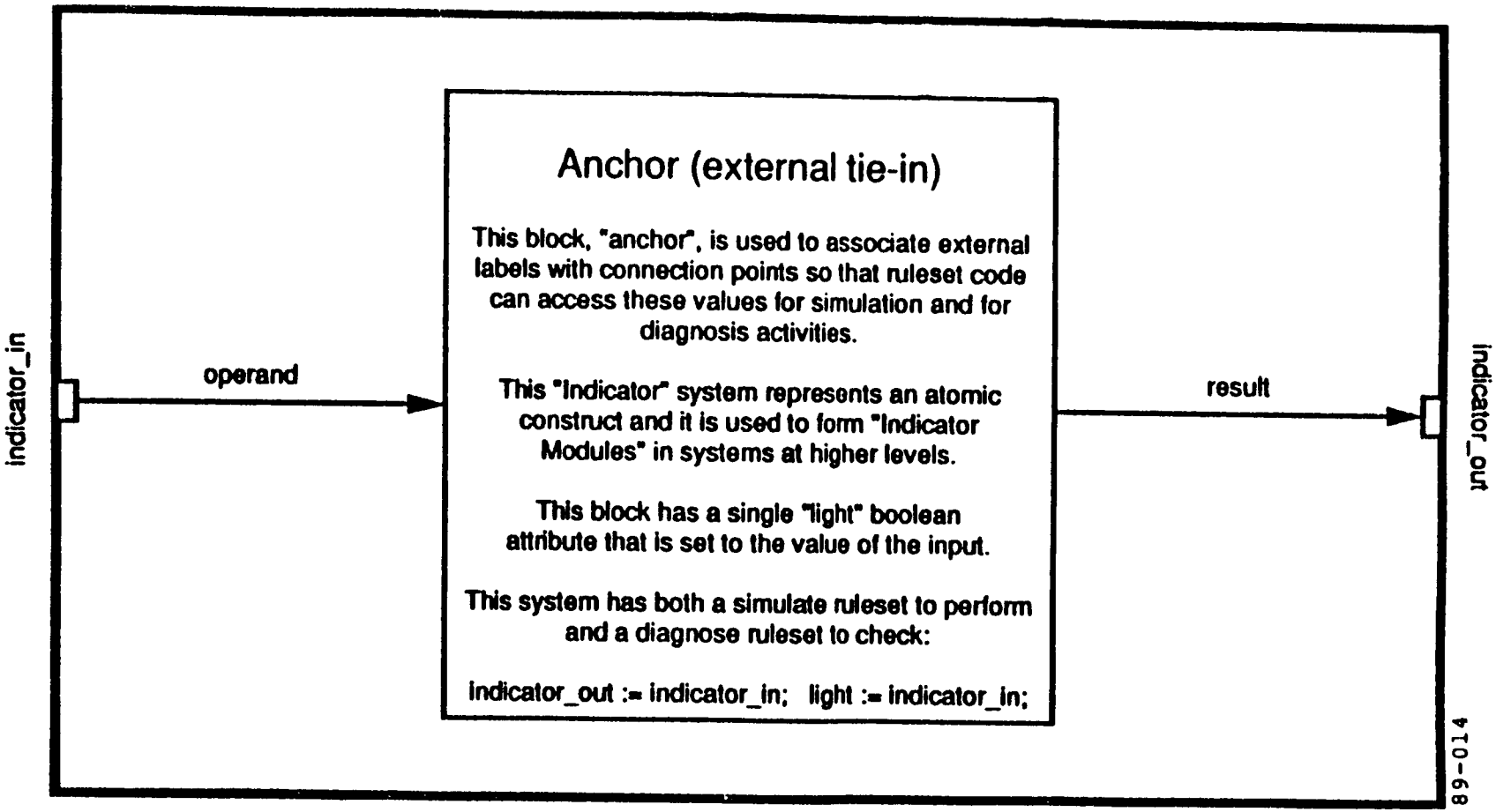


Figure 4.5

Example System Model: Binary Adder (Indicator_System) Lowest System Level Representation

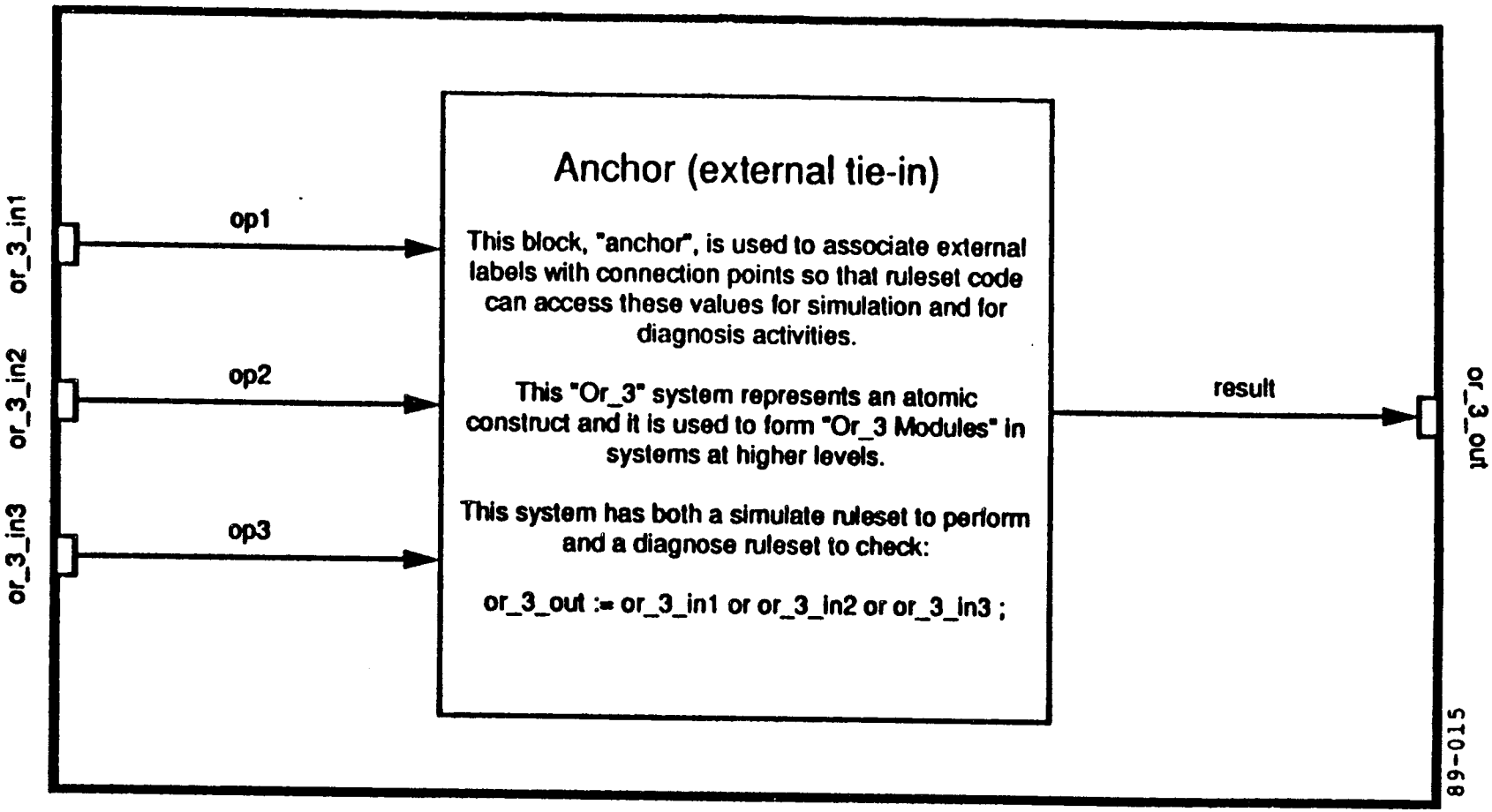


Figure 4.6

Example System Model: Binary Adder (Or_3_System) Lowest System Level Representation

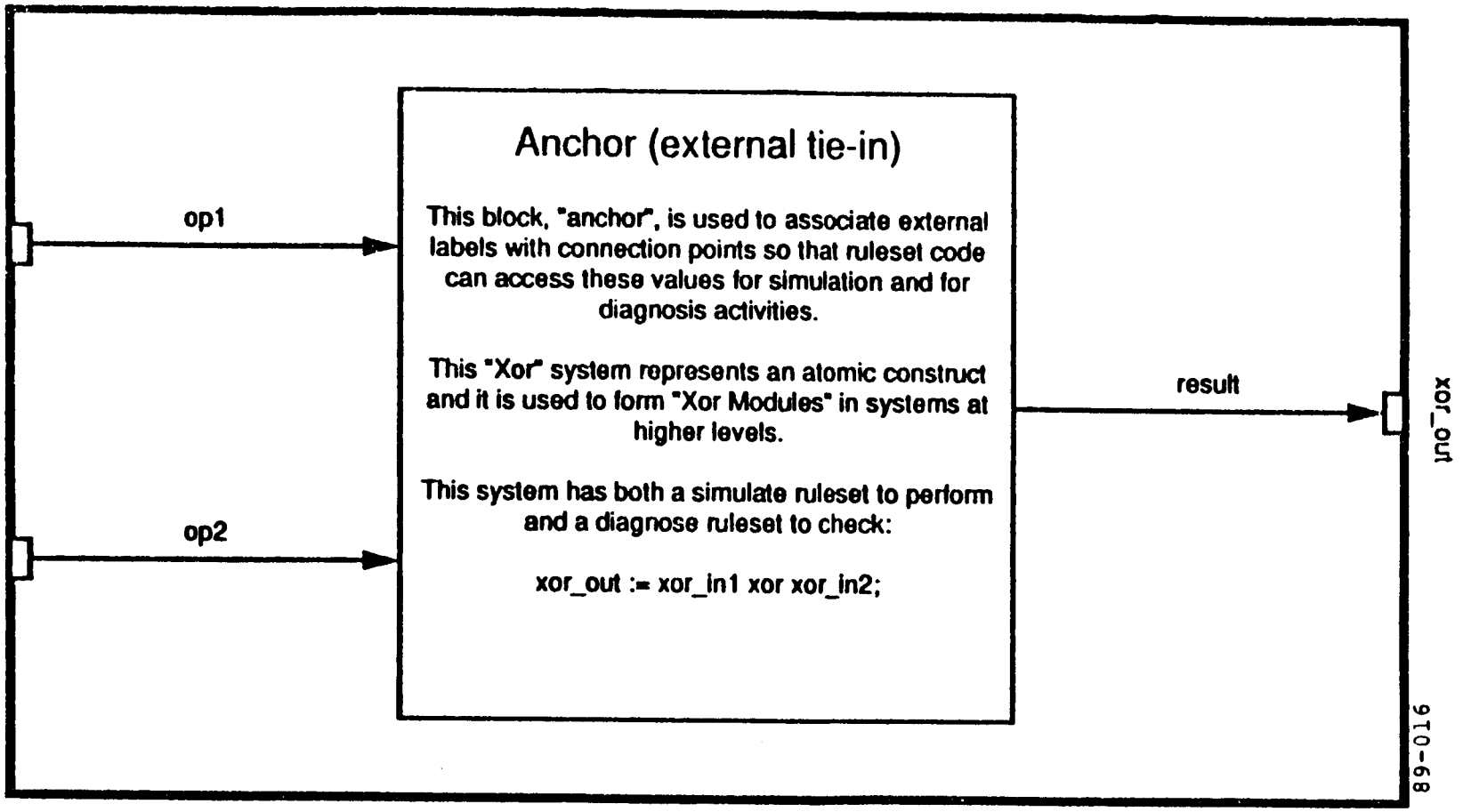


Figure 4.7

Example System Model: Binary Adder (Xor_System) Lowest System Level Representation

