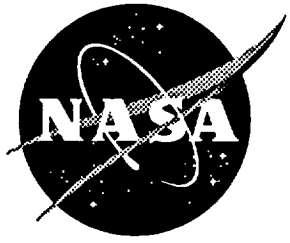
NASA Contractor Report 187588



Hardware and Software Reliability Estimation Using Simulations

Frederic L. Swern
Stevens Institute of Technology, Hoboken, New Jersey

Grant NAG1-587

February 1994

National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virginia 23681-0001

CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

1. INTRODUCTION

1.1 The Flight Control Scenario

New developments in aviation technology will significantly improve an aircraft's performance while reducing its operating cost. Many of the new techniques deteriorate aircraft stability; to make it flyable by the average pilot, computer based flight control systems are used to modify its handling characteristics. A flight control system mates well with the fly by wire concept, the latter clearing the aircraft of a kluge of cables and pulleys that previously activated the control surfaces. Some examples of the new technology are the X29 forward swept wing fighter, an unstable but highly maneuverable aircraft, and the Boeing 7J7 aircraft, to fly in commercial service in the next decade.

It is typical for a flight control system to have embedded processors performing the real-time control applications. A typical configuration contains an input processing subsystem that captures sensory data, a processor to implement the control laws, and an output processing subsystem to communicate the computed commands to the control surfaces. The computers that implement these control systems have become "flight critical"; that is, if the flight control system becomes inoperative, the pilot may be unable to control the aircraft. This differs radically from the autopilot on previous aircraft - if the autopilot failed, it was only necessary to revert control to the pilot in an orderly manner. The FAA requires that the probability of failure for a commercial aircraft be on the order of 10^{-9} per hour of flight. In order to meet this overall figure, the probability of failure of the flight control system must be an order of magnitude lower.

It is beyond the state of the art at the present time to manufacture a single processor with a failure rate this low, even if one considers only random hardware failures. A state-of-the-art system composed of a single input subsystem, processor, and output subsystem would exhibit a mean time to failure of 2000 hours. The accepted method of constructing a highly reliable system is to use multiple redundant processors and employ failure logic in such a manner that, when one processor fails, it is removed from the system. The remaining processors continue to safely control the aircraft. It is reasonable to assume that such hardware failures are independent events, and, if the failure logic and redundancy management are perfect, then the probability of system failure is simply the probability of all the processors failing at the same time. With currently available hardware, between three and four processors are required to achieve a probability of system failure of 10^{-9}. When the failure rate of the actuating system is considered, it is necessary to increase flight controller reliability to a probability of failure on the order of 10^{-10}.

For a redundant system to operate successfully, the system must start each flight with all channels operative and fault free. The channels must be interconnected in such a way that it is possible to detect and isolate a failure in any channel and reconfigure the system without that channel. Usually, the outputs of all the channels are compared, and voting is used to detect and isolate a failure. However, when the system is operating with only two channels, a secondary detection means, such as self-test, is used to detect and isolate failures. Because the system operates in a dynamic environment, there is a limit to the time that this reconfiguration process can take. That is, during the reconfiguration time the aircraft is operating without control; there is a time after which the aircraft may no longer operate safely in this state. In the worst case scenario this time is equal to the length of time the aircraft can survive a hardover on the control surface.

It is not necessary for each channel to be an autonomous flight controller. Rather, monitoring may be done at a subsystem level and the voted output of a subsystem fed forward to the next redundant level. This adds move voting planes to the system, which relaxes the requirements of fault detection at each comparator-monitor. There is no limit to the number of voting planes that may be added to the system provided that it is demonstrated that each voting plane detects all the failures of the subsystems it votes on, and the overall probability of system failure (including failure of the additional monitoring hardware) is within acceptable limits.

Many system designs already exist for modularly decomposing a large avionics system into smaller subsystems that may be configured in a manner similar to the one just described. In the past, the processor itself could be partitioned into smaller modules, with each module containing its own independent redundant architecture, i.e., multiple replication with a comparator and voter. The probability of system failure could then be calculated from hardware considerations only (see, for instance, FTMP [1]). With the advent of microprocessors into the avionics world, the partitioning of the processor for monitoring is unfeasible. Therefore, a system architecture using the processor as a building block is used. Such an architecture is desirable from an economic standpoint, and has been studied extensively in the past [2,3].

There is an intuitive appeal to the system designer in placing the comparator at the processor output because it protects the control surfaces of the aircraft from receiving erroneous signals. Because of the complexity of the processor and its complement of memory devices, it becomes increasingly difficult to show when or if a single fault within the processor will reach the comparator and be detected. Under these conditions, it is possible for faults to accumulate within the processors in such a manner that, when a particular fault is propagated to the comparators, faults have occurred in more than one redundant channels. The latency period of the fault may have allowed a similar fault to occur in another

channel resulting in premature system failure.

Recapping, in order to demonstrate the required survivability of an avionics system it is necessary to show that:

- the reliability of each of the redundant hardware modules is sufficient,
- the pre-flight test insures that all channels are fault-free,
- the voter/monitor detects and isolates all faults,
- secondary detection means detect and isolate faults as required,
- fault isolation time is within required time limits, and
- all faults propagate to a comparator quickly enough so that latent buildup does not reduce system reliability.

1.2 Latency as a Factor in Highly Reliable Systems

It has been shown by many investigators [4,11,14] that the predominant cause of failure in highly reliable computing systems is not through exhaustion of spares. If faults in any portion of the system are not correctly detected by the failure logic, recovery will fail. For instance, faults in nonreplicated hardware may cause incorrect computations and this will not be detectable; these faults cause single point failures. The probability of system failure cannot be decreased below that due to single point failures by adding more spares (unless, of course, the spares make these single point failures redundant). The inability to detect all faults within a system is referred to as imperfect coverage, and the coverage of a system is defined as the probability that a fault in the system is detected so that proper recovery can occur.

The computation of coverage is complicated by a group of faults that may or may not be detected depending on the operational state of the system. Since the state of the system is a complicated function of its operating environment, one cannot say for sure when and if these faults will be detected. The time from occurrence to detection of these faults is called the latency time. When the latency time of a fault is significant, other mechanisms exist for system failure to occur:

- a latent fault in one module of a redundant system could combine with a new fault in a second module of the system to defeat the comparator, and
- two latent faults can exist in two separate modules of the system, and a particular state of the environment could activate them.

In order to properly estimate the probability of system

failure, it is necessary to associate probabilities with each mechanism that could cause it to occur. While the above latency mechanisms are well understood, a practical methodology for measuring the parameters necessary to compute the probabilities does not exist.

1.3 Software Errors Modeled as Latent Faults

Designers have gained experience with generic hardware design problems in existing equipment, and the use of good engineering techniques has reduced them to an acceptable level. However, the experience of computer system designers is that software posses a significantly larger problem in validating error free operation than its complementary hardware. Reliability analysis is often based on the premise that the probability of component failure is random when exposed to the stresses and strains of normal operation. Moreover, two identical components subject to the same environment fail independently. However, software failures are not failures per se. Software never deteriorates with age; rather all failures result from design errors. In use, these errors are subject to "random excitation" and result in unpredicted system response.

To perform a system level reliability analysis, a "probability of failure" be associated with each software module. It would be more correct to say that the probability of excitation of existing latent design errors is required. In this sense, latent design errors are like latent hardware faults, and it seems reasonable to explore whether the same techniques can be used to measure software probability of failure as are being used to measure fault latency herein.

1.4 Objectives of the Studies

The overall objectives of this work are to provide estimates of the system characteristics that are necessary to calculate the probability of system failure due to latency. It would be desirable to supply both a methodology for measuring latency, and an estimate of its effects in a typical avionics application.

It is also desirable to measure the probability of failure of the software embedded in a flight critical piece of hardware using similar techniques to those used for measuring hardware latency.

1.5 Simulation as a Tool in the Reliability Estimation for Both Hardware and Software in a Highly Reliable System

To achieve the above objectives, simulations were constructed

that were appropriate for the type of system being tested (i.e., hardware or software). The actual reliability estimation process consisted of three tasks:

- Markov models must be constructed to relate the probability of system failure to measurable system characteristics. Markov models of latent faults were developed, and have been expanded herein. Markov models of the software modules were also constructed.

- A simulation of the system must be constructed and its characteristics measured. A fault simulation language called GGLOSS was developed under NASA funding [15]. GGLOSS is a special purpose logic simulation program which is gate level and very high speed. GGLOSS is capable of simulating multiple processors with different software programs running concurrently. It is envisioned that GGLOSS will be used to demonstrate the reliability of a flight control system. While many high quality gate level logic simulator are available in the marketplace, they cannot be used for this purpose because:

- they cannot accommodate the number of gates required, and

- their speed of execution is too slow.

The circuit description is presented to GGLOSS in a standard Partslist format. It is important to emphasize that GGLOSS was never intended as a circuit design tool. Consequently, it lacks many of the features normally contained in a commercially available simulator. It is assumed that the circuit will operate properly in the unfaulted state.

GGLOSS performs statistical fault analysis of the system. That is, GGLOSS inserts a number of random faults into the processor while it executes its software. The results are recorded, and a postprocessor reports the failure statistics to the user in a format meaningful for conducting reliability studies.

A separate simulation of the process environment (aircraft landing, etc.) was constructed for software validation.

- The Markov model must be solved. Models of a simple processor with simple flight control programs were used to measure system characteristics that were then used to numerically solve the Markov models for probability of system failure. Results from simulations for software validation were used to evaluate software reliability.

2. THE GGLOSS SIMULATION

2.1 The GGLOSS Simulation Methodology

Examining the faulted behavior of a processing system requires a simulation methodology that emulates logic network operation. In addition, statistical techniques are needed to inject faults into the system in such a manner that the simulation results can be used to accurately predict failure characteristics of the hardware. The required methodology was designed by considering:

- a mathematical model of the faults to be considered

The mathematical model for the faults in GGLOSS is a stuck-at-zero or stuck-at-one fault at each node of a gate level equivalent circuit of the processing system. The use of pin level faults was considered in earlier work [6], but the results were too optimistic to be reliable. That is, the advent of VLSI hides many of the intermediate nodes of the system within a chip, and these faults are not tested under pin level assumptions. It also appears that a fault on an intermediate node is more likely to remain latent than a fault on a pin.

To model LSI and VLSI circuits, each device is represented by its manufacturers supplied gate level equivalent circuit.

- a simulation technique that incorporates the faults and accurately predicts the response of the processing system

The simulation technique used in GGLOSS is parallel fault simulation. Parallel simulation takes advantage of the fact that logical operations incorporated in the instruction set of a digital computer operate independently on each bit of the computer word. If the simulation is written in such a manner that only boolean operations are used, then each bit position in the host computer word can be used to simulate a different version of the same logic network. Each version finds the response to a different fault. On a VAX computer, thirty two different faults can be simulated on the processor at the same time.

To increase the speed of GGLOSS, it is assumed that all sequential logic being simulated is synchronous and no pulse generation is allowed. This lack of pulse, or transient, signals allows GGLOSS to compute only the steady state output of each combinational network. Modern logic design precludes the use of asynchronous circuitry.

- a statistical technique for controlling fault injection and interpreting the results of the simulation

To obtain failure statistics for the system, a probability of

occurrence is assigned to each fault in its fault set. The following rules were utilized for assigning probabilities to gate level faults:

- The failure rate for each device is obtained from the applicable military handbook,
- The failure rate of the device is equally distributed over the gates of the equivalent circuit.
- The failure rate of a gate is equally distributed over the nodes of that gate.
- Stuck-at-one and stuck-at-zero faults are equally likely.

The probability of occurrence for all the faults in the fault set is summed to obtain the probability of a fault occurring within the system.

If every fault is simulated for every possible input, the conditional detection probabilities of each fault is known. These detection probabilities can be combined with the probability of occurrence of each input to obtain overall detection characteristics for the system, such as fault coverage, latency time, etc. Often, the number of possible faults is too large to allow simulation of every one. In this case, Monte Carlo techniques can be used to obtain approximate results by proceeding in the following manner:

- Faults are placed in a numerical ordering for selection purposes
- A cumulative distribution function for the ordering is generated based on the probability of occurrence of each fault
- A random number is selected within the range of the cumulative distribution function
- This number is mapped back through the CDF to a specific fault

The last two steps in the above process are repeated until a list of faults is generated. These faults are simulated, and the results are used to calculate the overall detection characteristics of the system. The error in a detection characteristic obtained in this manner depends on the value of the detection characteristic, the number of faults simulated, and the desired confidence level. However, calculation of error is straight forward as illustrated in [6]. When this error is considered unacceptably high, the Monte Carlo process is repeated for a larger fault sample.

Choice of Simulation Technique

The behavior of a particular digital network can be checked under the fault model either analytically, or by one of three fault modeling techniques.

Analytical techniques require examining the topology of a network to determine what faults can and cannot be detected at the output. If the boolean equations describing the network were available for the unfaulted case and for a faulted case, then they could be solved in such a way as to predict under what input conditions the fault could be detected at the network outputs. One way of doing this is to examine the faulted gate and determine what values on the input nodes induce a boolean value at the gate output that is different from the faulted case. The input nodes are then traced backwards to the network inputs to determine if an input set exists that induces this gate output. This process is known as failure excitation. The output of the faulty gate is then traced to the network output to see if its effects are measurable. This process is known as path sensitization. If both failure excitation and path sensitization are present for each failure, then it is detectable under the proper input vector. An analytic technique that performs the above calculations on boolean equations is called the D algorithm.

The general simulation of an electrical circuit involves the solution of differential equations representing lumped circuit parameters. Such a simulation will correctly predict both the steady state and transient behavior of the system. Simulation of logic circuits is usually simplified by requiring that only the steady state behavior be simulated. The steady state solution can be updated after the network has reacted to an input stimulus. Let the output of every gate represent a state variable of the system. Then the circuit be represented by a set of simultaneous boolean difference equations of the form:

$$O_x(k+1) = B(I_1(k), I_2(k), \ldots, I_n(k))$$

where O_x is the output of gate x at the k+1 interval

B is a boolean expression of its arguments

I_n is the nth input to gate x

The exact form of these equations depends on what additional assumptions have been made about the circuit.

In the unit delay simulation, it is assumed that each gate exercises a boolean equation of its inputs, but with a small time delay (usually on the order of 5-10 nanoseconds, depending on the logic family). The boolean equations can be iterated on a time base that represents this unit delay. For a given set of circuit inputs, the boolean equations are solved on a digital computer repeatedly until the network reaches an equilibrium or steady state value, which is the response of the network to these circuit inputs. As in any iterative solution technique, it is not

guaranteed that convergence will occur. However, convergence will occur if the network is feedforward only. It is noted that:

- unit delay simulation correctly predicts that portion of the transient response that can be represented as a boolean change, and

- if the network is poorly designed, there is no guarantee that a steady state solution will be reached.

One disadvantage of unit delay simulation is the amount of iterations required to reach steady state. Even in a network of N gates that is feed forward, this can amount to N evaluations of the entire network, or N^2 equations. However, by proper ordering of the equations, this can be reduced to one evaluation of the network equations. This ordering will be referred to as a p-ordering.

In many applications, only small portions of the circuit change boolean value for a given interval of time. The propagation of such a change is often "blocked" somewhere along the network by a single gate which feeds forward its previous value even after its inputs have changed. To take advantage of this phenomenon, one could retain the previous state outputs of the gates and evaluate only those gates whose inputs have changed at the beginning of each iteration. This process is repeated until no inputs to any gates have changed from the previous iteration, at which time the network has reached a steady state. The changing of input values for a gate is referred to as an "event", and this type of simulation is called event driven simulation. Statistically, most network simulations result in fewer gate evaluations using this technique. One disadvantage of the technique is the amount of simulation code required to do the bookkeeping.

The logical operations indicated by the boolean equations are performed by computer instructions that operate independently on each bit of the computer word. If the simulation program is written in such a manner that boolean operations only are used, then each bit position in the host computer word can be used to simulate a different version of the same logic network. However, each logic network can have a different boolean value on its inputs. In this manner, the same amount of host computer time can be used to simulate up to 32 different networks on a VAX host, each reacting to a different possible set of inputs. Let the presence of a stuck-at-one, stuck-at-zero fault be modeled as an input to the system for each gate. Then a large number of faults can be simulated in a small amount of host time using this method, which is commonly called parallel fault simulation.

One disadvantage of parallel fault simulation is that it must be implemented on top of a unit delay simulation, which is often time inefficient. However, by p-ordering the gates in the simulation, the parallel simulation can be made very efficient.

GGLOSS was initially designed to be a parallel logic

simulator. (However, later improvements allow other simulation techniques to be incorporated into the simulators GGLOSS produces). To increase the speed of GGLOSS, it was also assumed that all the sequential logic being simulated was synchronous, i.e. no asynchronous pulse generation is allowed. The lack of asynchronous circuitry allows GGLOSS to compute only the steady state output of each combinational network ignoring transient response. Modern logic design precludes the use of asynchronous circuitry.

Most microsequencer controlled logic is designed so that all flip-flops change state at the same instant of the system clock cycle. This is usually either the leading or trailing edge of the clock, and the remainder of the clock cycle time is used for the logic network to stabilize. GGLOSS works most efficiently with this type of logic network by evaluating the boolean logic equations only once for each clock cycle. This requires some special treatment of clocking in flip-flops, which occurs within the flip-flop macros.

Some logic networks which operate synchronously use both the leading edge of the clock pulse for certain flip-flops to trigger and the trailing edge of the clock pulse for triggering others. In the "toy" microprocessor, there were two timing signals that used the opposite edge of the clock from the rest. This created a problem for GGLOSS, and two solutions were developed to overcome it:

- modify the simulation to evaluate all the gates once every leading clock edge and once every trailing clock edge. This has the advantage that the user need not concern himself with which edge a particular flip-flop is triggered. However, it has the disadvantage that it requires twice as much host CPU time to execute each simulated clock cycle. The modification to the simulator requires changing the model of the flip-flop as given in the BLISS library of logic functions.

- force those flip-flops that are clocked from timing signals on the opposite edge to be evaluated last. This can be done by using the "fictitious clock" concept which was mentioned previously. In this case, the fictitious clock represents a half clock cycle delay.

2.2 Some New Features Added to GGLOSS

The original version of GGLOSS was constructed by J. McGough and F. Swern at Bendix, and called IGGLOSS. One objective of this work was to improve IGGLOSS to make it useable for validation studies. Some improvements that were initially made to IGGLOSS were:

Automatic Fault Injection

The fault insertion mechanism was modified so that a large number of randomly chosen faults can be injected, and the simulator will return fault detection statistics. To accomplish this, the following changes were made to GGLOSS.

The GGLOSS compiler inserts provisions in the simulation for faulting every node of every gate in the model. Each prospective fault is assigned a number, and an index dataset is created that associates each number with the name of the gate and node it faults.

The user supplies an input dataset that contains a probability of failure for each gate type used in the simulation. GGLOSS uses this data to compute a cumulative probability of failure distribution for the logic being tested. When GGLOSS is run, it asks the user how many faults to inject. Faults are then chosen at random using the cumulative distribution and the random number generator on the VAX. The list of gates to be faulted is written out to a dataset. While the BLISS simulation is running, it reads the list of gates to be faulted, and simulates up to 31 faults. If more than 31 faults were to be injected, the simulation is repeated until the fault list is exhausted.

At the end of each simulation run, the name of the fault simulated is looked up in the 'fault index' dataset and placed in the detect dataset. A list of faults not detected is also produced.

Accommodate 32 different ROMS

The simulation was modified to optionally run a single copy of 32 different machines executing in parallel, each running from a different copy of ROM. Because of the parallel simulation technique, the machines would be running bit synchronous. In order for the machines to communicate with one another, a common memory area can be set up that can be accessed by each machine. A single fault is injected into one of the machines, and the recovery mechanism of the system can be tested.

Accommodate multiple faults in a single machine

The simulation was modified so that multiple fault situations might be studied. In this case, a total of 31 faulted machines is simulated, but two or more faults will be injected per machine. A prompt from GGLOSS during compilation of the simulation sets the number of faults injected per machine.

Specification of Initial Conditions

The user may wish to specify initial conditions for each simulation. To simplify the process of initialization, GGLOSS assumes that all initial conditions are zero. The user may then specify an initial condition of one for the output of any gate(s) in the simulation.

This is accomplished by prompting while the simulation is being compiled. The user specifies the name of the output signal of the gate as given in the partslist. The prompt is repeated over and over again until the user indicates he is satisfied with the set of initial conditions.

Extended Test Pin Coverage

One output of the GGLOSS simulation is a table of faults followed by the test points that detect these faults. An entry is made into this table every clock cycle that the signal at the test point differs from the nominal case. At the end of the simulation, a complete fault dictionary has been constructed.

Memory Mapped I/O

One can now simulate inputs and outputs to the outside world, using memory mapped I/O. The actual values of input signals are contained in a dataset, one dataset record for each clock cycle that an IO signal is referenced. It is assumed that the user knows the proper sequence of input values required for each particular experiment a priori. An output signal is considered to be a monitoring point, and any deviation in output between different machines is recorded as detection of the failure.

Shared Memory

When GGLOSS is used to simulate a redundant processor complex, each group of processors can communicate with one another through shared memory. The shared memory is available to all the processors in the complex so that processor interaction and software monitoring and recovery can be simulated.

Intermittent Faults

An intermittent fault model was be developed and programmed into the simulation. This particular fault model allows two types of intermittent faults: single incident and cyclic. In the cyclic fault model, the user specifies the frequency and duration of intermittency. In the single incident model, the user specifies

the starting and ending clock cycle of the fault.

Expand Library of Bliss Coded Primitives

This task involved using a catalog of MSI and LSI logic to expand the GGLOSS primitive library to include most of the commonly available blocks used in logic design.

Faults in RAM

This task involves modifying the RAM model so that stuck at faults can be simulated. One may currently inject faults in RAM by specifying the word address, bit number, and stuck-at-one or stuck at zero. A dataset is supplied to GGLOSS which describes the memory locations to be faulted.

Tasks to be Modified or not to be Undertaken

The initial plans for modifying IGGLOSS contained items which were later modified as new directions in the development of GGLOSS as a simulator were planned:

Network Partitioning

A parallel fault simulation of a processor contains a large number of prospective gates to be faulted. However, the procedure only allows 31 different gates to be faulted each time the simulator is run. In the Bendix BDX930 simulator, simulation time was reduced significantly by dividing the processor into four partitions. Four different simulators were constructed, each allowing only one quarter of the processors gates to be faulted. After a large number of faults were selected, they were sorted into groups corresponding to the partitions. By running each group on its corresponding simulator, the correct results were obtained at a significant savings in execution time.

Discussions with RTI at Langley indicate that future development of the simulator would change the fault simulation mechanism used in the simulator. In the new technique, every gate would contain a unconditional branch statement that would bypass fault simulation. Memory modification techniques would be used to change the unconditional branch statements for those gates that are faulted. The advantage of this technique is lower overhead in the fault injection process.

Because of the lower overhead in the new technique, there is little to be gained by partitioning the network as described above. Therefore, this task is being held in abeyance pending future decisions on the fault simulation mechanism to be used.

Multiple Fictitious Clocks

The original role of the fictitious clock was to force the

evaluation order of the logic. Multiple fictitious clocks was supposed to allow different clock periods to be used within the simulation.

Many of the problems in getting the simulator to operate correctly with the model of a processor centered around the use of fictitious clocks. It was easier building a circuit model without them. However, they were useful in simulating a half clock cycle delay.

It is unclear at this point what fictitious clocks will be used for in the simulation. However, some thought will be given to the problem of how circuit timing (i.e., half cycle delays, etc.) can be properly simulated.

Fault Collapsing

This task requires comparing faults in a large fault list and deciding which faults have exactly the same effect on the network. This would be done from the topology of the network. Once a list of equivalent faults has been found, each equivalency group need be simulated only once. In a large combinatory network, this can represent a considerable savings in execution time. However, more work is needed to synthesize the algorithm to do this, and work on this task is continuing.

2.3 Processing Systems Described Across Multiple SDL Files

One problem with the IGGLOSS version of GGLOSS was that an entire processing system had to be described as a single module using a single SDL file. In systems containing a large number of logic gates, this was cumbersome and severely stressed the storage capacity of the GGLOSS program. A better approach is a modular one, breaking the simulation down into different SDL files and allowing GGLOSS to translate each SDL file separately. In the current version of GGLOSS, large processing systems may be represented in this manner.

Decomposition into modules is very natural to digital design, as digital systems are built hierarchically from chips into boards, and boards into systems. It is possible for a VLSI chip to be modeled by a software module and used multiple times within the simulation. The chip's pinout gives the signals that cross the module boundaries. In other cases, partitioning occurs solely due to module size, and choosing signals to cross module boundaries is not as natural as using a chip's pinout.

Each module is represented by a separate SDL file, and is compiled by GGLOSS into a separate BLISS program. To construct a modular simulator, it is the responsibility of the user to supply the appropriate SDL files to GGLOSS, including:

- at the top of the hierarchy, a single SDL file that describes the system using partitions, chips, and gates to be described in lower levels,

- additional SDL files to describe partitions of the system using both gates and lower level modules as required,

- any additional SDL files describing modules at lower levels,

- a library of primitives representing gates (same as the old GGLOSS library file), and

- sufficient gate failure statistics for GGLOSS to compute the failure rate of a module, and all modules in the hierarchy.

An example of a modularly constructed simulation is given in Figure 2.3.1.

CONNECTIVITY OF MODULES

There is a caveat in this partitioning process when working with parallel simulations. In the parallel method of simulation, the ordering of statement execution is extremely important. When GGLOSS works with a single SDL file describing a large system, its "P - ordering" algorithm almost always assemble a simulation module that will work properly when there are no feedback loops in the system. However, partitioning implies that GGLOSS cannot easily

"shuffle the statements from different modules" to find this correct ordering, and a poor partitioning may result in an unusable simulator. When this occurs, there will be no correct "P - ordering" of the higher level modules. GGLOSS can help the user by giving diagnostic information to help pinpoint which submodules or gates it is having trouble ordering.

Consider first how the user would handle this partitioning problem. One solution would be to break every module down into a number of submodules such that the interconnection characteristics of the submodules guarantees the existence of an overall ordering for the system. A possible technique would be to translate each module into BLISS using GGLOSS and note what problems GGLOSS has in "P - ordering" the module. Modules which GGLOSS has trouble ordering can then be broken down into smaller submodules, and the process repeated until a good simulator is built.

It seems desirable to solve the problem within GGLOSS, rather than hand it to the user. GGLOSS must examine the topology of each module, and determine the number of submodules that are required. In this case, all the submodules will be contained in a single BLISS module, existing as subsections of that module. The BLISS code would contain a variable indicating which submodule the caller wishes to execute. For "P - ordering" purposes, higher level GGLOSS routines treat the single BLISS module as separate submodules, and generate calls indicating which submodule to execute.

To develop the topological algorithms, let the inputs to a module be represented by a binary vector I and the outputs be represented by a binary vector $\underline{O}$. Then there exists a binary transfer matrix T that relates the circuit outputs to the circuit inputs, i.e.,

$$\underline{O} = T \underline{I}.$$

Each element of T represents a boolean transfer function between a particular input and a particular output. The exact values of the elements of matrix T are not of interest, only whether or not the element is a logical zero. A matrix T′ is constructed by substituting logical ones in matrix T for nonzero transfer elements, and T′ represents the connectivity of the module. An example of the T′ matrix is given in Figure 2.3.2. If all the elements of T′ are nonzero, then there is nothing further that can be done to break this module into submodules, since each output requires the knowledge of all the inputs for its computation.

When some of the elements of a specific row are zero, its corresponding output can be computed without knowing the values of those inputs whose columns contain the zeroes. Hence, each row in T′ which contains a different pattern is a candidate for a separate submodule. The construction of the submodules and the order of their call must be such that the proper evaluation of the module

occurs; however, this can be ascertained using the "P - ordering" algorithm as it already exists.

MODULE DATA FILES

When GGLOSS compiles a module from SDL into BLISS, it generates a file containing data about the module. This file contains:

- the name of the module,

- the number and values of any initial conditions on the module,

- the number of memory elements in the module,

- the failure rate of the module and a cumulative fault distribution among all the gates of the module,

- detection points within the module,

- whether the input pins are in common or passed as parameters, and

- the template of the calling statement including the order of the arguments.

This file is created in the GDAT: area of disk and has the file extension of .FTB.

The file has three uses:

- Computing the gate level equivalent of higher level modules

In order to properly compile a multi-module simulation, modules must be compiled working up the hierarchy chain. Each succeeding level of the hierarchy requires information about lower levels; specifically, the calling template of all lower level modules and their gate level equivalents. The gate level equivalent and its cumulative failure rate for each module is read directly from the FTB file.

- Generating complex simulations without recompiling each module

In order to avoid recompiling all modules each time the simulator is rebuilt, the INCLUDE command can be used to read the required data from the FTB file. This is convenient and time saving if a library of VLSI chips is built to be used in constructing complicated network simulations.

