June 1994

UILU-ENG-94-2225
CRHC-94-11

Center for Reliable and High-Performance Computing

A Graphical Environment for Object-Oriented Simulation Model Environment

Daniel Paul Olson

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S) National Aeronautics and Space Administration
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION National Aeronautics and Space Administration Office of Naval Research, & ComputerSciences Corp
6c. ADDRESS (City, State, and ZIP Code) 1308 W. Main Street Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) Hampton, VA Arlington, VA San Diego, CA
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 7a 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 7b	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
A Graphical Environment for Object-Oriented Simulation Model Environment

12. PERSONAL AUTHOR(S)
Daniel Paul Olson

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM ____ TO ____	1994 June 15	41

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis contains an overview of the work that has been done to develop GRIND (a GRaphical INterface for Depend), which is also a design environment and code generator. GRIND's major features are described, and a general tutorial shows how to use the tool to create simulation models. Three models that were constructed using GRIND are presented to illustrate the utility of the tool. Remarks on how the work can be extended are included.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ OTIC USERS	Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

A GRAPHICAL ENVIRONMENT FOR OBJECT-ORIENTED SIMULATION MODEL DEVELOPMENT

BY

DANIEL PAUL OLSON

B.S, University of Illinois, 1993

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1994

Urbana, Illinois

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation to my thesis advisor, Professor Ravi K. Iyer, for all of his assistance and guidance throughout this thesis work. I would also like to thank my peers at the Center for Reliable and High-Performance Computing for their help with an assortment of questions that I posed to them. Additionally, I would like to thank Mark Boyd and Ann Patterson-Hine at NASA Ames Research Center for helping me to understand the SSF-DMS. Finally, I thank my family for all of the support and encouragement they gave me throughout my college experience.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1 Motivation

Today's fault-tolerant computing systems require innovative solutions to dependability problems. To meet these needs, a highly instrumented, simulation-based CAD environment, called DEPEND, has been developed that allows designers to study a system in detail [1, 2]. The CAD tool provides a framework based on C++ that allows the evaluation of highly dependable systems. An important feature of DEPEND is its library of C++ objects which can be used to rapidly model components typically found in fault-tolerant systems. These objects provide extensive, automated fault injection facilities which can simulate realistic fault scenarios. In addition, the objects of the DEPEND Object Library provide several key features that are necessary for fault simulation [3]:

1. They provide ways to signal a change in the status of a component due to a failure, so that remedial actions can be simulated.

2. They have the capability to model the intercomponent dependencies under fault conditions. For example, a failed server may not be able to initiate reintegration without control from a healthy control server. Such dependencies can be easily modeled with DEPEND.

3. They provide several automatic fault statistics collection facilities that provide measures such as MTTF and availability. They can also provide a detailed list of every fault injected, repair action attempted, and their status.

DEPEND's system-level simulation environment differentiates itself from many tools which require prototype systems for fault-injection studies [4, 5, 6, 7, 8]. While the results that come from analyzing prototype systems are indispensable, it is also very important to be able to study an architecture in the early stages of the design in order to evaluate dependability and performance tradeoffs. Another key characteristic of DEPEND is its ability to model the execution of actual application code while realistic faults are injected into the system [9].

GRIND, the subject of this thesis, provides an alternative to coding C++ directly. GRIND is a menu-driven X-Windows application which facilitates the creation of DEPEND models. With this interface, one is able to visualize the architecture of the system being modeled and how it functions. Hardware components are represented using icons while the software aspects of the system are specified using a graphical flow-chart representation. Development of models can also be performed more quickly because GRIND's menu structure presents most of DEPEND's features to the user so that less time is

spent referring to the manual and debugging typos. While a graphical interface provides a speedier and more intuitive way of entering models, much of DEPEND's power can not be harnessed graphically, which means that direct C++ coding must be used to create especially complex models. However, since GRIND's output is a file containing a well-formatted C++ program, one can speed up the creation of a complex model by first using GRIND to create a simpler, more abstract model, and then extending the model by directly editing the C++ code generated by GRIND.

DEPEND users without C++ experience will find that GRIND is essential. The graphical interface has a visual programming environment that is designed such that one can specify the functional behavior of a system without having to know more than the basics C++ syntax. However, GRIND is not designed solely for those who do not know C++. GRIND also caters to the experienced C++ programmers by providing a text editing environment which one can use to efficiently specify the C++ code of functions that are to be used in the DEPEND model. We believe that this tool will be useful in speeding the creation of a model, reducing the number of mistakes made, and helping the user visualize the system being simulated.

1.2 Related Work

Graphical representation can be quite useful in that one is spared the job of writing large amounts of code in an esoteric simulation language; thus, the time spent in the analysis stage is reduced. A number of analysis packages have included graphical

interfaces and have been effective in speeding the process of preliminary analysis of the design.

NEST is a graphical environment for simulation and rapid prototyping of distributed networked systems [10]. A system is graphically represented as a set of nodes interconnected by arcs. Nodes model different points in an interconnection network while the arcs represent the communication channels between them. Actual code (with modification) can be included in the model to accurately simulate processes executing on the nodes. The behavior of the system with server and link failures can also be simulated. NEST's graphical interface is similar to GRIND in that the user constructs a visual image which directly corresponds to the hardware architecture of the system, while the behavioral description is kept separate. However, NEST's range of application is limited to distributed systems, and its graphical environment does not extend to the specification of the system's behavior.

Similar to the DEPEND/GRIND environment, SES Workbench has a graphical interface to a simulation language which is an extension of the C programming language [11]. It differs, though, in that its graphical language is based on transactions which propagate across a directed graph of nodes. SES Workbench's concept of using transactions and directed graphs reflects the fact that its simulation language has its origin in queueing networks. Another example of another graphical tool based on queueing networks is RESQ [12].

A number of other analysis tools using graphical interfaces are based on extensions of stochastic Petri nets. Petri nets, with their idea of places and transitions, also have their own natural way of being represented graphically. UltraSAN is an example of a software package based on stochastic activity networks, a stochastic extension to Petri nets [13].

1.3 Thesis Overview

This thesis contains an overview of the work that has been done to develop GRIND. Chapter 2 describes GRIND's major features and gives a general tutorial of how to use the tool to create simulations models. Three models that were constructed using GRIND are presented in Chapter 3 in order to illustrate the utility of the tool. Chapter 4 concludes the thesis with remarks on how this work can be extended.

2. GRIND—A GRAPHICAL INTERFACE FOR DEPEND

GRIND is a front-end tool which interfaces to DEPEND in a simple way. The graphical interface is used to specify the hardware and software architectures of the system to be modeled and to set the parameters that govern how the simulation is to run. When the user has completed the specification, GRIND then extracts the model from its internal data structures and creates a DEPEND source code file. All that is remaining is to compile the source code and run the executable to obtain the raw simulation results. Different variations of the same model can be tested by making the necessary changes to the model within GRIND, recompiling, and then rerunning.

When creating a functional simulation model, one can view the simulation model as having three aspects: the hardware architecture of the system to be modeled, the functional behavior of the system, and the manner in which the simulation is to execute. The following three sections describe the features of GRIND that address these issues.