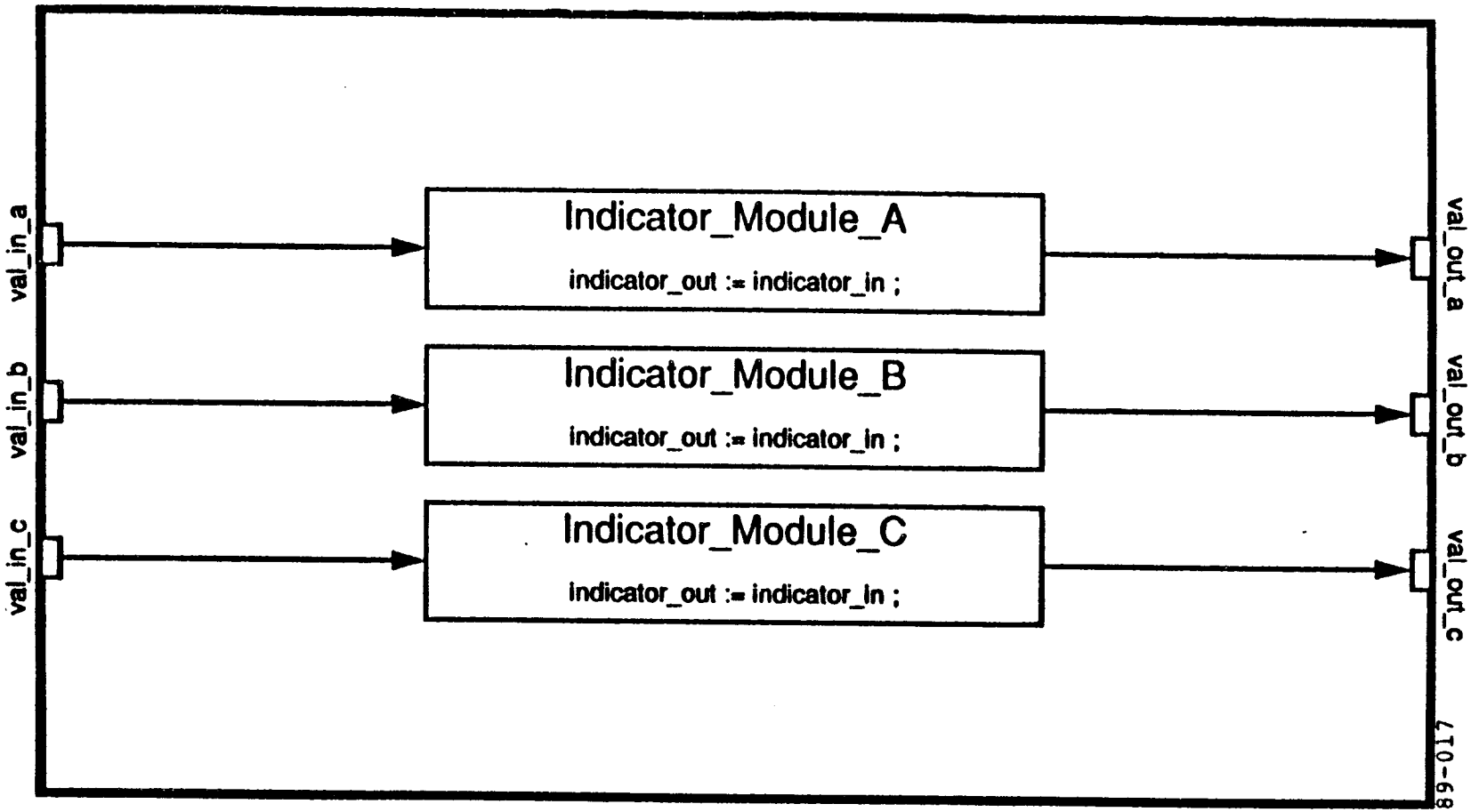


Figure 4.8

Example System Model: Binary Adder (Value_Module) Lower System Level Representation

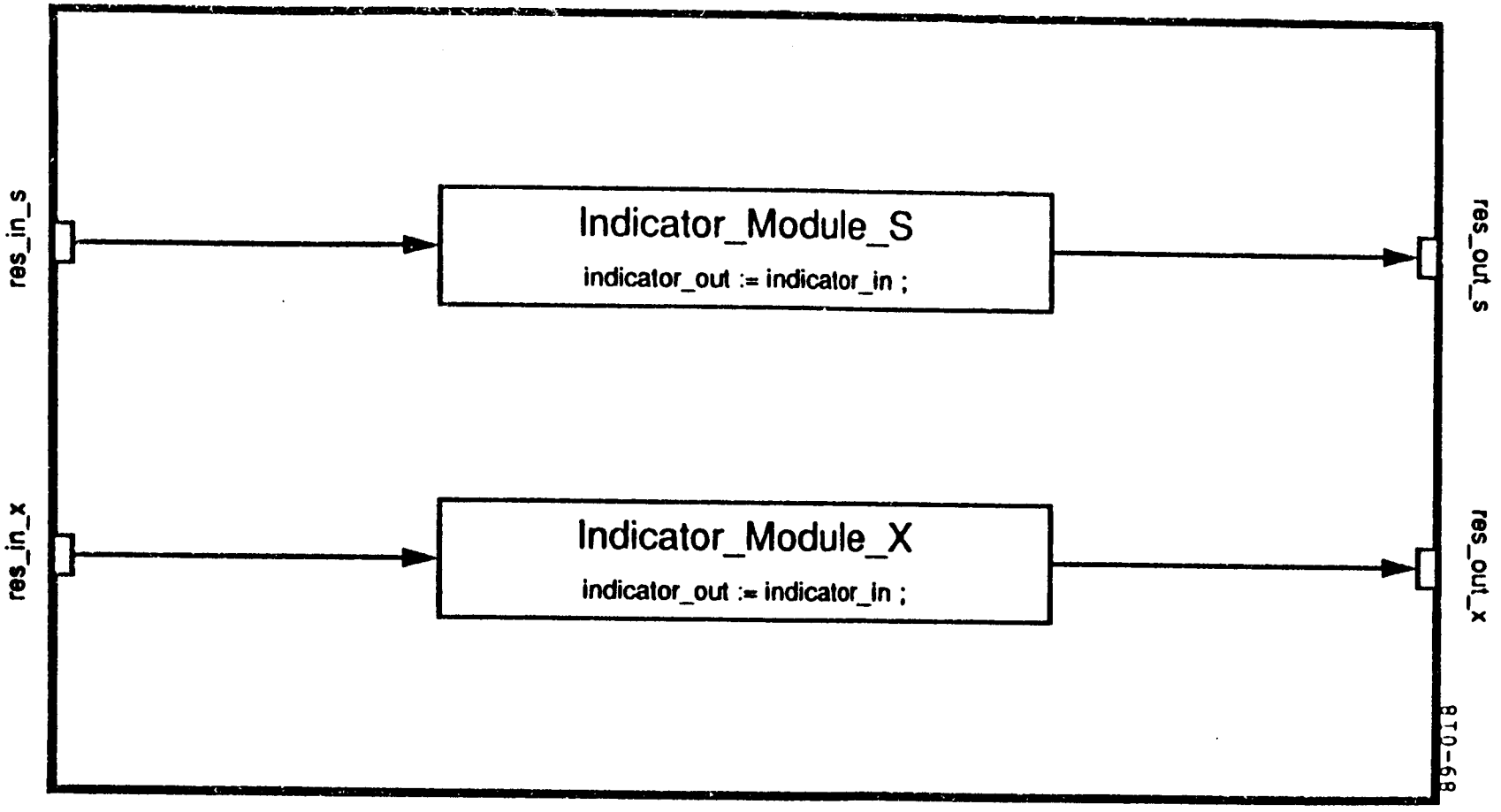


Figure 4.9

Example System Model: Binary Adder
(Result_Module)
Lower System Level Representation

Example System Model: Binary Adder
(Sum_Generation_Module)
Lower System Level Representation