- Generating new fault simulation lists without recompiling the simulation

In the Monte Carlo environment, it may be desired to run the simulator again with a different random selection of faults. This

can easily be done by INCLUDEing the highest level module and generating a new faults set with the FLTGEN command.

```
$ R GLOSS
GLOSS>
COMPILE GDAT:JEANX1.SDL,FAULT
TOTAL NUMBER OF FAULTS IS          906
         14  COMPONENTS IN THE FAILURE RATE TABLE,
FAILURE RATE FOR THE MODULE IS   1.3111077E-08
 NUMBER OF STATEMENTS =          132
GLOSS>
COMPILE GDAT:JEANX2.SDL,FAULT
TOTAL NUMBER OF FAULTS IS          376
         14  COMPONENTS IN THE FAILURE RATE TABLE,
FAILURE RATE FOR THE MODULE IS   5.1111111E-08
 NUMBER OF STATEMENTS =           61
GLOSS>
COMPILE GDAT:JEANX3.SDL,FAULT
TOTAL NUMBER OF FAULTS IS          410
         14  COMPONENTS IN THE FAILURE RATE TABLE,
FAILURE RATE FOR THE MODULE IS   6.5111112E-09
 NUMBER OF STATEMENTS =           67
GLOSS>
COMPILE GDAT:JEANXT.SDL,FAULT,TABLE=GDAT:TOYFAIL.DAT
TOTAL NUMBER OF FAULTS IS         1692
         14  COMPONENTS IN THE FAILURE RATE TABLE,
FAILURE RATE FOR THE MODULE IS   2.4733000E-08
 NUMBER OF STATEMENTS =            4
GLOSS>
MEMORY RAM,TYPE=RAM,ADDBITS=8,-
GLOSS>
   DATABITS=8,INIT=GDAT:TOY.MEM,FAULT,LENGTH=32
GLOSS>
BIND TOY,CYCLES=300,PRINT
```

A Modularly Constructed Simulation

Figure 2.3.1

		INPUTS					
		BIN1	BIN2	BIN3	BIN4	CIN	ALUEN
OUTPUTS	ALUO1						1
	ALUO2						1
	ALUO3						1
	ALUO4						1
	CARRY	1	1	1	1	1	
	C7NM1	1	1	1	1	1	
	C7NM2	1	1	1	1	1	

An Example of a Connectivity Matrix T'

Figure 2.3.2

2.4 Simulation of Functional Blocks

GGLOSS is capable of simulating up to 32 different faults within the same parallel run. In practice, one of the parallel executing simulations is always the good system, acting as a reference to determine when fault detection occurs. This means that 31 different faults are being evaluated during each simulation run on a VAX machine. The fault set of even a small processor contains more than ten thousand faults - most of which are not active during a given simulation run. Thus, a large portion of the logic is running "true-value"; that is, it is not simulating faults, but rather propagating the results of faults occurring in other logic gates.

Unfortunately, gate level logic simulation does not efficiently simulate this propagation phenomena. In most cases it is faster to functionally characterize portions of the logic that are unfaulted. For instance, it may take a hundred statements to simulate an ALU chip at the gate level while it takes only a few statements to functionally describe the chip. When a functional simulation of a module executes significantly faster than the equivalent gate level simulation, there may be a decrease in GGLOSS execution time by including that functional module. The objective of this portion of the study was to develop a methodology that would allow simulations to be constructed consisting of a combination of modules described functionally and at the gate level. In addition to decreased simulation time, there are other advantages of the functional simulation technique:

- ability to use a simplified description of peripheral hardware,
- ability to debug large simulations using simplified functional descriptions of hardware not yet simulated at the gate level, and
- ability to include environmental and analog simulations.

CONSTRUCTION OF FUNCTIONAL LEVEL MODULES

Functional level modules are constructed using the following groundrules:

- functional level modules simulate input/output relationships only

Functional level modules use the same pinout signals as gate level modules, but they need only simulate the functional relationship between the pins. This means that the internals of the module can be simplified in such a way as to reduce the amount of necessary computation, as long as it produces the correct answer at the output pins. However, to construct such simplification requires an individual to examine the circuit in each module and

synthesize a "custom" subroutine representing that module.

- functional level modules simulate one version of the network only

Functional level modules do not utilize parallel simulation, and therefore simulate one version of the network only. Thus, a functional level module must be called once for each active version in the parallel simulation. To correctly keep track of the different versions, the functional level module is passed a state storage array to store all the results of computation for this version. The same functional module can be called repeatedly in a loop to simulate all the active versions of the system as long as the calling module changes the state storage array it passes with each call.

- functional level modules may contain analog simulations

Functional level modules may contain continuous differential equations simulated in the standard manner, i.e. using any of the well known integration techniques such as Eulerian integration, Runge-Kutta, etc. These modules share in common with the control routines a time base which is used to synchronize any analog simulation with logic simulation.

SIMULATION ARCHITECTURE WITH FUNCTIONAL MODULES

To make the maximum use of functional computation, all modules but one are modeled at the functional level while the remaining module is modeled at the gate level using parallel simulation. Faults are injected at the gate level in the latter module only. An interface routine is provided that translates from the functional level output to gate level input, and back again.

Often it is not necessary for functional modules to be executed thirty two times to represent the thirty two versions being simulated. If a fault does not manifest itself by propagating to the output of the parallel module, then its functional state is the same as that of the unfaulted version. Even when a fault propagates to the output of the parallel module, its functional state may be the same as another fault being simulated which presents the same outputs from the parallel module. Faults which have shared the same functional state from the start of the simulation until a particular time will be termed equivalent, for they manifest themselves in exactly the same manner outside the faulted circuitry within that time frame.

Equivalent faults are modeled by a single execution of the functional modules. At the start of a simulation run, there is exactly one equivalence class for all faults including the unfaulted network since faults have not yet manifest themselves. As the simulation continues, the outputs of the parallel module are tested for a pattern that indicates a new equivalence class. When a new equivalence class is found, a new functional "machine" is set

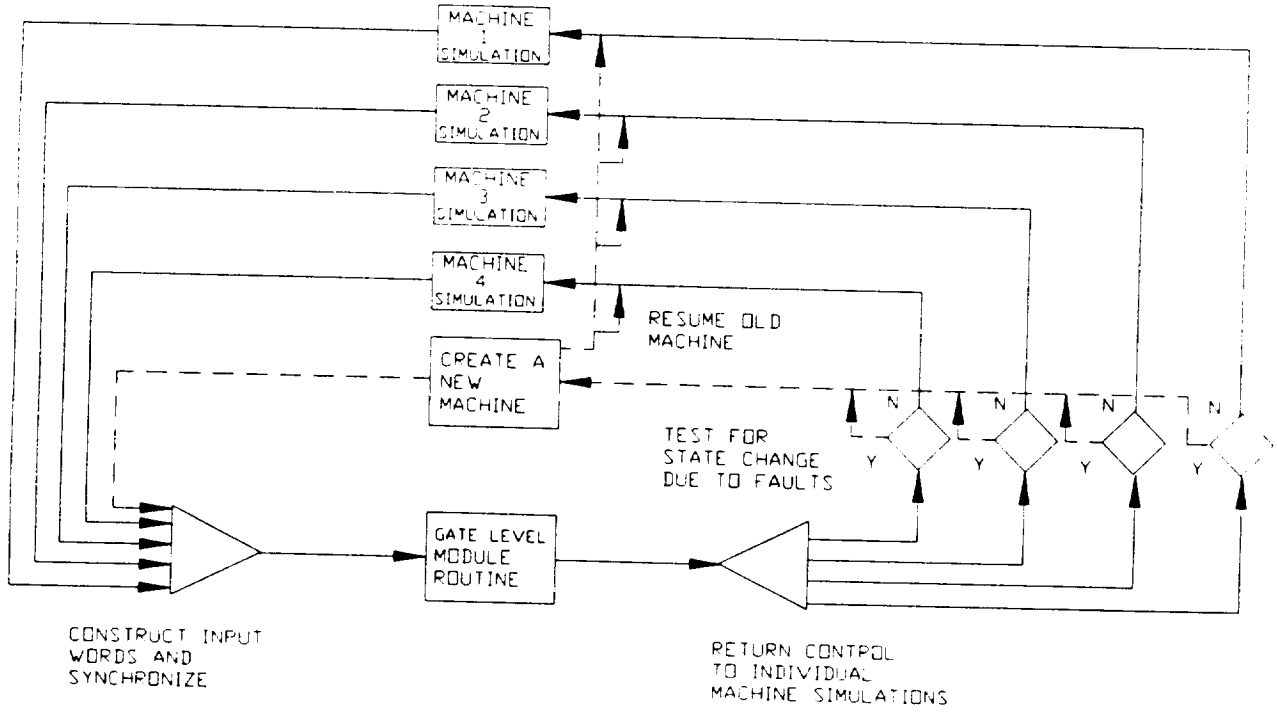
up, copying the state storage array from the "machine" that spawned it. This process continues, spawning new machines as required. However, when a fault is successfully detected, the machine representing its equivalence class is terminated. At any instant, the simulation must execute its functional modules as many times as there are active equivalence classes. The process is shown diagrammatically in Figure 2.4.1.

The logistics of simulating faulted machines is simplified by storing a fault mask for each equivalence class. A one in a particular bit position indicates that the parallel fault represented by this bit position is part of this equivalence class. Initially, the simulation starts out with one equivalence class and a fault mask of all ones. The fault mask is used to pack the input words to the parallel module from the outputs of the functional modules, i.e., when the functional machine for a particular equivalence class has an output of one to be input to the parallel module, its fault mask is ored into the appropriate parallel module input word. On return from the parallel module, each output is checked against the fault mask for each equivalence class - if all the bits that are one in the fault mask are the same (all either one or zero) in the output, then execution proceeds. If some of the bits that are one in the fault mask are mixed in the output, then a new machine is spawned. The procedure is shown in Figure 2.4.2.

TESTING OF FUNCTIONAL LEVEL SIMULATION

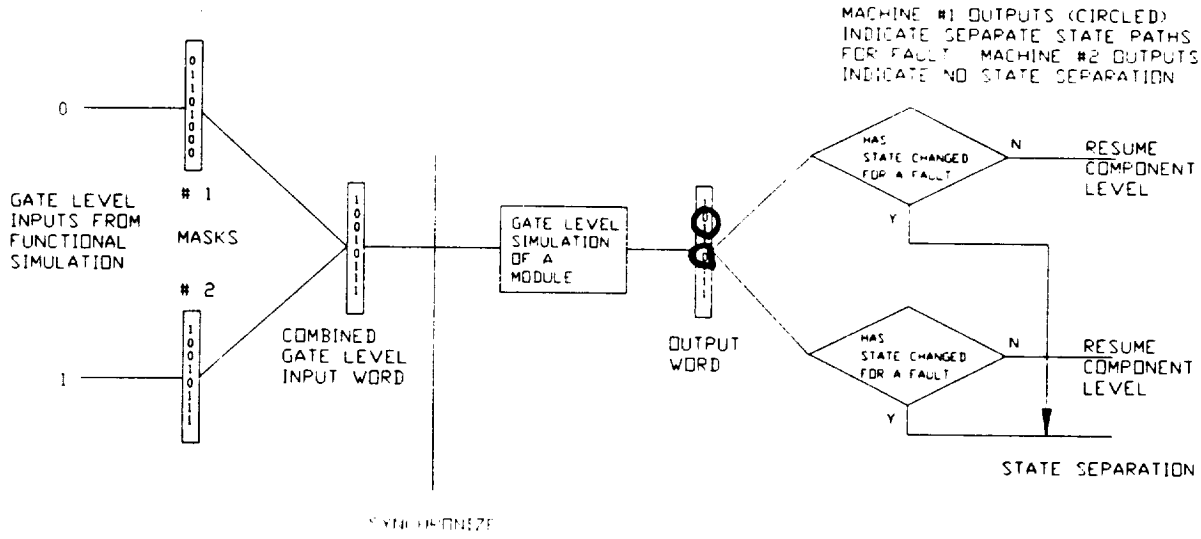
Functional level simulation was tested by implementing the ALU portion of the toy processor as a functional module. A partial listing of the module is shown in Figure 2.4.3. An interface module was constructed in BLISS that performed the checking and spawning operations described above, and is shown in Figure 2.4.4. Another FORTRAN module was programmed to keep track of both the average number of equivalence classes that existed in the simulation and the high water mark. The results of running this simulation is shown in Figure 2.4.5.

The functions performed by the ALU in the toy processor were simple to program. However, constructing and debugging the interface code in both BLISS and FORTRAN occupied much of the development time for this task. It was concluded that most of this code could have been generated by GGLOSS, and development of this capability should be part of the next contract period.



Modeling of Multiple "Machines" Using Functional/Gate Simulation

Figure 2.4.1



Interfacing Functional Modules With Gate Level Simulation

Figure 2.4.2

```
      SUBROUTINE FTALU(CCLK,CCLR,MD0,MD1,MD2,MD3,MD4,MD5,MD6,MD7,ENBL,
     1 ADDSUB,CNM1,ALU0Y,ALU1Y,ALU2Y,ALU3Y,ALU4Y,ALU5Y,ALU6Y,ALU7Y,
     2 ALU71,ALU72,ALU7Q,ALU7QB,CSTEP,STATE)
C
C     FUNCTIONAL SIMULATION OF TOY ALU
C
      IMPLICIT INTEGER*4 (A-Z)
      DIMENSION STATE(100)
C
      IF (CSTEP.EQ.0) THEN

      MD=0
      IF(MD0.LT.0) MD=MD+1
      IF(MD1.LT.0) MD=MD+2
      IF(MD2.LT.0) MD=MD+4
      IF(MD3.LT.0) MD=MD+8
      IF(MD4.LT.0) MD=MD+16
      IF(MD5.LT.0) MD=MD+32
      IF(MD6.LT.0) MD=MD+64
      IF(MD7.LT.0) MD=MD-128
      STATE(5)=CCLK
      STATE(6)=CCLR
C
         IF (ADDSUB.EQ.0) THEN
            STATE(2)=MD
            ELSE
            STATE(2)=-MD-1
            ENDIF

         IF (CNM1.LT.0) THEN
            STATE(3)=STATE(1)+STATE(2)+1
            ELSE
            STATE(3)=STATE(1)+STATE(2)
            ENDIF

         IF (ENBL.LT.0) THEN
            STATE(4)=STATE(1)
            ELSE
            STATE(4)=-1
            ENDIF

      ELSE

         IF (STATE(5).LT.0) STATE(1)=STATE(3)

         IF (STATE(6).EQ.0) STATE(1)=0

      ENDIF
```

Example of a Functional Level Module

Figure 2.4.3

```
IF(BTEST(STATE(4),0)) THEN
  ALU0Y=-1
  ELSE
  ALU0Y=0
  ENDIF

IF(BTEST(STATE(4),1)) THEN
  ALU1Y=-1
  ELSE
  ALU1Y=0
  ENDIF

IF(BTEST(STATE(4),2)) THEN
  ALU2Y=-1
  ELSE
  ALU2Y=0
  ENDIF

IF(BTEST(STATE(4),3)) THEN
  ALU3Y=-1
  ELSE
  ALU3Y=0
  ENDIF

IF(BTEST(STATE(4),4)) THEN
  ALU4Y=-1
  ELSE
  ALU4Y=0
  ENDIF

IF(BTEST(STATE(4),5)) THEN
  ALU5Y=-1
  ELSE
  ALU5Y=0
  ENDIF

IF(BTEST(STATE(4),6)) THEN
  ALU6Y=-1
  ELSE
  ALU6Y=0
  ENDIF

IF(BTEST(STATE(4),7)) THEN
  ALU7Y=-1
  ELSE
  ALU7Y=0
  ENDIF

IF(BTEST(STATE(3),7)) THEN
  ALU71=-1
  ELSE
  ALU71=0
  ENDIF

IF(BTEST(STATE(2),7)) THEN
  ALU72=-1
  ELSE
  ALU72=0
  ENDIF
```

Example of a Functional Level Module

(Continued)

Figure 2.4.3

```
MODULE FUNCTION(ADDRESSING_MODE(EXTERNAL = LONG_RELATIVE,
            NONEXTERNAL = LONG_RELATIVE)) =
BEGIN
!
GLOBAL ROUTINE TOYALU(I1,I2,I3,I4,I5,I6,I7,I8,I9,I10,I11,
      I12,I13,O1,O2,O3,O4,O5,O6,O7,O8,O9,O10,O11,O12,J,K) : NOVALUE =
!
!  NAME:  FUNCTIONAL INTERFACE ROUTINE
!
!  PURPOSE: INTERFACES BETWEEN GGLOSS PARALLEL GATE LEVEL SIMULATOR
!           AND BLOCKS WRITTEN AT FUNCTIONAL, NONPARALLEL LEVEL
!
!  INPUTS:
!
!  OUTPUTS:
!
!  DESCRIPTION:
!
!

  BEGIN

  MACRO CHECKIN(INPUT) =
      X = .INPUT AND .MASK[.I];
      IF (.X NEQ 0) AND (.X NEQ .MASK[.I]) THEN
         (NMASK = .NMASK + 1;
         MASK[.NMASK] = .X;
         MASK[.I] = .MASK[.I] AND NOT .X;
         INCR L FROM 0 TO 99 DO FSTATE[100*.NMASK+.L]=.FSTATE[100*.I+.L];
         EXITLOOP);%;

  OWN I, REG, JMASK, O01,O02,II1,II2,II3,II4,II5,II6,II7,II8,II9,II10,
      II11,II12,II13,O03,O04,O05,O06,O07,O08,O09,O010,O011,O012,X,L,
      MONE ;
  OWN
   FSTATE: VECTOR[3200],
   MASK: VECTOR[32],
   NMASK;
  EXTERNAL ROUTINE FTALU : FORTRAN;
  EXTERNAL ROUTINE PRI : FORTRAN;
  EXTERNAL ROUTINE CNTF : FORTRAN;
!
  REQUIRE 'GDAT:BLISCOM.R32';
!
!
 IF .IGOP EQL 1 THEN
     (NMASK=0; MASK[0] = -1; STOPMASK = -1; INCR I FROM 0 TO 3199 DO
      FSTATE[.I]=0;IGOP=0; MONE = -1; CNTF(MONE,MONE););
```

Example of a Functional Level Interface Routine

Figure 2.4.4

```
JMASK = 0;
INCR I FROM 0 TO .NMASK DO
   (MASK[.I]=.MASK[.I] AND .STOPMASK;
    IF .MASK[.I] NEQ 0 THEN JMASK = .JMASK + 1;);

CNTF(JMASK,NMASK);

DO BEGIN
   JMASK = .NMASK ;
   INCR I FROM 0 TO .NMASK DO
      BEGIN
        CHECKIN(.I1);
        CHECKIN(.I2);
        CHECKIN(.I3);
        CHECKIN(.I4);
        CHECKIN(.I5);
        CHECKIN(.I6);
        CHECKIN(.I7);
        CHECKIN(.I8);
        CHECKIN(.I9);
        CHECKIN(.I10);
        CHECKIN(.I11);
        CHECKIN(.I12);
        CHECKIN(.I13);
      END;
   END
   UNTIL .JMASK EQL .NMASK ;

 .O1 = 0; .O2 = 0; .O3 = 0; .O4 = 0; .O5 = 0; .O6 = 0; .O7 = 0; .O8 = 0;
 .O9 = 0; .O10 = 0; .O11 = 0; .O12 = 0;


INCR I FROM 0 TO .NMASK DO
   IF .MASK[.I] NEQ 0 THEN
   BEGIN
      II1 = (IF (.MASK[.I] AND ..I1) NEQ 0 THEN -1 ELSE 0);
      II2 = (IF (.MASK[.I] AND ..I2) NEQ 0 THEN -1 ELSE 0);
      II3 = (IF (.MASK[.I] AND ..I3) NEQ 0 THEN -1 ELSE 0);
      II4 = (IF (.MASK[.I] AND ..I4) NEQ 0 THEN -1 ELSE 0);
      II5 = (IF (.MASK[.I] AND ..I5) NEQ 0 THEN -1 ELSE 0);
      II6 = (IF (.MASK[.I] AND ..I6) NEQ 0 THEN -1 ELSE 0);
      II7 = (IF (.MASK[.I] AND ..I7) NEQ 0 THEN -1 ELSE 0);
      II8 = (IF (.MASK[.I] AND ..I8) NEQ 0 THEN -1 ELSE 0);
      II9 = (IF (.MASK[.I] AND ..I9) NEQ 0 THEN -1 ELSE 0);
      II10 = (IF (.MASK[.I] AND ..I10) NEQ 0 THEN -1 ELSE 0);
      II11 = (IF (.MASK[.I] AND ..I11) NEQ 0 THEN -1 ELSE 0);
      II12 = (IF (.MASK[.I] AND ..I12) NEQ 0 THEN -1 ELSE 0);
      II13 = (IF (.MASK[.I] AND ..I13) NEQ 0 THEN -1 ELSE 0);
      FTALU(II1,II2,II3,II4,II5,II6,II7,II8,II9,II10,II11,II12,II13,
          001,002,003,004,005,006,007,008,009,0010,0011,0012,
          CSTEP,FSTATE[100*.I]);
      .O1 = ..O1 OR (.001 AND .MASK[.I]);
      .O2 = ..O2 OR (.002 AND .MASK[.I]);
      .O3 = ..O3 OR (.003 AND .MASK[.I]);
      .O4 = ..O4 OR (.004 AND .MASK[.I]);
      .O5 = ..O5 OR (.005 AND .MASK[.I]);
      .O6 = ..O6 OR (.006 AND .MASK[.I]);
      .O7 = ..O7 OR (.007 AND .MASK[.I]);
      .O8 = ..O8 OR (.008 AND .MASK[.I]);
      .O9 = ..O9 OR (.009 AND .MASK[.I]);
```

Example of a Functional Level Interface Routine

(Continued)

Figure 2.4.4

```
$ RUN GSIM:EXEC
 ELAPSED:     0 00:00:41.00  CPU: 0:00:16.94  BUFIO: 304  DIRIO: 12  FAULTS  468
TOTAL CYCLES WAS           600
HIGH WATER MARK MACHINES WAS             1
AVERAGE NUMBER OF MACHINES WAS     1.000000
 ELAPSED:     0 00:00:42.94  CPU: 0:00:17.17  BUFIO: 304  DIRIO: 14  FAULTS  125
TOTAL CYCLES WAS           600
HIGH WATER MARK MACHINES WAS             2
AVERAGE NUMBER OF MACHINES WAS     1.998333
 ELAPSED:     0 00:00:42.91  CPU: 0:00:16.94  BUFIO: 303  DIRIO: 10  FAULTS  0
TOTAL CYCLES WAS           600
HIGH WATER MARK MACHINES WAS             2
AVERAGE NUMBER OF MACHINES WAS     1.9988333
 ELAPSED:     0 00:00:39.77  CPU: 0:00:16.08  BUFIO: 304  DIRIO: 14  FAULTS  0
TOTAL CYCLES WAS           600
HIGH WATER MARK MACHINES WAS             2
AVERAGE NUMBER OF MACHINES WAS     1.9988333
 ELAPSED:     0 00:00:44.34  CPU: 0:00:16.54  BUFIO: 305  DIRIO: 15  FAULTS  0
TOTAL CYCLES WAS           600
HIGH WATER MARK MACHINES WAS             5
AVERAGE NUMBER OF MACHINES WAS     2.275000
 ELAPSED:     0 00:00:44.46  CPU: 0:00:17.39  BUFIO: 304  DIRIO: 15  FAULTS  0
TOTAL CYCLES WAS           600
HIGH WATER MARK MACHINES WAS             8
AVERAGE NUMBER OF MACHINES WAS     2.335002
 ELAPSED:     0 00:00:43.49  CPU: 0:00:17.49  BUFIO: 303  DIRIO: 10  FAULTS  0
TOTAL CYCLES WAS           600
HIGH WATER MARK MACHINES WAS             8
AVERAGE NUMBER OF MACHINES WAS     2.348333
```

Performance Results for a Single Functional Level Subroutine

Figure 2.4.5

2.5 Simulation of Memory Blocks

Processors and sequencers to be simulated with GGLOSS contain a number of memory elements such as PROM used to contain microcode, PROM used to contain program code, RAM used as scratchpad, and memory mapped I/O. The previous GGLOSS memory simulation was inadequate in simulating all these different memories, and this situation was addressed by an improved memory block simulation technique. The GGLOSS memory simulation now allows the construction of twenty (an arbitrary number which can be increased) different memory chips with a variety of characteristics that allows enough flexibility to simulate practical processors.

Memories are handled on a functional basis, allocating VAX storage for either read only or read/write memory. The former is allocated only once; the same copy is read by each of the thirty two different machines. The latter is allocated thirty-two times so that each machine can have its own copy of RAM to modify as necessary. It is noted that there is no attempt to model memory at any lower level, such as static vs. dynamic, and/or including memory cycle timing. It was concluded that this type of simulation would be desirable in many cases; however, it is best modeled by using gate level logic to simulate the timing and then interface the processing system to the functional simulation included in GGLOSS.

Other features that have been retained in the memory simulation:

- Multiple processors may be simulated by neighboring bit positions on the VAX communicating through common memory.

- Only a portion of ROM and/or RAM need be simulated to conserve space in large memory simulations. Failures which result in access to unsimulated portions of the memory will return a word of all ones.

- Both ROM and RAM can be initialized.

- Memory mapped inputs are simulated by subroutine, and may either be linked to functional blocks or read a dataset containing input values.

- Memory mapped outputs are now detection addresses which may be used in the same manner as detect pins to initiate post-processing.

- A single memory chip may contain ROM, RAM and memory mapped I/O. This feature allows simplified simulation of processor operation without simulating the memory mapping logic.

- Both ROM and RAM memory may be selectively faulted by chip. ROM

faults randomly invert bits in memory while RAM faults are of the stuck-at-one, stuck-at-zero variety.

Memories are simulated in SDL using the MEMR primitive. However, the partsname for the memory is an eight character name that is used to reference that memory at simulation time. The GGLOSS MEMORY directive, described in the appendix, is used to associate memory attributes with memory chips using the partsname association. MEMORY also creates an initialization program to handle ROM loading.

2.6 Improving The GGLOSS Architecture for Other Simulation Methodologies

One design objective of GGLOSS was to produce a package that would simulate gate level hardware level with faults at speeds high enough so that phenomena such as fault latency could be studied. Parallel simulation methodology was chosen for maximum speed with faulted logic. Further, GLOSS reads a description of the logic under test and compiles a computer program that simulates that logic. However, after implementing GGLOSS, it soon became apparent that GGLOSS should be expanded to support more complex simulations including simulation modes other than parallel.

To accomplish this, building a simulation with GGLOSS was reorganized into a two stage process: -

o Build a P-ordered sequence of macro calls that represent the logic to be simulated. The translation process consists of three steps:

- Generate data declarations for every logic signal in the circuit being simulated.
- P-order the SDL description of the logic into an ordered list of simulation "blocks".
- Translate each simulation block into a macro sequence that describes its operation in a "neutral format".

o Translate the macro sequence into a target language for the simulation engine. This process also consists of three steps:

- Read in a macro definition file that expands each macro sequence generated above into a target language for simulation.
- Expand the macro sequences for each simulation block.
- Compile the tables required for table driven algorithms.

The above process was implemented, and shows many advantages over previous GGLOSS implementations. It allows GGLOSS to run on a variety of different computers, using a variety of different simulation languages, using a variety of different simulation techniques. More specifically, the languages GGLOSS can produce simulators for might include:

o BLISS

o FORTRAN

o C

o PASCAL

- Assembly Language

while the types of simulators that might be produced are:

- Parallel
- Event Driven
- Deductive
- Concurrent

The flexibility of GGLOSS should be apparent: GGLOSS can produce an optimum simulator for each situation with a minimum of effort on the part of the GGLOSS user.