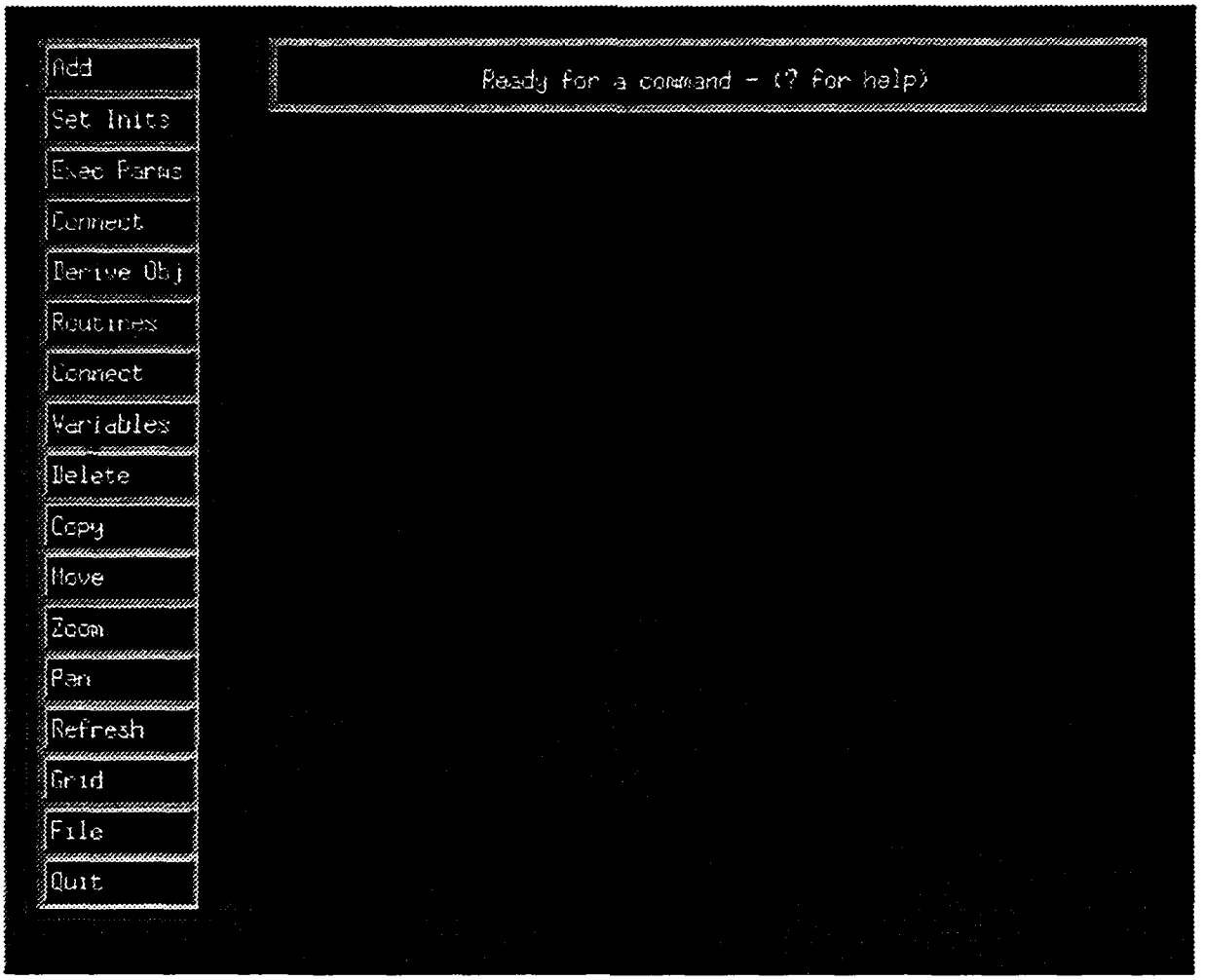

Figure 2.1: The window that the user first sees when invoking GRIND.

2.1 Specifying the Hardware Architecture

Figure 2.1 shows the display which first appears when GRIND is first started. The column of buttons represents the commands which can be invoked from the main window. These commands are invoked by pressing the respective button and are described in the following paragraphs.

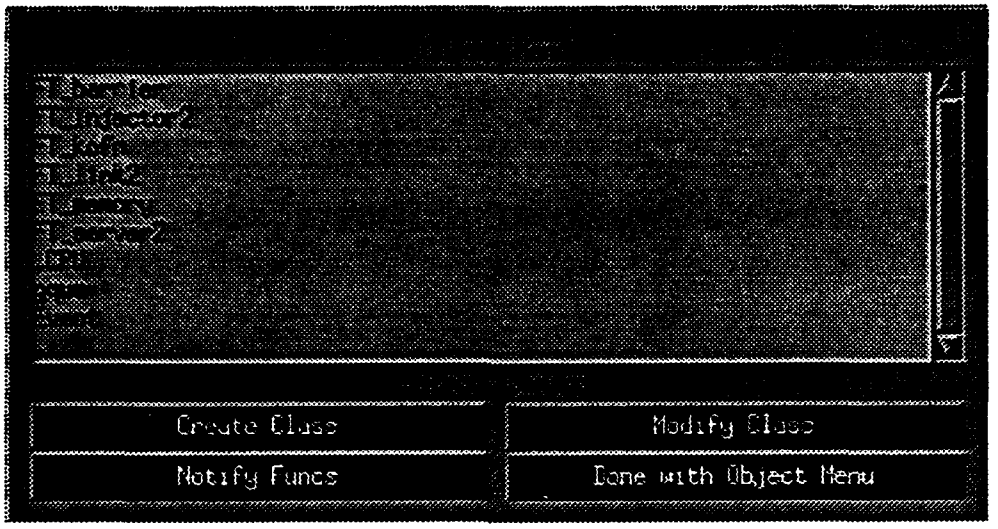

Figure 2.2: The Object Menu. Note that **FT_server2** is the parent class of the CPU object.

When creating a simulation model, the first action that the GRIND user will likely perform is to create derived classes from one of the default objects found in the DEPEND Object Library. A derived object is usually constructed for each of the different types of subcomponents that are going to be included in the model. Derived classes make it possible to add workload or supply additional behavioral description to the default DEPEND objects. Therefore, if the user thinks that it is remotely possible that a workload will be modeled on a particular object or that additional functionality may have to be included in it, it would be best to create a derived class for that object at the start.

Derived classes can be created and modified within the **Object Menu** which is accessed through the **Derive Obj** command button. Figure 2.2 shows the **Object Menu**. A new object can be created by selecting the **Create Class** button, selecting a parent class from the list, and then entering the name of the new class. The user then has the option

of adding component routines and component variables (discussed in Section 2.2) to the new object.

The Add command provides the means to add components to a model. When the user clicks on the Add button, a submenu appears listing each of the objects from the DEPEND Object Library as well as each of the derived objects that have been defined. Selecting one of these objects causes an icon of that object to be tied to the cursor which can be placed anywhere in the schematic by clicking the mouse button. Once an icon is placed in the schematic, the object that it represents becomes a part of the model. It should be noted that though there is only a single listing for each DEPEND object and for each derived class in the Add sub-menu, these objects can be instantiated multiple times to represent different components that have similar characteristics. One should also know that not all of the objects in the DEPEND Object Library are available through GRIND. Table 2.1 lists the DEPEND objects which are supported by GRIND. It does not matter in regard to the simulation model where the icon is placed within the schematic, but the user will want to orient the objects so that they reflect the architecture of the system being modeled. To give the user more control over how the objects are displayed, GRIND has many of the display features common to schematic capture tools such as zoom in/out, pan, display/hide grid, copy, move, and delete.

Once an object is added to the model, it has to be configured such that it accurately models its corresponding component. In the textual DEPEND environment, configuring the objects involves the tedious process of calling many initialization methods for each

Table 2.1: The objects from the DEPEND Object Library which are supported by GRIND.

Name	Description
FT_barrier	Allows an arbitrary number of coroutines to synchronize.
FT_injector2	Automatically injects faults based on statistical distributions. Offers workload based injections. Can inject correlated and latent faults.
FT_kofn	Models a k-out-of-n system. Automatic fault injection. Hot and cold sparing policies. Automatic repair and reconfiguration with specified coverages
FT_link2	Simulates communication channels. Several types of faults: link dead, packet corruption, packet loss, and user-defined faults. Automatic retry.
FT_memory	Simulates a basic random access memory. Can inject permanent and transient faults (bit flips) with latencies.
FT_server2	Simulates a server with spares. Three sparing policies: no spare, graceful degradation, stand-by sparing. Automatic repair and reconfiguration with specified coverage. Automatic injection of faults.
gque	General purpose singly linked queue.
Event	Manages an event. Coroutines can wait on an event or set an event.

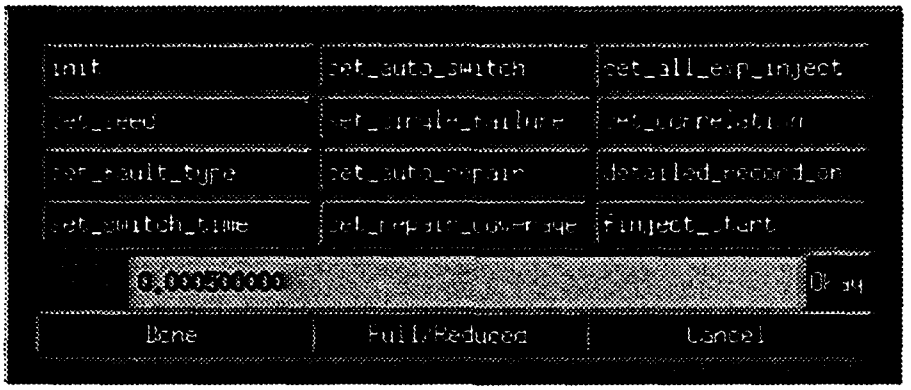

Figure 2.3: **Set Inits** menu for an **FT_server2** object.

of the objects. GRIND speeds the initialization of the objects by providing a menu of parameters for each object. In order to access the menu for a particular component, the user selects the **Set Inits** command and then clicks on the icon for that component. Figure 2.3 shows the **Set Inits** menu for an **FT_server2** object. The parameters shown in the figure are a reduced set of the more commonly used initialization methods for the **FT_server2** object; a full listing of the parameters appears when the **Full/Reduced** pushbutton is selected. Default values are assigned to many of the commonly used parameters to further reduce the work of the user. Parameters typical of many of the objects include the failure distribution, the fault latency, the number of spares, and the sparing policy.