Figure 4.10

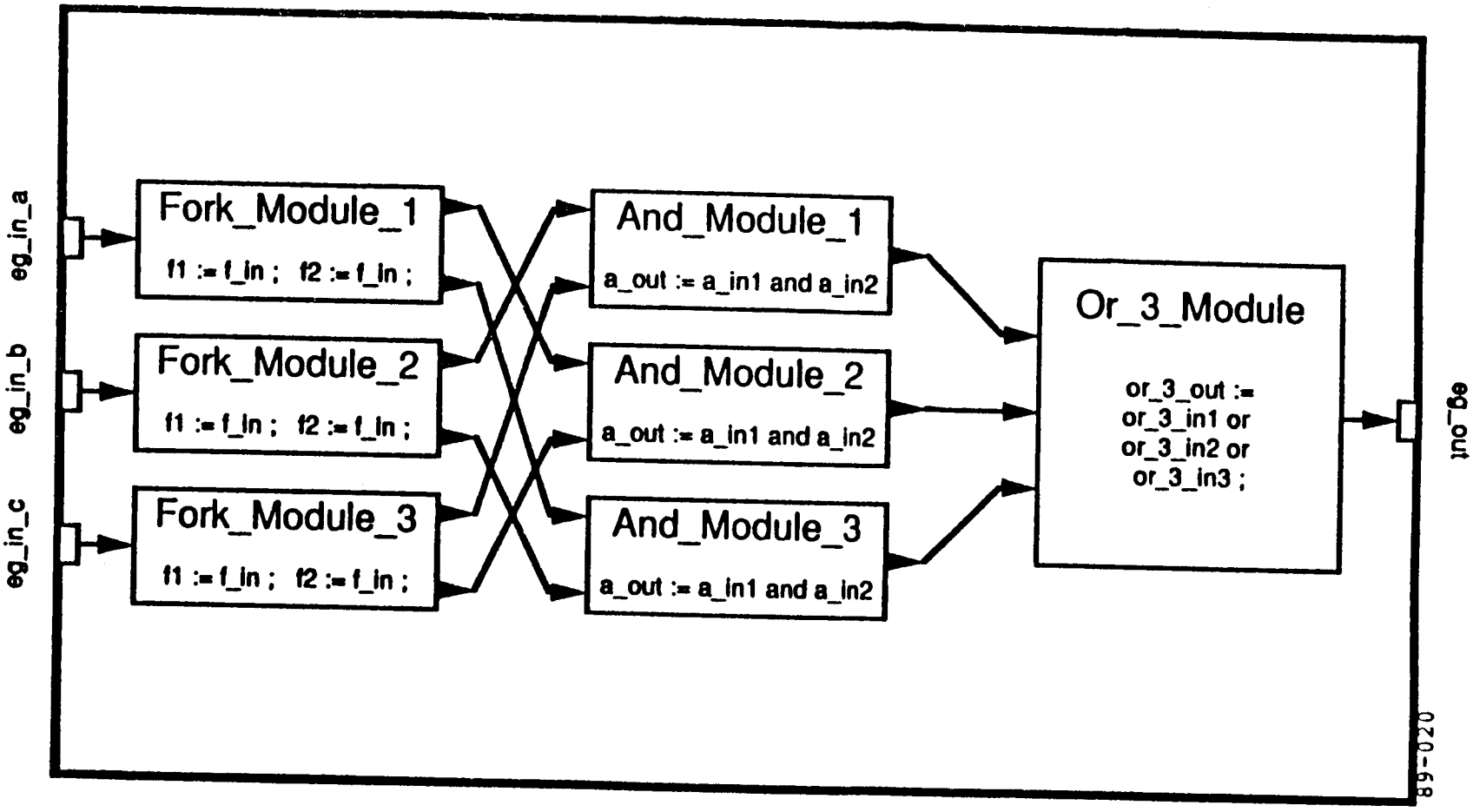


Figure 4.11

Example System Model: Binary Adder (Excess_Generation_Module) Lower System Level Representation

5. RULE SET PROCESSOR PROTOTYPE ARCHITECTURE

The Rule Set Processor prototype (RSP I) represents the working software component of the result of our investigation into the development of a tool for constructing real time expert system software for time critical applications.

The RSP prototype is a single computer program written in the Ada programming language. It was developed and tested on a Sun Microsystems SUN 3/160, a system that uses a Motorola MC68020 microprocessor and four megabytes of memory for a hardware platform along with Sun UNIX 3.4 as a software development environment. The Ada compiler in use is from Verdix Incorporated. The RSP Ada source is intended to be portable to any system that passes the standard Ada validation suite and possesses reasonable memory and file capabilities.

5.1 RSP I Architecture Overview

Written according to commonly accepted modular programming style, the prototype is implemented as a short main program along with thirty-two separate Ada packages organized according to functional purpose. The main program, Ada procedure "rsp", occupies a single source file; each package occupies two separate files: a package specification source file and a package body source file. These sixty-five files together contain three hundred and sixteen subprograms and a total of 14,355 source lines. The coding portion required approximately three months of concerted effort with parallel testing.

This is a brief description of the Ada source files used in the rule set project. The functional interrelations of the packages are described in further in this section of this document.

The main program is found is the file "rsp.a". All of the other Ada source files are paired: for each package, there is a single specification file and a single body file. Each package specification filename is sixteen characters long. The first four characters are "rsp_" and the last eight characters are "_pkg_s.a". The fifth through eighth characters identify the contents of the file. The corresponding package body file has the same name except the fourteenth character is a "b" (for body) instead of an "s" (for specification). Each package name is twelve characters long starting with "rsp_" and ending with "_pkg" with the middle four characters identifying the contents of the file. Thus, the package name is taken from the filename by dropping the last four characters.

A package may be referred to in abbreviated manner by supplying only its four distinguishing letters; for example, package rsp_cont_pkg (specification in the file rsp_cont_pkg_s.a and body in the file rsp_cont_pkg_b.a) may be indicated by the simpler term "package CONT". All subprograms contained in the packages start with the four characters "rsp_" and continue with the next five characters "XXXX_" where the four X letters are the four distinguishing letters of the package that hold the definition of the subprogram. This subprogram naming convention allows for the immediate recall of the defining package of any function or procedure.

All site dependencies are declared in the first section of the specification of rsp_defs_pkg in the file rsp_defs_pkg_s.a. The complete

history of the Ada source files in the RSP prototype is given in Appendix C. These dependencies include definition of the integer and float types used by the project. The rule set project also uses the package "text_io" from the standard Ada environment.

5.2. ISD Substructures

The program written by the RSP user in the User System Description Language (USDL) describes a single system. (The acronym in use for this program is "USD" for User System Description.). This single system, referred to as the root system after its compilation by the RSP, is represented as a complex internal system description (ISD). The ISD generated by the RSP contains all the information necessary to represent the described system completely including both its topological information (structures and connectivity) and its procedural information (rules and statements). The ISD also contains the statically allocated storage required for the interpretation of the USD.

The Internal System Description is built from linking together an assemblage of objects of several different Ada record types. Most record types correspond directly with USDL structures, and in those cases where an arbitrary number of language structures may appear in the same context (e.g., a sequence of statements or rules) these records are connected together in a two way linked list. The head of such a list corresponds to the first item encountered in that context of a particular type, and the tail of a list corresponds to the last item so encountered. Some object types used in the ISD may also refer to interior objects of the same type, either directly or indirectly, thus allowing for the power of topological recursive representation.

The entire ISD is built via dynamic allocation of its substructures. The single system in the USD program corresponds to the anchor of the ISD - an object of type "syst_t" (a system record) that represents the root system of the ISD. All parts of the ISD are reached through the pointer to this root system record.

The substructure record types listed in this section are those most closely connected with syntactic structures in the USDL. Other record types are also in use but are not listed here for sake of brevity.

5.2.1 ISD Substructure: System Record/Ada type "syst_t"

For the root system and for each subsystem in the model, an Ada record of type "syst_t" is allocated and initialized. Note that system records (and their contents) will be present in the ISD for subsystem indication via duplication from their original definition. This means that the "syst_t" structure built by parsing is used as a template; copies are made of this template for each use of the system as as subsystem representation of a block. These records are connected in linked lists (when associated with a parent system); and with descriptions of blocks/blocktypes (when associated with a subsystem representation).

The syst_t record contains the following information: 1) The user assigned name of the system; 2) A linked list of blocks; 3) A linked list of

blocktypes; 4) A linked list of declare items; 5) A linked list of paths; 6) A linked list of rulesets; 7) A linked list of systems; 8) A linked list of externals; 9) Links to previous and next system records.

5.2.2 ISD Substructure: Block Record/Ada type "comp_t"

An object of Ada type "comp_t" (from "component") is allocated and attached to the ISD for every block in the modeled system. These records are connected in linked lists and are associated with systems. A "comp_t" record contains the following: 1) The user supplied name of the block; 2) The user supplied name of the block's parent (if any); 3) A link to the type parent (if any); 4) A link to the description (attributes, lines, subsystem); 5) A count of connection points (lines, in and out); 6) Links to previous and next block records.

5.2.3 ISD Substructure: Blocktype Record/Ada type "ctyp_t"

An object of Ada type "ctyp_t" (from "component type") is allocated and attached to the ISD for every blocktype in the modeled system. These records are connected in linked lists and are associated with systems. A "ctyp_t" record contains the following: 1) The user supplied name of the blocktype; 2) The user supplied name of the blocktype's parent (if any); 3) A link to the type parent (if any); 4) A link to the description (attributes, lines, subsystem); 5) A count of connection points (lines, in and out); 6) Links to previous and next blocktype records.

5.2.4 ISD Substructure: Declare Item Record/Ada type "decl_t"

An object of Ada type "decl_t" (from "declare item") is allocated and attached to the ISD for every declare item in the modeled system. These records are connected in linked lists and are associated with systems, rulesets, and rules. A "decl_t" record contains the following: 1) The user supplied name of the declare item; 2) The basetype of the item; 3) The current value associated with the item; 4) Links to the previous and next declare items.

5.2.5 ISD Substructure: Path Record/Ada type "path_t"

An object of Ada type "path_t" (from "path connection") is allocated and attached to the ISD for every path in the modeled system. These records are connected in linked lists and are associated with systems. A "path_t" record contains the following: 1) The user supplied name of the path (if any); 2) A link to the source block; 3) A link to the source line; 4) A link to the destination block; 5) A link to the destination line; 6) Links to the previous and next paths.

5.2.6 ISD Substructure: Ruleset Record/Ada type "rset_t"

An object of Ada type "rset_t" (from "ruleset") is allocated and attached to the ISD for every ruleset in the modeled system. These records are connected in linked lists and are associated with systems and rulesets. A

"rset_t" record contains the following: 1) The user supplied name of the ruleset; 2) A linked list of declare items; 3) A linked list of rules; 4) A linked list of rulesets; 5) Links to previous and next rulesets.

5.2.7 ISD Substructure: Rule Record/Ada type "rule_t"

An object of Ada type "rule_t" (from "production rule") is allocated and attached to the ISD for every rule in the modeled system. These records are connected in linked lists and are associated with rulesets. A "rule_t" record contains the following: 1) The user supplied name of the rule; 2) A linked list of declare items; 3) A link to the expression that forms the test of the rule; 4) A link to the affirmative statement ("then part"); 5) A link to the alternative statement ("else part", if any); 6) Links to the previous and next rules.

5.2.8 ISD Substructure: External Record/Ada type "xtrn_t"

An object of Ada type "xtrn_t" (from "external") is allocated and attached to the ISD for every external in the modeled system. These records are connected in linked lists and are associated with systems. An "xtrn_t" record contains the following: 1) The user supplied name of the external; 2) A link to the associated block; 3) A link to the associated line; 4) Links to previous and next externals.

5.3 Top Level Control

RSP prototype execution begins with the procedure rsp in the Ada source file rsp.a. This procedure forms the main program of the prototype and it contains only a few statements. Procedure rsp first calls an initializer (rsp_cont_init), then calls the main command cycle (rsp_cont_cycle), and finally calls a terminator routine (rsp_cont_term). This procedure also provides for trapping exceptions unhandled elsewhere.