To illustrate how the GGLOSS works, Figure 2.6.1 shows the output of the first stage of building a simulation while Figure 2.6.2 shows the macro definitions for the second stage that produces a FORTRAN parallel simulator. The next two sections describe how GGLOSS constructs Event Driven and Deductive simulators.

```
$MODULE I_O_REGS
$MEXTNM MPSNEW
$MEXTNM ALU
$MEXTNM ALULATCH
$MEXTNM CNTRSTOR
$MEXTNM CS_REG
$MEXTNM IC_AD_RG
$MEXTNM MAINSTOR
$MEXTNM MAPSTORE
$MEXTNM GP_REGS
$MEXTNM TEMPREGS
$MEXTNM T1750A
$ROUTINE I_O_REGS
$REND
$EXTERN2 IINJECT0X          IINJECT1X          I_O_REGS
$EXTERN2 OEDBIRX            ICLRDBIRX          I_O_REGS
$EXTERN2 GCDBIRX            SERIALX            I_O_REGS
$EXTERN2 GCSIGREGX          TESTMODEX          I_O_REGS
$EXTERN2 OEDBORX            ICLRDBORX          I_O_REGS
$EXTERN2 G0DBORX            CLK1X              I_O_REGS
$EXTERN2 GOOD_MACHX         IDB_N15X           I_O_REGS
$EXTERN2 IDB_N14X           IDB_N13X           I_O_REGS
$EXTERN2 IDB_N12X           IDB_N11X           I_O_REGS
$EXTERN2 IDB_N10X           IDB_N9X            I_O_REGS
$EXTERN2 IDB_N8X            IDB_N7X            I_O_REGS
$EXTERN2 IDB_N6X            IDB_N5X            I_O_REGS
$EXTERN2 IDB_N4X            IDB_N3X            I_O_REGS
$EXTERN2 IDB_N2X            IDB_N1X            I_O_REGS
$EXTERN2 IDB_N0X            IB_N15X            I_O_REGS
$EXTERN2 IB_N14X            IB_N13X            I_O_REGS
$EXTERN2 IB_N12X            IB_N11X            I_O_REGS
$EXTERN2 IB_N10X            IB_N9X             I_O_REGS
$EXTERN2 IB_N8X             IB_N7X             I_O_REGS
$EXTERN2 IB_N6X             IB_N5X             I_O_REGS
$EXTERN2 IB_N4X             IB_N3X             I_O_REGS
$EXTERN2 IB_N2X             IB_N1X             I_O_REGS
$EXTERN4 IB_N0X             I_O_REGS
$LOCAL NNN259064_2_1
$LOCAL NNN316064_2_1
$LOCAL NNN373064_2_1
$LOCAL NNN430064_2_1
```

Example of GGLOSS Macro Output

Figure 2.6.1

```
          MACRO
          DFC %R1,%R2,%I1,%I2,%I3,%L
          .GNAME %R1
          .GNAME %I1
          .GSAVE %R1
          .GNAME %I2
          .GSAVE %R1
          .GNAME %I3
          .GSAVE %R1
          .GNAME %R2
          .GNAME %R1
C
100%$RETVAL          CONTINUE
       %R1 = (((%R1,.AND.(.NOT.%I1,)).OR.(%I2,.AND.%I1))
          .IF %I3,=XXXX NOI3
     1     .AND.%I3
NOI3      .ANOP
     2       )
        %R2 = (.NOT.%R1,).OR.(.NOT.%I3,)
          $TESTNXT %R1
          $TESTNXT %R2
          GOTO %$DISPAT
C
          MEND
```

Example of GGLOSS Macro Definition

Figure 2.6.2

2.7 Event Driven Simulation with the GGLOSS Architecture

An event driven simulation is based on the premise that a logic change at the input to a network propagates forward until it reaches a gate whose output remains unchanged; at this point, it doesn't affect other gates in the network. Because of the nature of Boolean algebra, a large number of gates in a network have outputs that remain unchanged under changing inputs in normal operation. Therefore, time would be saved if only those gates whose inputs changed state were evaluated each clock cycle of the network.

To determine which gates need to be evaluated, one constructs an evaluation list within the simulator. At first, network inputs are examined to see which ones have changed state. Then, gates that have direct connections to network inputs that have changed state are placed on the evaluation list. Gates are chosen from the evaluation list one at a time for computation. As each gate is evaluated, its output is checked against its previous state. If the output has changed state, an event has occurred, and all gates that this gate fans out to are now placed at the end of the evaluation list. This process continues until there are no more gates to evaluate on the list, which will occur if the logic is stable.

In order to implement event driven simulation, GGLOSS requires

- a dispatcher to control computation from the evaluation list, and

- a fanout table to link the evaluation of one gate to those gates it fans out to.

The GGLOSS implementation of event driven simulation adds two new mechanisms to speed up the simulation process:

- The evaluation list is P-ordered. An event driven simulator need not be P-ordered; the fanout table guarantees that proper signal propagation is maintained. However, implementing the evaluation list in P-ordering guarantees the least number of gate evaluations for a given logic network.

- Event driven simulation is mixed with parallel simulation. This means that succeeding gates are evaluated if ANY of the 32 parallel machines have a logic state change, assuming that machine is still active. While this takes longer than straight event driven simulation, it produces a larger number of results.

To implement event driven simulation, GGLOSS contains support in its macro translating stage to build a fanout list from the logic statement macros themselves. Figure 2.7.1 shows an example of the FORTRAN code generated to implement an event driven

simulator.

```
C
C
10259         CONTINUE
      ST(K+83) = (((ST(K+83).AND.(.NOT.NNN301265_2_1)).OR.(NNN409272_2_1
     1.AND.NNN301265_2_1))
     2       )
      TEMP$ = NQ374_85
      NQ374_85 = .NOT.ST(K+83)
      TESTB = ((NQ374_85.XOR.TEMP$).AND.STOPMASK)
      IF(TESTI.NE.0) THEN
         DO 20259 J=1,IARRAY(259,2)
            DISP(IARRAY(259,J+2) = 1
20259    CONTINUE
         ENDIF
         DISP(259)=0
         GOTO 9999
C
C
10263         CONTINUE
      ST(K+84) = (((ST(K+84).AND.(.NOT.NNN301265_2_1)).OR.(NNN449272_2_1
     1.AND.NNN301265_2_1))
     2       )
      TEMP$ = NQ374_86
      NQ374_86 = .NOT.ST(K+84)
      TESTB = ((NQ374_86.XOR.TEMP$).AND.STOPMASK)
      IF(TESTI.NE.0) THEN
         DO 20263 J=1,IARRAY(263,2)
            DISP(IARRAY(263,J+2) = 1
20263    CONTINUE
         ENDIF
         DISP(263)=0
         GOTO 9999
```

Example of GGLOSS Event Driven Simulator

Figure 2.7.1

2.8 Deductive Simulation using GGLOSS

Deductive simulation is a technique in which all faults can be checked for detectability with a single execution of the simulator. In other forms of logic simulation, each fault is translated into an equivalent logic network, and that logic network is simulated. The results of simulating the faulted logic network is compared with a simulation of the good network to determine detectability. In deductive simulation, only a good logic network is simulated. However, at the end of each cycle of the master clock, the states of the good network are checked to see if faults at a each point in the network could propagate and change the state of the network output. If so, they are detectable.

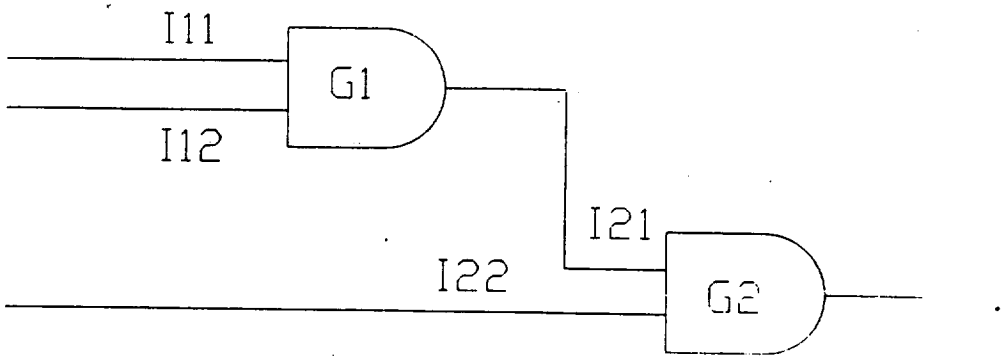
This implies that the simulator should perform two operations each clock cycle of the emulated logic network:

- all gates are evaluated propagating logic values forward from inputs to output in the normal GGLOSS fashion, and

- Detectability is deduced by examining the network backwards checking from its output gates toward its inputs.

Each output gate is checked by assuming a fault at each input pin of that gate. If the fault changes the output state of the gate, it is detectable. Gates that drive the output gates are checked in a like manner except only those faults that change the driver gate outputs to values that are detectable at the output gates are considered detectable. And so on working through the gates until the input gates.

GGLOSS performs deductive simulation by assigning variables to each logic node that indicate both its logic value and its detectability status. Detectability is determined by gate specific logic that performs the backward propagation. Figure 2.8.1 shows an example of the backwards logic for two and gates.



- Detectability Logic for a Two Input AND Gate with an Input Fault:

  IF THE OTHER INPUT IS ONE, FAULT OF THE OPPOSITE LOGIC VALUE ARE DETECTABLE

- In the Above Logic Diagram, Deductive Logic is:

```
G2:  IF (I21 = 1 AND OUTPUT IS DETECTABLE) THEN
        I22 = (FAULTS OF OPPOSITE LOGIC VALUE ARE DETECTED)
     ENDIF
     IF (I22 = 1 AND OUTPUT IS DETECTABLE) THEN
        I21 = (FAULTS OF OPPOSITE LOGIC VALUE ARE DETECTED)
        DISPATCH G1
     ENDIF
G1:  IF (I11 = 1 AND I21 IS DETECTABLE) THEN
        I12 = (FAULTS OF OPPOSITE LOGIC VALUE ARE DETECTED)
     ENDIF
     IF (I12 = 1 AND I21 IS DETECTABLE) THEN
        I11 = (FAULTS OF OPPOSITE LOGIC VALUE ARE DETECTED)
     ENDIF
```

Example of Deductive Logic Algorithm in GGLOSS

Figure 2.8.1

2.9 Postprocessor Requirements

In order for GGLOSS to be a useful tool in statistical fault simulation, a postprocessor must exist to analyze the large amount of data generated by a GGLOSS simulation and produce useful output. To perform the computations, it is necessary that the postprocessor be passed information from the simulator, including:

- the number of frames during which the fault was detected, and the probability of the inputs simulated during those frames occurring,
- the output to the comparator(s) that resulted in detection, and
- the time base and fault set size for the simulation.

One of the main uses of GGLOSS is to study the reliability of computer system architectures; the postprocessor should present to the reliability engineer various quantities of interest. It would be impossible to present a list of statistics and graphical data required for all various studies that GGLOSS can be used to perform. Such a list requires knowing in advance the exact nature of these studies, and the general applicability of GGLOSS to a variety of problems prevents this. It seems more prudent to include the following capabilities in the postprocessor:

- Present basic information, such as probability of a fault being detected and a list of all faults not detected.
- Present information of interest to those studying latency, such as
  - average latency time of all faults,
  - percentage of faults showing the same error at a comparator,
  - histogram of fault latency times,
- Present accuracy and confidence level of the results.
- Compare the results of a GGLOSS run to previous GGLOSS runs showing the change in the above parameters. This might be done in histogram form to illustrate changes that occur when a particular system design parameter is varied.
- Present data on disk that can be used by an external statistical analysis package to generate other quantities of interest (or graphical data) without rewriting the postprocessor.

3. USE OF GGLOSS TO STUDY LATENT FAULTS

3.1 Causes of Failure in Highly Reliable Systems

Before analyzing the reliability of a flight control system using a simulation methodology, an examination of its architecture is necessary. Most implementations utilize redundant sensor modules connected through A/D converters to the control computer module. The outputs of the processors are connected through D/A converters to actuator modules and then to the control surfaces utilizing redundancy where required to maintain reliability.

In some architectures, each sensor/processor/actuator combination represents a separate and autonomous channel, communicating with the other two or three channels for failure management purposes only. Monitoring is done at selected points within the channel, e.g. at the output of the sensors and the output of the processors. The number of monitoring points (sometimes called voting planes) in each channel is a function of the reliability of each module within that channel and the ability of the system to reconfigure without that module. The values of each monitoring point is compared across all the channels. If one channel disagrees with the other channels at a particular point, the output feeding that point is disconnected and succeeding modules receive their input from a functioning neighbor. Additional hardware modules called comparators perform the comparison and fault isolation functions, and they must be able to properly handle their own failures.

In other architectures, each monitor point contains a voter module connecting it to succeeding module inputs. The voter receives signals from the same points in all the channels and transmits a value according to its algorithm (such as mid value) which is correct even if a single module fails. The result is that the failure of a single module does not affect the operation of the system in a manner perceivable to the pilot or passengers. Other architectures exist in which the comparators and/or voters are implemented in software, using interprocessor communications busses.

While monitoring may occur at many points within the channel, the focus for the studies conducted for this report is monitoring at the output of processors embedded within the channel. A block diagram of such a processing system is shown in Figure 3.1.1. To simplify the studies, the comparator is assumed to be implemented in hardware, as software comparators require resolving interactive consistency problems [16].

With this architecture, the factors that may affect system reliability, in order of decreasing importance, are:

COMPARATOR COVERAGE

It is apparent that the above architectures are heavily dependent on the ability of the comparators/voters to handle any errors that might occur. When errors exist that are not detected and corrected properly, the failure rate of the system is low, even if the number of redundant modules is large. The probability that an error is detected by a comparator is referred to as its coverage. This problem was explored by Bavuso [4]. As an example, consider a channel consisting of a processor whose MTBF is 1000 hours. Under the assumption of perfect coverage, it is possible to build a triplex system with a probability of system failure of 10^{-9} for a one hour flight. However, if the coverage of the comparator in this system is only .99, the probability of not handling properly one faulted processor in the one hour flight is 10^{-5}. Even if the comparator coverage is raised by an order of magnitude, the probability of system failure is still far from acceptable. It is noted that this is not a problem that can be solved by adding more channels - in fact, comparator coverage dictates the highest reliability that can be attained through redundancy.

COVERAGE OF SECONDARY DETECTION MEANS

If the coverage of the comparator can be increased sufficiently, then other factors become dominant in determining system reliability. In a triplex system with perfect comparator coverage, fault isolation may become a significant factor. When more than two processors are available in the system, the comparator mechanism will always properly isolate a single fault to the processor in which it occurs. However, when only two processors are left operative, a secondary fault isolation means, such as self-test, must be used to determine which processor is faulty. The probability of correct fault isolation by this secondary means is a function of its coverage, and is usually on the order of .95. The probability of recovery from a second fault is reduced by this imperfect fault isolation. However, with quad and quintiplex systems, this is not a significant problem until the third or fourth failure.

FAULT LATENCY

With perfect coverage of the comparator and secondary detection means fault latency becomes the dominant factor in determining system reliability. A highly redundant system has a greater total hardware component count and therefore a greater probability of sustaining a fault in one of its components. If the fault is detected immediately, then reconfiguration occurs, deleting the faulted processor. However, in a computer based control system voting only occurs at the end of each control iteration (frame) and the probability of detecting a random fault occurring during that iteration is considerably less than one (.5 to .7 as measured by experiments [6]). The fault may be detected on the next iteration, or on some subsequent iteration; the average time until detection is different for each fault in the processor

and is dependent on the software being executed and on the flight maneuvers.

If the fault remains latent for a significant amount of time, it is possible for a second latent fault to occur. The two latent faults may possess different propagation characteristics, and one may reach the output of a processor before the other one does. In this case, the system will recover successfully.

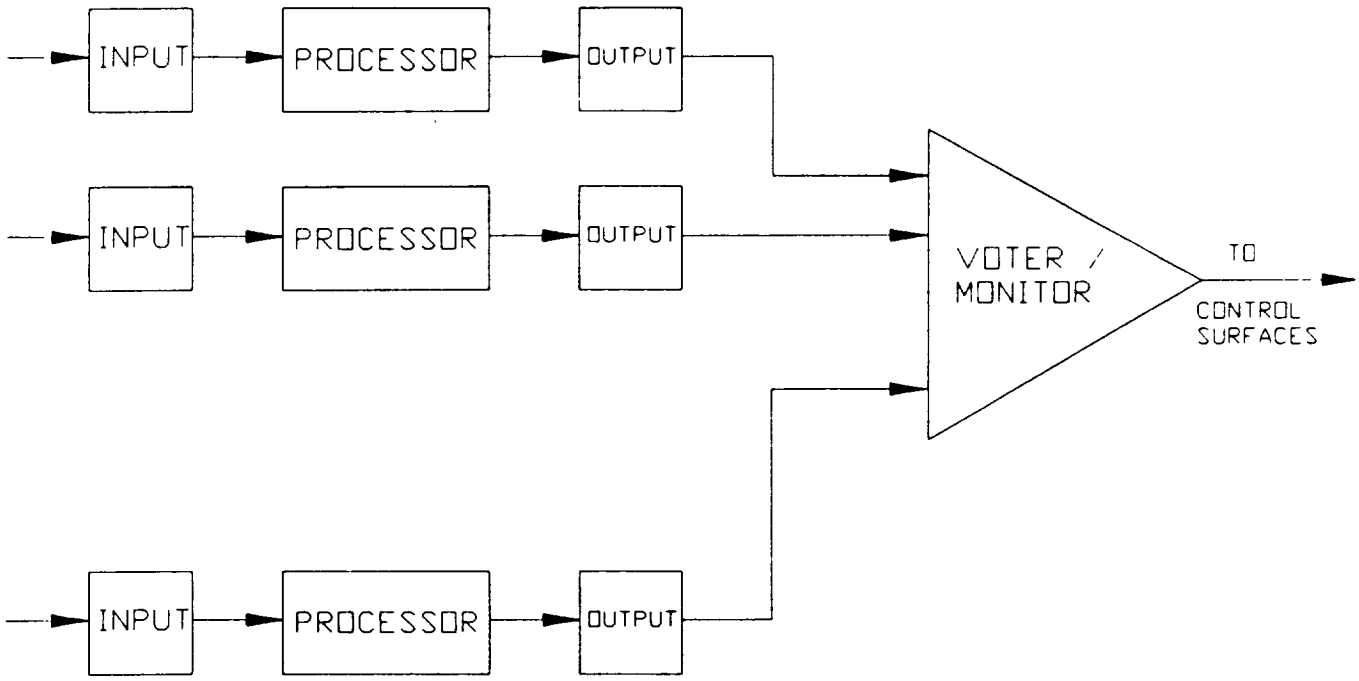
It is also possible for both latent faults to have a common excitation means and reach the comparator at the same iteration. If this occurs, then two possibilities exist:

- both latent processors present the same output to the comparator. The comparator is defeated if this is a triplex system. If this is a quad system, it will see two differing sets of outputs and it must use a secondary means to decide which processors are operating correctly. Many quad systems are not architected with a secondary fault isolation means.

- each latent processor presents a different output to the comparator. If this is a triplex system, it can recover by initiating its secondary fault isolation means. Quad systems could recover directly; however, the reconfiguration algorithm must remove both faulted processors from the system.

NEAR-COINCIDENT FAULTS

Once the fault is isolated, the system must reconfigure without the failed processor. The system is likely to fail if another fault occurs while it is reconfiguring from the first fault. If reconfiguration time is significant, the probability of failure during this exposure time can be calculated as was done in [7]. Note that this definition of near-coincidence is different that of some authors, who consider latent faults to also be near-coincident.



Block Diagram of an N Module Redundant Configuration

Figure 3.1.1

3.2 Latent Fault Scenario

One important use of GGLOSS is to investigate the probability of system failure due to latent faults. Latent faults are a significant factor in ultrahigh reliability configurations. If some parameters associated with latency can be measured, models could be constructed to predict the probability of failure of such a system. While analytic expressions for latent parameters are extremely complex, measuring their value using GGLOSS provides a usable technique for reliability estimation.

In order to discuss latent faults, some formal definitions are required:

Fault Let the processor be represented by a set of components G_1, G_2, ..., G_k and an interconnection mapping of the components in G. Each member in G has a functional model defined in terms of its inputs and outputs. A fault is defined as a malfunction of one of the members of G such that its functional model has been altered.

Each member of the set G has an associated set containing all possible alterations under consideration in a particular reliability assessment. This set is referred to as the fault model.

Failure The propagation of a fault so that the output of the processor is erroneous will be termed a failure of that processor, a failure of the channel containing that processor, or simply, a failure.

System failure The propagation of an erroneous value past the comparator to a surface actuator for at least one iteration of the control program due to a failure in one or more channels of the system will be considered a system failure.

Detection The successful detection and isolation of a failure to its originating channel by a voting comparator or a self-test program and the subsequent disconnection of that channel will constitute detection.

Fault Latency The time from the occurrence of a fault, as might be measured by a suitably placed test probe within the processor (if placement of such a probe were possible), until its subsequent appearance at a comparator will be called the fault latency time.

Indistinguishable Faults Those faults whose latency times can be shown to consistently be infinite will be termed indistinguishable. All other faults will be termed distinguishable.

Previous work in reliability estimation for flight control systems has often utilized a worst case approach. It is assumed that all latent faults result in system failure (sometimes referred to as malicious excitation). If the overall reliability under this

assumption is still acceptable, it follows that the actual system will be more reliable. Bavuso [4] constructed a TMR model based on this premise. He showed that the overall reliability of a TMR system is strongly dependent on the coverage of both the comparator and the secondary detection means. However, as the comparator deviates slightly from perfect coverage, the reliability of the system decreases drastically. Nagel [5] and McGough and Swern [6] showed that the coverage of a comparator in triplex systems is probably considerably less than perfect.

3.3 Latency Effects as a Function of Architecture

Markov models for the triplex and quad systems were constructed and are given in Figures 3.3.1 and 3.3.2 respectively. The probability that two latent faults result in the same bad output for a given input was measured using GGLOSS and was seen to be approximately .1. This value was used in the Markov model to estimate the probability of system failure due to latency for various average latency times. The results obtained from these model are shown in Figure 3.3.3. From the table, the following results were indicated:

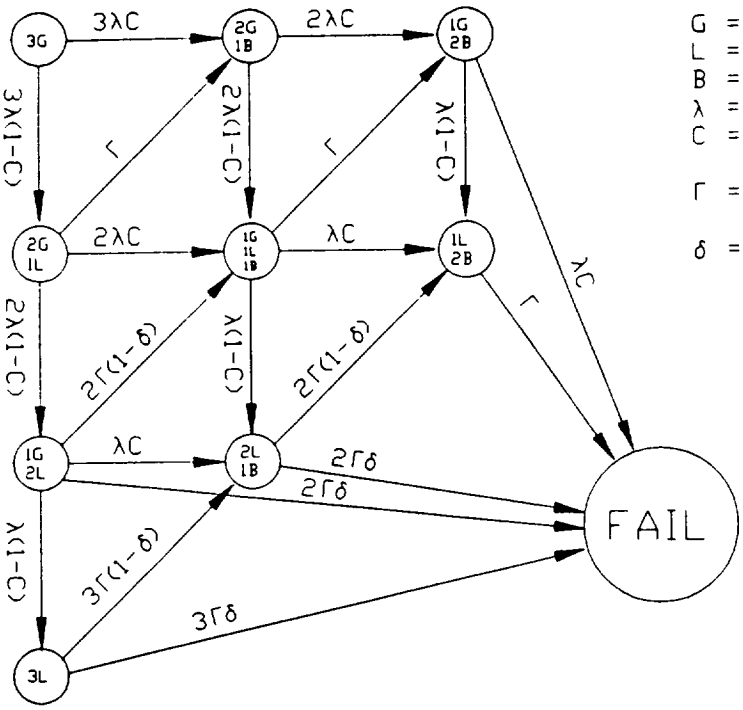
- in the triplex system, probability of system failure was not significantly affected by small latency times until the average latency time increases to 30 seconds. The worst effects of latency occurred when the average time was 1 - 2 hours, and reliability decreased by an order of magnitude.

- in the quad system, if the comparator could not recover from a tie vote, system reliability is significantly affected for latency times as low as 1 second. Reliability decreased by an order of magnitude. When the comparator initiates a secondary fault isolation upon a tie vote, this situation is significantly improved (by a factor equivalent to the coverage of the fault isolation means). The worst effects of latency occurred again when the comparator could not recover from a tie vote and the average latency time was 1-2 hours; the reliability of the quad system was decreased below that of the triplex system!!!

Latent faults can have a significant effect on the reliability of a redundant processing system. The extent of this effect must be carefully measured to insure that it is not dominant cause of system failure. From the Markov models it was concluded that:

- Fault latency can become the limiting factor in the construction of ultra-reliable computing system when the desired probability of system failure is less than 10^{-9}.

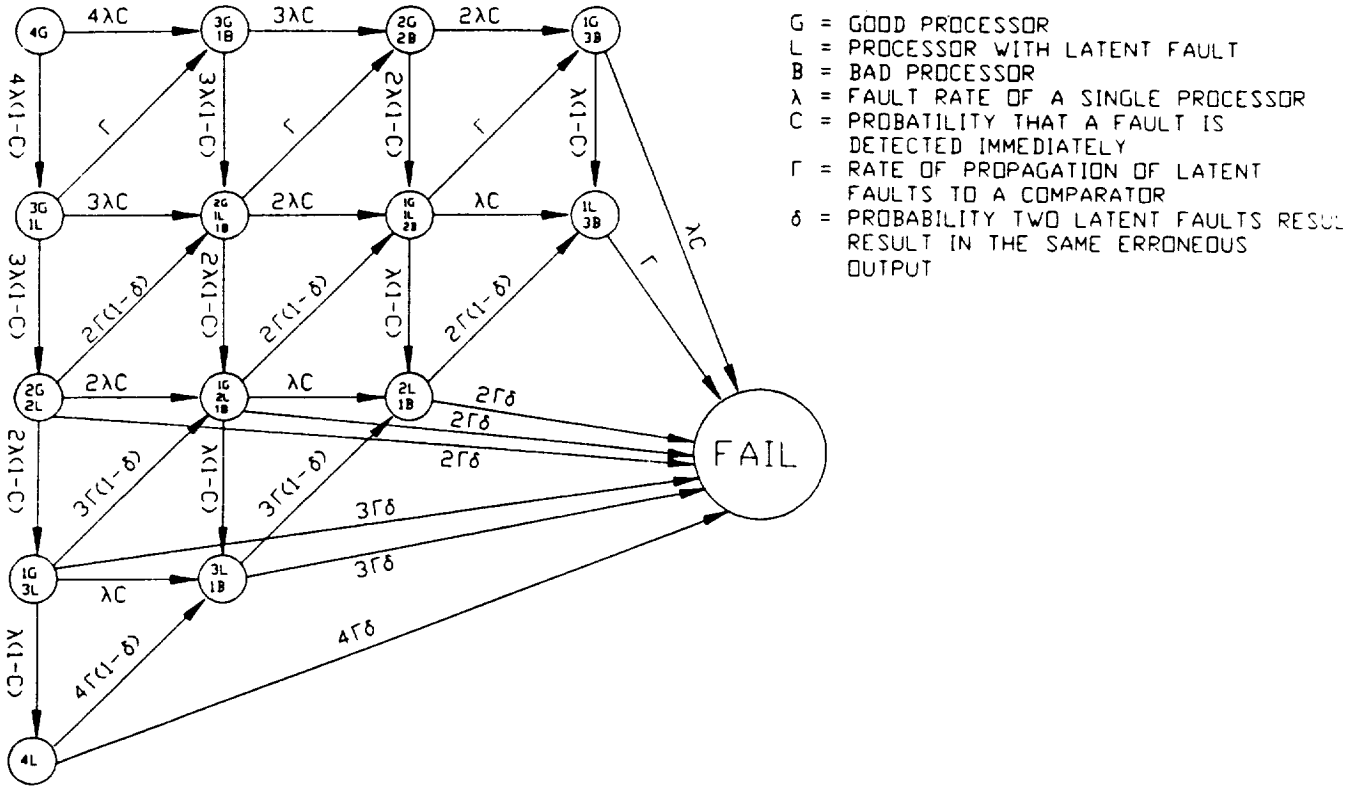
- When latency effects are significant, even the benefits of a fourth processor in improving system reliability may be lost. Extremely long latency times cause the quad system to become less reliable than the triplex.



G = GOOD PROCESSOR
L = PROCESSOR WITH LATENT FAULT
B = BAD PROCESSOR
λ = FAULT RATE OF A SINGLE PROCESSOR
C = PROBATILITY THAT A FAULT IS DETECTED IMMEDIATELY
Γ = RATE OF PROPAGATION OF LATENT FAULTS TO A COMPARATOR
δ = PROBABILITY TWO LATENT FAULTS RESUL RESULT IN THE SAME ERRONEOUS OUTPUT

Markov Model of a Triplex System Including Latency But Without Repair

Figure 3.3.1



Markov Model of a Quad System Including Latency But Without Repair

Figure 3.3.2

PROBABILITY OF FAILURE AS A FUNCTION OF LATENCY

FOR δ = .1

	C = .5	C = .7	C = .85
TRIPLEX			
Γ = 2/HR	2.9 E -8	1.1 E -8	3.3 E -9
Γ = 1/MIN	3.4 E -9	1.8 E -9	1.2 E -9
Γ = 1/SEC	1.0 E -9	1.0 E -9	1.0 E -9
QUAD			
Γ = 2/HR	5.7 E -8	2.0 E -8	5.0 E -9
Γ = 1/MIN	4.8 E -9	1.7 E -9	4.4 E -10
Γ = 1/SEC	8.7 E -11	3.1 E -11	8.6 E -12

FOR δ = .2

	C = .5	C = .7	C = .85
TRIPLEX			
Γ = 2/HR	5.7 E -8	2.1 E -8	5.9 E -9
Γ = 1/MIN	5.8 E -9	2.7 E -9	1.4 E -9
Γ = 1/SEC	1.0 E -9	1.0 E -9	1.0 E -9
QUAD			
Γ = 2/HR	1.1 E -7	4.1 E -8	1.0 E -8
Γ = 1/MIN	9.0 E -9	3.5 E -9	8.7 E -10
Γ = 1/SEC	1.6 E -10	6.1 E -11	1.6 E -11

C = PROBABILITY THAT A FAULT IS DETECTED IMMEDIATELY

Γ = RATE OF PROPAGATION OF LATENT FAULTS TO THE COMPARATOR

δ = PROBABILITY THAT TWO LATENT FAULTS RESULT IN THE SAME BAD OUTPUT

Figure 3.3.3

3.4 Description of Latent Fault Studies

Experiments on a sophisticated processing system running a complete flight program are impractical at the present time because of system complexity and the large number of factors that affect the interpretation of the results. However, a simplified processor (herein referred to as the "toy processor") is available that is used for these studies while it runs an extremely simple program representing the flight controller. It is hoped that the results of these studies will furnish data that may be used to construct better model of reliability for a more complex system.