After the components have been added to the model, their interconnections can be defined. GRIND represents a connection from one object to another with a line from an output pin of an icon to an input pin of another icon. A "pin" is simply a point along the border of an icon to which a connection line can be attached. An input pin is denoted by a short bristle while output pins are represented by small circles. Each object has one

input pin and can have multiple output pins. For a visual example, see Figure 3.2 on page 23 showing a GRIND display which contains a number of components connected together. The user sets these interconnections through the `Connect` command.

2.2 Specifying the Behavioral Description

Within the textual DEPEND environment, the behavior of a system's components is defined using C++ functions. These DEPEND C++ functions correspond to "routines" in GRIND. For every routine specified in the GRIND environment, either a C++ function or method is created in the DEPEND output file.

There are four types of routines within GRIND: global, component, notify, and iteration. Global routines are not associated with a particular object so they are therefore not well-suited to simulate workloads running on components. Rather, they are usually used to start up and maintain a simulation by performing such tasks as initializing or setting global variables or initiating workloads. The menu for adding or editing global routines is accessed through the `Routines` command on the main GRIND window.

In contrast to global routines, component routines are always associated with derived classes. They correspond to methods (functions associated with classes) in the C++ language. Since a component routine is tightly coupled with an object, it is usually used to model a workload running on the object that it is associated with. The `Object Menu` allows the user to add component routines as well as variables to the derived classes.

GRIND allows one to define a number of different *notify routines* for derived classes. A notify routine is a special type of component routine which is called when a specific event (such as a fault injection or an automatic repair) occurs. For instance, if a derived class has its `fault` notify routine defined, whenever an object of that class has a fault injected, the `fault` notify routine is called for that object. Notify routines are defined by the user through the `Object Menu`. Iteration routines are special types of global routines and are described in Section 2.3.

There are two ways to define methods/routines within the GRIND environment: textually and graphically. Whenever it is time to define a routine, GRIND prompts the user as to which mode of entry is desired. Those who are comfortable with C++ syntax but still enjoy GRIND's framework for constructing models may desire to use a standard text editor and type in the actual code for that routine. Alternatively, the user may desire GRIND's visual programming environment for entering routines. The goal of these features is to allow the specification of a wide range of behaviors with the least amount of work.

GRIND's visual programming environment allows a user who has only basic programming skills and a limited knowledge of C++ to create a routine. Every routine that is visually created is represented by a "flow chart" which consists of nodes and arrows. Flow charts are analogous to those that programmers often draw before coding a program. Each flow chart has its own window containing a command panel and a display similar to the main window. The nodes of a flow chart correspond to C++ statements

Table 2.2: Description of each of the node types that can be used within GRIND's visual programming environment.

Node	Type	User Action
For	control	Specify index variable. Specify starting value for variable. Specify ending value for variable.
While	control	Specify the condition for looping.
If	control	Specify the condition for branching.
Done	control	(No action necessary. This node causes the simulation to halt. Its use is discussed Section 2.3.)
Assignment to Variable	action	Select global variable from menu. Specify expression to be assigned to variable
Global Routine	action	From menu, select global routine that is to be called. Within the menu, specify arguments for the routine.
Component Routine	action	Select component whose method is to be called. Select method and specify arguments from menu.
Local Comp Routine	action	Select method and specify arguments from menu. (Not available when defining global routines.)
Message	action	Specify message that is to be sent to output.
Custom	action	Type C++ statement.

while the arrows define the order in which the nodes will be executed. GRIND divides the different types of nodes into two categories: control and action. Control nodes control the flow of execution through the flow chart by providing the capabilities of branching and looping. Action nodes cause actions to be taken such as assigning values to variables, invoking routines, and sending messages to the display. Nodes are placed and connected in a manner similar to the way components are placed and connected in the main GRIND display. Table 2.2 contains a listing of the types of control and action nodes, and Figure 2.4 shows GRIND's visual programming environment with a sample flow chart. The "User Actions" entry within the table for each node lists the actions that have to be

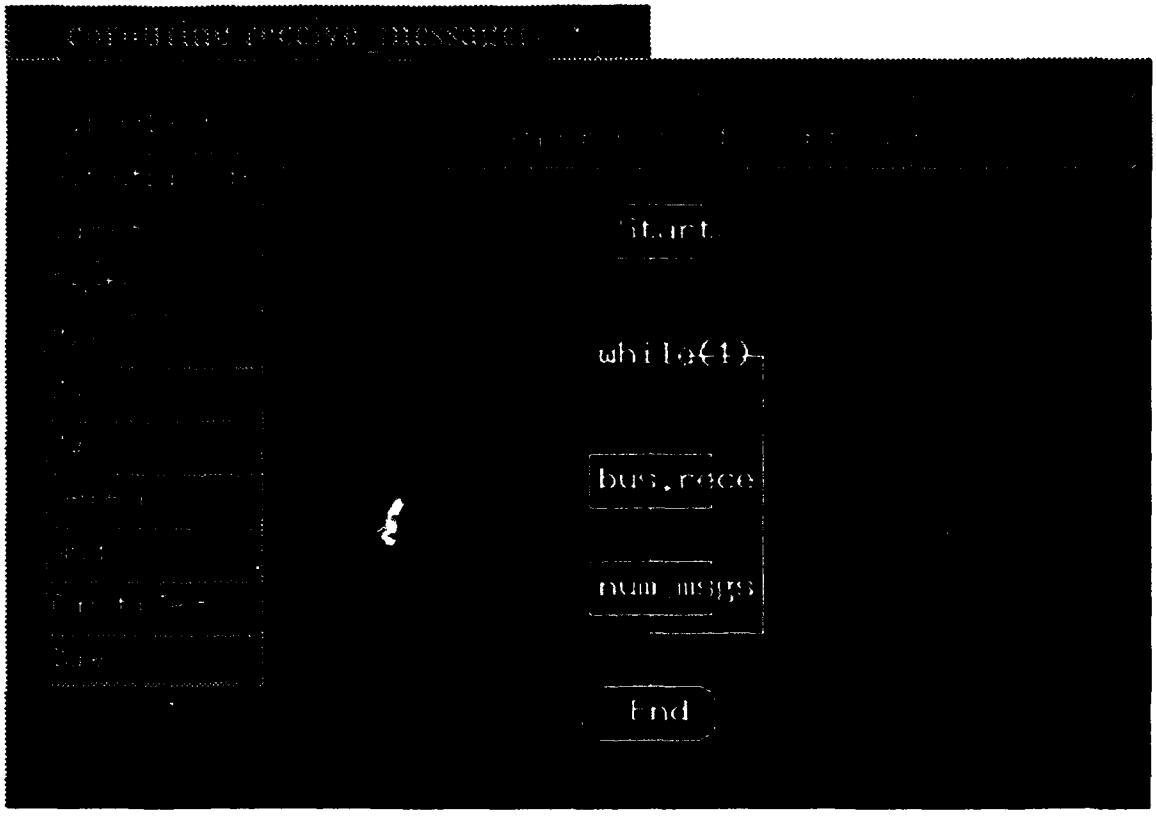

Figure 2.4: The graphical specification of a routine called **receive_msg**. This routine uses a **while** control node to implement an infinite loop, a **Component Routine** action node to simulate a message being received from a bus, and an **Assign to Variable** action node to update a global variable called **num_msgs**.

performed by the user in order to specify the contents of that node. These actions are performed after the node's icon has been placed within the window.

2.3 Controlling the Simulation and Maintaining Statistics

GRIND has a number of features to help the user set up the simulation. In order to obtain statistically valid results, a user will often want to run many simulations for the same model with a different random number generator seed for each simulation. For this reason, GRIND outputs C++ code which loops a user-defined number of times, with a simulation run executing once for every iteration of the loop.

To control the various simulation runs, a number of parameters are available for the user to define through the *Exec Parms* menu. The first parameter is the number of iterations of the simulation that are to be executed. The user will want to make this number large enough so that the simulation results will be statistically valid.