The package CONT provides for the high level control of the prototype. This package contains the main initializer controller (rsp_cont_init) that orchestrates all required initialization functions by calling subsidiary initialization routines of all other packages as appropriate. Package CONT also contains the main terminator routine (rsp_cont_term) that functions analogously to the initializer.

Package CONT contains the procedure rsp_cont_cycle that implements the command read-parse-dispatch loop. A command input line, either from a configuration file or from the user, is read by the routine rsp_cont_read. Tokens in the command line are parsed by rsp_cont_token_parse and checked by rsp_cont_token_check. The command dispatch itself is handled by the routine rsp_cont_dispatch. Each nonblank command line is viewed as a sequence of one or more tokens of which the first is interpreted as one of the available commands and following tokens are considered parameters to that command. Each command is fully processed after dispatch and before continuing the read-parse-dispatch loop.

5.4 Command Processing

A command read in and checked by package CONT is passed by the procedure rsp_cont_dispatch into a command specific procedure in package DPCR for processing. For each command type available, there is a corresponding procedure in package DPCR by the name of "rsp_dpcr_X", where X represents the spelling of the command. For example, the procedure rsp_dpcr_exit handles the "exit" command.

Some of the commands available require only a small amount of processing code, and their implementations can be entirely enclosed in their single handling routine. Other commands require much more extensive processing, and so involve many other portions of the prototype. In all cases, the command processing required is completed upon the return from the called dispatch routine.

5.5 Parsing and Compilation

The two main functions of the RSP prototype are compilation and interpretation. The compilation process is the translation of the User System Description source into an Internal System Description and is performed by the recursive descent parser (from package PARS). The interpretation of the ISD is handled by the interpretation subsystem (from package EXEC).

The term "parsing" is used to refer to the generation of the derivation sequence by which the USD source is built from the syntactical specification of the language. The word "compilation" indicates the processing, executed in parallel and as part of the parsing activities, that converts the externally represented (USD source) semantics of a system into an internal representation (ISD). The parsing detects syntactic errors and ensures that only correctly formed USDL sources can be then sent to a later interpretation stage; the compilation forms the ISD so that the later interpretation stage has something to work upon.

A proposed function of further RSP investigation is to incorporate a "translation" capability into the program. Such a translation facility would use the ISD produced by the parser and compilation processing and "translate" this structure into an external text file to be used as a source to a conventional software compiler. For example, a USD would be parsed, its generated ISD interpreted, and the final tested version would then be translated into a language (e.g., Ada, FORTRAN, 1750 Assembler) to be further processed.

5.5.1 Recursive Descent Parser

The parsing and compilation of a USD source is directed by use of the "compile" command. This command causes the indicated user source to be parsed, errors to be detected (if any), an ISD to be generated (if no errors), and a listing output file to be produced.

The mode of parsing is known as "recursive descent", a powerful technique for processing those languages that have recursive definitions and that allow single symbol disambiguation of alternate grammatical derivations. The USDL, like many languages, has recursive definitions such as expressions within

expressions and systems within systems. The USDL also has, by design, the above mentioned disambiguation property where it is easily decidable (by the parser) which syntactic derivation is the correct one at any point in the parse. The main reasons for choice of the recursive descent method are that it can be quickly implemented and it can be easily modified to allow for changes in the language during the construction of the prototype.

The recursive descent parser works in a "top-down" fashion - it starts with the working assumption that it has a complete program, and then proceeds to fill in the specifics by calling subsidiary routines to handle the various syntactic forms included in a complete program. The scanning of the input source is a one pass, forward scan with no backup. Each syntactic form (e.g., a program, a system, a rule, an expression) has a corresponding function in the parser. Each of these functions has a specific task: to parse the corresponding syntactic form and to create a corresponding data structure that holds the semantic information from the USD source for that form. The returned data structure is then connected to other parts of the ISD under construction. The final result is a complete ISD ready for interpretation or translation.

The parser is activated by a call to the procedure rsp_pars_compile from the dispatch routine rsp_dpcr_compile. This routine performs various initialization tasks first and then calls the syntactic form parsing function rsp_syst_rdp_syst to parse the entire USD source. When the parsing function completes, it returns a pointer to the structure representing the parsed system description; if no errors where detected during the parse, this pointer becomes the root pointer of the entire ISD. If one or more errors were detected, the ISD root pointer remains null.

The names of the routines in the RSP that are used to parse a syntactic form have a common format. Each function name takes the form "rsp_XXXX_rdp_YYYY" where XXXX is the defining package and YYYY is the name (usually shortened to four letters) of the parsed syntactic form. For some examples, the routine rsp_path_rdp_path parses a path description and the routine rsp_expr_rdp_expr parses a general expression. Table 5.1 lists all such functions in the RSP prototype along with the structure pointer types returned.

Table 5.1: RSP Prototype Functions

```
rsp_comp_rdp_comp returns compptr_t (blocks)
rsp_ctyp_rdp_ctyp returns ctypptr_t (blocktypes)
rsp_desc_rdp_attr returns attrptr_t (block attributes)
rsp_desc_rdp_desc returns descptr_t (block descriptions)
rsp_desc_rdp_line returns lineptr_t (block lines)
rsp_desc_rdp_subs returns subsptr_t (block subsystems)
rsp_expr_rdp_exp1 returns exprptr_t (level 1 expressions)
rsp_expr_rdp_exp2 returns exprptr_t (level 2 expressions)
rsp_expr_rdp_exp3 returns exprptr_t (level 3 expressions)
rsp_expr_rdp_exp4 returns exprptr_t (level 4 expressions)
rsp_expr_rdp_exp5 returns exprptr_t (level 5 expressions)
rsp_expr_rdp_exp6 returns exprptr_t (level 6 expressions)
rsp_expr_rdp_expr returns exprptr_t (general expressions)
rsp_path_rdp_path returns pathptr_t (paths)
rsp_prim_rdp_prim returns exprptr_t (primary expressions)
rsp_rset_rdp_rset returns rsetptr_t (rulesets)
```

Table 5.1: RSP Prototype Functions (continued)

```
rsp_rule_rdp_rule returns ruleptr_t (rules)
rsp_stmt_rdp_advn returns stmtptr_t (advance statement)
rsp_stmt_rdp_asgn returns stmtptr_t (assignment statement)
rsp_stmt_rdp_bloc returns stmtptr_t (compound statement)
rsp_stmt_rdp_call returns stmtptr_t (call statement)
rsp_stmt_rdp_dfrl returns stmtptr_t (read statement)
rsp_stmt_rdp_dfwl returns stmtptr_t (write statement)
rsp_stmt_rdp_elab returns stmtptr_t (elaborate statement)
rsp_stmt_rdp_exit returns stmtptr_t (exit statement)
rsp_stmt_rdp_ifte returns stmtptr_t (if-then-else statement)
rsp_stmt_rdp_null returns stmtptr_t (null statement)
rsp_stmt_rdp_puls returns stmtptr_t (pulse statement)
rsp_stmt_rdp_rest returns stmtptr_t (reset statement)
rsp_stmt_rdp_rtrn returns stmtptr_t (return statement)
rsp_stmt_rdp_scva returns stmtptr_t (accept statement)
rsp_stmt_rdp_scvd returns stmtptr_t (display statement)
rsp_stmt_rdp_stmt returns stmtptr_t (general statement)
rsp_syst_rdp_decl returns declptr_t (declare item)
rsp_syst_rdp_syst returns systptr_t (system)
rsp_xtrn_rdp_xtrn returns xtrnptr_t (external)
```

Note that some structure types are returned by more than one function. The parser is organized in this fashion so that the relatively complex forms (statements and expressions) can be handled in a divide and conquer strategy while retaining a single data representation.

Each of the above parsing routines are written according to a single plan, although the details will differ somewhat for each function. The actions of this general plan are:

1) Initialize a data structure of the type corresponding to the syntactic form to be parsed;

2) If appropriate, test the current source input to ensure that the routine was called is the correct routine for the input - and report an internal consistency fault if the wrong routine was called;

3) Scan and record any items associated with a prologue (if any) of the syntactic form;

4) For each item scanned in the source input, either record information directly into the local working data structure, or invoke subsidiary parsing routines to construct the appropriate data structures and then record the resultant pointer values into the local data structure;

5) Scan and record any items associated with a epilogue (if any) of the syntactic form;

6) Return as the function result a pointer to the working data structure constructed to represent the information gained from the parse of the syntactic form.

(For each step in the plan, complete syntactic and semantic error detection and reporting also takes place.)

5.5.2 Lexigraphical Analyzer

The RSP prototype uses a lexigraphical analyzer (a.k.a. lexer) to provide an interface between the main functions of the parser and the text of the USD source. The main function of the lexer is to read characters sequentially from the user source and to detect and assemble lexical tokens. Lexical tokens are the atomic syntactic forms of the USDL. These tokens include: literal values (integers, floats, booleans, and strings), identifiers (user supplied names), reserved words (keywords), and reserved symbols (punctuation and the end-of-file marker).

The lexer resides in package LEXR. The main lexer subprogram is the procedure rsp_lexr_next_token which scans the user source and generates a value for the variable "tokn" (also defined in package LEXR). This variable is declared to be of the record type "tokn_t" and its components contain the pertinent information about the token required for parsing and ISD generation. The lexer also detects and reports certain simple source errors. As the lexer scans the input text lines it also sends these lines to the listing generation system so that a listing file may be produced.

5.5.3 Structure Allocation and Initialization

The data structures used to construct the ISD require routines for allocation and initialization. These routines are located in package ARCH (from "architecture"). For each type of record that can be used in the ISD, there is a corresponding routine that allocates and initializes storage for that type. These routines are called by the parser as required. The specification portion of package ARCH contains all of the Ada type declarations for these record types. The motivating idea for this centralization is aid development and maintenance efforts by simplifying the layout of the strongly interrelated definitions required for the ISD.

5.5.4 Error Management

The RSP prototype provides for the detection and reporting of those errors associated with parsing (lexical, syntactic, and semantic). The RSP detects and reports sixty-three different error types. Error processing is enclosed in package ERRS.

When an error is detected, a call to the procedure rsp_errs_scan_post is made with the error type and the error text position passed. This procedure records the error information along with other error information for that line (if any). This collection of information (error type index / source column number) is saved for further listing processing so that the error indications are reported to the user in a useful fashion.

5.5.5 Scope Management

Some USDL syntactic forms have an associated lexical scope. A lexical scope is the contiguous source region where a name is available. For example, a name associated with a declare item in a ruleset is available only in the scope of that ruleset, while the name of a declare item in a system is available throughout the system. In order to detect various semantic errors

and to correctly generate the ISD, the parser requires knowledge of the current scope and of enclosing scopes.