The current set of experiments were performed independent of dynamic flight maneuvers, i.e. inputs are assumed to be uncorrelated. All variations in flight path are due to a noise input such as mild turbulence, etc. While a particular flight maneuver might imply a change in control law to another mode of operation, this is not being modeled at this time.

The toy processor contains approximately 1600 possible faults comprising its fault set. The control program to be run is given in Figure 3.5.1. This program contains a single input and a single output. The output will be considered to drive a comparator while the input will take on random values. The methodology for the experiments was to run each of the 1600 faults with as many different input variable sequences as possible. The program will be run for a small number of control algorithm iterations and the time (in iterations) that the faulted system output disagrees with the output from a nonfaulted machine will be recorded (detection time) if the fault is detected. The data will then be used to calculate the quantities of interest.

3.5 Probability of System Failure Due to Latent Faults

Definition of Parameters Associated with Latency

Consider the aircraft in automatic control terminology, with the processor and its software together forming the controller and the aircraft and its environment represent the controlled vehicle or, in modern control parlance, the plant. Together, the plant and controller form an autonomous, stochastic and discrete system. If the environment could be frozen and repeated for every flight, then the aircraft path from take-off to landing would be deterministic. Random variations in the path that are observed from flight to flight come from turbulence, electrical noise and other phenomena the aircraft might experience in normal operation.

Such a system can be represented on a suitable discrete state space with the dynamics of the system represented by a suitable transformation function. This function maps the state space onto itself for each iteration of the control program. Let

$$J = \{ 1 , 2 , \ldots , M \} \tag{3.5.1}$$

represent the set of numbers that can be stored in a register or memory location, where M is the largest such unsigned integer. The control equations operate on a state vector X of dimension n, where

$$X \in J^n . \tag{3.5.2}$$

A system defined on X without any input would be a purely deterministic system. In order to introduce the normal variations that occur during a flight, let W represent a random input to the state equations. Let it be further assumed that the system is time invariant. This is not a strict requirement in the work that is to follow; however, it simplifies the resulting expressions and obviates the need for an absolute time base. The system equations are

$$\underline{X}(k+1) = g(\underline{X}(k),W(k))$$

$$Y(k) = h(\underline{X}(k)) \tag{3.5.3}$$

where

$W \in J$ is a random variable representing noise,

$Y \in J$ is the comparator input,

$g: J^{n+1} \rightarrow J^n$ is a mapping that takes the processor from its present state into the next state, and

$h: J^n \rightarrow J$ is a mapping that relates the comparator input to the state variables.

It can be seen that g is a nonlinear function dependent upon the processor architecture, the software, the control algorithm, and the response of the aircraft.

There exists no general form for solutions to (3.5.3). However, the systems under consideration are realizable which implies the existence of a transition function ϕ which takes $\underline{X}$ from a particular set of initial conditions at a particular instant to some future point on the flight path. Let W represent a particular noise history, and let j represent a particular iteration of the control program with associated state vector $\underline{X}(j)$. Then the output at the comparator after m iterations from j can be obtained from the relation

$$Y(j+m) = h(\phi(\underline{X}(j),W,m)). \qquad (3.5.4)$$

Let us assume that a fault occurs at iteration j. This causes an alteration of the system into a new system, although the new system is still defined on the same state space (some states in the original space may be unreachable by the new system due to the fault). The mappings g, h, and ϕ, are changed by the fault so that

$$\underline{X}(k+1) = g_n(\underline{X}(k),W(k))$$

$$Y(k) = h_n(\underline{X}(k)) \qquad (3.5.5)$$

where g_n and h_n are the mappings for a particular fault n chosen from the space of all possible faults of the processor, η. For completeness, let $n=0 \in \eta$ be the unfaulted state.

Let n1 and n2 be processor faults, $n1,n2 \in \eta$. Let a comparator be connected between two like processors at Y, one processor containing fault n1, and one processor containing fault n2. Then the Detection Function is given by

$$D_{n1}(\underline{X},W,n2,m) = 0 \text{ if } h_{n1}(\phi_{n1}(\underline{X}(j),W,m)) = h_{n2}(\phi_{n2}(\underline{X}(j),W,m)) \qquad (3.5.6)$$

$$= 1 \text{ if } h_{n1}(\phi_{n1}(\underline{X}(j),W,m)) = h_{n2}(\phi_{n2}(\underline{X}(j),W,m)).$$

If n1 = 0, then $D_0(\underline{X},W,n,m)$ is the Detection Function for comparing the output of a good processor to the output of a processor with fault n. Further, the Latency Time of a Fault is defined by

$$T(\underline{X},W,n) = \min \{ m \mid D_0(\underline{X},W,m,n) = 1 \}. \qquad (3.5.7)$$

Let χ represent the upper bound of the space of all possible states of the system in a normal flight, and Ω represent the upper bound of the space of variations of the noise W. A latent fault is one whose latency function L(n) is one, where L(n) = 0 if $D_0(X_k,W_l,n,1) = 1$ for all $k\epsilon\chi$, $l\epsilon\Omega$ and L(n) = 1 otherwise. If

$$p(f_n) = \text{failure rate of each component}$$

λ = failure rate of the processor

then the rate of occurrence of latent faults is

$$\lambda_{FL} = \sum_{n \epsilon \eta} p(fn)\ L(n).$$

Then the expected value of T can be computed for each fault n as

$$\tau(n) = \sum_{k=1}^{\chi} \sum_{l=1}^{\Omega} p(\underline{X}_k)\ p(W_l)\ T(\underline{X}_k, W_l, n)\ L(n) \qquad (3.5.8)$$

and the overall expected value of latency for the system is just the weighted sum of the latency times for each fault

$$\bar{\tau} = (1 / \lambda_{FL}) \sum_{n=1}^{\eta} p(f_n)\ t\ (n) \qquad (3.5.9)$$

where

λ = probability of a fault occurring in a processor.

Consider the effect of a latent fault on the probability of failure of a redundant system. McGough [7] discusses the probability of a second fault occurring before recovery from the first fault is complete. Because the recovery time is small, he considers the faults to be concurrent. With latent faults the computations are similar except that latency time can be much longer than recovery time.

Two faults will be considered similar when the first erroneous value that reaches the comparator after the fault occurs is the same for both. By previous definition, the existence of two failures giving similar signals to the comparator for at least one iteration leads to a failure of the system. This is somewhat conservative because, unless the inputs to the comparator are equivalent for a large number of iterations, it is possible for the dual faults to be detected before a harmful output arrives at the aircraft surface. When the latency time of the first fault is nonzero, the probability of failure can be approximated by the product of two terms

$$P(\text{ failure }) = P(\text{ 1st })\ P(\text{ 2nd } \mid \text{ 1st }) \qquad (3.5.10)$$

where

P(1st) = probability that a fault occurs and is not detected, and

P(2nd | 1st) = probability that a second similar fault occurs given that the first fault occurred and was not detected.

The probability that the first fault occurs and is not detected in a NMR system is simply

$$P(\text{ 1st }) = N\,\lambda\,(\,1 - C1)\,\Delta T \qquad (3.5.11)$$

where

ΔT = time from the beginning of the flight

C1 = percentage of faults detected by the comparator immediately after they occur

N = number pf redundant modules in the system.

A more formal definition of C1 appears later in conjunction with triplex systems. In (3.5.11) it is assumed that a constant value for C1 can be fount that characterizes the system under consideration. The average time it takes to detect a fault can be computed from (8). Then the conditional probability required for (3.5.10) is

$$P(2nd \mid 1st) = (N-1)\,(1-C1)\sum_{n=1}^{\eta} P(f_n)\,\tau(n)\,\delta(n) \qquad (3.5.12)$$

where $\delta(n)$ is a function that denotes the probability of a new fault giving the same output at the comparator as the old fault. More formally,

$$\delta(n) = \sum_{k=1} P(\underline{X}_k) \sum_{l=1} P(W_l) \frac{\sum_{\substack{i=1 \\ i\neq n}}^{\eta} P(f_i)\,(1-D_n(\underline{X}_k,W_l,i,1))}{\sum_{i=1}^{\eta} P(f_n)\,D_0(\underline{X}_k,W_l,i,1)} \qquad (3.5.13)$$

A more useful quantity may be defined in terms of the overall probability of a second fault giving the same output as the first. Let

$$\sum_{n=1}^{\eta} P(f_n)\,\tau(n)\,\delta(n) = \lambda\,\bar{\tau}\,\bar{\delta} \qquad (3.5.14)$$

where $\bar{\tau}$ is as defined in (9), and $\bar{\delta}$ is a new quantity measuring the expected value of $\delta(n)$. $\bar{\delta}$ is considered defined by (3.5.14). Then the overall probability of failure due to a latent fault is

$$P(\text{ failure }) = N\,(N-1)\,\lambda^2\,(1-C1)^2\,\Delta T\,\bar{\tau}\,\bar{\delta}. \qquad (3.5.15)$$

Experiments Measuring Parameter Values

Using the processor and software enumerated in section 3.4, a study was performed to obtain some values for the parameters enumerated above. The toy processor was simulated using GGLOSS while it executed a simple flight control program. The program is given in Figure 3.5.1. The system architecture in which the processors operate is shown in Figure 3.5.2. The characteristics of this architecture are used to compute system reliability.

In this architecture, each processor operates autonomously and outputs to a hardware monitor which consolidates its inputs to a single value which in turn drives the control surface. The comparator contains a majority voter that prevents a failed channel from driving the control surface, and a comparator which disengages the failed channel from the system. It is assumed that the monitor is capable of detecting and isolating all failed channels that present an output to the monitor that is different from the output of a good channel. That is, the hardware coverage of this comparator is one hundred percent.

Because of the autonomy of each processor and the symmetry of the architecture, it is unnecessary to simulate more than one processor of the n-plex to determine its failure characteristics. The output of this faulted processor is compared with the output of a good processor at the end of each control program iteration. If the outputs are the same, the fault has not been detected. If the outputs differ, detection has occurred. Once detection occurs, the experiment ends.

The methodology used in the experiments was to inject a fault into the processor, and iterate the flight control program with a random number as an input. The output is then compared with the output of a good processor running the same problem. The results are recorded. The processor is then reinitialized and the process is repeated with another random number.

This process was repeated for 128 unique random inputs. The flight control program was written to have no "memory", i.e. each computation is independent of the results of the previous computation. The program has two inputs of eight bits; hence the study represents only a small portion of the 65,536 possible input combinations that could be presented to the program. However, the 128 inputs exercised half of the possible values that could be presented to the eight bit bus structure of the processor which made this a significant exercise of its hardware.

All gate faults were considered equally probable for this study. While GGLOSS is capable of accepting different failure rates for each gate type, the values were all set the same.

The output of the simulation is summarized in Table 3.5.1, which contains:

- the number of iterations for which the fault was detected from

a total of 128 iterations with different inputs,

- the average number of iterations it took to detect this fault, which is just the previous number divided by 128. This number is an estimator for $\tau(n)$ as given by (8),

- when the fault output an erroneous value to the comparator, the average number of faults that output the same value to the comparator for the same input. This number is an estimator for $\delta(n)$ as defined by (3.5.13).

The value of $\delta(n)$ is obtained by first computing $\delta(\underline{W},n)$ for each successful detection. GGLOSS records the erroneous outputs of each channel for each input. $\delta(\underline{W},n)$ is computed by counting the number channel outputs that are the same for a particular input and dividing that by the total number of erroneous outputs for that input. It is assumed that all inputs are equiprobable. Then $\delta(n)$ is just the average value of all the $\delta(\underline{W},n)$ computed for this fault.

The faults can be grouped by iteration time to obtain a latency table. This was done, and is shown in Table 3.5.2. Using the data in the table and (3.5.9), one obtains

$$\overline{\tau} = 4.18 \text{ iterations},$$

and, from (3.5.14)

$$\overline{\delta} = .0706.$$

From the table, it is noted that the probability of detecting a fault on the first iteration after it occurs is .489. This is the quantity C1 referred to in (3.5.11). Let it be further assumed that the iteration rate of the processor is ten times per second and the duration of the flight is one hour. Then from (3.5.15) the probability of system failure due to latency for a triplex system with a failure rate of .001 is

$$P \text{ (Failure Due to Fault Latency)} = 1.16 \times 10\text{-}11.$$

Now this is not a significant factor in system reliability, even for a commercial flight. However, if for the same system the average detection time were to increase to two minutes due to environmental conditions, then

$$P \text{ (Failure Due to Fault Latency)} = 1.55 \times 10\text{-}9.$$

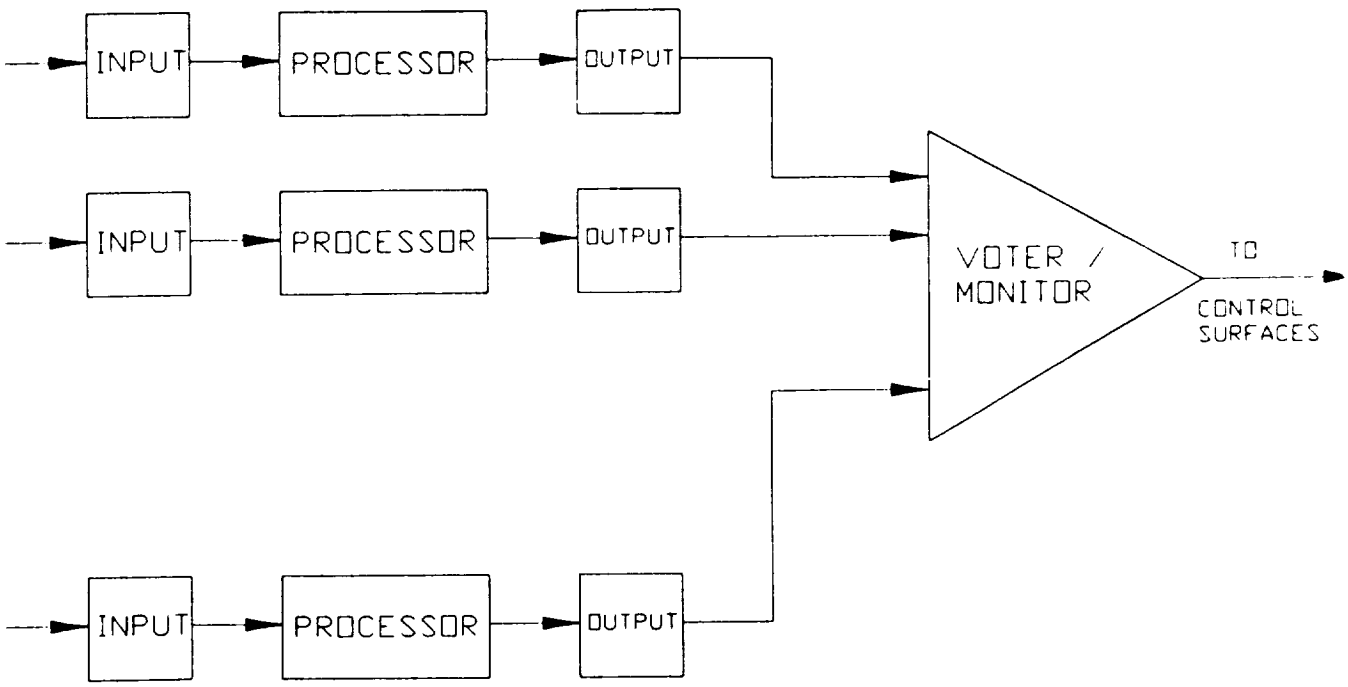
Table 3.5.2 also shows that the percentage of faults that are never detected is 20.7. It was desired to analyze why these faults were not detected. This was done, and is given in Table 3.5.3. In general, these faults can be grouped into four categories:

- faults that would be detected if more inputs were considered,
- faults that will never be detected by this program, but could be detected by some other program,
- faults that represent overdesign, etc. and will never be detected, and
- faults that are anomalies of the simulation and don't really exist.

```
00:     60      START:  STA     TEMP
01:     A0              SUB     TEMP        ;  CLEAR ACCUMULATOR
02:     F0              ADD     IPORT       ;  GET SENSOR INPUT
03:     B0              SUB     IPORT       ;  NEW - OLD
04:     61              STA     OUT         ;  NEW OUTPUT VALUE
;
05:     60              STA     TEMP
06:     A0              SUB     TEMP        ;  CLEAR ACCUMULATOR
07:     E1              ADD     OUT         ;  CAN BE NO OVERFLOW
08:     71              STA     OPORT       ;  OUTPUT VALUE
09:     00              BNO     START
;
;       VARIABLES
;
20:     00      TEMP:   DC      0
21:     00      OUT:    DC      0
;
;       PORTS
;
        30      IPORT   EQU     48
        31      OPORT   EQU     49
```

Simple Flight Control Program

Figure 3.5.1



NMR Architecture

Figure 3.5.2

FAULT			NO. DETECTS	AVG TIME (ITER.) $\tau(n)$	FAULTS WITH SAME OUTPUT (PERCENTAGE) $\delta(n)$
D3U38:	A*	0	128	1.00	
D3U38:	A*	1	128	1.00	
D3U38:	B*	0	128	1.00	
D3U38:	B*	1	0	0.00	
D3U38:	Y*	0	128	1.00	
D3U38:	Y*	1	128	1.00	
D4U38:	A*	0	128	1.00	
D4U38:	A*	1	0	0.00	
D4U38:	B*	0	128	1.00	
D4U38:	B*	1	0	0.00	
D4U38:	Y*	0	0	0.00	
D4U38:	Y*	1	128	1.00	
D1U27:	A*	0	86	1.49	0.03
D1U27:	A*	1	53	2.42	0.04
D1U27:	Y*	0	53	2.42	0.04
D1U27:	Y*	1	86	1.49	0.03
D1U28:	A*	0	128	1.00	
D1U28:	A*	1	76	1.68	0.06
D1U28:	Y*	0	76	1.68	0.06
D1U28:	Y*	1	128	1.00	
D1U29:	A*	0	44	2.91	0.10
D1U29:	A*	1	26	4.92	0.08
D1U29:	Y*	0	26	4.92	0.08
D1U29:	Y*	1	44	2.91	0.10
D1U30:	A*	0	128	1.00	
D1U30:	A*	1	93	1.38	0.09
D1U30:	Y*	0	93	1.38	0.09
D1U30:	Y*	1	128	1.00	
D2U27:	A*	0	128	1.00	
D2U27:	A*	1	20	6.40	0.04
D2U27:	Y*	0	0	0.00	
D2U27:	Y*	1	20	6.40	0.04

Tabular Postprocessor Output for Flight Control Program

Table 3.5.1

TIME (ITERATIONS)	FAULTS	PERCENT OF TOTAL
UNDETECTED	351	20.7
ALWAYS DETECTED	829	48.9
2	220	13.0
3	49	2.9
4	28	1.7
5	6	.4
6	55	3.2
7	15	.9
8	29	1.7
9	26	1.5
10	13	.8
11	9	.5
12	11	.6

Latency Distribution of the Faults

Table 3.5.2

NUMBER OF UNDETECTED FAULTS	351	20.17%
- NOT UTILIZING ALL OF RAM/ROM	140	
- NO INPUT VALUE STIMULATED FAULT	73	
- BRANCH INSTRUCTION NOT USED	70	
- SIMULATION ANOMALIES	44	
- MEMORY SHUT OFF DURING UNUSED CYCLES	12	
- OVERDESIGN OF PROCESSOR	12	
TOTAL NUMBER OF FAULTS INJECTED	1692	

Resolution of Undetected Faults

Table 3.5.3

3.6 Latency Characteristics of a Processor/Program Combination

Development of a Markov Model

A large portion of the flight regime consists of the cruise mode, which is a holding or tracking mode. The deterministic component of the system response is invariant; the perturbations occur due to noise and turbulence. The system will operate in a fixed region of the trajectory $\underline{X}$ with variations due to W; for the work that is to follow, these variations will be considered gaussian. The remaining equations in this section will be derived independent of $\underline{X}$. The probability of the program detecting a failure in the first iteration of the control algorithm after the fault occurs is

$$P(D) = (1/\lambda) \sum_{n=1}^{\eta} \sum_{l=1}^{\Omega} P(f_n)\ P(W_l)\ D_0(W_l,n,1). \qquad (3.6.1)$$

Let a new measure of fault detection detectability be defined as

$$D^*(n) = \begin{cases} 1, \text{ if } \max_{k>0} \left| \sum_{l=1}^{\Omega} D_0(W_l,n,k) \right| > 0 \\ 0, \text{ if } \max_{k>0} \left| \sum_{l=1}^{\Omega} D_0(W_l,n,k) \right| = 0. \end{cases} \qquad (3.6.2)$$

Then the probability that a fault will be detected at all is

$$P = \sum_{n=1}^{\eta} P(f_n)\ D^*(n) \qquad (3.6.3)$$

where (1-P) is the probability that a fault is indistinguishable.

As a basis for modeling system behavior, consider that a large number of faults are sensitive to only one bit in a particular component of $\underline{X}$. If this bit varies randomly, then the probability of detecting a fault with this sensitivity is .5 for each iteration. Other faults are sensitive to a combination of two bits. Their probability of detection is .25 for each iteration. The concept may be extended so that a class of faults θ_i is defined as

$$f_n \in \theta_i \text{ if } \sum_{l=1}^{\Omega} P(W_l)\ D_0(W_l,n,1) = 1/i$$

$$f_n \in \theta_0 \quad \text{if} \quad \sum_{l=1}^{\Omega} P(W_l)\, D_0(W_l, n, 1) = 1 \tag{3.6.4}$$

where $i \in \Phi$.

The probability that a fault belongs to class i is

$$P_i = (1/\lambda) \sum_{n \epsilon \theta_i} P(f_n) \tag{3.6.5}$$

so that (3.6.1) may be rewritten as

$$P(D) = P_0 + \sum_{\substack{i \epsilon \Phi \\ i>0}} P_i/i. \tag{3.6.6}$$

From (3.6.4) - (3.6.6) one may construct a four state Markov model of this process that is identical to the urn model constructed by Nagel [5]. The model is shown diagrammatically in Figure 3.6.1. Define

$$P' = P(D)/P \; ; \quad \sigma = (P(D) - P0)/(P - P0) \tag{3.6.7}$$

Let F represent the cumulative probability of detection in any iteration up to k

$$F_1(D) = P\, P'$$

$$F_2(D) = P\, P' + P\, (1-P')\, \sigma$$

$$F_3(D) = P\, P' + P\, (1-P')\, \sigma + P\, (1-P')\, (1-\sigma)\, \sigma$$

$$F_k(D) = P\, P' + P\, (1-P')\, \sigma + P \sum_{n=1}^{k-2} (1-P')\, (1-\sigma)^n\, \sigma \tag{3.6.8}$$

A more interesting model can be constructed with five states where the fault class $\theta 2$ represents a state, and the remainder of fault classes are combined to form the remaining state. Now define the auxiliary parameters

$$P' = P(D)/P \quad ; \quad P'' = (P(D) - P_0 - P_2/2)/(P - P_0 - P_2)$$

Then

$$F_1(D) = P\, P'$$

$$F_2(D) = P\, P' + P\, (1-P')\, (P_2/2 + (1-P_2)\, P'')$$

$$F_k(D) = P\, P' + P\, (1-P')\, P_2 \left[1/2 + \sum_{n=1}^{k-2} (1/2)^n\, 1/2\right] +$$

$$P\ (1-P')\ (1-P_2)\ [P'' + \sum_{n=1}^{k-2} (1-P'')^n\ P'']$$ (3.6.9)

A Markov model representing the above equations is shown in Figure 3.6.2.

The Markov diagram can be extended to more states. Define the auxiliary function

$$\beta(k,\alpha) = \alpha + \sum_{n=1}^{k-2} (1-\alpha)_n\ \alpha$$ (3.6.10)

Now let θ3, θ4, θ5, etc. represent states in the Markov diagram. Let Φ* represent the subset of Φ that are states in the diagram, noting that θ0 is not a member of Φ*. Now form equations for this model corresponding to (3.6.9):

$$P' = P(D)/P \quad ; \quad P^* = \sum_{i\epsilon\Phi^*} P_i$$

$$P'' = (P(D) - P_0 - \sum_{i\epsilon\Phi^*} P_i/i)/(1 - P_0 - P^*)$$

$$F_1(D) = P\ P'$$

$$F_2(D) = P\ P' + P\ (1-P')\ [\ (1-P^*)\ P'' + \sum_{i\epsilon\Phi^*} P_i\ /i\ ]$$

$$F_k(D) = P\ P' + P\ (1-P') \sum_{i\epsilon\Phi^*} P_i\ \beta(k,1/i) +$$

$$P\ (1-P')\ (1-P^*)\ \beta(k,P'')$$ (3.6.11)

A general Markov model is illustrated in Figure 3.6.3.

Evaluation of Transition Probabilities

It would be constructive to compare the results of a more complex model with the results obtained for the urn model described in Nagel's original study [5]. The urn model is given by (3.6.8). The transition probabilities for the simple flight control program used in the last section were calculated based on the data in Table 3.5.3. From that table is seen that

P = .793 = probability that the fault is detected eventually

P' = .735 = (P0+ P2/2 + P3/3 + P4/4 + ...)/P = probability that the fault is detected on the first iteration, given that it will eventually be detected

σ = .388 = probability that the fault is detected during each subsequent iteration, obtained from (3.6.7)

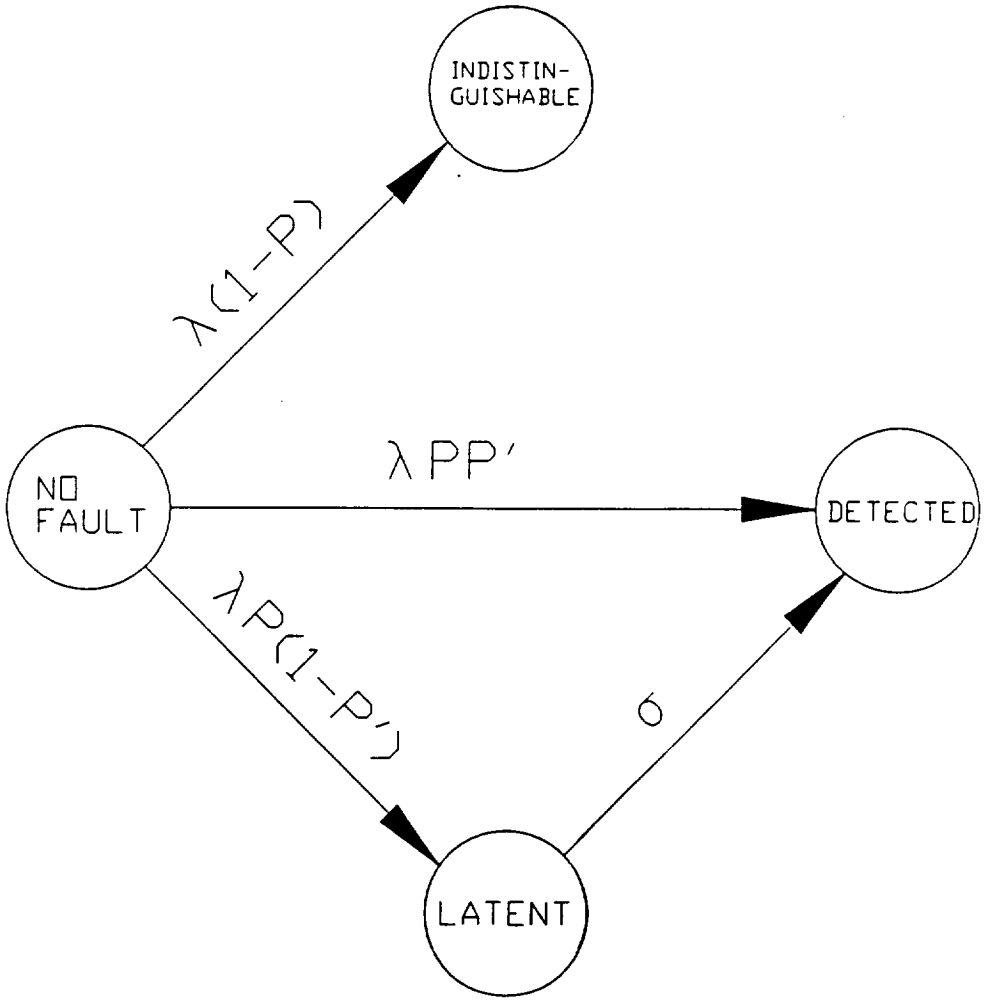
An earlier work of McGough and Swern [6] quotes the following numbers for these parameters:

	P	P′	s
SERCOM	.55	.84	.49
LINCON	.55	.97	.24
QUAD	.58	.86	.66
FCS	.66	.94	.87

A complete description of these programs is given in [5], and is omitted here.