The next parameter in the **Exec Parms** menu that has to be set is the *termination condition*. The termination condition parameter determines when a simulation run is to be terminated. There are four possible values for the termination condition: `time`, `fail`, `time/fail`, and `done`. If `time` is used, the simulation will finish after a user-defined amount of simulated time. This time can also be set within the **Exec Parms** menu. The `fail` value is used when the user wants the simulation to terminate when the modeled system fails. `Time/fail` is a combination of `time` and `fail`. Whichever events occurs first will cause the current simulation iteration to terminate. If the *termination condition*

is set to **done**, a simulation run will not terminate until a **done** node is encountered in one of the routines. The sole purpose of including a **done** node in a routine is to signal that a simulation iteration is to be halted. It should be noted that if a **done** node is encountered when the termination condition is set to **time**, **fail**, or **time/fail**, the current iteration will be stopped even if the system has not yet failed and the specified amount of time has not yet elapsed.

When the termination condition is set to either **fail** or **time/fail**, the user defines what it means for a system to fail by specifying what set of components must be alive in order for the system to be alive. This is done through the **Edit System Dependencies** command within the **Exec Parms** menu. The combination of components is specified in a logical AND-OR expression. For instance, a system containing five components, V, W, X, Y, and Z, can be defined as being functional as long as [V AND W] OR [X AND Y] OR [Z AND X AND V] are alive.

In order to give the user control over each simulation run and provide a means to collect and reset statistics, GRIND provides four *iteration routines*: pre-iteration routine, iteration start routine, iteration end routine, and post-iteration routine. The *pre-iteration* routine is called only once before any of the simulation runs begin to execute. It can be used to initialize variables which will be accumulated throughout all of the simulation iterations. The *iteration start* routine is called at the beginning of each simulation run. Workloads and other processes that are to run throughout a simulation are generally initiated by the iteration start routine. This routine might also initialize variables which are

used to collect statistics for a single run. The *iteration end* routine is invoked every time a simulation is terminated. The results of a simulation are often output by this routine. Also, if faults are being injected during the simulation, the user will want to repair each component whose status might be "faulty." The final iteration routine, *post-iteration*, is called after the last iteration end routine has completed. The main use of this routine is to output statistics which were gathered throughout all of the simulation iterations. Each of these iteration routines can be defined through the `Routines` command in GRIND's main window.

3. EXAMPLE APPLICATIONS OF GRIND

This chapter presents three models that were constructed using GRIND. An analysis was performed using each of these models, and the results of the analysis are also presented. These examples are included in order to demonstrate GRIND's range of application.

The first model is an example of a first-order static analysis in which each subcomponent of a computing system is given a failure rate and faults are injected until the system as a whole fails. GRIND constructs the output code such that many simulations can be executed sequentially so that a distribution of times to failure can be obtained. Such an analysis can be performed on virtually any system using GRIND.

The second analysis uses an extended version of the first model in order to perform a memory failure latency analysis. A workload is simulated which performs reads and writes to memory while transient failures are injected to random memory locations. Rough utilization figures are also obtained for each of the computing elements.

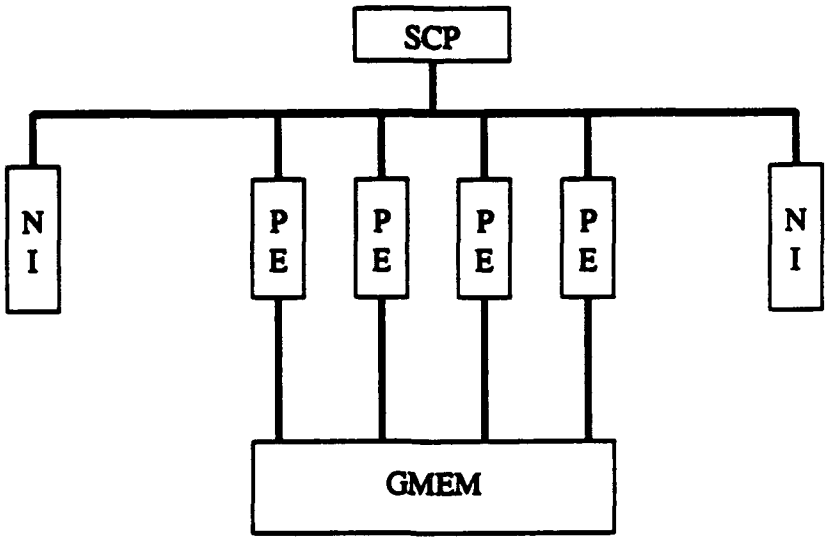

Figure 3.1: Common Integrated Processor (CIP) block diagram.

The final example is a slightly more complex model of a different computing system. Like the second model, the third includes the concepts of workload and transient failures, but this model differs in that it simulates and measures a limited amount of fault propagation.

3.1 First-Order Static Analysis of an Avionics System

3.1.1 The Common Integrated Processor (CIP)

The two models presented in this section and in Section 3.2 are based on the Common Integrated Processor (CIP), a processing module found in one of Hughes' avionic computer systems. The system consists of four processing elements (PEs), two redundant network interfaces (NIs), a global memory (GMEM), and a standard control processor (SCP). See Figure 3.1 for a block diagram of the CIP. Either NI can be used by any of the PEs to send or receive data from the outside world. The SCP is responsible for

distributing tasks among the four PEs which communicate with each other using the GMEM.

As far as we could tell from its documentation, a CIP module does not by itself have any hardware fault-tolerant features. For the purpose of illustrating GRIND's capabilities, we will consider a version of the CIP module that has a limited amount of redundancy. We will assume that the reason for having multiple PEs is for fault-tolerance considerations as well as for increasing computing power. Let's assume that three of the four processors are required to maintain the minimum throughput requirements. Also, since the two NIs each connect the four processors to the outside world, we will assume that one network interface has enough bandwidth so that if one fails, the system as a whole can continue to operate.

3.1.2 The model

This analysis focuses mainly on the system's four processing elements. Since tasks are likely to be running on a PE when it fails, the process of reconfiguring to use only three PEs is likely to be complex and itself prone to failure. Thus, the model includes a reconfiguration coverage for the processing elements. Given the failure rates of each of the subcomponents, an interesting analysis would be to see how sensitive the reliability of the module as a whole is to this reconfiguration coverage. This would give engineers an idea of how much effort has to be invested in designing a robust reconfiguration process.

Constructing this model using GRIND was a straightforward process. The first step was to create a derived class for each of the different types of subcomponents in the model. Once a derived class is created, one can create variables and methods for that class in addition to those inherited from the parent class. In this example model, a PE class was derived from the FT_kofn object. Because FT_kofn is the parent of PE, PE inherits all of FT_kofn's functionality, making it able to model the processing-element 3-out-of-4 system. Similarly, a GMEM class was derived from the FT_memory object, a SCP class from the FT_server2 object, and an NI class from the FT_server2 object. Since no workload (involving such activities as processor utilization, message passing, and memory access) is incorporated into this model, there was no need to further specialize the derived classes.

Once the derived classes were established and objects from these classes were added to the model, the initialization methods of these objects were set through the *Set Inits* menu. Through this menu, the fault injection rates as well as some of the other configuration parameters were set. See Figure 3.2 for a GRIND display showing the objects of the model and a listing of the initialization methods of pe_, an object of the class PE. The subwindow shown in Figure 3.2 is the *Set Inits* menu for the pe_ object, through which we specified it to be a 3-out-of-4 system with a 66.7% reconfiguration (*switch*) coverage. The failure rates for each of the objects (listed in Table 3.1) were also set through this menu.