Scope information is processed by routines in package SCOP. This package has routines to push (establish) a new scope and to pop (disestablish) the current scope. These routines are called by those parsing routines that handle syntactic forms which have associated scopes. Also included are functions that search for identifiers throughout the enclosing scopes to locate corresponding ISD structures. The scoping information handled here is recording in a linked list data structure, one record for each scope. This list is the only data structure in the RSP other than the ISD that records information about the USD source.

5.5.6 Source Listing Processing

The RSP prototype generates an output listing text file as part of the compilation process. This generation is performed by routines in package LIST. The output file contains several items:

1) One informative header that identifies the file and the version of the RSP program responsible for the compilation;

2) A listing of the user source with line numbering;

3) When required, error indicators composed of error type indices (small integers) adjacent to the line and column of detection in the user source;

4) When required, a decoder table after the end of the user source listing that contains earlier reported error type indices (numerically ordered) along with corresponding descriptive messages.

5.6 ISD Interpretation

The RSP prototype creates an ISD upon the successful compilation of an error-free user system description. This ISD can then be interactively interpreted by the ruleset interpretation routines. Interpretation is triggered by one of the appropriate commands, each of which starts the interpretation of a particular ruleset embedded in the root level system. These commands are: "preset", "diagnose", and "simulate", each of which initiates the interpretation of the ruleset with the same name. The entry point for ruleset interpretation is the procedure rsp_exec_rset; it is called with a pointer into the ISD that identifies the data structure corresponding to the indicated ruleset.

5.6.1 ISD Sequencing

The correct interpretation of a ruleset requires that the various portions of the ruleset are interpreted using the correct ordering. This means that certain static and dynamic information concerning rules and statements may cause the path of interpretation to vary from the normal top-down sequential convention. In all cases, the processing of any part of

interpretive code in the ISD corresponds to a potentially recursive call of a procedure in the RSP prototype.

A ruleset is interpreted (rsp_exec_rset) by interpreting its rules in top-down order until some condition occurs to suspend, abort, or normally terminate ruleset interpretation. After that last rule is processed, the flow of interpretation returns to the invoker of the ruleset. If the ruleset was invoked by an interactive command, interpretation terminates and control is returned to the interactive level.

A rule is interpreted (rsp_exec_rule) by first evaluating (rsp_eval_expr) its test expression and then either executing (rsp_exec_stmt) the rule's affirmative statement (test true) or its alternative statement (test false).

A statement (rsp_exec_stmt) is interpreted by a dispatch to a routine that corresponds to the statement kind begin interpreted. Most statement kinds do not modify the normal flow of interpretation, but a few do change this flow.

A compound statement (rsp_exec_bloc) is interpreted by processing each of its enclosed statements in top-down order, subject to possible diversions by these enclosed statements.

An if-then-else statement (rsp_exec_ifte) is interpreted in a manner analogous to a rule's interpretation. Its expression is evaluated and the appropriate statement is then interpreted.

A return statement (rsp_exec_rtrn) is interpreted by the immediate termination of the enclosing ruleset and a return to the ruleset invoker.

An exit statement (rsp_exec_exit) is interpreted by the immediate termination of the inclosing ruleset and of all invoking rulesets and the termination of the interpretation session with a return to the interactive command level.

A call statement (rsp_exec_call) is interpreted by the temporary suspension of the enclosing ruleset and a transfer to another ruleset. When the called ruleset concludes, interpretation (generally) returns to the caller.

An elaborate statement (rsp_exec_elab) is interpreted in the same fashion as the call statement except that the current system level is changed to a subsystem representation of the indicated block thus allowing multilevel representation of systems and system knowledge. When the elaborated ruleset concludes, interpretation (generally) returns to the caller, and the current system level returns to its prior status.

5.6.2 ISD Expression Evaluation

Various parts of the ISD (declare items, lines, line history, and attributes) will assume scalar values. These objects are collectively referred to as variables as their values may vary with interpretation. These variables, along with literals (constant values) may be referenced and combined in expressions in order to represent the numerical and logical

aspects of the system model. (Most RSP interpretive evaluation is coded in package EVAL.)

An expression is an arrangement of one or more of these values along with zero or more operator symbols according to the syntactic specification of the USDL. Expressions are parsed by a set of routines that are organized into a hierarchy that correctly records the desired evaluation precedence by constructing an expression tree with nodes representing operators and operand. During interpretation, an expression tree is evaluated (rsp_eval_expr) using a normal recursive infix transversal starting at the root of the tree. For each of the available operator kinds (thirty-two in all) there is a separate routine that handles that particular operator. Each of one of these routines handles evaluation for all appropriate scalar basetype (boolean, float, and integer).

5.6.3 ISD Scalar Location and Access

The Internal System Description data structure contains storage for all of the scalars in the system model. Storage for a scalar is bound to the defining record in the ISD. Scalars associated with declare items are allocated in decl_t records and scalars associated with block attributes are allocated in attr_t records (inside of blocks). Scalar storage for the one or more history values associated with lines are allocated in linked lists connected to line_t records.

Access and manipulation of scalar storage is handled in packages SASA and HIST. These routines are called mostly by the expression evaluation routines to load or store values. Routines also exist in these packages for the resetting of initial object states and to propagate values throughout the topography of the system model.

5.7 I/O Utilities

All input/output operations are performed with the standard family of Ada routines. For purposes of modularity, access to these routines is itself compartmentalized into its own package (package IOCP).

All files used are organized as basic sequential access text files opened for either reading or for writing. File formats used by the RSP are kept simple so as to also possible interfacing with other software development tools.

6. CONCLUSIONS AND RECOMMENDATIONS

6.1 Conclusions

The central goal of this research effort is the exploration of techniques for the implementation of expert system programming for the domain of real time dynamic systems. The work includes both an examination of an established, general purpose expert system shell approach (using CLIPS) and a new language/interpreter approach (USDL/RSP) that is tailored for simulation and diagnosis of real time dynamic systems. These two methods have some similarities, but also have important differences.

The selected general purpose expert system shell approach employs a fairly large program, written in the C language, that uses a powerful and general pattern matching capability for inference processing. A drawback of this method is that extensive computational facilities are required for running the expert system shell, either in its interpretive mode or in its standalone application mode. Moreover, most shells are written in a computer language that is not yet well standardized and for which optimized compilers may not be readily available for those dedicated processors commonly used for real-time onboard tasks.

The USDL/RSP approach also employs a fairly large program, comparable in size with a general purpose interpreter. However, this program is only used during the interactive development phase (where plenty of computational resources are present) where size is not as important as a consideration as it is during the production phase (in a more restrictive embedded requirement). Although not yet operational, the translator portion of the RSP program should produce compact, quick, and compilable Ada source for the target processor. The reasoning behind this prediction is based upon the differences in the inference mechanisms involved: the extensive resource requirements of pattern matching of a general purpose shell versus the direct, no-search, programmer-directed ruleset approach of the RSP. While the ruleset method may not be among the first chosen for general purpose reasoning in general domains, its use of a standardized language (Ada) along with the speed and integration requirements of real time applications make its a good candidate for further development.

The general purpose expert system shell approach requires extensive programming skills by the developer in order to construct the system model. Usually the general flavor of programming is that of the LISP computer language with a slight mixture of the constructs of a procedural language such as C. Usage of a LISP style is certainly appropriate for problems that require search and pattern matching where the system model and the inference mechanism used in reasoning about the system is not well known even to the domain experts, as many years of artificial intelligence programming practice has demonstrated. However, much of the power of expression may be wasted because a framework for the representation of elemental dynamic objects required for event monitoring and fault diagnosis is not readily available in these general purpose shells.

The RSP approach also requires considerable programming skills by the developer. However, the User System Description Language is designed with two important features in mind: first, the USDL is explicitly intended for the modeling of dynamic systems; second, the USDL style is intentionally similar to that of the Ada language. The result of the first feature is to eliminate

excess baggage (and thus lessen chance for programming error), the result of the second feature is to reduce the probable effort required in training programmers.

The central conclusion of this investigation is that use of a dedicated language/interpreter is appropriate for applying expert system programming methodologies to the domain of onboard fault diagnosis and event monitoring applications. Furthermore, we have identified the desirable knowledge representation and inference mechanism facilities for such a dedicated real-time expert system shell and incorporated these feature into the USDL/RSP design:

- RSP provides a framework for specifying a physical system model which includes both the topological representation (elemental dynamic objects, physical interconnections between these objects) and the procedural representation (functional dynamic system representation).

- RSP provides a hierarchical dynamic representation mechanism allowing multiple hierarchical representations of a dynamic system at several levels of abstraction. For instance, such a representation mechanism can be used to look at a flight control system as a single block with pilot inputs and actuator commands, or as interconnection of several interconnected dynamic objects such as limiters, shaping filters, notch filters, etc.

- RSP provides the necessary tools in its ruleset for building powerful inference strategies which can move up and down (elaborate statement) or horizontally (call statement) within the specified knowledge hierarchy. Such a mechanism can be used to construct top-down, bottoms-up or hybrid diagnosis strategies.

6.2 Recommendations

The central recommendation of this research is to continue the examination and development of the promising dedicated real-time expert system shell the USDL/RSP strategy. Such examination and development can be targeted towards specific goals:

Goal: The application of the USDL/RSP approach to a real world onboard fault diagnosis application. This goal means the selection of a particular real-time application such as a reconfiguration strategy for a self-repairing flight control system (Caglayan et al. 1987), or a sensor failure detection and isolation system (Caglayan et al. 1988), or a fault diagnosis system for a flight control system of a specific aircraft, development of an expert system based on the USDL/RSP approach for the selected application, and demonstration of the developed expert system in a real-time simulation.

Such a goal is the real test of the USDL/RSP method. Because of the successes in meeting the initial goals of this investigation, we believe that future exploration of the USDL/RSP is indicated, and that such exploration include a treatment of a real world, real-time application. In order to achieve this goal, the following modifications would be needed to the RSD prototype design:

Goal: Expansion of constructs in the User System Description Language. As currently specified, while the USDL is capable of extensive modeling efforts, it lacks several useful features commonly found in conventional languages: structured types (arrays and records), subprogram formal parameters, and a facility for separate compilation. The addition of these features would be a natural extension of the corrent effort as the current syntactic specification is designed with such extension in mind.