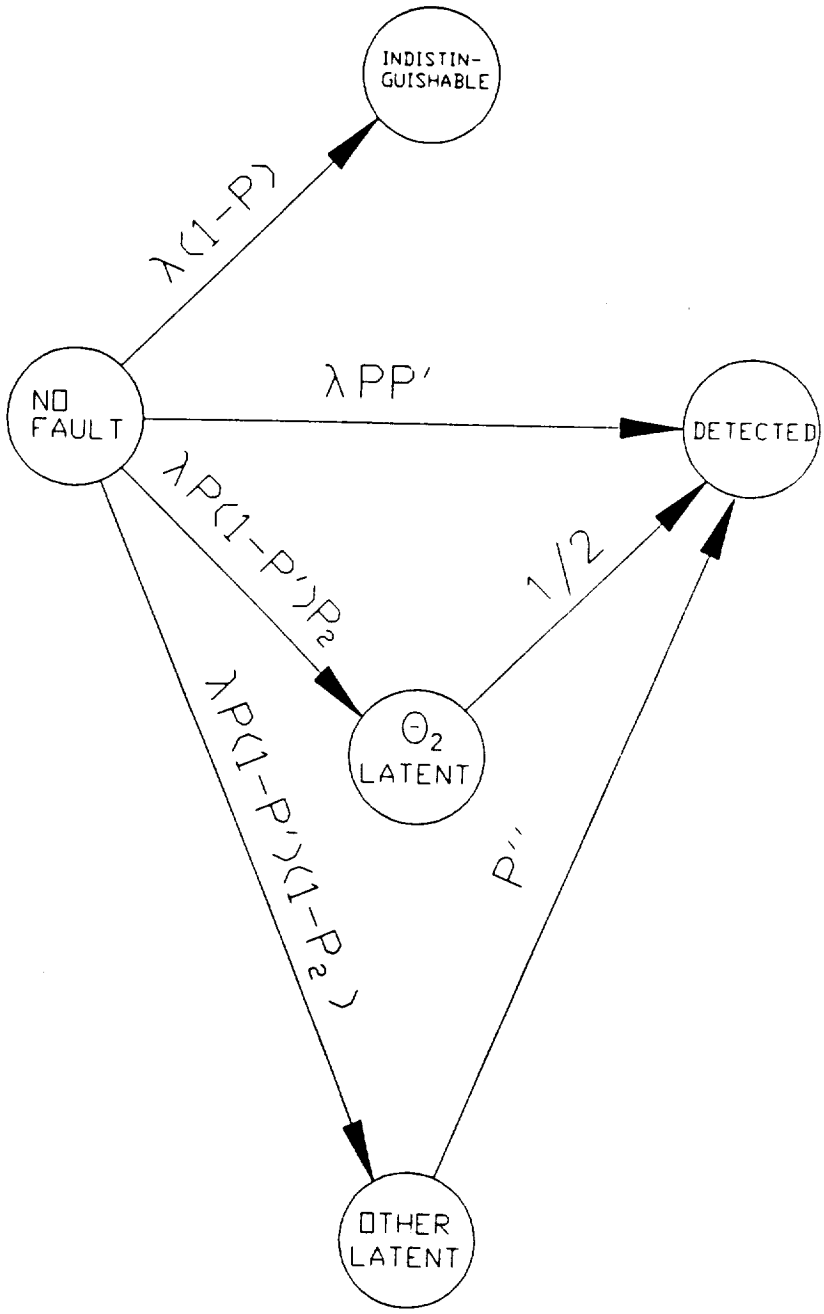
A comparison of results is extremely hard at this point because the above programs are of different sizes and run on different computers. The simplicity of the "toy" processor may account for the larger value of P because even a simple program exercises all of its hardware. Simple programs generally have lower values of σ, a trend which is also true here. First iteration detection P′ may be a function of the hardware architecture of the processor. However, the above is merely conjecture and more studies would have to be done to confirm them.

Using (3.6.9) a more sophisticate model of the processor/program combination can be obtained. The relevant transition probabilities are summarized in Table 3.6.1; however, it seems preferable to present the data in graphical form as in Figure 3.6.4. It was expected that a large number of faults would be detected after two iterations. However, it was conjectured that the number of faults detected would decrease as a function of the number of iterations thereafter. Such is not the case; the figure shows distinct peaks at 6 and 8 iterations. While the peaks could represent experimental anomalies, this effect warrants further studies.



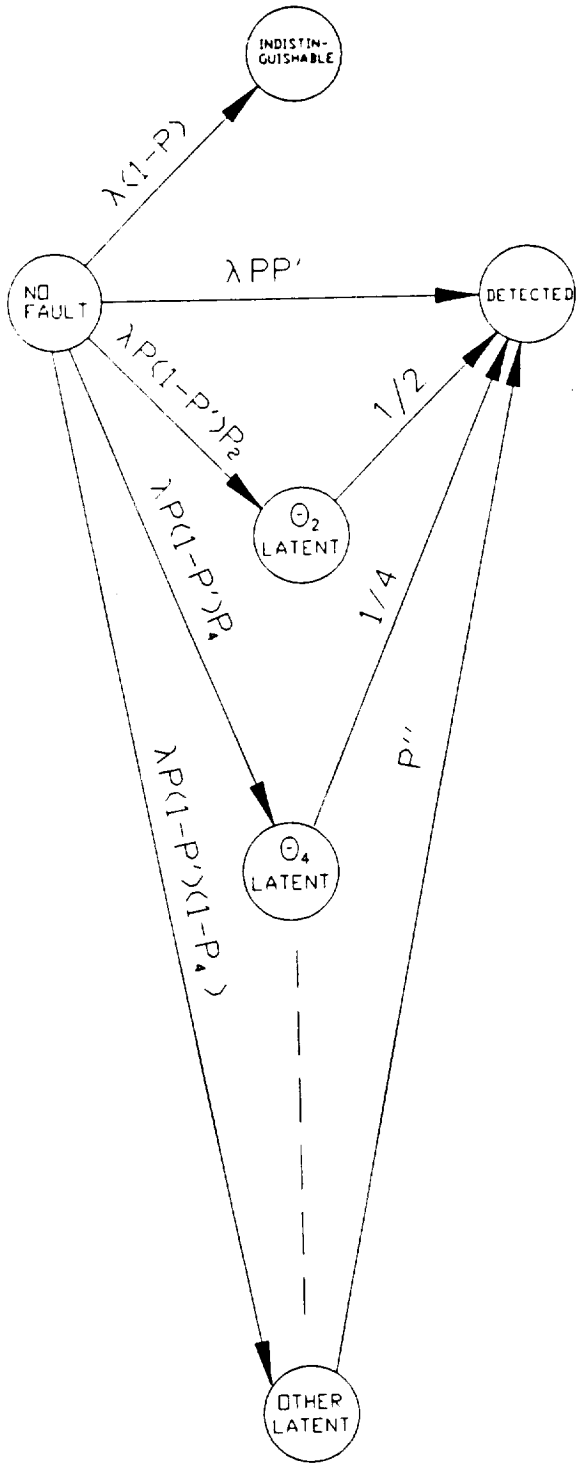
Urn Model Markov Diagram

Figure 3.6.1



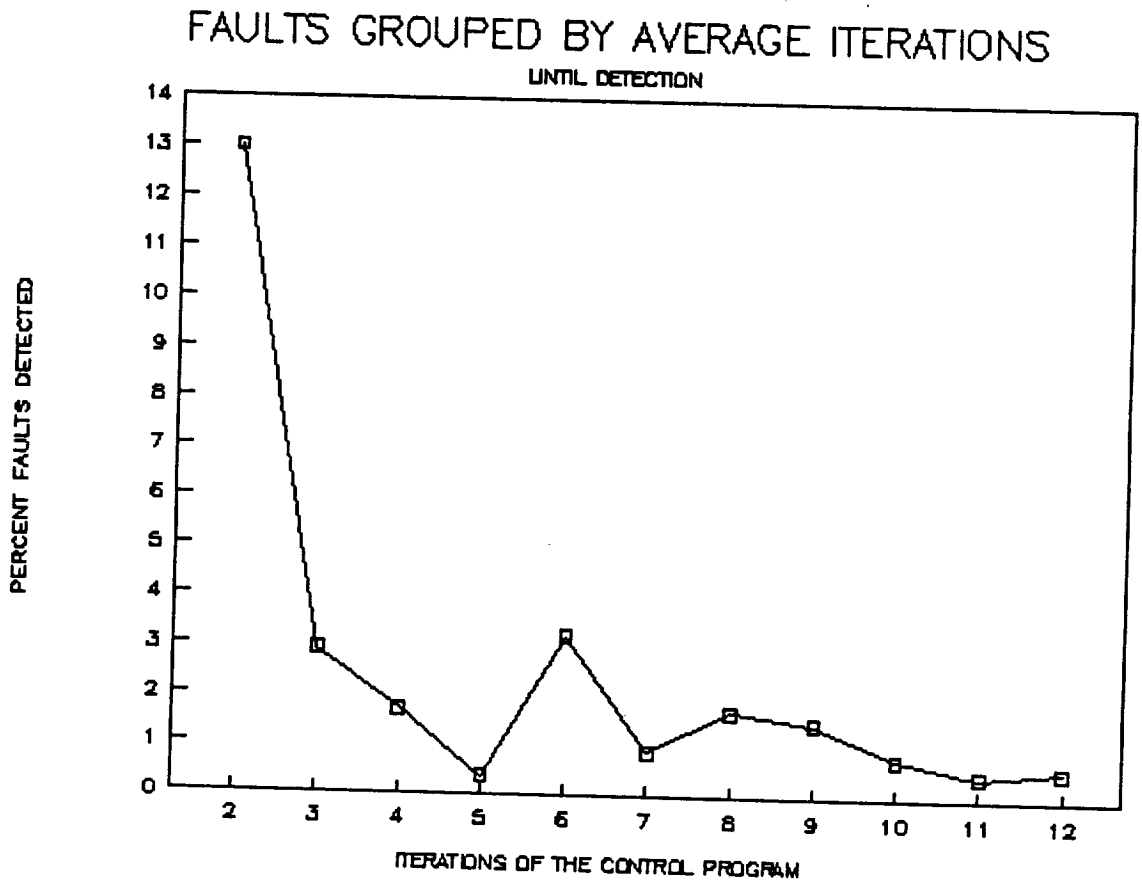
Three State Program Model Markov Diagram

Figure 3.6.2



Generalized Program Markov Diagram

Figure 3.6.3



Graphical Form of Processor/Program Results

Figure 3.6.4

FAULT CLASS	PROBABILITY
P	.79
P_0	.489
P_2	.130
P_3	.029
P_4	.021
P_6	.032
P_8	.041
P_{12}	.028
P_{18}	.023

Model Parameters for Processor/Program Results

Table 3.6.1

3.7 Measurements of a TMR System

TMR System Architecture

The Triple Module Redundant system under consideration is shown in Figure 3.7.1. It contains three similar computational channels replicated from the analog sensors to the actuators that command the aircraft surfaces. Each channel is a complete flight controller which operates independently of the other two. For fault detection, a voting monitor is implemented in hardware at the output to the actuators. This type of monitor is reminiscent of earlier analog systems, and analog comparators, already proven in field service, are used. The voter is a mid value select device which masks the fault from reaching the surface and causing erroneous control of the aircraft (which would constitute a system failure). Logic exists in either hardware or software so that, when one channel fails, it is disengaged. The remaining two channels continue controlling the aircraft, comparing their outputs one against the other. When a discrepancy occurs, both processors will execute a self-test program to determine which channel has failed. If one channel passes self-test and one channel fails, then the good channel will continue controlling the aircraft and the bad channel will be disengaged. If both channels pass (or fail) the self-test, then one channel will arbitrarily be chosen to control the aircraft and the other disengaged.

Modeling the TMR System

It would be useful to study a Markov model of this system with latent faults included. To construct this model, some new parameters must be defined to form the transition rates. Let

$$\eta_t\ (\underline{X}_k, W_k) = \{\ n \mid n \in \eta,\ T(\underline{X}_k, W_k, n) = t\ \} \qquad (3.7.1)$$

represent the set of faults that are detected at iteration t. The probability of detection of a fault at t is given by

$$\gamma_t(\underline{X}_k, W_k) = (1/\lambda) \sum_{n \epsilon \eta_t} P(f_n). \qquad (3.7.2)$$

The probability that the fault is detected within a certain number of iterations, a, given that a fault occurred, is

$$C1(\underline{X}_k, W_k, \alpha) = \sum_{t<\alpha} \gamma_t \qquad (3.7.3)$$

This is a general form of the same C1 that was defined earlier as the percentage of faults detected by the comparator. However, in (3.7.3), C1 is seen to be a function of the initial conditions on the aircraft and the random noise.

The rate of fault detection per iteration is also required. Denoting this function by Γ, it may be estimated by

$$\Gamma(\underline{X}_k, W_k) < \max_{t>\alpha} | \gamma_t | \qquad (3.7.4)$$

Both (3.7.3) and (3.7.4) are functions of t and a. One object of the experiments in the next section is to test the conjecture that, for a given $\underline{X}$ and W, the variation in C1 and Γ over a relatively short time (with respect to the duration of the flight) is small. Then C1 and Γ are chiefly functions of the flight conditions $\underline{X}$ and the noise W. If the noise W is expected to be random, then it too will cause relatively small variations in C1 and Γ.

The remaining factor that might adversely effect the probability of system failure is the change in flight conditions $\underline{X}$. The implication here is that failures that are not detected during the cruise portion of the flight regime might accumulate, although the probability of their effecting the control surface during this mode is low. However, when switching to landing modes, the values of C1 and Γ might change in such a manner that the overall probability of failure of the system is adversely effected.

To complete construction of a Markov model for the studies to follow, let α=1 in (3.7.3) and (3.7.4) above. For each value of $\underline{X}$ and W, there are values associated with C1 and Γ. These parameters, along with other parameters defined in previous sections, are used to form the transition probabilities. Since $\underline{X}$ and W are functions of time with respect to the beginning of the flight, the model is still Markov; however, it is non-homogeneous. Figure 3.7.2 shows the model used for analysis in the studies to follow.

TMR System Experiments

Two groups of experiments were done on the TMR architecture. In the first experiment, the sensitivity of the TMR Markov model with latent faults was determined as a function of the latent fault detection rate. The Markov model given in Figure 3.7.2 is a function of five parameters:

- λ single channel failure rate,
- C1 comparator primary software coverage as defined in (3.7.3),
- C2 secondary detection means coverage,
- Γ propagation rate of latent failures, and
- δ the probability that two latent failures are excited and produce the same erroneous output at the comparator.

A value of λ of .001 failures per hour was used in all computer

studies for this experiment. From the previous studies, a value for C1 of .5 was considered reasonable. C2 and d were allowed to take on three discrete values:

C2: .95, .99, 1.0

δ: .3, .1, .01

The parameter Γ was allowed to vary over its full range. The resulting probability of system failure was plotted as a function of Γ in Figure 3.7.3.

The results can be interpreted from architectural considerations. A propagation rate of latent faults of zero implies that (1.0-C1) of the possible hardware faults never affect the operation of the processor. Then the processor operates uses less hardware, and the resultant probability of failure is smaller. On the other hand, if the propagation rate of latent faults is very high it is as if no fault ever becomes latent and C1 = 1.0. Thus the two endpoints of the curves are easily predictable neglecting the dynamics of the latency itself.

In the middle region of Γ, its effect depends on whether or not two latent faults are likely to have the same erroneous output. When this is not the case, latency actually improves the survivability of the system as a certain percentage of the faults may remain latent until the end of the flight. When the probability of two latent faults giving the same erroneous output is high, then system performance suffers dramatically. However, there is a latent failure propagation rate which is most disastrous for the system, and this rate can be calculated based on the other parameters.

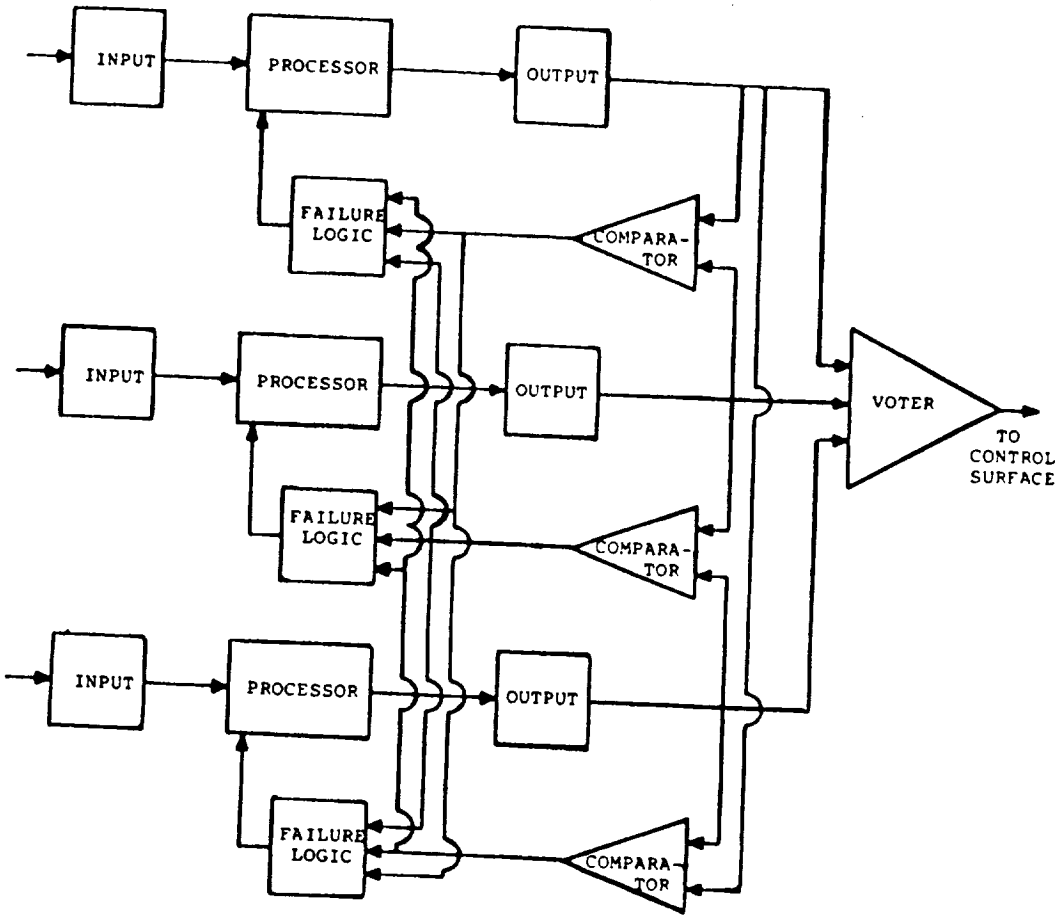
A second flight control program was constructed that contained both a flight control algorithm and a software comparator and voter. A block diagram of this architecture is shown in Figure 3.7.4, and the flight control program is given in Figure 3.7.5. The computed output of each processor was passed to the other processor using shared memory. Each processor compared its output with both its neighbors output, and only transmitted that output to the control surface if there was agreement with at least one neighbor. If there was no agreement, then the processor went into a loop.

In this situation, a failure of the processor can also mean a failure of the comparator/monitor. While this particular algorithm might not be a realistic way to monitor a flight control system, it was deemed an good test of the GGLOSS simulator because GGLOSS simulated all three processors at the same time running slightly different control programs.

The study showed that the software monitor was capable of detecting 85 percent of hardware faults that were detectable during the iterations that the simulator ran. That is, if the software

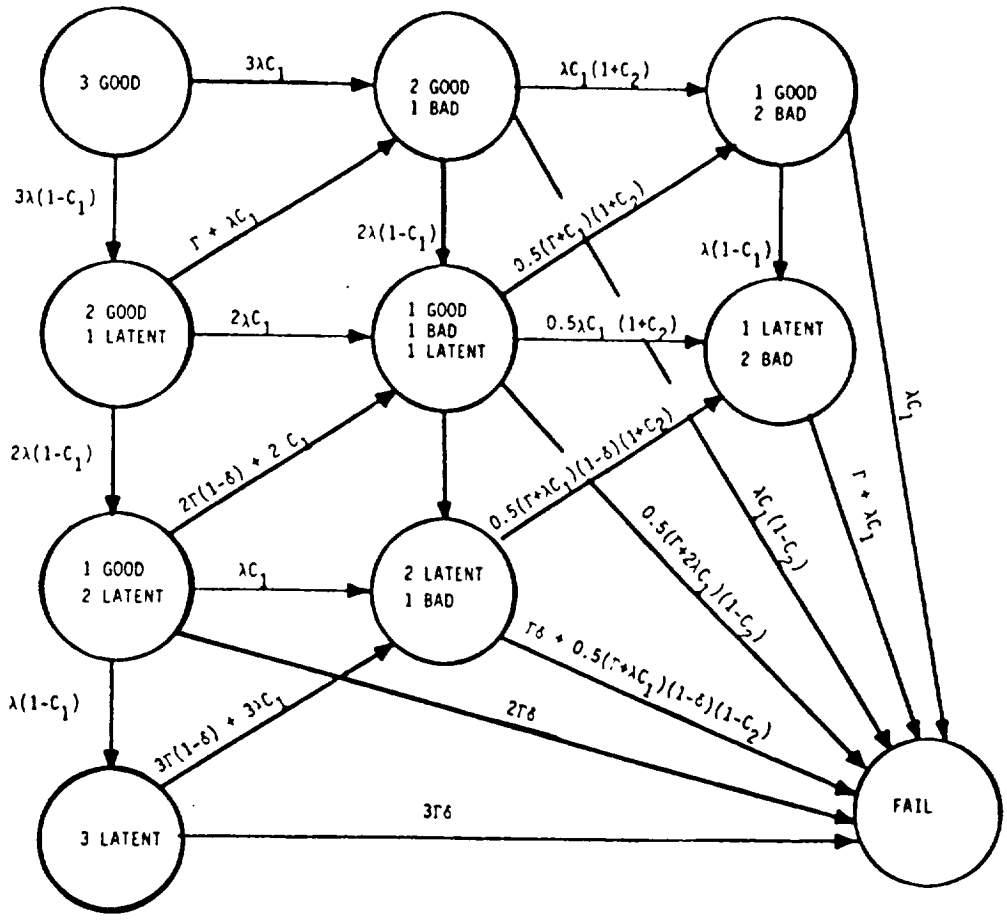
comparator were compared to a perfect hardware comparator, it would detect failures 85 percent of the time. With respect to latency, the results were similar to what was reported in the previous sections.

The results are summarized in Table 3.7.1.



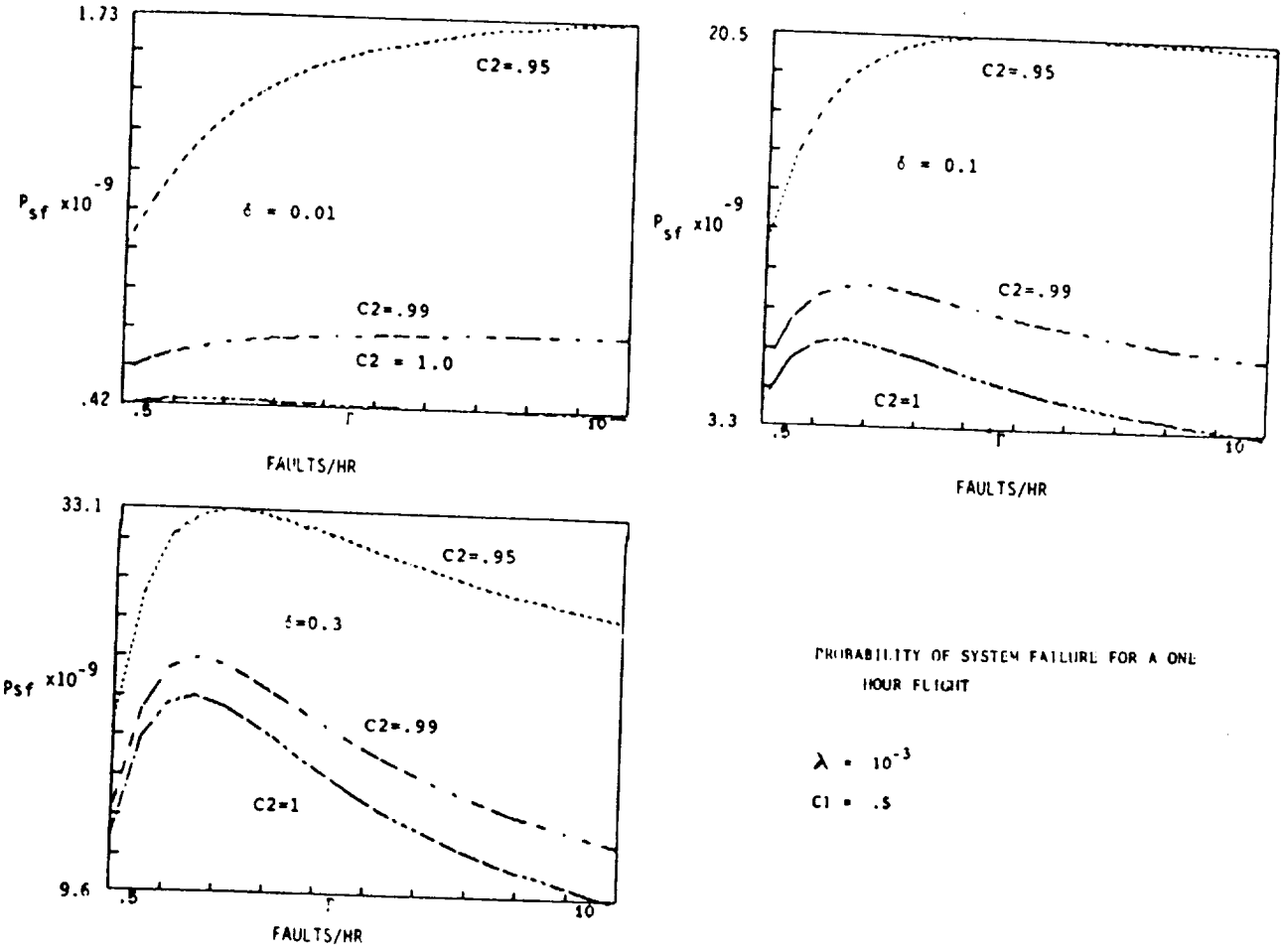
TMR Architecture

Figure 3.7.1



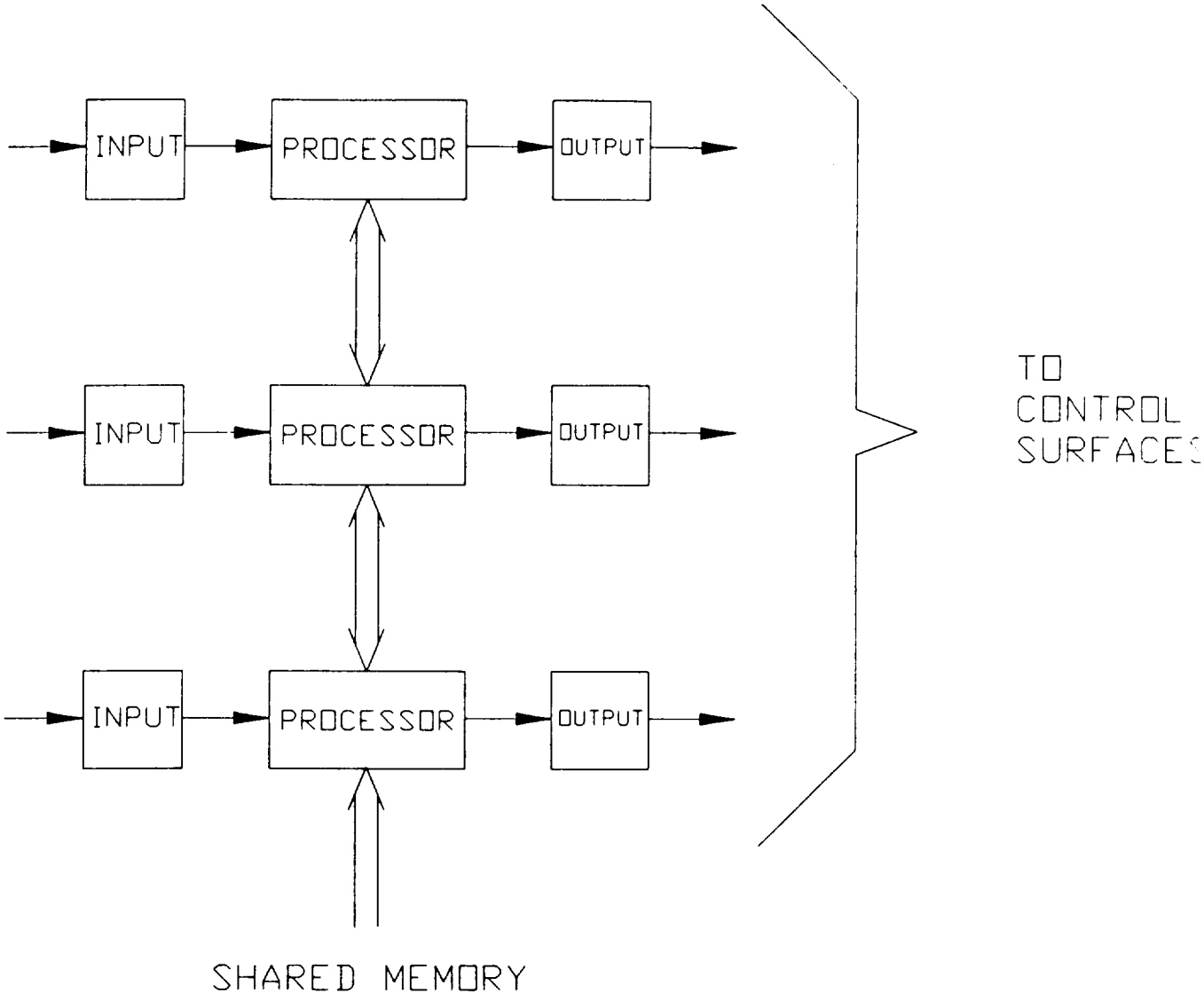
TMR Markov Diagram

Figure 3.7.2



TMR Study Results

Figure 3.7.3



Software Monitoring System Architecture

Figure 3.7.4

```
00:    60          START:    STA    TEMP
01:    A0                    SUB    TEMP      ;  CLEAR ACCUMULATOR
02:    F0                    ADD    IPORT     ;  GET SENSOR INPUT
03:    B0                    SUB    IPORT     ;  NEW - OLD
04:    61                    STA    OUT       ;  NEW OUTPUT VALUE
05:    64                    STA    OUT3      ;  PUT IT IN COMMON
;                                                MEMORY
;
;      FAILURE LOGIC
;
06:    60                    STA    TEMP
07:    A0                    SUB    TEMP      ;  CLEAR ACCUMULATOR
08:    E2                    ADD    OUT1      ;  GET FIRST PROC.
;                                                OUTPUT
09:    A1                  SUB    OUT       ;  SUBTRACT THIS OUTPUT
0A:    DE                    ADD    M127      ;  INDUCE OVERFLOW
0B:    DF                  ADD    M1        ;  OVERFLOW ONLY IF ALL
;                                                OK
0C:    10                   BNO    TRY2      ;  NOT GOOD, TRY AGAIN
;
0D:    F1                    ADD    OUT       ;  PUT OUTPUT IN REG
0E:    71                    STA    OPORT     ;  OUTPUT THIS VALUE
0F:    00                  BNO    START     ;  DO ANOTHER ITERATION
;
10:    60          TRY2:     STA    TEMP
11:    A0                    SUB    TEMP      ;  CLEAR ACCUMULATOR
12:    E3                    ADD    OUT2      ;  GET SECOND PROC.
;                                                OUTPUT
13:    A1                  SUB    OUT       ;  SUBTRACT THIS OUTPUT
14:    DE                    ADD    M127      ;  INDUCE OVERFLOW
15:    DF                  ADD    M1        ;  OVERFLOW ONLY IF ALL
;                                                OK
16:    1A                    BNO    BAD       ;  FAILS ON TWO TRIES
;
17:    F1                    ADD    OUT       ;  PUT OUTPUT IN REG
18:    71                    STA    OPORT     ;  OUTPUT THIS VALUE
19:    00                  BNO    START     ;  DO ANOTHER ITERATION
```