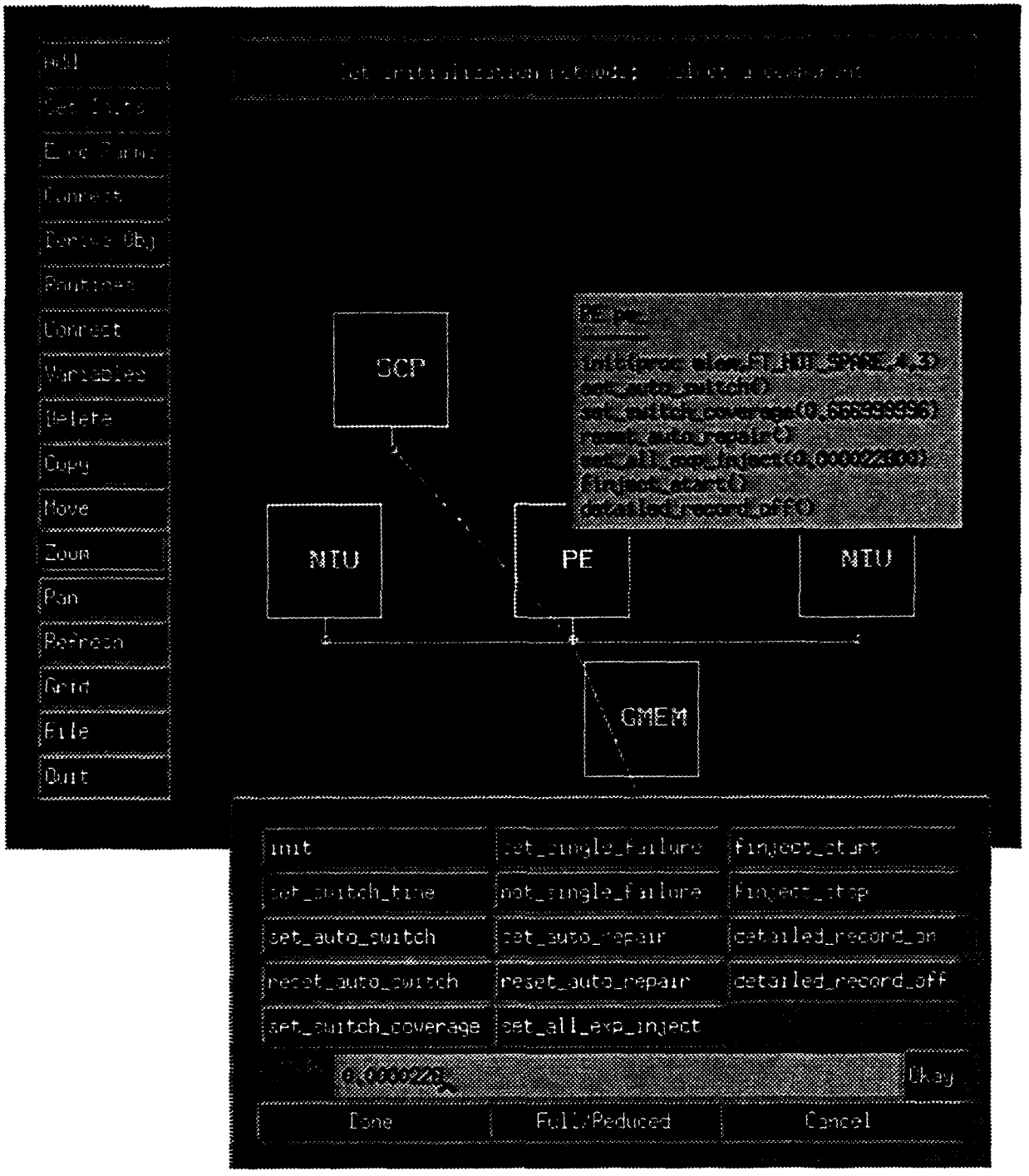

Figure 3.2: GRIND hardware display showing the *Set Inits* menu for the processing elements.

Table 3.1: List of assumed failure rates.

Subcomponent	MTTF	Failure Rate
SCP	5 years	2.28×10^{-5} failures/hr
PE	5 years	2.28×10^{-5} failures/hr
GMEM	3 years	3.80×10^{-5} failures/hr
NI	8 years	1.43×10^{-5} failures/hr

In this model, we defined the `fault` notify routine for each of the different derived class types (`SCP`, `PE`, `NI`, and `GMEM`) such that the name of a component is sent to the standard output when that component has a fault injected. This will allow us to know which component is responsible for each failure and will thus help us to ascertain which components are reliability bottlenecks.

Through the *Exec Parms* menu, we specified that the simulation was to run ten-thousand times, with each simulation terminating when the system as a whole fails. The *Exec Parms* menu also allowed us to specify what combination of components had to be alive in order for the system as a whole to be alive. We specified this combination such that two failures can potentially be tolerated: one NI failure and one PE failure if the processing-element reconfiguration is successful.

With the above information, GRIND was able to create a C++ program which was compiled and then run. The output of the executable is simply a listing of ten-thousand times to failure along with the component failures that precipitated each system failure. With the aid of a standard statistical analysis package, a curve estimating the reliability of the system was easily constructed.

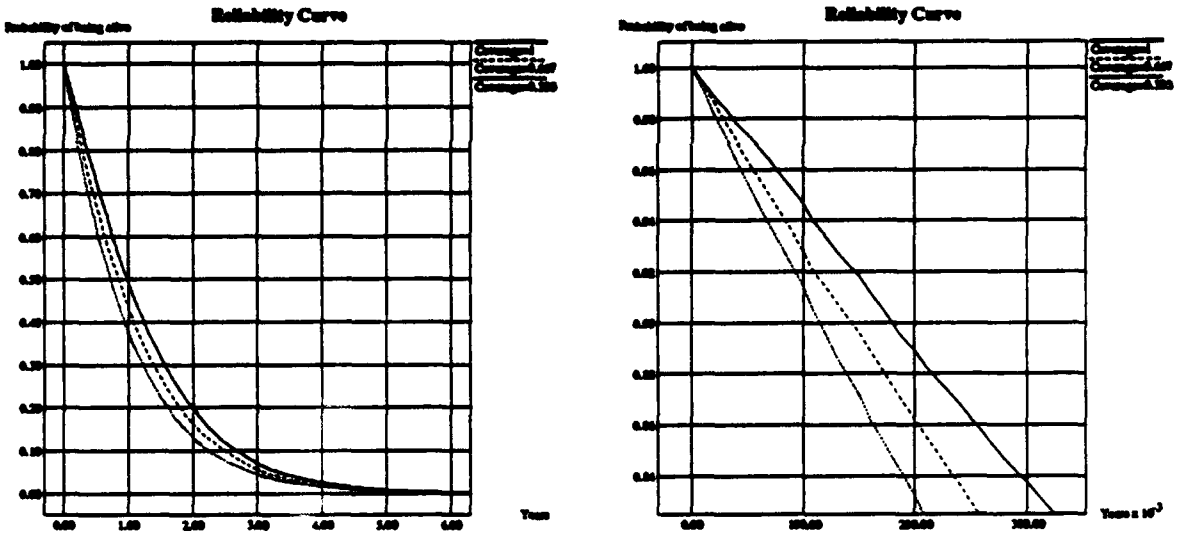

Figure 3.3: Reliability curves for CIP system. The graph on the right is an enlarged version of the one on the left.

Table 3.2: List of reliability statistics. MTTF is the mean time to failure. The figures for the 90 and 95 percent confidences represent the length of time that one can be 90 and 95 percent certain that the system will stay alive.

	Cvrg=1.000	Cvrg=0.667	Cvrg=0.333
MTTF	1.25 yr	1.12 yr	1.00 yr
90% conf.	1600 hr	1200 hr	1000 hr
95% conf.	830 hr	610 hr	480 hr

Three simulation executables were created with reconfiguration coverage values set to 1.00, 0.667, and 0.333. Awk and SAS were used to process the raw output from the models. The reliability curves are shown in Figure 3.3 and some reliability statistics are given in Table 3.2. Judging by the full reliability curves and the MTTF values, it does not appear that the reconfiguration coverage has a great effect on the reliability of the system (a 25% decrease in MTTF from coverage equalling 1.000 to a coverage equalling 0.333). But since this system is primarily geared for avionics applications in which mission times are measured in hours, it is more important to view the reliability in the short run. For

Table 3.3: Percent of system crashes attributed to the respective subcomponents.

Subcomponent	Cvrg=1.000	Cvrg=0.667	Cvrg=0.333
SCP	25.3%	22.3%	20.0%
PE	30.0%	37.6%	44.6%
GMEM	41.6%	37.1%	33.3%
NI	3.2%	2.9%	2.1%

this reason, a zoomed-in version of the reliability graphs and durations for the 95th and 90th percentiles is also given. From these, we can see that as the reconfiguration coverage decreases from 1.000 to 0.333, there is a 42% decrease in the length of time that we can be 95% certain that the system will stay alive.

Because the simulation executable records each component failure, we were able to determine which component caused each crash. Table 3.3 breaks down the percent of system failures that each subcomponent was responsible for. From this table, we can see that the processing elements become the main reliability bottleneck of the system (increasing its share of system failures by almost 50%) as the coverage decreases to 0.333.

3.2 Model for Memory-Failure Latency Analysis

The second model of the Common Integrated Processor simulates a mock workload running on the system in order to determine the latency of memory faults and to gain rough estimates of the utilization of the processing elements and the standard control processor. The latency of a memory fault is defined here as the time from the corruption of a memory location to the reading of that location. If the corrupted cell is written over,

nothing is recorded except that the error was avoided. Utilization is the percent of time that a resource (e.g., a SCP or a PE) is being used. The utilization figures calculated for the PEs are the average utilization figures for each of the four PEs. Therefore, it would take four continuous jobs running simultaneously to attain a PE utilization of 1.00.