Goal: Correction of certain deficiencies in the Rule Set Processor interpreter. RSP I is a partial implementation of the RSP prototype design. For instance, it does not provide interpretation for certain hierarchical models. Also, the RSP compilation occasionally emits spurious complaints about incorrectly perceived semantic source errors. An important extension to the RSP interpreter is to provide a more comprehensive interactive debugging facility; such a facility should greatly increase developer productivity as have similar high level debugging tools for conventional programming languages.

Goal: Implementation of a convenient system component library mechanism. Currently, the RSP program requires the developer to manually combine component specifications into a single source file prior to compilation. A much more desirable approach is to have the RSP program handle the librarian task of configuring systems that involve the reuse of software. This library mechanism of the RSP corresponds with the addition of a separate compilation facility in the USDL syntax.

Goal: Implementation of the RSP translator mechanism. The current RSP program is not yet able to translate the Internal System Model into an standalone application source language program to be embedded onto a flight computer. This effort requires not only the programming of the ISD-source translator, but also the resolution of the interfacing issues involved in the porting of the expert diagnostic knowledge.

7. REFERENCES

Abbott, K.H., Schutte, P.C., Palmer, M.T. and Ricks, W.R., "Faultfinder: A Diagnostic Expert System with Graceful Degradation for Onboard Aircraft Applications, Symposium on Aircraft Integrated Monitoring Systems, Friedriechshafen, W. Germany, Sept. 15-17, 1987.

Braunston, L., Farrell, R., Kant, E., and Martin, N., Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming, Addison-Wesley, Reading, MA, 1986.

Brown, J.S., Burton, R.R., and Bell, A.G., "SOPHIE: A Sophisticated Instructional Environment for Teaching Electronic Troubleshooting," Bolt Beranek and Newman, Inc., Report No. 2790, Cambridge, MA, 1974.

Buchanan, B.G., and Feigenbaum, E.A., "DENDRAL and Meta-DENDRAL: Their Applications Dimension," Artificial Intelligence, Vol. II, 1978.

Caglayan, A.K., Rahnamai, K., Moerder, D.D. and Halyo, N., (1987) "A Hierarchical Reconfiguration Strategy for Aircraft Flight Control Systems Subjected to Actuator Failure/Surface Damage," AFWAL-TR-87-3024, May 1987.

Caglayan, A.K., Godiwala, P.M. and Satz, H.S., "User's Guide to the FINDS Computer Program," NASA CR 178410, June 1988.

Davis, R., Shrobe, H., Hamscher, N., Wreckert, K., Shirley, M., and Polit, S., "Diagnosis Based on Descriptions of Structure and Function," AAAI Proceedings, Carnegie-Mellon Univ., Pittsburg, PA, August 1982.

Davis, R., "Reasoning from First Principles in Electronic Troubleshooting," Int. J. Man-Machine Studies, Vol. 19, pp. 403-423, 1983.

Davis, R., "Diagnostic Reasoning from Structure and Behavior," Artificial Intelligence, Vol. 24, pp. 347-410, 1984.

Davis, R., "Diagnosis via Causal Reasoning: Paths of Interaction and the Locality Principle," Proceedings AAAI-83.

Davis, R., "Knowledge-Based Systems: The View in 1986," in AI in the 1980s and Beyond - An MIT Survey, editors Grimson and Patil, The MIT Press, Cambridge, MA, 1987.

Davis, K., "Diagnostic Expert System for the B1B," IEEE AES Magazine, April, 1988.

Disbrow, J.D., Duke, E.L., and Regenie, V.A., "Development of a Knowledge Acquisition Tool for an Expert System Flight Status Monitor," NASA TM 86802, January 1986.

Duda, R.O., Gaschnig, J., Hart, P.E., Konolige, K. Reboh, R., Barrett, P., and Slocum, J., "Development of the PROSPECTOR Consultation System for Mineral Exploration Inc., Final Report, Project No. 6415, SRI International Inc., Menlo Park, CA, 1978.

Duke, E.L. and Regenie, V.A. "Description of an Experimental Expert System Flight Status Monitor," NASA TM 86791, October 1985.

C-2

Duke, E.L., Regenie, V.A., Brazee, M., and Brumbaugh, R.W., "An Engineering Approach to the Use of Expert Systems Technology in Avionics Applications," NASA TM 88263, May 1986.

Forbus, K.D., "Qualitative Process Theory," A.I. Memo No. 694, M.I.T. Artificial Intelligence Laboratory, Cambridge, MA, February 1982.

Forbus, K.D., "Qualitative Process Theory," Ph.D. Thesis, M.I.T., Dept. of Electrical Engineering and Computer Science, 1984.

Forbus, K., "Interpreting Observations of Physical Systems," IEEE Trans. on SMC, Vol. 17, No. 3, May 1987.

Forbus, K., "Qualitative Physics: Past, Present and Future," in Exploring Artificial Intelligence, Morgan Kaufman Publishers, 1988.

Forgy, C.L., "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," Artificial Intelligence, Vol. 19, 1982.

Govindaraj, T., "Qualitative Approximation Methodology for Modelling and Simulation of Large Dynamic Systems: Applications to a Marine Steam Power Plant," IEEE Trans. on SMC, Vol. 17, No. 6, 1987.

Giarratano, J.C., "CLIPS User's Guide, Version 4.1," AI Section, NASA Johnson Space Center, 1987.

Gupta, A., "Parallelism in Production Systems: The Sources and the Expected Speed Up," Expert Systems and their Applications, Fifth International Workshop, Avignon, France, 1985.

Laffey, T.J., et. al., "Real-Time Knowledge Based Systems," AI Magazine, Vol. 9, No. 1, Spring, 1988.

Malcolm, J.G. and Highland, R.W., Analysis of Built-In-Test (BIT) False Alarm Conditions, RADC-TR-81-220, Rome Air Development System, Griffiss, AFB, NY, 1981.

Malin, J.T., "Processes in Construction of Failure Management Expert Systems from Device Design Information," IEEE Trans. on SMC, Vol. SMC-17, No. 6, 1987.

Palmer, M.T. Abbott, K.H., Schutte, P.C., and Ricks, W.R., "Implementation of a Research Prototype Onboard Fault Monitoring and Diagnosis System," 1987 Computers in Aerospace Conference, Wakefield, MA, October 7-9, 1987.

Rasmussen, J., "The Role of Hierarchical Knowledge Representation in Decision Making and System Management," IEEE Trans on SMC, Vol. 15, No. 2, 1985.

Regenie, V.A. and Duke, E.L., "Design of an Expert-System Flight Status Monitor," NASA TM 86739, 1985.

Schutte, P.C., Abbott, K.H., Palmer, M.T. and Ricks, W.R., "An Evaluation of a Real-Time Fault Diagnosis Expert System for Aircraft Applications," Proceedings IEEE CDC, Los Angeles, CA, December 9-11, 1987.

Shortliffe, E.H., Computer-based Medical Consultations: MYCIN, American Elsevier, New York, NY, 1976.

Shoham, Y., Reasoning About Change: Time and Causation from the Standpoint of Artificial Intelligence, The MIT Press, Cambridge, MA, 1988.

Stefik, M., Aikins, J., Balzer, R., Benoit, J., Birnbaum, L., Hayes-Roth, F. and Sacerdoti, E., "The Organization of Expert Systems: A Prescriptive Tutorial," Xerox Report No. VLSI-82-1, Palo Alto, CA, 1982.