Flight Control Program for Triplex Study

Figure 3.7.5

```
;
;    WHEN A FAILURE IS DETECTED, LOOP HERE FOREVER
;
1A:    60          BAD:      STA    TEMP
1B:    A0                    SUB    TEMP      ;  CLEAR ACCUMULATOR
1C:    E0                    ADD    TEMP      ;  SHOULD BE NO
;                                                OVERFLOW
1D:    1A                    BNO    BAD       ;  LOOP FOREVER
;
;    DATA CONSTANTS
;
1E:    7F          M127:     DC     127
1F:    01          M1:       DC     1
;
;    LOCAL VARIABLES
;
20:    00          TEMP:     DC     0
21:    00          OUT:      DC     0
;
;    SHARED MEMORY VARIABLES
;
22:    00          OUT1:     DC     0
23:    00          OUT2:     DC     0
24:    00          OUT3:     DC     0
;
;    MEMORY MAPPED I/O
;
       30          IPORT     EQU    48
       31          OPORT     EQU    49
```

Flight Control Program for Triplex Study
(continued)

Figure 3.7.5

SYSTEM AVERAGE DETECTION TIME: 4.853175
ACTIVE FAULT COMPARATOR COVERAGE: 0.8514851

DET. TIME	NO. FAULTS	PERCENT OF TOTAL
0.00	10	12.7
1.00	53	67.1
2.00	3	3.8
3.00	2	2.5
4.00	1	1.3
5.00	5	6.3
6.00	1	1.3
7.00	1	1.3
8.00	1	1.3
9.00	1	1.3
10.00	1	1.3

Results of Triplex Study

Table 3.7.1

3.8 Latency Characteristics as a function of System Mode

Earlier sections developed both a Markov model representing the latency characteristics of a processor/program combination and a Markov model representing the effects of latent faults on the reliability of a triplex system. It was shown that the latency time associated with a fault varies significantly across the fault set of a particular processor. Further, latency time can be a key factor in determining the probability of system failure in a triplex system. The present work continues this investigation by considering how the execution of different portions of the control program during a flight mission affects system reliability.

The parameters that characterize fault latency are strongly dependent on the actual hardware and software in operation during a given mission. In order to obtain a feel for these parameters, a simple processor was modeled (the "toy" microprocessor") running a simple flight control program. Stuck-at-one and stuck-at-zero faults were injected into the processor using GGLOSS and the number of iterations until detection was recorded. To examine the effects of modal change, a longitudinal pitch control system was implemented possessing both a vertical speed mode and a flare mode (to simulate landing). Modal change occurred at 40 feet as a function of programmed logic. A block diagram of the control system is given in Figure 3.8.1, and the actual programs are listed in Figure 3.8.2.

It is desirable to define a parameter set that characterizes latency for a given processor/program combination. Let each processor in a redundant system have an associated fault set n so that $n \in \eta$ represents a fault on a processor. The system guides the aircraft through a trajectory represented by the vector $\underline{X}$ and experiences a noise history W. Recall from section 3.6 that a detection function $D_0(\underline{X},W,n,m)$ was defined such that $D_0 = 0$ if the output of the processor is the same as that of a good processor after m iterations and $D_0 = 1$ if the output differs from a good processor after m iterations. Then the latency time of the fault is defined by

$$T(\underline{X},W,n) = \min \{ m \mid D_0(\underline{X},W,m,n) = 1 \}. \qquad (3.8.1)$$

Let $\lambda(fn)$ represent the rate of occurrence of fault n, calculated according to the rules discussed in section 2.1, and λ_t represent the failure rate of the processor as a whole. Then, given that the processor has a fault, the probability that fault n occurs is

$$P(f_n) = \lambda(f_n)/\lambda_t. \qquad (3.8.2)$$

Consider flying a mission represented by a certain time history of $\underline{X}$. When the average latency time of a fault occurring during this mission is i iterations, that fault belongs to fault class θi.

Define

$$P(i,n) = \begin{cases} 0 & \text{if } \sum_{k\epsilon\Omega} P(W_k)\ T(\underline{X},W_k,n) \neq i \\ 1 & \text{if } \sum_{k\epsilon\Omega} P(W_k)\ T(\underline{X},W_k,n) = i \end{cases}$$

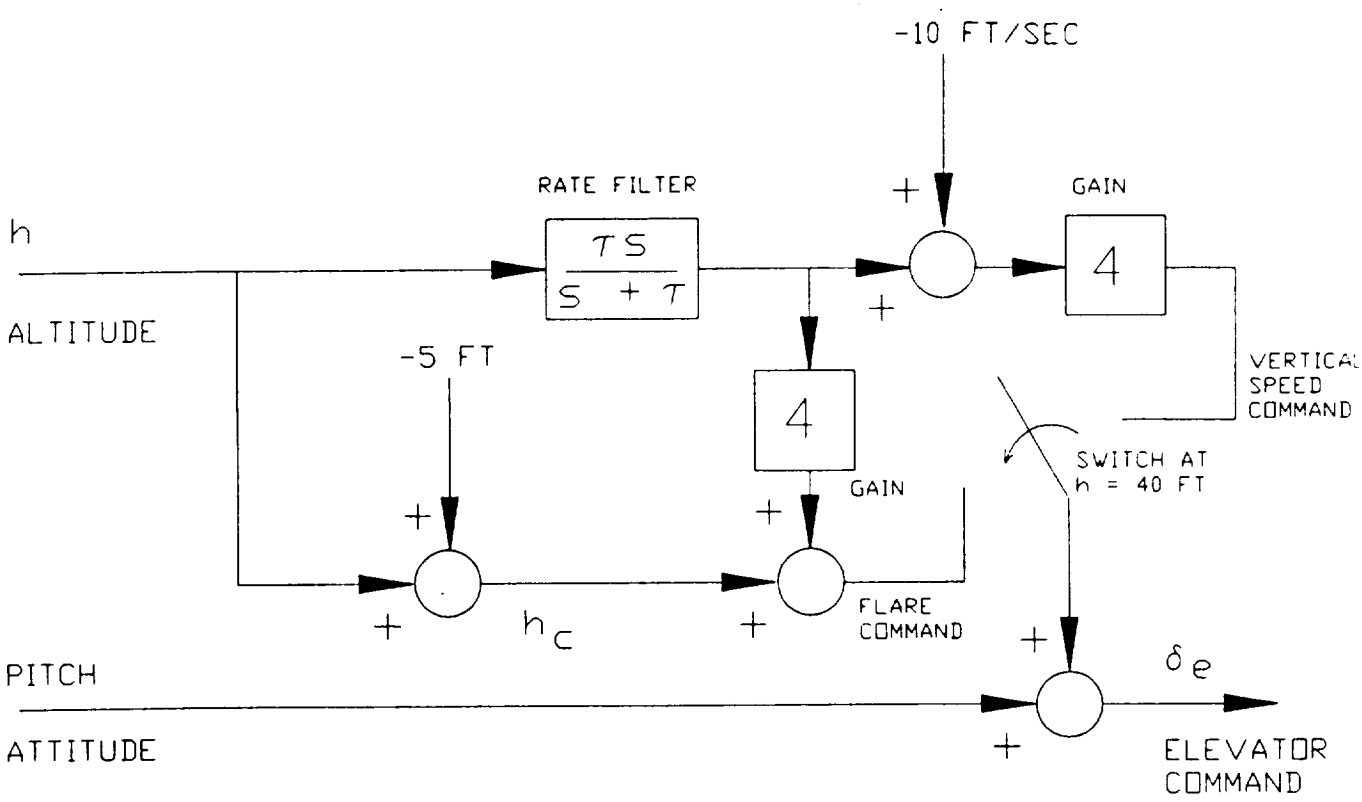
so that, if a fault occurs in a processor, the probability that it belongs to fault class i is

$$P(\theta_i) = \sum_{n\epsilon\eta} P(i,n)\ P(f_n). \qquad (3.8.3)$$

For a given processor/program combination and a given mission, the latency behavior of the system can be characterized by the set $P(\theta_i)$ where θ_i represents the latency class of the fault and i varies from 2 to ∞. In practice, there are only a finite number of θ_i with nonzero probability. A simple way to present latency characteristics is to plot $P(\theta_i)$ versus θ_i.

The latency distributions for each of the three software modules (vertical speed, flare, and modal logic) appears in Figures 3.8.3, 3.8.4, and 3.8.5. The majority of the faults are always detected in the first iteration of the control program, as occurred in previous work [6]. However, a significant number remained latent for a number of iterations. The average latency time of those faults which remained undetected varied among the modules; the significant data is summarized in Figure 3.8.6.

A significant result of this study was the wide variance in latency parameters for each software module.



Block Diagram of a Simple Flight Control System

Figure 3.8.1

```
                ;
                ;     PITCH CONTROLLER FOR TOY MICROPROCESSOR
                ;
                ;     INCLUDES ALTITUDE RATE HOLD AND FLARE
                ;
                ;
00: 7B    ST1:      STA     TEMP        ;   ZERO OUT
01: BB              SUB     TEMP        ;      THE ACCUMULATOR
                ;
                ;     INPUT AND PROCESS THE ALTITUDE DATA
                ;
02: FC              ADD     HPORT       ;   INPUT CURRENT ALTITUDE
03: 75              STA     H           ;     STORE FOR ALTITUDE HOLD MODE
04: A9              SUB     FIVEFT      ;   FORM FLARE ALTITUDE COMMAND
05: 77              STA     HC          ;     STORE IN FLARE COMMANDED ALTITUDE
                ;
                ;     IMPLEMENT FLARE MODE
                ;
06: B7              SUB     HC          ;   CLEAR ACCUMULATOR
07: F5              ADD     H           ;   GET ALTITUDE
08: B6              SUB     HOLD        ;   COMPUTE ALTITUDE RATE
09: 78              STA     HDOT        ;     STORE ALTITUDE RATE
                ;
0A: F8              ADD     HDOT        ;   MULTIPLY HDOT BY TWO
0B: 7B              STA     TEMP        ;   THEN MULTIPLY BY FOUR
0C: FB              ADD     TEMP        ;      TO GET EQUIVALENT ALTITUDE
                ;
0D: F7              ADD     HC          ;   NOW ADD CURRENT FLARE ALTITUDE TO
0E: 7A              STA     DELHD2      ;       GET ERROR AND STORE
```

Flight Controller Program Listing

Figure 3.8.2

```
            ;
            ;     ALTITUDE RATE FILTER UPDATE
            ;
0F: BA              SUB     DELHD2      ;   CLEAR ACCUMULATOR
10: F5              ADD     H           ;       GET ALTITUDE
11: 76              STA     HOLD        ;   STORE IN FILTER STATE VARIABLE
            ;
            ;   IMPLEMENT ALTITUDE RATE MODE
            ;
12: B6              SUB     HOLD        ;   CLEAR ACCUMULATOR
13: F8              ADD     HDOT        ;   GET ALTITUDE RATE
14: AA              SUB     HDOTC       ;   COMPUTE ALTITUDE RATE ERROR
15: 7B              STA     TEMP        ;   NOW
16: FB              ADD     TEMP        ;      MULTIPLY
17: 7B              STA     TEMP        ;            BY
18: FB              ADD     TEMP        ;               FOUR
19: 79              STA     DELHD1      ;   STORE ALTITUDE RATE ERROR
```

Flight Controller Program Listing

(Continued)

Figure 3.8.2

```
                ;
                ;    MODAL LOGIC
                ;
1A: B9                SUB     DELHD1      ;   CLEAR ACCUMULATOR
1B: F5                ADD     H           ;   GET CURRENT ALTITUDE
1C: EB                ADD     H40         ;   FIND OUT IF OVER 40 FEET
1D: 25                BNO     FLLOOP      ;    NO, IN FLARE MODE
                ;
1E: 7B                STA     TEMP        ;    CLEAR ACCUMULATOR
1F: BB                SUB     TEMP        ;
20: F9                ADD     DELHD1      ;   COMMAND IS ALTITUDE RATE ERROR
                ;
                ;    PITCH INNER LOOP
                ;
21: FD    OUT1:       ADD     TPORT       ;   ADD TO PITCH SIGNAL
22: 7E                STA     DEPORT      ;    SEND OUT TO ELEVATOR
23: BE                SUB     DEPORT      ;    CLEAR ACCUMULATOR
24: 00                BNO     ST1         ;    LOOP AGAIN
                ;
25: 7B    FLLOOP:     STA     TEMP        ;   CLEAR ACCUMULATOR
26: BB                SUB     TEMP
27: FA                ADD     DELHD2      ;   COMMAND IS FLARE COMMAND
28: 21                BNO     OUT1        ;   GO OUTPUT TO ELEVATOR
                ;
                ;    ROM CONSTANTS FOR PITCH
                ;
29: 05    FIVEFT    DC       5          ;   FIVE FOOT BIAS FOR FLARE
2A: 0C    HDOTC     DC      12          ;   ALTITUDE RATE COMMAND
2B: 57    H40       DC      87          ;   FORTY FOOT SWITCH POINT
```

Flight Controller Program Listing

(Continued)

Figure 3.8.2

C-2.

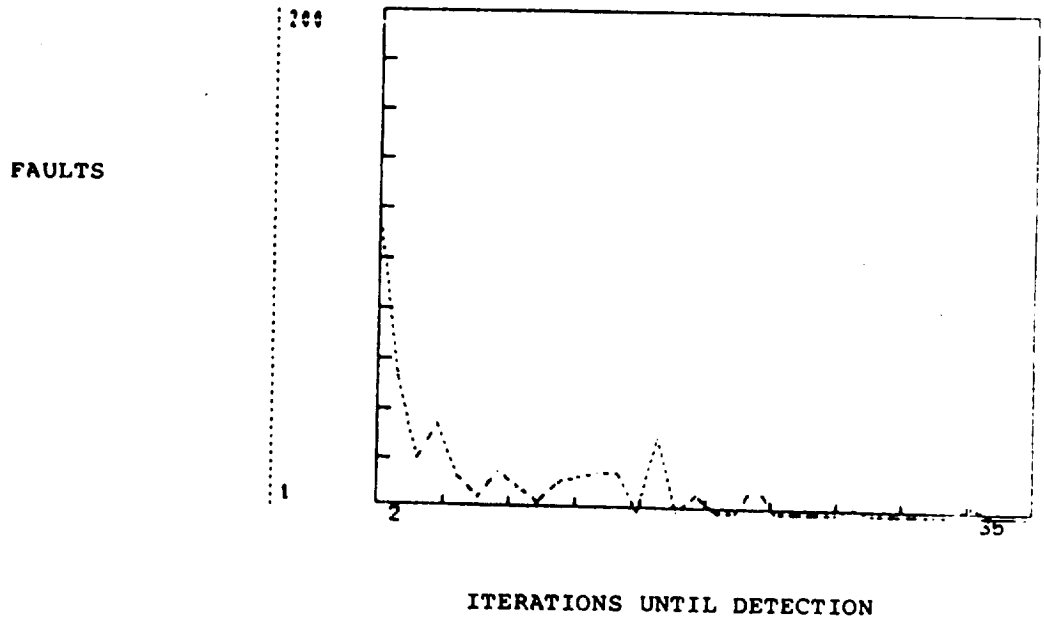
```
            ;
            ;   RAM VARIABLES AND MEMORY MAPPED I/O
            ;
     35             ORG     53
35:  00     H       DC      0           ;   ALTITUDE
36:  00     HOLD    DC      0           ;   OLD ALTITUDE FOR RATE FILTER
37:  00     HC      DC      0           ;   FLARE VARIABLE
38:  00     HDOT    DC      0           ;   COMPUTED ALTITUDE RATE
39:  00     DELHD1  DC      0           ;   FLARE ERROR
3A:  00     DELHD2  DC      0           ;   ALTITUDE RATE ERROR
3B:  00     TEMP    DC      0           ;   FOR CLEARING ACCUMULATOR
            ;
3C:  00     HPORT   DC      0           ;   ALTITUDE MEMORY MAPPED INPUT
3D:  00     TPORT   DC      0           ;   PITCH MEMORY MAPPED INPUT
3E:  00     DEPORT  DC      0           ;   ELEVATOR MEMORY MAPPED OUTPUT

NO ERRORS DETECTED
```

Flight Controller Program Listing

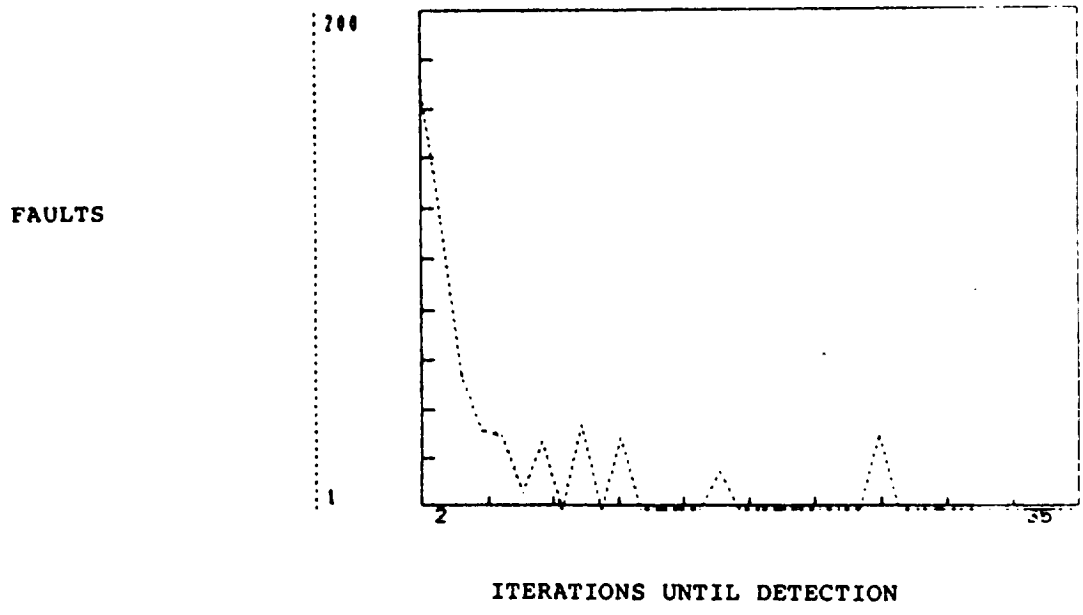
(Continued)

Figure 3.8.2



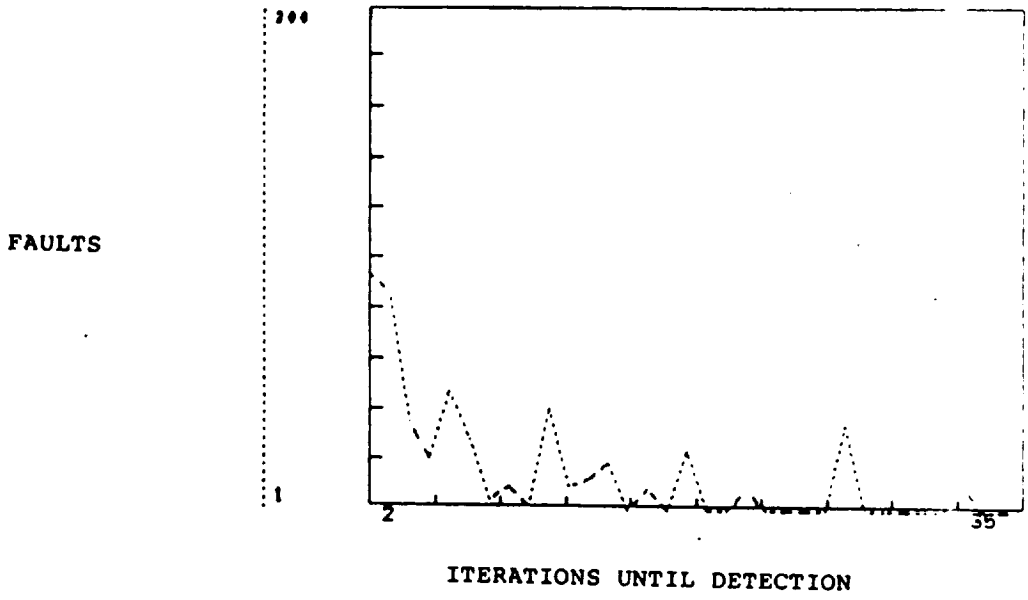
Latency Distribution for the Vertical Speed Program Only

Figure 3.8.3



Latency Distribution for the Flare Program Only

Figure 3.8.4



Latency Distribution for the Modal Logic Only

Figure 3.8.5

COMPARISON OF RESULTS FOR DIFFERENT PROGRAMS RUN ON THE SIMULATOR

	VERTICAL SPEED	FLARE	MODAL LOGIC	SIMPLE LOOP (PREVIOUS REPORT)
PERCENT FAULTS DETECTED	84.8	84.33	82.5	79.8
AVERAGE DETECT. TIME (ITER.)	11.00	5.05	14.8	4.18
PERCENT SIMILAR FAULTS	21.0	21.1	38.6	7.0
NO. FAULTS NEVER DETECTED	275	283	314	351
NO. FAULTS DET. EVERY ITER.	936	831	784	829
NO. FAULTS DET. IN 2 ITERS.	118	168	94	220
NO. FAULTS DET. IN 3 ITERS.	54	116	87	49

Figure 3.8.6

3.9 Effects of Modal Change on the Reliability of Systems with Latent Faults

In order to evaluate the probability of system failure in a processing system running a particular program, it is necessary to combine the results of the previous two sections. Consider the Markov diagram for a triplex system given in Figure 3.3.1. Rather than characterizing fault latency by a single propagation rate G, one may expand the Markov model to include more latency classes as defined in section 3.8. A Markov model of a triplex system with two latency classes is given in Figure 3.9.1. The process can be continued to a larger number of latency classes. However, as the number of latency classes increases, the number of states of the Markov diagram also increases, and the model becomes difficult to solve. Hence, it may be more feasible to consolidate latency classes when the latency times of the faults contained in each class are close in magnitude.

Once a Markov model has been constructed for a particular system architecture, it is possible to use this model to analyze the probability of system failure in a multi-phase mission. Many missions require the system to run a particular portion of the operational flight program for one phase of the flight mission, then switch to another portion of the OFP for some other phase. This modal changing of the OFP which occurs at phase change of the mission has a marked effect on system reliability.

Let $\underline{P}(t)$ represent a vector of state occupancy probabilities for the Markov model, where the size of $\underline{P}$ corresponds to the number of states in the model. For a particular phase of the mission, denoted by i, it is possible to define a matrix G_i that represents the transition rates among the states of the model so that

$$\frac{d\,P}{d\,t} = G_i \; P. \qquad (3.9.1)$$

It is possible to solve (3.9.1) for the transition matrix ϕ_i that relates the state occupancy probabilities at any time t to the state occupancy at time t_0 yielding

$$P(t) = \phi_i(t,t_0)\; P(t_0). \qquad (3.9.2)$$

Consider a two phase mission. At the beginning of the first phase, it is assumed that all processors are functioning properly, which implies one particular Markov state with probability one. For convenience, let this starting state be the first element of $\underline{P}$ so that

$$P(0) = (\,1,\ 0,\ 0,\ 0,\ \ldots\ ,\ 0).$$

At a particular time t1 phase one ends, and the occupancy probabilities are therefore

$$P(t_1) = \phi_1(t_1,0)\ P(0).$$

When the system changes mode, G_1 changes to G_2 and ϕ_1 changes to ϕ_2. However, the state occupancy probabilities $\underline{P}$ at the end of phase one do not represent the state occupancy probabilities at the beginning of phase two. This is because the states of the Markov model which represent latency are based on the latency class of the faults which have occurred, which itself is a function of the program being executed. In other words, while the fault stays the same when the mode changes, the latency class which characterizes it may change. As an example, a fault that was not detectable by the program in phase one may become detectable by a program in phase two. It is necessary to compute the probability of migration among fault classes when mode transition occurs. This is done in a very straightforward manner by using the same fault set for examining each mode. One can compute the probability that a fault in a particular class in phase one is in some other class in phase two, and construct a switching matrix S. It is only slightly more complex to construct a matrix Q that relates the state occupancies at the end of phase one to the state occupancies at the beginning of phase two, based on S and the definition of states in the Markov diagram.

Let $P'(t_1)$ be the state occupancy probabilities at the beginning of phase two. Then

$$P'(t_1) = Q\ P(t_1)$$

and

$$P(t_2) = \phi_2(t_2,t_1)\ P'(t_1) \qquad (3.9.3)$$

gives the state occupancy probabilities at the end of phase two, and hence the probability of system failure of the mission.

As an example, consider the following mission flown by the flight controller defined in section 3.3:

- Vertical speed mode from take-off to destination for a flight time of approximately one hour, and

- Flare mode upon arrival at destination for a time of approximately ten seconds.

Some latency class data for each phase of the mission is given in Figures 3.9.2 and 3.9.3. A switching matrix S was constructed from the simulation data, and is given in Figure 3.9.4. Examining the switching matrix, two interesting points come to light:

- the matrix is not as strongly diagonal as might be expected,

- a significant number of faults transition from never detected to always detected.

The data in this matrix was used to construct a matrix Q, and the Markov model was solved for the two phase mission. The results can be summarized as follows:

Condition	Probability of system failure
No latency effects	1.01×10^{-9}
Vertical Speed Mode only	1.46×10^{-9}
Entire flight mission	30.36×10^{-9}

The above data supports the hypothesis that there is a significant migration of faults from one class to another during mode switching. CERTAIN FAULTS WHICH MAY BE UNDETECTABLE UNDER A PARTICULAR MODE MAY BECOME DETECTABLE WHEN THE MODE CHANGES, SUCH AS:

- memory addressing faults (to the memory containing the new mode), and

- other hardware not exercised by the previous mode.