3.2.1 The software model

Two component routines were added to simulate the workload: `dispatch_tasks` was added to the SCP object and `task` was added to the PE object. These methods pick random values from normal and exponential distributions in order to model the software's functionality. It should be noted that the latency results presented here are most likely not representative of the dependability of the CIP since the analysis is highly dependent on the patterns in which the software accesses memory. The analysis will have value only if the behavior of the software and the rate at which the jobs arrive are accurately modeled.

`Dispatch_tasks` models a process running on the SCP which repeatedly issues jobs to the processing elements. The amount of time between task dispatches is picked from an exponential distribution with a mean of 0.05 second and the amount of processing time required for each dispatch is picked from a normal distribution with a mean of 0.01 second and a standard deviation of 0.0075 second. Each task is sent to the PE object as a whole, which, in turn, decides how the job is to be distributed to the four processors. This decision is handled automatically by the `PE object` (a feature which it inherited from

the FT_kofn object) and is done in the most intelligent way. Therefore, this simulation assumes that the SCP is able to determine which is the best processing element to send a task. This assumption does not affect the utilization figures that are given here, but does perhaps decrease the latency of the memory faults since the response time of each task is minimized.

Each job initiated by dispatch_tasks to run on a processing element is modeled by the task method. Memory locations within GMEM are accessed sequentially by task. Dispatch_tasks gives task the starting address and the number of memory locations that are to be read or written to, each of which is picked at random from the ranges [0,300] and [50,200], respectively. Dispatch_tasks also decides randomly whether task is going to read or write to those locations. The amount of processing that task performs for each access is selected from an exponential distribution with a mean of 0.001 second. Figure 3.4 shows the graphical representation used to specify the task method. From this figure, one can see GRIND's flow-chart style of graphically specifying routines.

As was mentioned earlier, the object used to model the global memory, GMEM, is a derived class of the FT_memory object. FT_memory allows one to model large memory spaces by maintaining a sparse data structure of memory locations. With each memory location, FT_memory keeps a status flag telling whether that location is "uninitialized," "valid," or "faulty." Since this model's representation of the software does not deal with actual data, only these status flags are accessed by the task method. If task performs a read on a memory location whose flag indicates that the cell is faulty, the time at which

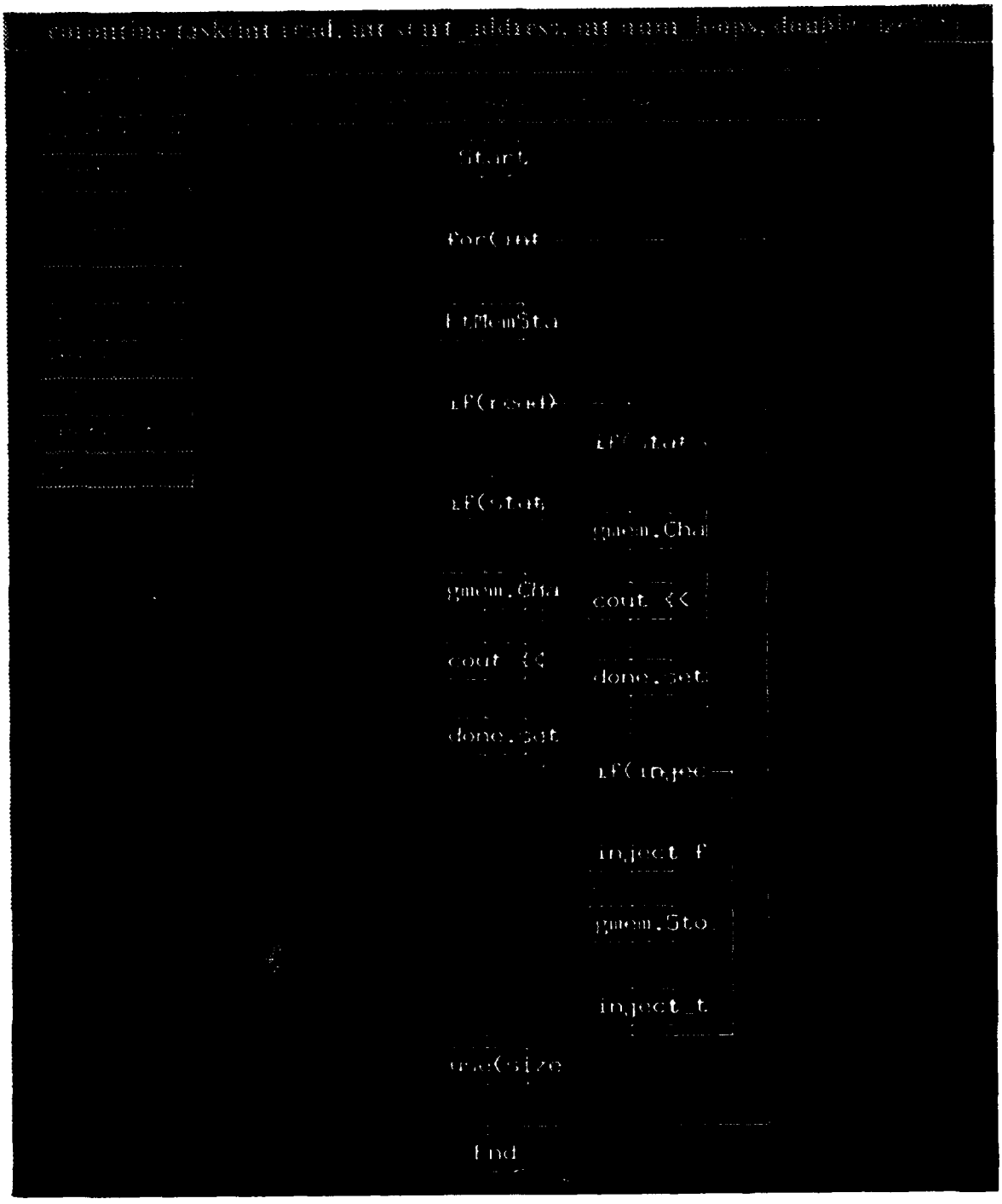

Figure 3.4: The graphical representation of the task method.

Table 3.4: Statistics from the simulations using the two workload intensity levels. Note that the *% Masked* figures refer to the percent of errors which were avoided.

Workload level	Mean Latency	% Masked	Ave SCP Util.	Ave PE Util.
1 Process	0.336 sec	45.6%	0.190	0.447
3 Processes	0.131 sec	47.2%	0.509	0.736

the access occurred is recorded and that simulation is halted. If a write is performed on a faulty location, it is noted that an error was avoided and the simulation is stopped. A global routine which waits for a random length of time and sets the status flag of a random memory location to "faulty" was defined in order to inject failures into the memory.

3.2.2 Simulation results

One-thousand memory-failure latencies and utilization samples were obtained for two workload intensity levels. The first intensity level had only one of the `dispatch_tasks` processes continuously issuing tasks to the processing elements. The second simulation ran three `dispatch_tasks` processes, each initiating jobs.

Table 3.4 lists the averages of the results gained from the simulations. As one would expect, the results indicate an inverse relation between the intensity of the workload and the latency of memory faults. Since more tasks are being issued to the processing elements, more reads are occurring in a given period of time; thus, the faulty memory locations are encountered sooner. Figure 3.5 gives the distributions of memory-failure latencies for the two scenarios. From the figure, one can see that the distributions are

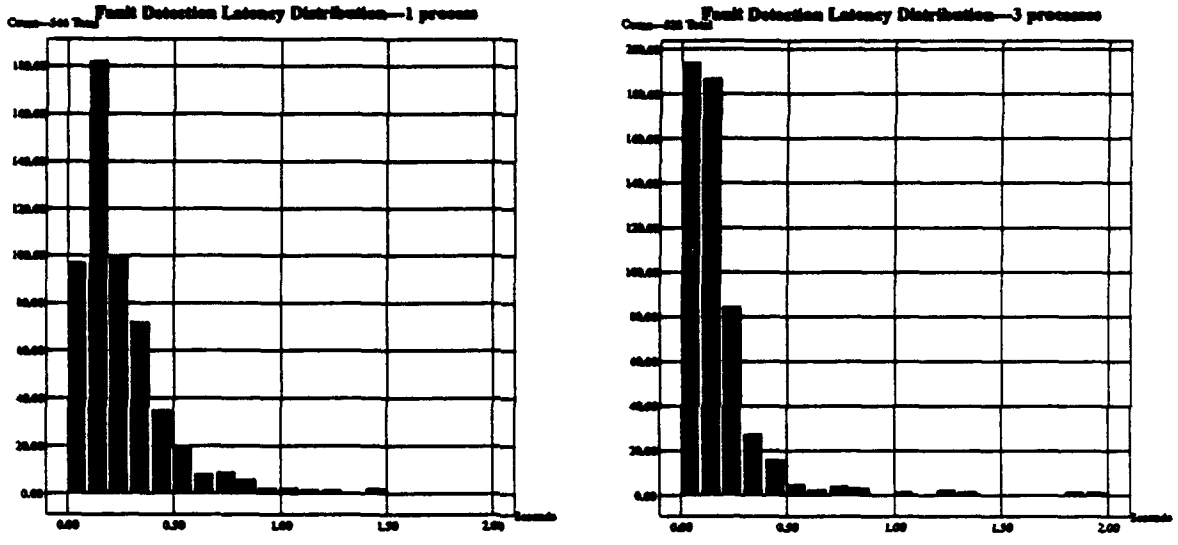

Figure 3.5: Distributions of memory error latencies for a system executing a single `dispatch_task` process and for another system running three `dispatch_task` processes.

fairly tight, with few memory faults remaining latent for more that a second for the one-process scenario and more than a 0.5 second for the three-process scenario.