APPENDIX A: USER SYSTEM DESCRIPTION LANGUAGE SYNTAX SPECIFICATION

```
-- ; Syntactic Specification for the User System Description Language
-- ; USDL version 1.1
-- ; 05 Oct 1988
--
-- <accept-prompt-string> ::=
--   <string-constant>
--
-- <accept-statement> ::=
--   accept <accept-prompt-string> <variable> ;
--   accept <variable> ;
--
-- <additive-expression> ::=
--   <prefix-expression>
--   <prefix-operator> <prefix-expression>
--
-- <additive-operator> ::=
--   +
--   -
--
-- <advance-statement> ::=
--   advance ;
--
-- <assignment-statement> ::=
--   <variable> := <expression> ;
--
-- <attribute-basetype> ::=
--   basetype <basetype-indicator>
--   <empty>
--
-- <attribute-body> ::=
--   <attribute-constant> <attribute-basetype> <attribute-default>
--
-- <attribute-constant> ::=
--   constant
--   <empty>
--
-- <attribute-default> ::=
--   default <literal>
--   <empty>
--
-- <attribute-definition> ::=
--   attribute <attribute-id> is <attribute-body> ;
--
-- <attribute-id> ::=
--   <identifier>
--
-- <attribute-indication> ::=
--   <component-id> . <attribute-id>
--
-- <basetype-indicator> ::=
--   boolean
--   float
--   integer
--
```

```
-- <boolean-literal> ::=
--   false
--   true
--
-- <call-statement> ::=
--   call <ruleset-id> ;
--
-- <component-body> ::=
--   <component-item>
--   <component-item> <component-body>
--
-- <component-description> ::=
--   <component-head> <component-body> <component-tail>
--
-- <component-head> ::=
--   block <component-id> is general begin
--   block <component-id> is type <component-type-id> begin
--
-- <component-id> ::=
--   <identifier>
--
-- <component-item> ::=
--   <attribute-definition>
--   <line-definition>
--   <subsystem-definition>
--
-- <component-tail> ::=
--   end ;
--   end <component-id> ;
--
-- <component-type-body> ::=
--   <component-type-item>
--   <component-type-item> <component-type-body>
--
-- <component-type-description> ::=
--   <component-type-head> <component-type-body> <component-type-tail>
--
-- <component-type-head> ::=
--   blocktype <component-type-id> is general begin
--   blocktype <component-type-id> is type <component-type-id> begin
--
-- <component-type-id> ::=
--   <identifier>
--
-- <component-type-item> ::=
--   <attribute-definition>
--   <line-definition>
--   <subsystem-definition>
--
-- <component-type-tail> ::=
--   end ;
--   end <component-type-id> ;
--
-- <compound-statement> ::=
--   begin <compound-statement-list> end ;
--
```

```
-- <compound-statement-list> ::=
--   <statement>
--   <statement> <compound-statement-list>
--
-- <conjunctive-expression> ::=
--   <relational-expression>
--   <relational-expression> <relational-operator> <conjunctive-expression>
--
-- <conjunctive-operator> ::=
--   and
--   cand
--   cor
--   or
--   xor
--
-- <declaration-description> ::=
--   declare <declaration-id> : <basetype-indicator> ;
--
-- <declaration-id> ::=
--   <identifier>
--
-- <declared-item-indication> ::=
--   <declaration-id>
--
-- <digit> ::=
--   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
--
-- <digit-sequence> ::=
--   <digit>
--   <digit> <digit-sequence>
--
-- <display-label-string> ::=
--   <string>
--
-- <display-statement> ::=
--   display <display-label-string> ;
--   display <display-label-string> <expression> ;
--   display <expression> ;
--
-- <elaborate-statement> ::=
--   elaborate <component-id> using <ruleset-id> ;
--
-- <empty> ::=
--
--
-- <exit-statement> ::=
--   exit ;
--
-- <exponential-expression> ::=
--   <primary>
--
-- <exponent> ::=
--   E + <digit-sequence>
--   E - <digit-sequence>
--   E <digit-sequence>
--
```

```
-- <expression> ::=
--   <conjunctive-expression>
--   <conjunctive-expression> <conjunctive-operator> <expression>
--
-- <external-description> ::=
--   external <external-id> is <line-indication> ;
--
-- <external-id> ::=
--   <identifier>
--
-- <floating-literal> ::=
--   <digit-sequence> . <digit-sequence>
--   <digit-sequence> . <digit-sequence> <exponent>
--
-- <history-indication> ::=
--   <component-id> . <line-id> . history [ <expression> ]
--
-- <identifier> ::=
--   <letter>
--   <letter> <letter-digit-underscore-sequence>
--
-- <if-statement> ::=
--   if <expression> then <statement> else <statement> end if;
--   if <expression> then <statement> end if;
--
-- <integer-literal> ::=
--   <digit-sequence>
--
-- <letter> ::=
--   a | b | c | d | e | f | g | h | i | j | k | l | m |
--   n | o | p | q | r | s | t | u | v | w | x | y | z |
--   A | B | C | D | E | F | G | H | I | J | K | L | M |
--   N | O | P | Q | R | S | T | U | V | W | X | Y | Z
--
-- <letter-digit-underscore> ::=
--   <digit>
--   <letter>
--   <underscore>
--
-- <letter-digit-underscore-sequence> ::=
--   <letter-digit-underscore>
--   <letter-digit-underscore> <letter-digit-underscore-sequence>
--
-- <line-basetype> ::=
--   basetype <basetype-indicator>
--   <empty>
--
-- <line-body> ::=
--   <line-mode> <line-basetype> <line-history>
--
-- <line-definition> ::=
--   line <line-id> is <line-body> ;
--
-- <line-history> ::=
--   history <integer-literal>
--   <empty>
```

```
--
-- <line-id> ::=
--   <identifier>
--
-- <line-indication> ::=
--   <component-id> . <line-id>
--
-- <line-mode> ::=
--   mode <mode-indicator>
--   <empty>
--
-- <literal> ::=
--   <boolean-literal>
--   <float-literal>
--   <integer-literal>
--
-- <mode-indicator> ::=
--   in
--   out
--
-- <multiplicative-expression> ::=
--   <exponential-expression>
--   <exponential-expression> * <multiplicative-expression>
--
-- <multiplicative-operator> ::=
--   *
--   /
--
-- <null-statement> ::=
--   null ;
--
-- <path-description> ::=
--   path from <line-indication> to <line-indication> ;
--   path <path-id> is from <line-indication> to <line-indication> ;
--
-- <prefix-expression> ::=
--   <multiplicative-expression>
--   <multiplicative-expression> <multiplicative-operator> <prefix-expression>
--
-- <prefix-operator> ::=
--   +
--   -
--   not
--
-- <primary> ::=
--   <literal>
--   <variable>
--   ( <expression )
--
-- <pulse-statement> ::=
--   pulse <component-id> ;
--
-- <read-statement> ::=
--   read ;
--   read <variable> ;
--
```

```
-- <relational-expression> ::=
--   <additive-expression>
--   <additive-expression> <additive-operator> <relational-expression>
--
-- <relational-operator> ::=
--   =
--   /=
--   <
--   <=
--   >
--   >=
--
-- <reset-statement> ::=
--   reset ;
--
-- <return-statement> ::=
--   return ;
--
-- <rule-body> ::=
--   <rule-body-declaration-sequence> <rule-body-test>
--
-- <rule-body-declaration-sequence> ::=
--   <declaration-description>
--   <declaration-description> <rule-body-declaration-sequence>
--   <empty>
--
-- <rule-body-test> ::=
--   if <expression> then <statement> end if ;
--   if <expression> then <statement> else <statement> end if ;
--
-- <rule-description> ::=
--   <rule-head> <rule-body> <rule-tail>
--
-- <rule-head> ::=
--   rule <rule-id> is begin
--
-- <rule-id> ::=
--   <identifier>
--
-- <rule-tail> ::=
--   end ;
--   end <rule-id> ;
--
-- <ruleset-body> ::=
--   <ruleset-body-declaration-sequence> <ruleset-body-rule-sequence>
--
-- <ruleset-body-declaration-sequence> ::=
--   <declaration-description>
--   <declaration-description> <ruleset-body-declaration-sequence>
--   <empty>
--
-- <ruleset-body-rule-sequence> ::=
--   <empty>
--   <rule-description>
--   <rule-description> <ruleset-body-rule-sequence>
--
```

```
-- <ruleset-description> ::=
--   <ruleset-head> <ruleset-body> <ruleset-tail>
--
-- <ruleset-head> ::=
--   ruleset <ruleset-id> is begin
--
-- <ruleset-id> ::=
--   <identifier>
--
-- <ruleset-tail> ::=
--   end ;
--   end <ruleset-id> ;
--
-- <statement> ::=
--   <accept-statement>
--   <advance-statement>
--   <assignment-statement>
--   <call-statement>
--   <compound-statement>
--   <display-statement>
--   <elaborate-statement>
--   <exit-statement>
--   <if-statement>
--   <null-statement>
--   <pulse-statement>
--   <read-statement>
--   <reset-statement>
--   <return-statement>
--   <write-statement>
--
-- <string-character> ::=
--   <letter>
--   <digit>
--   ! | | # | $ | % | ^ | & | * | ( | ) | _ | + | - | = | ~ | ` | [ | ] |
--   { | } | \ | < | > | , | . | / | : | ' | ; | ? |   | |
--
-- <string-character-sequence> ::=
--   <empty>
--   <string-character>
--   <string-character> <string-character-sequence>
--
-- <string-constant> ::=
--   " <character-sequence> "
--
-- <subsystem-definition> ::=
--   subsystem <system-id> ;
--
-- <system-body> ::=
--   <system-item>
--   <system-item> <system-body>
--
-- <system-description> ::=
--   <system-head> <system-body> <system-tail>
--
-- <system-head> ::=
--   system <system-id> is begin
```

```
--
-- <system-id> ::=
--   <identifier>
--
-- <system-item> ::=
--   <component-description>
--   <component-type-description>
--   <declaration-description>
--   <external-description>
--   <path-description>
--   <ruleset-description>
--   <system-description>
--
-- <system-tail> ::=
--   end ;
--   end <system-id> ;
--
-- <underscore> ::=
--   _
--
-- <user-system-description> ::=
--   <system-description>
--
-- <variable> ::=
--   <attribute-indication>
--   <declared-item-indication>
--   <history-indication>
--   <line-indication>
--
-- <write-statement> ::=
--   write ;
--   write <expression> ;
--   write <string> ;
--   write <string> <expression> ;
--
-- ; end of syntactical specification
```

APPENDIX B: USER INTERFACE SPECIFICATION

The RSP (Rule Set Project) is an expert system development program with extensive emphasis on the description, simulation, and diagnosis of topologically complex realtime systems.

This appendix describes the Rule Set Project user interface. The user interface is the interaction between the RSP computer program and a user of the system, either performed with an interactive terminal or via batch processing. The interface operates with a standard character file stream style that would be appropriate for most text oriented terminals and also for text I/O in a windowing environment.

The RSP user interface is a command driven system. The RSP program first signs on to initialize the interface, and then repeatedly requests and processes user commands. The RSP program terminates by one of the following: an explicit user command, an end of file condition on the command input stream, or by the occurance of an abnormal operating event.

The use of a simple command driven system is easy to learn, can be quickly and confidently implemented, helps enable RSP program host independence, and is easily extendable.

The RSP program sign-on consists of various program identification items including: program name, program version, the time and date of program generation, and the current time and date. The intent here is not just for the sake of verbosity but instead to aid in configuration control and deficiency reporting. The sign-on messages, along with all non-abnormal reporting, is written to the standard output text file stream in order to allow for high level redirection should the host operating system support file redirection.

The RSP program command cycle is invoked after the program sign-on. Each run through this cycle has the following actions:

1) A command prompt ": " (colon space) is output.
2) A user RSP command line is read from the input.
3) The user command is parsed and checked for validity.
4) The validated user command is processed.

The RSP program allows for use of a configuration file. If the file "config" exists and is readable, the program will read and process commands from this file immediately after the program sign-on. For these commands, both the prompt and the command are printed to the standard output. The intent is that interactive outputs of the RSP system are identical for identical inputs irregardless of the source of the commands (either an initialization file or by manual input).

After the commands on the initialization file (if any) are processed, the normal interactive cycle will commence. The exception is that if some command on the initialization file caused the system to terminate early.

Eventually, the RSP program will complete command processing, either because of an explicit command or by an abnormal operating event. This concludes the command cycle and provides for normal program termination

activities. The program closes any open data files, reports interesting operation statistics, and signs off.

Each command line entered is parsed for validity immediately after it is entered. Blank input lines are ignored. A valid command line is composed of tokens separated by whitespace characters. A token is a sequence of non white space characters; a white space character is a character from the set {blank, tab, form feed}. Token charcters are those with ASCII values from "!" (hex value 21) upto "~" (hex value 7e). Characters outside these values signify an error in the command line.

The first token on the input command line should match one of the commands in the RSP program. If the first token does not match an available command, the line is considered to be erroneous and the user is encouraged to try the "help" command for assistance. If the first token does match an available command, the remaining tokens (command parameters), if any, are scanned for for consistancy with the indicated command. Note that command token matching, like the rest of the text processing activities of the RSP system, is fully case sensitive.

Erroneous command lines are diagnosed, reported, and then ignored. Unless an abnormal operating event has occurred, the command cycle is re-entered.

RSP Command Table (all commands in lower case)

Command	Action
clearflag	Clear zero or more internal processing flags.
compile	Generate internal representation of current source file.
describe	Present a brief descption of zero or more symbols.
dribbleoff	Deactivate auxillary output dribbling.
dribbleon	Activate auxillary output dribbling.
dump	Produce RSP program developer diagnostic file dump.
exit	Terminate RSP processing (preferred form).
halt	Terminate RSP processing.
help	Present a brief description of available RSP commands.
listing	Specify an output file for listing purposes.
noop	No operation.
object	Specify an output text file for object processing.
quit	Terminate RSP processing.
setflag	Set zero or more internal processing control flags.
preset	Preset inter representation.??
simulate	Simulate internal representation.
diagnose	Diagnose internal representation.
source	Specify an input text file for source processing.
status	Present a brief status report.
stop	Terminate RSP processing.
translate	Generate object output based upon internal representation.