THESE FAULTS MAY COLLECT AND BECOME LATENT FOR LONG PERIODS OF TIME PRIOR TO MODE CHANGE. Thus, mode switching may be the most significant contributor to fault latency, and hence to degrading system reliability.

G = GOOD PROCESSOR

B = BAD PROCESSOR

L_1 = PROCESSOR WITH CLASS 1 LATENT FAULT

L_2 = PROCESSOR WITH CLASS 2 LATENT FAULT

λ = FAULT RATE OF A SINGLE PROCESSOR

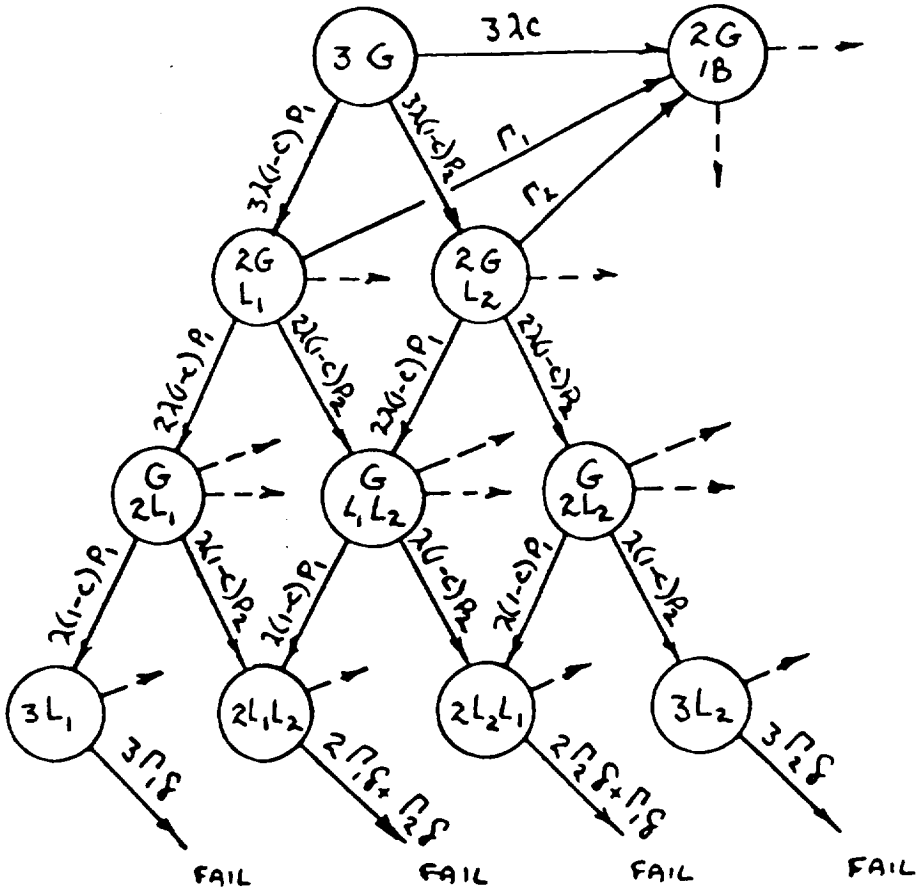
C = PROBABILITY THAT A FAULT IS DETECTED IMMEDIATELY

δ = PROBABILITY THAT TWO LATENT FAULTS RESULT IN THE SAME ERRONEOUS OUTPUT

Γ_1 = PROPAGATION RATE OF LATENT FAULT CLASS 1

Γ_2 = PROPAGATION RATE OF LATENT FAULT CLASS 2

(PARTIAL DIAGRAM - DASHED LINES ARE CONTINUED)



Markov Model of a Triplex System With Two Latency Classes

Figure 3.9.1

FAULTS DETECTED	86.8 %
AVERAGE DETECTION TIME	8.79 ITERATIONS
SIMILAR FAULTS	22.4 %
FAULTS DET. EVERY ITERATION	44.1 %
FAULTS DET. IN 2 ITERS.	10.3 %
FAULTS DET. IN 3 ITERS.	14.2 %
FAULTS DET. IN 4 ITERS.	3.4 %
FAULTS DET. IN 5 ITERS.	2.5 %
FAULTS DET. IN 6 ITERS.	3.0 %

Vertical Speed Mode Latency Parameters

Figure 3.9.2

FAULTS DETECTED	84.33 %
AVERAGE DETECTION TIME	5.05 ITERATIONS
SIMILAR FAULTS	21.1 %
FAULTS DET. EVERY ITERATION	49.1 %
FAULTS DET. IN 2 ITERS.	9.9 %
FAULTS DET. IN 3 ITERS.	6.9 %
FAULTS DET. IN 4 ITERS.	3.3 %
FAULTS DET. IN 5 ITERS.	2.0 %
FAULTS DET. IN 6 ITERS.	1.8 %

Flare Mode Latency Parameters

Figure 3.9.3

	NEVER	ALWAYS	1 ITER	2 ITER	3 ITER	4 ITER	5 ITER
NEVER	.9205	.0389	.1234	.0384	.0961	.0	.0
ALWAYS	.0452	.7960	.5489	.0256	.1153	.0	.4883
1 ITER.	.0	.0703	.1361	.3589	.0384	.0	.0930
2 ITER.	.0	.0820	.0723	.1025	.1538	.2000	.0
3 ITER.	.0	.0047	.0681	.0256	.2500	.2400	.0
4 ITER.	.0	.0	.0213	.1538	.0	.0	.0
5 ITER.	.0	.0	.0298	.0128	.1731	.0	.0

Latency Class Switching Matrix S from Vertical Speed to Flare

Figure 3.9.4

4. SOFTWARE RELIABILITY EVALUATION USING SIMULATION

4.1 Introduction

A significant problem in designing fault tolerant computer systems is to insure that generic failures, such as those caused by design errors, do not occur. Design errors cannot be detected by comparison monitoring techniques, and, when excited, can result in "single point" system failures. Engineers have gained considerable experience in processor design, and good engineering practices can reduce hardware design errors to an acceptable level. Software, however, poses a significantly harder problem. The focus of the following work is on delivering error free software to a particular reliability level; more specifically, validating the reliability of software that will be used in a highly reliable computer system.

One application of fault tolerant computer systems occurs in the avionics industry, when a processor complex is used to control the aircraft. If the application of the processor complex is such that its failure implies loss of control of the aircraft, it is termed flight critical. The software that controls the aircraft is termed the Operational Flight Program (OFP), and in a flight critical system the OFP must operate without degrading the reliability level of the hardware upon which it runs.

Reliability analysis of electronic systems is based on the premise that components fail in a random manner when exposed to the stresses and strains of normal operation. The failure rate of each component is often assumed constant, and values can be determined by collecting field data. Moreover, two identical components subject to the same environmental conditions fail independently. Complex systems can be modeled as a combination of simpler components. However, software faults are not component failures per se. Software never deteriorates with age; rather all faults result from design errors that have remained latent through the verification and validation phases. During system usage, these errors are subject to random excitation, possibly resulting in system failures.

Before an OFP can be used as a component of a flight critical application, it is required to associate with that program a probability of failure. Because of the nature of software, it would be more correct to say that the probability of excitation of existing design errors is required. In this sense, software errors are like latent hardware faults, and it seemed reasonable to explore whether techniques used to measure the effects of hardware fault latency on reliability can be used to measure software reliability. Measurement of hardware fault latency was discussed previously herein.

To test this hypothesis, an OFP was obtained that controlled an aircraft during autoland at ILS category III facilities, a flight

critical application. The OFP was meant to run on a triplex or quad computer architecture to ensure that the hardware's system failure rate was on the order of 10^{-9} per hour. It was assumed that the OFP had already gone through the verification phase; the methodology was formulated for its validation. A reliability model of the programs execution characteristics was developed, and the program itself was instrumented to measure the parameters of this model. When the validation test was complete, a reliability number was generated.

The procedure was then evaluated for its ease of use, predicted execution time for larger software programs, and ability to be automated. The outcome of the test showed the method to be a feasible approach for validation of critical software.

Previous Work on Software Reliability

Many researchers have proposed software reliability models [23,24,25]; however, most of the models have been based on cumulative failure data obtained during normal system operation. For critical software, this is to be avoided, as a single software failure could become catastrophic. Most models assume a particular distribution to predict the time between failures, and model parameters are estimated from the data. If the past failure history fits the model well, it can be used to predict the occurrence of future failures. Models have been constructed assuming the occurrence of failure follows the binomial distribution, the Weibull distribution, and the Poisson distribution; a good summary of the different models is given in [23] and also in [24].

Other researchers have dealt with measurement of errors during the validation phase, more similar to the problem dealt with here. Specifically, Nelson [26] has suggested measuring reliability from random excitation of the software. Specifically, Nelson suggests running the software with a series of n inputs, randomly chosen from the input domain E needed to make a run. The random sampling is done according to the probability of occurrence of the inputs in normal operation. If n_f is the number of inputs that resulted in software failures, then $R = (1-n_f/n)$ is the reliability of the software. Ramamoorthy and Bastani [27] have increased the complexity of the model to include probabilistic equivalence classes, a concept which takes into account the complexity of the program.

Fault Tolerant programming schemes exist, such as N version programming [28], which purport to increase the reliability of the software using hardware principles of redundancy management.

4.2 Flight Critical Software Requirements

It is necessary that the failure rate of flight critical systems meet government specifications, i.e., , 10-9 per flight hour for a commercial system and , 10-7 per flight hour for a military system, and that both the hardware and software that compromise the system must together meet these criteria. Further, it is necessary to demonstrate to the applicable certifying authority that the resulting system indeed will attain the required reliability.

System certification usually implies a combination of flight tests and computer studies. Computer studies can show simulated operation in normal and aberrant environments, demonstrating that the unfailed system will operate correctly in the expected environments to the required probability. Component failures may then be simulated according to their probability of occurrence to demonstrate that the system either recovers in an acceptable manner to common failures, or the combination of component failures necessary to induce an equipment failure is sufficiently unlikely as not to adversely affect the probability of system failure. The final result of these computer studies is an estimated probability of system failure during the intended mission of the aircraft.

Flight testing is expensive, and it is not possible to collect enough flight data to form a statistical basis for validation. Flight testing can check the computer model used in the simulation studies by demonstrating that the system operates correctly in normal usage, and operates in the manner predicted under a small number of induced failures.

Certifying that the system meets its reliability criteria is usually termed validation testing. Validation is independent of system development, and represents an acceptance test of the final product. When the system is relatively simple and the required reliability is low, certification can be done strictly at the systems level, i.e., sufficient testing of the entire system can be performed to form a statistically valid sample of system operation. However, when the system is complex and the required reliability is high, the system validation can be decomposed into a number of tests that exercise different portions of the system, the results being mathematically combined to obtain the overall reliability.

One such decomposition is to validate separately the hardware and the software. This makes sense, because many reliability modeling procedures are available for hardware [11,12] which reduce the complexity of the required tests (i.e., decomposition into components and evaluating the component reliabilities and their interrelationships). The same type of validation testing should be done on the software.

To perform software validation testing, one must prepare a

test that exercises every possible set of inputs that the system might encounter. This is clearly impossible. One must construct a test that will exercise enough of the possible environments and demonstrate that the probability of occurrence of untested scenarios is sufficiently small so as not to adversely affect system reliability.

Consider that software for flight critical applications is usually relatively small compared to, say, navigation software. The operating environment for critical software is well defined - it is specified in advance under exactly what conditions the system must operate satisfactorily. Description of the environment is often available in government documents, such as FAA advisory circulars [29,30].

Because of the criticality of the application, it is necessary to perform reliability estimation during validation testing, rather than as a consequence of flight experience. However, the total amount of computer and flight time required to validate the software must be "reasonable" - that is, time to validate must be sufficiently small that the software could be revalidated if the code is changed. The methodologies described herein should meet these goals.

4.3 Typical Programming Errors

Before discussing the methodology itself, it is instructive to consider errors that are typical of what one might encounter in a program. The intent is not to classify programming errors, but rather to examine a sampling of errors one might find in practice and consider how the validation technique would test for each.

- erroneous calculation of a control law

It is possible that, during coding, the design specification was misinterpreted and algorithms were implemented that were different from the requirements. Errors could be as simple as changing a constant multiplier by a small amount, or as complicated as omitting (or adding) logic paths. It is most likely that the errors would be in some sense small, or they would have been uncovered during verification testing. These errors might cause a negligible change in system operation during most environmental conditions. It is possible that, under the right conditions, catastrophic changes in system performance could occur. It is also possible that these changes might never affect system performance in more than a minimal manner.

- invalid reference to a memory location

It is possible that a statement used to test the value of a memory location could refer to the wrong memory location. Many tests are such that they are always (or rarely) satisfied. In this case, if the erroneous memory location tests the same way as the correct memory location most of the time, the error would be hard to detect. However, certain conditions might cause the test outcome to change, and this error will be activated.

Consider as an example, a test for maximum allowable value of a variable before its transmittal to the processor output. If the test references the wrong variable, and the incorrectly referenced variable is below the maximum value, the test will give the correct outcome whenever the correct variable is also below the maximum value. If the correct variable reaches its maximum value very seldom, this may be a hard error to detect. However, going over the maximum allowable value might cause the system to operate incorrectly and result in a catastrophic failure.

It is also possible that this incorrect reference could be to a memory location outside of the scope of the module. In this case, the error might be benign for all conditions the system might incur; however, recompiling and relinking the program might cause the error to become catastrophic for a number of conditions. Further, the program might be recompiled and/or relinked for problems that were not related to the module containing the error.

- overflow or divide by zero not properly handled in certain situations

It is possible that an exception handling routine does not operate correctly under certain conditions. The exception handler is usually entered from a variety of calling locations, in some designs activated by interrupt and in other designs by subroutine calls interspersed in the code. The nature of exceptions implies that they occur infrequently, hence the code to support them may be exercised very infrequently. The above conditions imply that any problems within the exception handler may remain latent. The absence of an exception handler also causes data dependent errors in other lines of the code.

- compiler/assembler not properly translating source code

It is possible that the compiler used to translate source code is not operating properly. When improper operation results in immediate failure of the resulting object code, this problem will be corrected immediately. When subtle errors occur, such as an incorrect interpretation of the compiler specifications in certain situations, these errors may remain latent.

Consider, as an example, a misinterpretation of the compiler specification on the execution of FORTRAN type "DO" loops when the index variable is less then or equal to zero. The misinterpretation could be either by the applications programmer or the compiler writer; reliability wise the result is the same. A data dependent error is created that may remain latent depending on the operating environment.

- improper timing of control code

It is possible that, under certain conditions, computations may not be finished in a major computational frame. A background task may access a variable used by a foreground task; the value of the variable is now a function of the program timing. Timing dependencies may create problems that only occur infrequently, creating latent errors.

- improper interpretation of the system requirements document

It is possible that, in translating the system requirements document through the many steps that result in code generation, errors were made. Good engineering practices will reduce the number of errors that actually appear in the code; however, subtle errors may remain. It is important to note that the cause of improper system operation often can only be blamed arbitrarily on one particular phase of the development process, i.e. even when the error is fully understood, it is not fully clear that it is due to poor design, poor coding, or incomplete specifications.

4.4 Development of the Methodology

Software does not deteriorate with age; if a program encounters a given set of input conditions, it will present the same output sequences repeatedly. If one could validate a flight program by presenting all possible sequences of input conditions to the software, and verify that the output sequences are correct, then the software is, by definition, error free! Clearly, this is an impossible task.

When validation is imperfect, software errors remain in the program and become latent. Latent errors can be excited by the right combination of environmental factors. If these environmental factors can never occur, then the software operation is still completely reliable. When these environmental factors do occur, the software fails. Then, the reliability of the software is equivalent to the probability that environmental factors do not occur that will excite latent errors. The fact that a certain release of software has flown successfully for a certain length of time implies a certain reliability for that particular release of the software, as it implies a probability of excitation of any latent errors.

What constitutes correct operation of the software? Errors might occur that cause the software to deviate from what the system designers had in mind, but the software might still acceptably control the aircraft. Therefore, the following definition is proposed of correct software operation:

> **Correct operation of the software is implied by an acceptable aircraft system in which the software is embedded.**

If the aircraft can be shown to operate acceptably (as defined in applicable flight documents and specifications) for all possible environmental conditions, then, by definition, the software is operating correctly.

It is apparent that some of the faults mentioned in the previous section may be present, and the software will operate correctly. If the environment in which the software is to operate is known a priori, then the probability of the software not operating correctly is a function of the probability of an environmental change that excites an error, or the error's latency time. More specifically,

- most faults with small latency times are uncovered during verification testing,

- if the latency time is sufficiently high, the error has a minor effect on reliability, and

- faults with moderate latency times have the worst effect on reliability.

Of course, the interpretation of what is a small, moderate, or high latency time is necessary to performing an evaluation of system reliability. Previous studies with hardware latency have shown this concept to be viable for failure analysis [17].

Note that by this definition of software reliability, if the environment changes significantly from the environment the system was designed for, the reliability of the software changes accordingly. Further, the reliability of a new release of software cannot be predicated on the reliability of the old release.

One may argue that the proposed definition of software reliability encompasses failure issues that are more correctly on the system level, and may include hardware design errors, system design errors, etc. It is the contention here that, in many cases, it is impossible to pinpoint whether an error is truly a "software error", a poor hardware design that is intolerant of its software, or a control algorithm that is hard to implement. In any case, the important requirement is a system reliability requirement with software embedded.

4.5 Mathematical Development of Software Reliability Models

Hardware reliability is estimated by dividing it into modules, and breaking the modules down into components. A mathematical model of the interconnection of the components and modules is constructed. Component reliability may be obtained from handbook data, or laboratory experiments, and the mathematical model may be evaluated to obtain the overall system reliability.

Software reliability modeling proceeds in a similar manner. A decomposition into modules for a program is normally part of the software design phase. A line of code will be used as an entity analogous to components in hardware; a line of code is represented by a typical line of a FORTRAN program (or other higher order language).

One might question whether the definition of this entity is too loose. A complete operational flight program might consist of a single line of FORTRAN code such as FLY_THE_PLANE, while the same program at an assembly language level might consist of a few thousand lines of code, each doing a significantly smaller operation. The definition of exactly what constitutes a line of code is not critical to the method, as the reliability of lower level lines may be easier to compute with greater accuracy than those of lines which imply a higher computational level.

Analyzing a Line of Code

A typical FORTRAN line of code, other than a subroutine call, associates a value to a variable based on some computations. For simplicity, it is assumed that a single output quantity is associated with each line. This is not considered confining, as a program line that manipulates two output quantities can be modeled as two lines of code. Each programmed variable can be represented as a discrete outcome of a computation contained in the set

$$J = \{ 1, 2, \ldots, \Omega \} \tag{4.5.1}$$

where Ω represents the number of different values that can be represented by a variable on this processor, i.e., 2^{16} for a sixteen bit processor and 2^{32} for a thirty two bit processor.

Each line of code can be represented mathematically by the function

$$y_k = \phi_k(y_1, y_2, \ldots, y_n) \tag{4.5.2}$$

where $y_k \in J$ is the output of the kth line,

$y_1, y_2, \ldots, y_n \in J$ are n quantities that the kth line of code uses as inputs, and

ϕ: $\Omega^n \rightarrow \Omega$ is the mapping represented by the code line.

Definition

An elementary error occurs when one member of the input space of ϕ is not correctly mapped onto its corresponding output.

The probability of an elementary error resulting in an erroneous computation is then the probability of that elementary error being excited. If only one elementary error exists at, say, input sequence $i \in \Omega^n$, and its excitation causes a system failure, then

$$P(\text{error}) = p(\text{excitation}) = p(e_i) \qquad (4.5.3)$$

where e_i is the event that the ith input sequence presents itself sometime during a single operating interval of the system (such as a landing, mission, etc.), and $p(e_i)$ is the probability associated with event e_i. The situation becomes a little more complex when there are multiple elementary programming errors in the line. If there are n elementary programming errors in a line, represented by input sequence events α_i, $i=1,\ldots,n$, $\alpha_i \in \Omega^n$, then

$$P(\text{excitation}) = p\left(\bigcup_{i=1}^{n} \alpha_i\right) < \sum_{i=1}^{n} p(\alpha_i) \qquad (4.5.4)$$

To verify correct design, one must show that all combinations of inputs to the line of code results in the proper output. This is clearly an impractical task, and unnecessary because:

- each input y_i has associated with it an expected region of operation, significantly reducing the input space for the line,

- the mapping for each input combination is not independent; rather they follows the algebraic rules as stated in (4.5.2). Therefore, errors will not associated with input combinations but rather with the algebraic rules of the code line.

The foregoing points significantly limit the number of independent errors that might be associated with each line. It therefore seems unfeasible to associate an infinite number of independent errors with each line, but rather some small finite number of errors associated with the structure of the line.

Definition

A programming error occurs when a single factor causes a region of the input space of ϕ to map incorrectly to its output. Multiple programming errors are independent events.

Each programming error implies a set of associated elementary

errors; conversely an elementary error may be implied by more than one programming error. A programming error may be represented by the set of n elementary errors that it implies, i.e.,

$$\psi_i = \{\alpha_1, \alpha_2, \ldots, \alpha_n\} \qquad (4.5.5)$$

where the index i enumerates the m different programming errors that may be associated with this particular line. Then the probability of error for each line is

$$P(\text{excitation}) = P\left(\bigcup_{i=1}^{m} f_i \left[\bigcup_{j\epsilon\psi_i} e_j\right]\right) < \sum_{i=1}^{m} P(f_i) \sum_{j\epsilon\psi_i} P(e_j)$$

$$= \sum_{i=1}^{m} S(f_i) \; P(\psi_i) \qquad (4.5.6)$$

where f_i is the event signifying that programming error i has occurred, and $S(f_i)$ is a function either one or zero.

In certain cases, it may be possible to identify analytically all the possible errors. Then, a test that checks for the presence of each error can be constructed. A proper test set can show that $S(f_i)$ is zero for all m errors, P(error) = P(excitation) = 0 from (4.5.6), and the line is correct. This is the basis for proof of correctness techniques.

Considering the types of errors that might occur, in most cases, both analytical identification of errors and test generation is not practical. However, if the number of possible errors in the line could be identified along with some information about the elementary error set ψ_i associated with each error, statistical testing techniques could be used test the state of $S(f_i)$.

The M Error Model

One approach to measuring P(excitation) for each line is to assume a model for (4.5.6) whose parameter are known or can be measured. The model used here contains exactly M possible errors, all having an equal probability of excitation, i.e. it is assumed that

$$P(\text{excitation}) < \sum_{i=1}^{M} S(f_i) \; P(\psi_i) = \frac{1}{M} \sum_{i=1}^{M} S(f_i). \qquad (4.5.7)$$

It is noted that P(excitation) has a binomial distribution, i.e., either an error in the line is excited, or it is no not. Consider a validation test that exercises the line of code in a random manner. The probability under the M error model that the validation test encounters no errors for E executions of the line and encounters an error the E+1 time the line is executed is

$$P(\text{error on } E+1) = [1.0 - P(\text{excitation})]^{E} P(\text{excitation}) \quad (4.5.8)$$

It is easy to show that, when the relation

$$E > \frac{1}{P(\text{excitation})} - 1,$$

holds, then P(error on E+1) is maximized when P(excitation) is minimized. From (4.5.7), P(excitation) is minimized under the M error model when one assumes that all the $S(f_i)$ equals zero but one, yielding P(excitation) = 1/M.

It is noted that, when right side of (4.5.8) were multiplied by E, it represents a binomial distribution. When E is very large, a binomially distributed random variable can approximated by a Poisson distribution, yielding

$$P(\text{error on } E+1) = e^{-P(\text{excitation}) E} P(\text{excitation})$$

$$= P(\text{error/excitation}) P(\text{excitation}). \quad (4.5.9)$$

Partitioning the M Error Model into Regions

In many applications, the complexity of each line of code is such that the data gained from measuring its probability of excitation for the M error model is not sufficient to characterize its execution characteristics. The line of code may be executed continuously, but the computations involved may differ for each execution, i.e. the assumption of equiprobable excitation for the M errors may not be correct. When this is the case, execution statistics can be improved by assuming that the line of code is composed of L regions of operation, and excitation anywhere within a region follows the previously defined error model. Furthermore, it is assumed that the errors are evenly dispersed among the regions.

The output of each line of code is divided into L equally spaced regions of operation in this model. While it would be preferred to divide the input space of each line into regions, this is infeasible. Since the input of most lines of code is obtained from the output of previous lines, any variable that takes infrequent excursions could possibly excite latent errors as an input to succeeding lines. The expression for the probability of an error occurring after E executions of a line of code is

$$P(\text{error on } E+1) = \sum_{R=1}^{L} e^{-(E_R L)/M} P(\text{excitation}) \quad (4.5.10)$$

where E_R is the number of executions of the line yielding output values of the line within region number R.

4.6 Reliability Model for an Operational Flight Program

The probability of failure of an operational flight program can be obtained by summing the probabilities of error for each of the N lines of the program using the conditional relation

$$P(\text{failure}) = \sum_{K=1}^{N} P(\text{error/excitation})\ P(\text{excitation}). \quad (4.6.1)$$

Let the program be subjected to a randomly generated validation test during which no software failures are recorded, and let a particular line of code be executed EK times during that test. Then (4.5.9) can be used to calculate P(error/excitation).

P(excitation) in (4.6.1) is now the probability of excitation of the line of code during a typical flight. It is assumed that the probability of encountering any particular environmental condition is constant from flight to flight; mathematically, the hazard function is constant. This implies an exponential distribution whose hazard function is the execution rate of the line of code. The execution rate of the line of code can be measured during the validation test. If the duration of a particular flight scenario is T_F, execution rate is W, then the probability of excitation during that flight is

$$P(\text{excitation}) = (1 - e^{-W_K T_F}). \quad (4.6.2)$$

Before the program is subject to validation, verification testing has been performed. Verification testing exercises each line of the program against the software specification; many errors are detected and corrected during this phase. Verification testing cannot guarantee that the program works as a whole within the computer system; this is the function of validation. However, verification testing will be considered an independent check of the program for errors. The probability of an error existing after program verification will be termed the verification coverage, represented by C_T. Then the probability of failure given in (4.6.1) can be modified to account for the verification test, and from (4.5.9) and (4.6.2)

$$P(\text{failure}) = C_T \sum_{K=1}^{N} e^{-E_K/M} (1.0 - e^{-W_K T_F}). \quad (4.6.3)$$

4.7 Validation by Simulation versus Flight Testing

In order to obtain a statistically significant sample for evaluating (4.6.3), a large number amount of data would have to be collected with instrumented software. It is infeasible to do this during the flight test phase; rather, a large number of simulation runs can be made. Simulation alone cannot fully validate the program because some errors may be present that can only be activated in a flight environment. One may question how much flight testing should be done as part of the overall validation scheme.

One approach to evaluating software reliability from flight test involves estimating what fraction of errors are never excited during simulation testing, but will be excited during flight. Call this fraction PF. Then the probability of failure of the software due to errors that will only be excited during flight test is

$$P(\text{failure}) = C_T \sum_{K=1}^{N} e^{-W_K T_T / PF} \left(1.0 - e^{-W_K T_F} \right) \qquad (4.7.1)$$

where T_T is the number of hours of flight test without incident, and W_K was obtained from the simulation phase.

4.8 Implementing Procedure for the Methodology

To implement validation testing using this model, one must perform repeated simulations of the system under sufficient operating conditions to generate statistical data to evaluate (4.6.3) and (4.7.1). For an avionics system, this can be done by simulating the aircraft, its environment, and the hardware interfacing the processors to the aircraft. Simulation in this context could be entirely contained in a computer simulation package, or could be achieved using a combination of flight hardware (processors, sensors, actuators, etc.) connected to a simulation of airframe dynamics.

The simulation flies repeated "mission segments" using the OFP under test, each mission segment under a different set of environmental conditions. A mission segment could be a take-off, landing, following a specific terrain, etc. The specification of what constitutes a mission segment comes from the design document for the particular system. Many mission segments involve execution of a particular portion of code for a certain period of time, then switching to some other portion of the program for the next segment. Latency times within one simulated mission segment are insignificant when compared to the latency time that may accumulate over many executions of a mission segment. Statistics are gathered each time a mission segment is run indicating what regions of each variable's output space were exercised. The probability of excitation for a particular region of a variable is its probability of excitation during a single running of a mission segment.

The methodology for performing the validation is as follows:

- simulate a large number of repetitions of the mission segments being validated, changing the environmental conditions each time,

- choose environmental conditions by mapping randomly across high and low probability scenarios,

- record the statistical results - results may be cumulative over many different simulation instances (i.e., one can always perform more simulations if required),

- insure that none of the simulated performance resulted in system failure, and

- postprocess the statistics to obtain reliability numbers.