On the other hand, there appears to be a fairly large variance in the utilization of the processing elements. For every memory fault that was injected, a sample was taken. Figure 3.6 gives the distributions of these utilization samples for the two levels of workload intensity. Significant levels of utilization were found in the range between 0.1 and 0.8 for the one-process level and between 0.4 and 1.0 for the three-process workload intensity level.

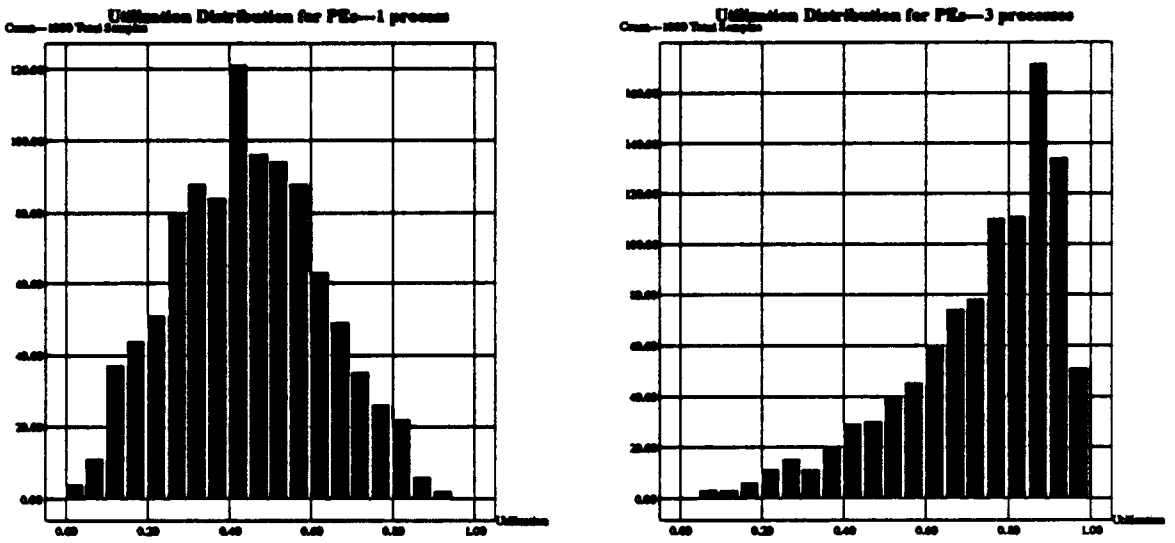

Figure 3.6: Distributions of processing-element utilization samples for a system running one `dispatch_task` process and for another system running three `dispatch_task` processes.

3.3 Transient Failure Analysis of the SSF-DMS

3.3.1 The architecture of the SSF-DMS

The Data Management System (DMS) of the Space Station Freedom (SSF) is designed to be a fault tolerant distributed computing system. Figure 3.7 shows a simplified overview of the DMS hardware architecture.[1] The DMS will be responsible for providing the computing resources required to support a number of functions that are crucial to the operation of the station. The design is centered around a set of general purpose computers called Standard Data Processors (SDPs) which are interconnected by an FDDI communication network. Also connected to the FDDI are a telemetry unit and a disk. The SDPs are connected by MIL-STD-1553B local buses to Intel 80386-based peripheral

[1]The description here is loosely based on the DMS proposed architecture as it existed prior to the SSF redesign announced in the spring of 1993.

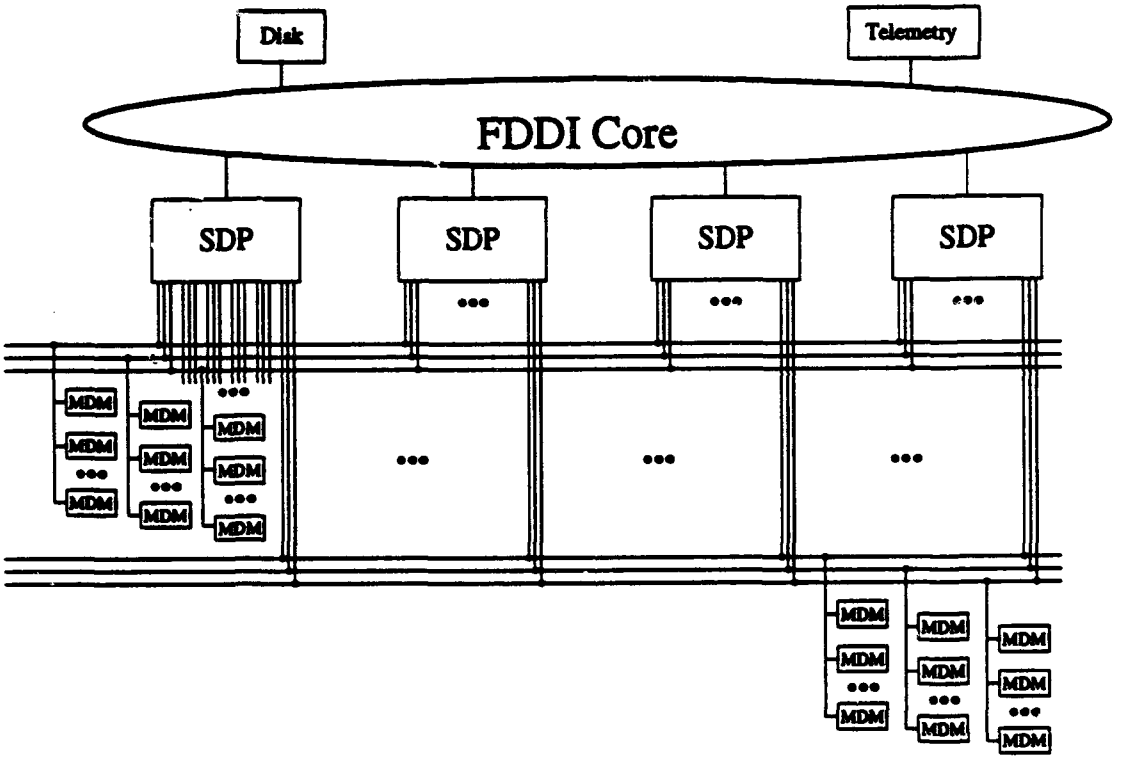

Figure 3.7: Overview of SSF-DMS Hardware Architecture.

processors called Multiplexor-Demultiplexors (MDMs), which are, in turn, connected to sensors and effectors. The MDMs are responsible for collecting data from banks of sensors and routing them to appropriate application programs executing in the SDPs. Each SDP is configured as a bus controller for a subset of the 1553B buses that are used to send requests to MDMs to obtain sensor reading values.

3.3.2 The model

In this analysis, we will be modeling transient failures in the MDMs and measuring the latency and propagation of these failures to disk or telemetry. The latency of an MDM failure is defined here as the time from which the MDM is injected with a failure until the time that MDM is detected to have an invalid sensor value. A mock workload will be simulated to run on a single SDP connected to five MDMs. This workload will continually fetch sensor values from the MDMs, process these values, and send the data to either the disk or the telemetry unit. During the time that a transient failure is active in an MDM, it is assumed that the MDM will be able to return a sensor value, but this value will be invalid. The SDP has a means of checking to see if a sensor value is out-of-bounds, but depending on the state of the workload, a given sensor value fetched from an MDM may or may not be checked. Since an invalid sensor value may still be within a reasonable range, it might not be detected as being out-of-bounds. Each MDM has a specfied probability that (when it fails) its invalid sensor values will be detected (out-of-bounds) if checked.

Table 3.5: Probabilities that a checked invalid sensor value will be detected as out-of-bounds.