RSP Processing Control Flags (Options):

Option	Description (action when set)
debug	Enables developer debugging operations.
tr_source	Trace: source echo to console
tr_token	Trace: token echo to console
verbose	Enables increased reporting of RSP internal operations.

Command Description: clearflag

The "clearflag" command is used to clear (set to false) zero or more RSP internal flag variables. The RSP system has several user accessible flags to control various aspects of processing. The current flag values can be printed with the "status" command. User accessible flags can be set with the "setflag" command.

The clearflag command takes zero or more additional parameters; each parameter is the name of an internal user accessible flag.

Command Description: compile

The "compile" command is used to generate a complete internal representation of the user system description found on the indicated source file (established by a prior "source" command). The compile command processing first performs a syntactical check identical to the "check command". If no errors are detected, an internal interpretable representation of the user system is constructed in memory and is initialized for either simulation (see the "simulate" command) or for translation (see the "translate" command).

The compile command takes no additional paramters; the source, listing, and object files should already be specified with previous "source", "listing", and "object" commands.

Command Description: describe

The "describe" command is used to present information about zero or more symbols. The informative symbol descriptions are generated from internal information resulting from a previous compile command. The intent is to provide a helpful feature for controlled access into the RSP system symbol table mechanism.

The describe command takes zero or more parameter tokens; each token is an identifier found in the RSP symbol table.

Command Description: diagnose

The "diagnose" command provides for the internal diagnosis of the user system descption by interpretation of the corresponding data structure resulting from compilation. A correct, compiled user system description must already exist.

The diagnose command takes no additional parameters.

Command Description: dribbleoff

The "dribbleoff" command is used to deactivate the output dribble facility. After the dribble is deactivated, console output is not longer echoed to the output dribble file.

The dribbleout command takes no additional parameters.

Command Description: dribbleon

The "dribbleon" command is used to activate the output dribble facility. After the dribble is activated, console output is echoed to the output dribble file. User command line input is also echoed to the dribble file.

The dribbleout command takes a second optional parameter, the name of the dribble file to receive the echo of the console output. If no additional parameter is specified, the file name "dribble" is used. In either case, the selected output file is cleared prior to use.

Command Description: dump

The "dump" command is used to provide a RSP system diagnostic dump onto a file. This feature is intended only for use by the RSP project development staff.

The dump command takes an optional second parameter: the name of the file to receive the dump. If no file name is present, the file "dump" receives the diagnostic dump.

Command Description: exit

The "exit" command is used to terminate execution of the RSP program. The processing of this command concludes the command cycle processing and initiaites RSP system normal shutdown operations. Synonyms for this command are: "halt", "quit", and "stop".

The exit command takes no additional parameters.

Command Description: halt

The "halt" command is a synonym for the "exit" command.

Command Description: help

The "help" command is used to provide a brief description of the available commands of the RSP program user interface.

The help command takes no additional parameters.

Command Description: listing

The "listing" command is used to specify an output text file to recieve the listing of the user system description produced by the "translate" command.

The listing command takes an optional second parameter: the name of the file upon which user system description listing is written. If no second parameter is present, the name "listing" is used for the output listing file.

Command Description: object

The "object" command is used to specify an output text file to recieve the Ada source code produced by the "translate" command.

The object command takes an optional second parameter: the name of the file upon which generated Ada realtime code is written. If no second parameter is present, the name "object" is used for the output object file.

Command Description: preset

The "preset" command provides for the internal presetting of the user system descption by interpretation of the corresponding data structure resulting from compilation. A correct, compiled user system description must already exist.

The preset command takes no additional parameters.

Command Description: quit

The "quit" command is a synonym for the "exit" command.

Command Description: setflag

The "setflag" command is used to set (set to true) zero or more RSP internal flag variables. The RSP system has several user accessible flags to control various aspects of processing. The current flag values can be printed with the "status" command. User accessible flags can be cleared with the "clearflag" command.

The setflag command takes zero or more additional parameters; each additional parameter is the name of an internal user accessible flag.

Command Description: simulate

The "simulate" command provides for the internal simulation of the user system descption by interpretation of the corresponding data structure resulting from compilation. A correct, compiled user system description must already exist. RSP simulation provides a rich, comprehensive, and user-directed examination of the described system including its user-supplied simulation ruleset.

The simulate command takes no additional parameters.

Command Description: source

The "source" command is used to specify the text file that contains a user system description to be processed by the RSP program.

The source command takes an optional second argument: the name of the input file that contains the user system description. If no second argument is supplied, the name "source" with be used for the input source file.

Command Description: status

The "status" command is used to present a brief status report about the current state of the RSP system. The report contains items such as: the names of the currently associated files, the values of the user accessible flags (options), the current state of the internal user described system, and the current resource utilization factors.

The status command takes no additional parameters.

Command Description: stop

The "stop" command is a synonym for the "exit" command.

Command Description: translate

The "translate" command is used to generate a realtime diagnostic expert system from the internal representation of the user described system into Ada source code. A correct compiled description must already exist, as must a specified output object file.

The translate command takes no additional parameters.

APPENDIX C: RSP PROTOTYPE ADA SOURCE FILES

rsp.a:
    Main program (procedure rsp)
rsp_arch_pkg_s.a and rsp_arch_pkg_b.a:
    User System Description architecture definitions/resources
rsp_comm_pkg_s.a and rsp_comm_pkg_b.a:
    User command knowledge
rsp_comp_pkg_s.a and rsp_comp_pkg_b.a:
    User System Description recursive descent parsing (components)
rsp_cont_pkg_s.a and rsp_cont_pkg_b.a:
    Control routines (command dispatch)
rsp_defs_pkg_s.a and rsp_defs_pkg_b.a:
    General definitions (constants, types)
rsp_desc_pkg_s.a and rsp_desc_pkg_b.a:
    User System Description recursive descent parsing (component descriptors)
rsp_dpcr_pkg_s.a and rsp_dpcr_pkg_b.a:
    Command processing (dispatch target routines)
rsp_dump_pkg_s.a and rsp_dump_pkg_b.a:
    Diagnostic dump resources
rsp_dupl_pkg_s.a and rsp_dupl_pkg_b.a:
    User System Description substructure duplication
rsp_errs_pkg_s.a and rsp_errs_pkg_b.a:
    User System Description parsing/lexing error reporting
rsp_eval_pkg_s.a and rsp_eval_pkg_b.a:
    User System Description interpretation (expression evaluation)
rsp_exec_pkg_s.a and rsp_exec_pkg_b.a:
    User System Description interpretation (main line system execution)
rsp_expr_pkg_s.a and rsp_expr_pkg_b.a:
    User System Description recursive descent parsing (expressions)
rsp_find_pkg_s.a and rsp_find_pkg_b.a:
    Post-translation User System Description probe/fetch
rsp_form_pkg_s.a and rsp_form_pkg_b.a:
    General formatting resources
rsp_hist_pkg_s.a and rsp_hist_pkg_b.a:
    User System Description history scalar access
rsp_iocp_pkg_s.a and rsp_iocp_pkg_b.a:
    I/O choke point routines and file information (all input and output
    operations)
rsp_lang_pkg_s.a and rsp_lang_pkg_b.a:
    User System Description language syntactics
rsp_lexr_pkg_s.a and rsp_lexr_pkg_b.a:
    User System Description lexigraphical scanner
rsp_list_pkg_s.a and rsp_list_pkg_b.a:
    User System Description listing processing
rsp_optn_pkg_s.a and rsp_optn_pkg_b.a:
    Program option knowledge
rsp_pars_pkg_s.a and rsp_pars_pkg_b.a:
    User System Description recursive descent parsing (main line)
rsp_path_pkg_s.a and rsp_path_pkg_b.a:
    User System Description recursive descent parsing (paths)
rsp_prim_pkg_s.a and rsp_prim_pkg_b.a:
    User System Description recursive descent parsing (primary items)
rsp_rset_pkg_s.a and rsp_rset_pkg_b.a:
    User System Description recursive descent parsing (rulesets)
rsp_rule_pkg_s.a and rsp_rule_pkg_b.a:

```
     User System Description recursive descent parsing (rules)
rsp_sasa_pkg_s.a and rsp_sasa_pkg_b.a:
     User System Description interpreter storage allocation and scalar access
rsp_scop_pkg_s.a and rsp_scop_pkg_b.a:
     User System Description declarative identifier scope processing
rsp_stmt_pkg_s.a and rsp_stmt_pkg_b.a:
     User System Description recursive descent parsing (statements)
rsp_syst_pkg_s.a and rsp_syst_pkg_b.a:
     User System Description recursive descent parsing (systems)
rsp_util_pkg_s.a and rsp_util_pkg_b.a:
     General utilities
rsp_xtrn_pkg_s.a and rsp_xtrn_pkg_b.a:
     User System Description recursive descent parsing (externals)
```

Report Documentation Page

NASA National Aeronautics and Space Administration

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
CR-179441		

4. Title and Subtitle	5. Report Date
Expert Systems for Real-Time Monitoring and Fault Diagnosis	April 1989
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
S.J. Edwards and A.K. Caglayan	H-1540
	10. Work Unit No. RTOP 505-66-71

9. Performing Organization Name and Address	11. Contract or Grant No.
Charles River Analytics Inc. 55 Wheeler Street Cambridge, Massachusetts 02138	NAS2-12725
	13. Type of Report and Period Covered: Contractor Report—Final
12. Sponsoring Agency Name and Address: National Aeronautics and Space Administration, Washington, D.C. 20546	14. Sponsoring Agency Code

15. Supplementary Notes

NASA Technical Monitor: E. Lee Duke, NASA Ames Research Center, Dryden Flight Research Facility, Edwards, California 93523-5000.

16. Abstract

The aim of this study is to investigate methods for building real-time onboard expert systems and to demonstrate the use of expert systems technology in improving the performance of current real-time onboard monitoring and fault diagnosis applications. The potential applications of the proposed research include an expert system environment allowing the integration of expert systems into conventional time-critical application solutions, a grammar for describing the discrete event behavior of monitoring and fault diagnosis systems, and their applications to new real-time hardware fault diagnosis and monitoring systems for aircraft.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Control systems Expert systems Fault detection Fault isolation	Unclassified—Unlimited Subject category 63

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	119	A06

NASA FORM 1626 OCT 86

*For sale by the National Technical Information Service, Springfield, Virginia 22161.