MDM #	Probability
0	0.45
1	0.50
2	0.30
3	0.20
4	0.20

As with the previous example, the workload modeled here uses exponential and uniform distributions to determine its pattern of execution since an actual algorithm is not available. The amount of time between sensor-value fetches is picked from an exponential distribution with a mean of 0.02 second and the MDM that is accessed for a given sensor value is picked from a uniform distribution. After a sensor value has been fetched, it has a 30% chance of being stored to disk. If it is not stored to disk, it then has a 30% chance of being sent to the telemetry unit. Table 3.5 gives the probabilities that a checked invalid sensor value will be detected as out-of-bounds.

An `SDP` class was derived from an `FT_server2` object and was used to represent the SDP module. A component routine called `gather` was added to this class to model the workload running on the SDP which continuously gathers sensor values and sends them to the disk or the telemetry unit.

Each MDM is modeled using an `MDM` object which was also derived from `FT_server2`. The `get_sensor_value` component routine returns a sensor value to its caller. This routine uses the `cond_ok` component routine which the MDM inherited from `FT_server2` to check so see if the value it returns should be invalid. Since actual data are not used

within this model, the actual sensor values in this simulation are only flags indicating whether the values are valid or invalid. `Get_sensor_value` is called directly by `gather` to obtain a sensor value from a particular MDM (an `FT_link2` is not used for communication between the SDP and the MDMs since bus failures and contention for the bus are not issues in this simulation).

A `DISK` class was derived from the `FT_memory` object and is used to model the system's disk. Added to it is the `store` component routine which takes a sensor value as a parameter. `Store` simply checks the sensor value that was passed and records a "disk error" if that value is invalid. Similarly, a `TELEM` class was derived from the `FT_server2` object in order to model the telemetry unit. It has a `send_data` component routine which takes a sensor value as a parameter. If the sensor value passed to this routine is invalid, a "telemetry error" is recorded.

3.3.3 The results

Two versions of the model described above were evaluated: one using a 25% probability that a checked invalid sensor value will be detected as out-of-bounds and another using a 75% probability. One-thousand simulation iterations were executed for each scenario. Each iteration simulated the system for ten years, with transient failures injected into each MDM using a rate of once per ten years (exponential distribution). The number of disk errors and the number of telemetry errors were recorded for each mission, and the latency of an MDM failure was recorded every time a sensor value was detected

Table 3.6: Statistics from the simulations using varying levels of out-of-bounds checking for sensor values.

% of sensor values checked	25%	75%
number of missions simulated	1000	1000
length of each mission	10 yrs	10 yrs
average # of disk errors per mission	5.05	2.66
average # of telem. errors per mission	3.38	1.84
average latency	0.335 sec	0.227 sec
% of faults detected	28.5%	54.0%

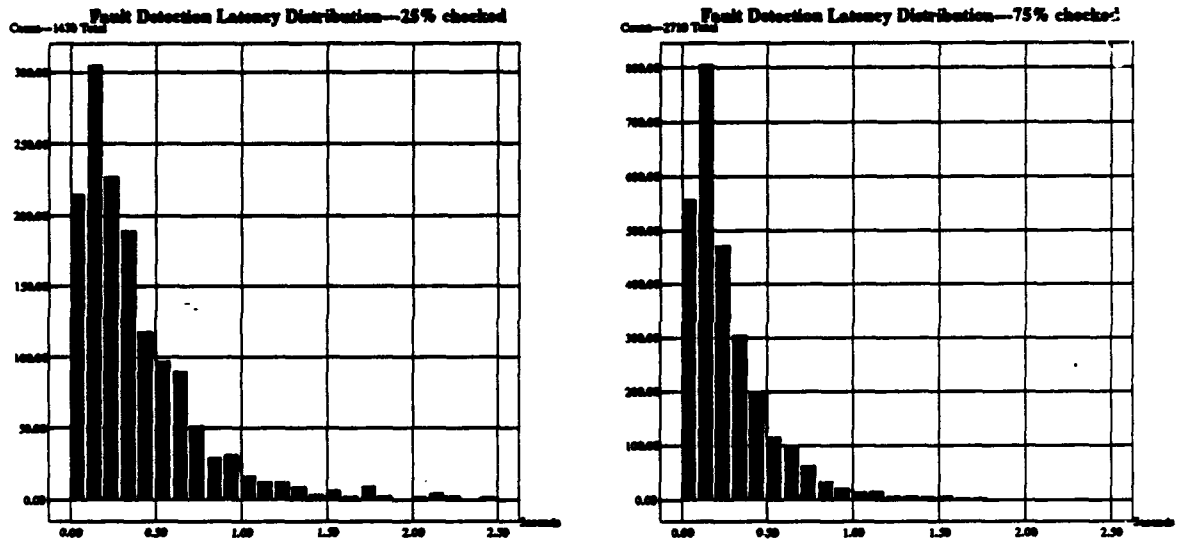

Figure 3.8: Transient sensor failure distributions.

as out-of-bounds. Transient failures whose durations expired before they were detected were also recorded.

Table 3.6 gives the average number of times that errors propagated to the disk and the telemetry unit, the average latency of MDM failures, and the percent of transient failures that passed undetected. Figure 3.8 shows the distribution of MDM error latencies. For the given workload, these results quantify how much the system's dependability improves when the percent of sensor values checked increases from 25% to 75%.

3.4 Summary of Example Applications

This chapter gave a brief overview of the DEPEND/GRIND environment and presented three models constructed using GRIND. It has been shown that GRIND can be used to capture aspects of both the system's hardware and software architectures. Though the results presented here have little value since they are based on fictitious parameters, accurate failure rates and a more precise description of the software can be included to obtain valid simulation results.

4. CONCLUSIONS

This document has motivated the need for GRIND, described its major features, and presented example applications of the tool. The most prominent contribution of this work has been the development of techniques to graphically specify fault-injection simulation models in an object-oriented environment. In particular, GRIND's flow-chart method of defining a system's behavior with menus to aid the user in specifying the contents of the flow chart's nodes is not included in any other graphical packages known to the author.

REFERENCES

[1] K. K. Goswami and R. K. Iyer, "DEPEND: A simulation-based environment for system level dependability analysis," Tech. Rep. CRHC Report #92-11, CRHC – University of Illinois at Urbana-Champaign, June 1992.

[2] K. K. Goswami and R. K. Iyer, "A design environment for prediction and evaluation of system dependability," in *Proceedings of the Ninth Digital Avionics Systems Conference*, October 1990.

[3] K. K. Goswami and R. K. Iyer, *The DEPEND Reference Manual.* University of Illinois – Center for Reliable and High Performance Computing, Urbana, Illinois 61801, Oct. 1990.

[4] J. Lala, "Fault detection isolation and reconfiguration in ftmp: Methods and experimental results," in *5th AIAA/IEEE Digital Avionics Systems Conference*, pp. 21.3.1–21.3.9, 1983.

[5] K. G. Shin and Y. Lee, "Error detection process - model, design, and its impact on computer performance," *IEEE Transactions on Computers*, vol. C-33, pp. 529–540, Jun. 1984.

[6] L. Young, R. K. Iyer, K. K. Goswami, and C. Alonso, "A hybrid monitor assisted fault injection environment," in *Third IFIP Conference on Dependable Computing for Critical Applications*, Sep. 1992.

[7] Z. Segall. et al., "FIAT - fault injection based automated testing environment," in *Proceedings of the 18th International Symposium on Fault-Tolerant Computing*, pp. 102–107, Jun. 1988.

[8] G. Kanawati, N. Kanawati, and J. Abraham, "FERRARI: A fault and error automatic real-time injector," in *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, Jul. 1992.

[9] K. K. Goswami and R. K. Iyer, "A simulation-based study of a triple modular redundant system using DEPEND," in *5th International Tests, Diagnosis, Fault Treatment Conference*, Sept. 1991.

[10] A. Dupuy, J. Schwartz, Y. Yemini, and D. Bacon, "NEST: A network simulation and prototyping testbed," *Communications of the ACM*, vol. 33, pp. 64–74, Oct. 1990.

[11] SES, Inc., Austin, TX, *SES/Sim Simulation Language Reference Manual*, Mar. 1989.

[12] C. Sauer, E. MacNair, and J. Kurose, "Resq: Cms user's guide," Tech. Rep. RA-139, IBM T.J. Watson Research Center, Yorktown Heights, NY, Apr. 1982.

[13] W. H. Sanders, W. D. Obal, M. A. Qureshi, and F. K. Widjanarko, "*UltraSAN* version 2: Architecture, features, and implementation," Tech. Rep. PMRL-93-17, Department of Electrical and Computer Engineering, University of Arizona, Oct. 1993.