Additional Software Components Needed for Validation

To form the validation package, the following software components were necessary in addition to the OFP:

- a simulation of a suitable airframe with dynamics to represent the mode of flight controlled by the OFP,

- inputs representing environmental changes that the aircraft might encounter, such as winds, instrument noise, terrain variation, etc.,

- a system monitor to insure that system operation does not deviate from specifications, and

- a statistics gathering routine to measure raw data needed to calculate parameters for (4.6.3) and (4.7.1).

The statistics gathering program is called after each program line with the value of the output variable. This program compiles the highest and lowest value attained by each variable, and passes this data to the postprocessing program. In the case of subroutine calls, it also divides statistics based on calling path. In this manner each subroutine is treated statistically as if it were in-line code.

5. VALIDATION OF AN OPERATIONAL FLIGHT PROGRAM

5.1 Description of the Operational Flight Program

To test the proposed methodology, a small (200 line) FORTRAN program was written to implement the pitch autoland mode of operation. This OFP included an ILS glideslope coupler and a flare mode based on sensed radar altitude. A simulation of a Boeing 707 airframe in landing configuration, with appropriate ground effects, was used to model system dynamics.

Environmental changes were modeled as noise superimposed on the system inputs, including

- wind turbulence,
- steady state winds,
- wind shear,
- instrument noise,
- landing guidance noise, and
- variation in runway characteristics.

Specifications for environmental changes were obtained from applicable FAA documents, notably [29] and [30].

A system monitoring routine insured that parameters of interest remained within bounds during simulation. According to FAA document [30], the following boundaries were checked continuously

- normal acceleration within 1 G,
- pitch attitude less than 10 degrees,
- altitude deviation from glide path less than 20 feet,

while the following boundaries were checked after touchdown

- descent rate at touchdown is less than 5 feet/sec, and
- range from glide slope transmitter is less than 2000 feet.

If any of the above boundaries were violated, a catastrophic failure of the software is assumed to have occurred, and the software is invalid. If such a failure should have occurred, the OFP would be repaired and the validation process started over again.

The mission segment simulated for the validation test started

at 1000 feet with the aircraft flying straight and level. The OFP guided the aircraft through capture of the ILS glideslope through touchdown on the runway. The total duration of real time simulated was approximately 100 seconds, depending on atmospheric conditions. Because of limitations on the amount of computer resources available to run the tests, the mission segment was repeated only 600 times. During each simulation, 131 variables were monitored in the OFP. A sample of the monitored code is shown in Figure 5.1.1. While this did not provide enough data to validate the software to flight critical levels, it did provide enough data to evaluate the methodology.

5.2 Results of the Study

The results of the 600 simulations of the mission segment were recorded and analyzed by a postprocessor. To evaluate reliability, it was assumed that the aircraft landed once per hour, and that the OFP was verified to 95 % coverage before validation.

Table 5.2.1 shows the probability of computing values within each of ten regions associated with a particular variable within the OFP, along with the number of simulated landings which entered that region of operation. The regional occupancy probability is based on the relative probability of occurrence of each of the conditions which caused the OFP to enter that region, which may not be proportional to the number of simulated landings that entered that region. In evaluating (4.6.3), it is noted that the probability of occurrence is needed to evaluate W_K while E_K is the number of simulated landings that entered the region.

The regional occupancy probability was computed for all variables within the OFP and was tabulated. It revealed that a large percentage of regions have a very high probability of occupancy while a small number of regions had a very low percentage. Few regions had occupancy probabilities in between. The data is summarized in Table 5.2.2.

The M error model was evaluated setting M equal to 31, a number chosen arbitrarily. The probability of failure measured by the simulation was calculated to be 4.38×10^{-4} while the probability of failure measured by a 100 hour flight test would be 1.88×10^{-3} under the assumption that 5 % of the errors could only be found during flight test (PF equals .05). In order to assess the dependence of the measured probability of failure on the parameter M, a plot was made of this relationship, and is shown in Figure 5.1.2.

The 600 simulation runs were performed on a MicroVAX computer and took about 10 hours of computer time to complete. If a larger member of the VAX computer family were used, the same results could have been obtained in approximately 1 hour of computer time. Examining (4.6.3), it is noted that P(error/excitation) is an exponential function of $-E_K/M$. Hence, when the regional mappings don't change significantly, increasing the number of simulation runs is equivalent to decreasing M. Figure 5.2.2 can then be used to estimate the number of hours necessary to validate the software to flight critical levels, in this case 60 hours of simulation would validate the OFP to approximately 10^{-9} probability of failure under the M error model with M equals 31. While the 60 hours is only an estimate, it is a feasible amount of computer time for a validation effort.

During the 600 computer runs, one line of code was never executed. Upon investigation, it was associated with a logical

input, normally associated with failure recovery code, that was never exercised. To properly validate this OFP, the process must be repeated with this input properly simulated.

Conclusions

It was concluded that the methodology outlined is a feasible approach to validating critical software. The computing resources necessary to validate programs up to, say, a few thousand lines would be large, but not infeasible, although it does not seem practical to validate very large programs by this method. Instrumentation of the software to be validated was straightforward and could be easily automated.

An examination of the statistics produced during validation can be used to point out those internal regions of operation seldom used, and the code can be rewritten, or testing can be improved to insure their reliability. The effects of a successful flight test on reliability can be evaluated.

The method is dependent on the value of M assumed for the M error model, and also on the value of PF assumed for flight test, and more work is needed to establish suitable values for these parameters. More work is also needed to verify the methodology with large samples and larger programs.

Region Boundaries		Simulated Landings Entering Region	Occupancy Probability
-.27	-.20	33	.0272
-.20	-.13	477	.904
-.13	-.05	600	1.
-.05	.01	600	1.
.01	.07	600	1.
.07	.14	600	1.
.14	.20	575	.999
.20	.27	428	.966
.27	.33	22	.0143
.33	.40	1	.000155

Probability of Regional Occupancy for a Single Variable

Table 5.1.1

Probability of Execution	Percentage of Regions
.000012	13.7
.0142	2.6
.0285	1.85
.942	13.4
.957	2.06
.971	6.61
> .985	59.5

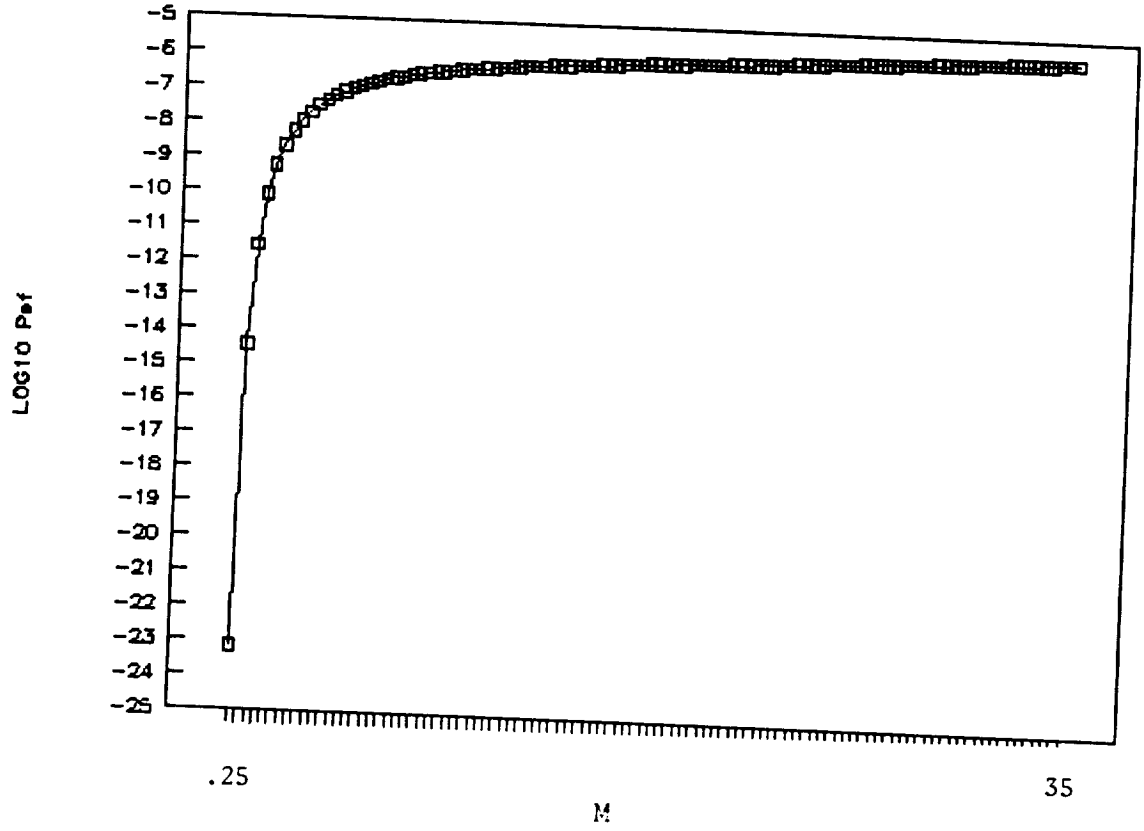
Probability of Execution of Regions of Variables

Table 5.1.2

```
C
C     ROLL ANGLE INPUT
C
          POINTS(3)=1.25*PHI
          CALL MON(1,3,POINTS(3))
C
C     RADIO ALTIMETER
C
          POINTS(4)=.5*HRA
          CALL MON(1,4,POINTS(4))
          IF(GSTRK) THEN
             POINTS(5)=4.7
             CALL MON(1,5,POINTS(5))
             ELSE
             POINTS(5)=15.0
             CALL MON(1,6,POINTS(5))
             ENDIF
          CALL LAG(POINTS(6),POINTS(5),5.,1.)
          CALL MON(1,7,POINTS(6))
          IF(HVALID) THEN
             POINTS(7)=POINTS(4)
             CALL MON(1,8,POINTS(7))
             ELSE
             POINTS97)=POINTS(6)
             CALL MON(1,9,POINTS(7))
             ENDIF
```

Sample of an Instrumented Program

Figure 5.1.1



Probability of Failure as a Function of M in the M Error Model

Figure 5.1.2

6. VALIDATION OF THE LAUNCH INTERCEPTER CONDITION PROGRAM

6.1 Description of the Program Under Test

The Launch Interceptor Condition Program is a variation on the operational flight program previously tested. The program describes a decision process which takes as input a set of (up to 100) radar points which might represent an incoming missile, or might represent just random noise. By testing the points, the program decides whether or not to launch an intercepting missile.

The program makes this decision by performing geometric calculations on the radar tracking points provided. Fifteen geometric tests are performed resulting in the generation of fifteen logical variables which represent intermediate indicators of whether or not a missile is present. These fifteen variables are combined using a logical weighting matrix to yield an overall answer of whether or not to fire the interceptor missile. The logical weighting matrix is generated randomly so that the characteristics of the program change from execution to execution for testing purposes.

Three versions of the LIC program were provided by NASA:

- a gold version, with no errors
- a program with an error in the geometric calculations
- a program with an invalid use of the arccosine function

These programs were originally used in a test of the N version methodology in which random sequences were generated representing radar tracks. Outputs of the N versions were then compared to uncover errors. However, the N version testing did not take into account the design specifications of the software, i.e., the radar tracks were not necessarily indicative of what the system would see in "normal" operation.

Validation testing using the methodology described in section 4 proceeds by defining the input specifications. To simulate radar tracking of an incoming missile, random numbers were used to generate the following quantities which were then combined:

- background noise for the radar,
- missile status (whether or not one was present),
- trajectory of incoming missile (when one is present), and
- radar echoes.

The radar track data was then fed to the LIC program using a single assumed weighting matrix for all tests. The output of the LIC program (launch or don't launch intercept missile) was compared to the missile status. The program was considered to have failed when it did not launch an intercept missile when a missile was present, or launched a missile needlessly.

6.2 Results of the Study

The Launch Intercept Condition program contained approximately 300 lines of code, and hooks were placed in the code to monitor the usage of each line. Twelve thousand simulation runs were made, and statistics were gathered.

A typical probability of regional occupancy for one of the lines of code is shown in Figure 6.2.1, this being a line that was frequently executed. Unfortunately, because of the nature of the code, there were many lines of code that were executed infrequently. This drastically reduces the overall reliability of the software, as execution of these lines of code might result in a software bug being uncovered. The overall probability of failure of the GOLD module was calculated to be

$$P(\text{failure}) = .5 \times 10^{-3}$$

The change in the probability of failure of the GOLD module as the number of cases tested was increased is shown in Figure 6.2.2.

The invalid arccosine function error was excited using the methodology. This was a data dependent error, and it was possible to run the program with data that did not cause it to fail. However, under the assumed input space,

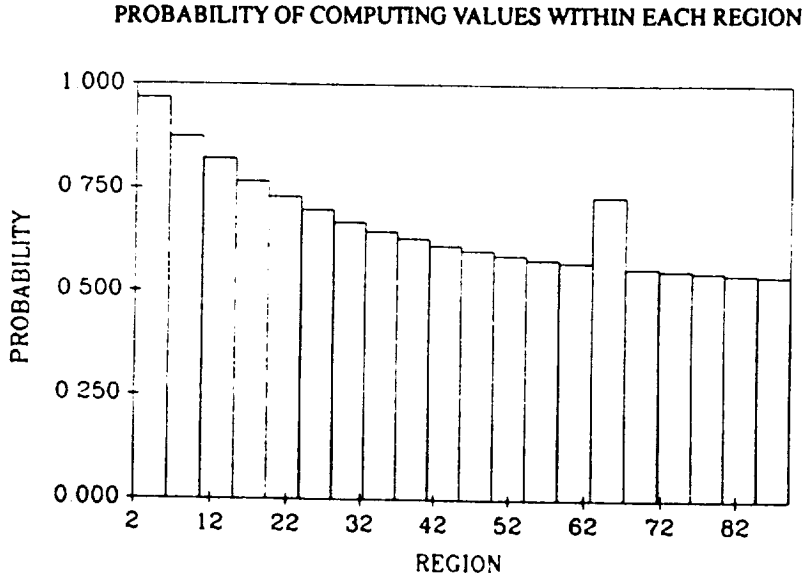
$$P(\text{excitation of invalid arccosine error}) = .5 \times 10^{-2}$$

The geometric calculation error was never excited in the 12,000 cases tested. From the large sample of input data it was concluded that:

- the error does not adversely affect system performance, and
- the error may never be excited with the input simulation used.

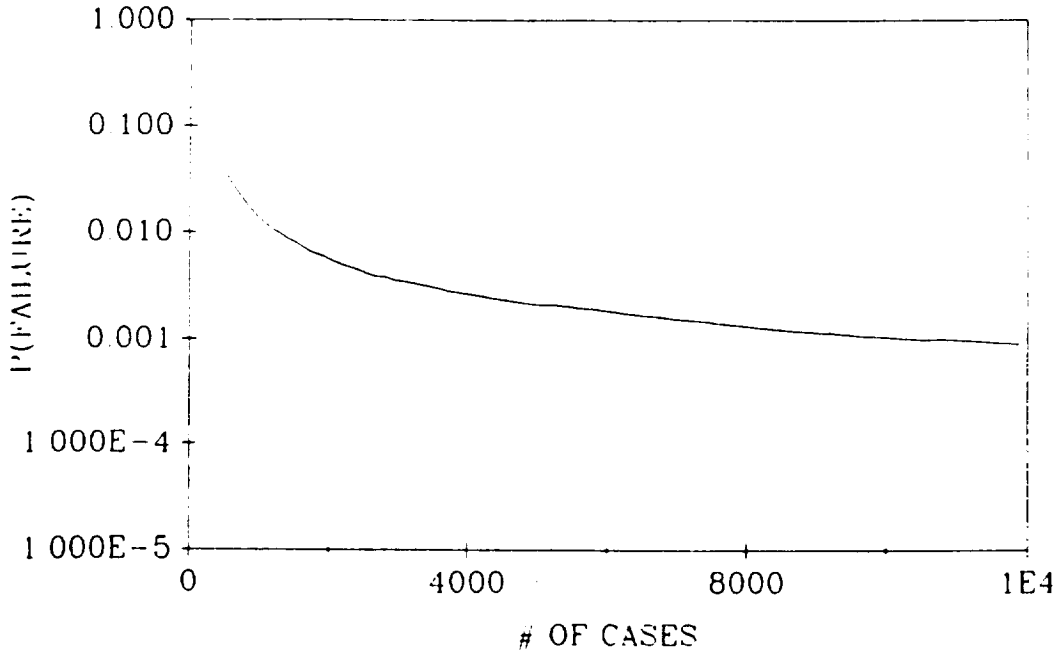
Considering the shape of Figure 6.2.2 and the high probability of failure of the GOLD module, the lack of detection of this error is not surprising.

- MAIN PROGRAM LINE NUMBER 19, EXECUTED 11,600 TIMES
- MINIMUM VALUE = 2; MAXIMUM VALUE = 89



Typical Probabilities of Regional Occupancy for a Single Output

Figure 6.2.1



Probability of Failure of the Gold Module

Figure 6.2.2

7. A SIMULATION METHODOLOGY FOR EVALUATING THE EFFECTS OF SINGLE EVENT UPSETS (SEUS)

7.1 Description of Single Even Upsets

New development in aircraft design will have a profound effect on the performance and cost of operating an airplane. the use of composite materials for the aircraft shell is very appealing because they are strong, yet light weight. Further, the airframe can be designed to reduce the drag, even when such a design results in deteriorated stability. Onboard computer systems acting as flight controllers can modify the handling characteristics of the aircraft to acceptable level. The result is an aircraft that performs well, and is cheaper to operate.

Computer chip technology has improved significantly, also. It is possible to build more functionality on a single chip, hence reducing the size of avionics equipment. However, the small thickness of the device and the high density of its circuitry make shell aircraft provided electrical shielding; composite materials do not. The metal cabinets which house an avionics computer may transients due to radiation may enter at signal and power connectors. Lightning and electromagnetic interference (EMI) represent possible radiation sources. Radiation effects may also be nuclear, even at low altitudes. While the physics of each type of radiation is different, their effect on a computer processing system is similar.

It is, of course, quite possible for high level radiation to cause permanent damage to the computing complex. This is a function of hardware design, not to be addressed here. It is a likely that even a mild dose of radiation will cause transient errors to occur in the processors. Such transient errors are termed Single Event Upsets. From a hardware standpoint, the processors are functionally capable of recovering from the transient if the error propagation from the SEU can be identified and stopped. However, if the SEU causes the software to malfunction, it may never recover.

Langley Research Center currently is investigating the susceptibility of an engine controller to SEUs caused by lightning. An analytical reliability model of the controller is described herein, which could be used as an aid for reliability prediction of this controller, and to generalize the results of those investigations to other aircraft controllers.

Recovery from an SEU

It is not clear whether a single event will cause multiple errors to be induced in a processor. If the upset affects only a single processor of a redundant set, it is possible that detection and isolation mechanisms will allow the system to recover from the event. However, if the upset affects more than one processor, disturbs the recovery algorithm, or affects a non-redundant processor, the system may not recover.

It is also possible that the event may never propagate to a point where it causes an error in controller operation. Should the upset never propagate to a latch or register within the CPU, it may never affect program execution.

Even if the event reaches a register, that register may not be used by the software at that instant. If this is the case, the system will again recover spontaneously from the upset.

Carrying the argument further, even if the event changes a software variable, the software may treat that change as noise, and still recover. However, if the software does fail, can it fail in such a way as to defeat the detection and recovery scheme? Remember, most detection and recovery schemes were designed to work correctly based on the assumption that faults occur solely due to component ageing.

7.2 Associating a Probability of Failure with an SEU

While the propagation and recovery mechanisms of SEU's are of interest to system designers, it is the probability of system failure due to SEU's that is of interest to the reliability engineer. Consider a single processor that experiences an SEU. When the SEU causes the processor to cease functioning, (branch to a bad ROM address, etc.) the watchdog timer will signal the system recovery mechanism. However, there is evidence [31] that in many instances, only a single bit in a single register will be affected, allowing the processor to continue executing instructions. It would be of interest to estimate the probability of failure (due to the SEU) when either a single bit, or a small number of bits, are affected.

Consider a reliability model in which the fault propagates hierarchically from the gate level to the system level. It is possible that recovery occurs at a particular level, halting error propagation to higher levels. Then the probability of recovery at each higher level is actually a conditional probability, i.e., the probability of recovery at that level given that the error has propagated to that level. If

$P_{\text{gate level recovery}}$ = Probability that the error never propagates from the affected gate to a register or latch

$P_{\text{register level recovery}}$ = Probability that the error never propagates from a register to a memory variable

$P_{\text{software recovery}}$ = Probability that the error in a memory variable is treated as "noise" by the software

and

PSEU = Probability that an SEU occurs within a given mission time

then

$$P(\text{SYSTEM FAILURE}) = PSEU * (1-P_{\text{gate level recovery}}) * (1-P_{\text{register level recovery}}) * (1-P_{\text{software recovery}})$$

In order to evaluate the above expression, it is necessary to obtain each of the required probabilities. PSEU can be obtained from the literature (such as [31]), from experiments, or circuit level modeling. The remaining probabilities can be obtained by gate level and functional level simulation for a particular hardware/software combination. Each of these parameters will now

be examined in detail.

Measuring $P_{\text{gate level recovery}}$

To measure the probability of gate level recovery, one can simulate the processor and peripheral hardware at the gate level, and inject transient faults. To do this in a meaningful manner, one must:

- define a mix of faults characteristic of the radiation environment to be studied,
- define the software to be executed on the processor, and
- define recovery within the processor.

The fault mix is a function of the type of SEU contemplated. If nuclear radiation effects are to be studied, the probability of a fault at any particular gate is a function of the gate density on the chip where that gate exists. In the absence of density data for a particular chip, one can assume that faults are equally likely to occur at any gate within the chip.

Lightning may also cause random effects across the chip due to electromagnetic radiation within the box. In an aircraft composed of composite materials, ship's wiring will pick up the lightning as EMI and transmit it to the processor via the processor's input connectors. Power lines can be filtered as they enter the box; however, radiation may be emitted from that portion of the input line prior to the filter. Feedthrough capacitors and chassis mounted line filters can help this situation; the amount of EMI emitted in any particular design must be measured for a particular hardware design. Signal lines cannot be filtered in the same manner as power supply lines, as this would obliterate the signal information. For lightning and other EMI sources, it is more likely that faults will originate on input or output gates than on gates located well within the processor.

A fault mix can be generated based on the above fault scenario descriptions.

It would be desirable to simulate the actual program the CPU will be executing to measure its characteristics. However, this would be too time consuming. Therefore, a small sample (about fifty) of the typical mix of instructions will be executed with a typical data mix. The mix can be obtained by measuring program execution (counting instructions, sampling data, etc.) in an unfailed environment.

Recovery of the processor is defined as no fault propagating to a register or latch. Should a fault propagate to a register, the register and its contents will be recorded for use in the next simulation phase.

To reiterate, the methodology for measuring recovery at the gate level is:

- choose a fault model and probability distribution,
- simulate about 50 instructions of program execution for each fault, and
- record the faults that never reach a register or latch, and which register or latch the remaining faults reach.

GGLOSS is a tool that can efficiently perform the above simulation tasks. The Stevens version of GGLOSS can run as a deductive simulator, yielding the required information in a timely manner.

Once the data is collected, a few assumptions are necessary for the final probability calculation. One must assume the number of clock cycles for which the fault exists. In the absence of other information, it is assumed that a fault is equally likely to occur at any time during program execution.

Measuring $P_{\text{register level recovery}}$

Register level recovery occurs when an error propagated to a register does not affect program execution. For instance, an error in the AX register of a microprocessor may occur just prior to that register being loaded with a constant value; - the previous (erroneous) value of the register is ignored, and the program executes correctly.

Gate level simulation data will predict what registers will contain errors due to an SEU. In order for the program to execute incorrectly, this error must propagate to memory (or a system output; the system outputs are usually memory mapped), or cause a program jump incorrectly. A register transfer level (or functional level) simulation of the processing complex can be used to measure the probability that the error propagates to affect program execution.

A register transfer level simulator will be used because:

- RTL simulators are simpler and require significantly less time to run, and
- a significant number of instructions must be simulated to obtain meaningful data.

A significant subset of the instructions should be simulated (~ 500 - 5000 instructions) depending on the software itself.

The output of the level of simulation should include:

- the probability that the error propagates to memory, and

- the probability that the error causes an extraneous jump.

Again, it is assumed that the SEU could occur at any time during execution.

Measuring $P_{\text{software recovery}}$

Should a memory variable be affected, the processor might still recover if the error is treated as noise by the processor. Each variable takes on a specific set of values during normal operation; if the error causes a variable to take on a value within this range, it may only show up as a noise point, and the system will recover.

On the other hand, if the variable takes on a value well outside of its operating range, the software may not function correctly.

Using the software validation methodology described previously [32], the validated operating range of each variable is available for comparison with that value created by the error. It is assumed that:

- If the error causes a variable to take on values within its operating range, then the system will recover.

- If the error causes a variable to take on values outside its operating range, this region of operation was never validated, and the system will fail.

By simulating the control program, instrumented for validation as in [32], the necessary probabilities can be calculated. It is also assumed that the error has an equal probability of occurring at any instant in time. Using this, an overall probability can be calculated.

Computing P(SYSTEM FAILURE)

The probability of system failure of a multiplex of processors is a function of the system architecture. With an architecture in mind, some description of the failure scenario must be made. One possible failure scenario is the following:

- all processors incur a simultaneous SEU

- the SEU affects each processor in a different manner

- majority voting is used

Then, if the architecture is an n-plex:

$$P(\text{SYSTEM FAILURE}) = PSEU * [(1-P_{\text{gate level recovery}})$$

$* (1-P_{\text{register level recovery}}) * (1-P_{\text{software recovery}})](n-1)/2$

Other expressions can be derived for other architectures and other failure scenarios.

8. CONCLUSIONS AND SUGGESTIONS FOR FUTURE WORK

It was concluded that simulation is a viable means for validating both hardware and software and associating a reliability number with each. This is useful in determining the overall probability of system failure of an embedded processor unit, and improving both the code and the hardware where necessary to meet reliability requirements. The methodologies were proved using some simple programs, and simple hardware models.

It would be desirable to test the GGLOSS methodology on more sophisticated hardware and more complex software. More specifically,

- Use GGLOSS to simulate a complete processing channel of a triplex or quad system,

- Simulate an actual OFP to surmise its latency characteristics,

- Estimate its probability of failure under the appropriate Markov model.

It would be desirable to test the software reliability methodology on a more complex OFP, and to demonstrate high reliability software (Probability of failure on the order of 10^{-9}).

It would be desirable to combine the methodologies to obtain results on the reactions of these processing systems to SEUs.

Appendix I. GGLOSS Users Manual

The Current Stevens Institute version of GGLOSS has been changed significantly from previous versions. These changes were made to achieve the following:

- make Stevens GGLOSS easier to use, more reliable, and easier to modify in the future, and

- add the capability of building a large simulation from modules representing components or partitions.

The resultant GGLOSS program has been tested by creating a partitioned version of the toy microprocessor, assembling a simulator, running it, and comparing the results with results obtained using the toy processor modeled under previous versions.

GGLOSS no longer prompts the user with a series of questions to decide on how to build the simulator. Rather, the main GGLOSS program has been rewritten to act as a command processor. This means that users may either type in interactively, or prepare a command file, of different functions that they would like GGLOSS to perform, along with parameters that either choose options or supply optional values. Only in certain instances (where it was deemed desirable) does GGLOSS prompt the user to enter data.

To run GGLOSS, type in (or set up a command file with)

```
R GGLOSS
```

The program will respond with its prompt

```
GGLOSS>
```

and wait for its first command. There are five different commands supported:

```
COMPILE
BIND
FLTGEN
MEMORY
INCLUDE
```

In addition a blank line may be entered (which will be ignored) and comments may be entered by starting the line with !. The command

```
END
```

will terminate the program.

Each command may be entered with optional parameters following it. Parameters are separated by a comma, and some parameters have an equal sign followed by data. If a command and its parameters

cannot fit on a single line, a dash (-) is used to indicate that the command is continued on the next line. However, each parameter must be completed on a single line. In the following command description, lower case letters represent parameters to be replaced by appropriate names while upper case letters represent parameters to be entered as shown.

GGLOSS COMMANDS

Examining each command in detail, along with its parameters:

COMPILEfilename,FAULT,IN=COMMON,IN=PARAMETERS,ICS,TABLE=tablefile

This command translates a single SDL file into an executable module. The parameters mean:

filename - the (complete) name of the SDL file

FAULT - signals the translating modules to compile a faulted module. If this parameter is omitted, the resulting module is true value (or fault free).

IN=COMMON \
IN=PARAMETERS/ - indicates whether the input pins for the module should be passed by subroutine parameter, or in common. If omitted, this parameter defaults to PARAMETER. It should be noted that PARAMETER is the only suitable method for inputting to a chip used in multiple places in the hierarchy; however, with PARAMETER, the user is limited to 64 pins. COMMON has no limitation as to the number of pins, and is meant for situations where the board has been partitioned, and there will be only one caller of the module.

ICS - indicates that the module contains initial conditions to be applied at the beginning of the simulation. If this parameter appears on the command line, the user will be prompted to supply the initial conditions by pin and value. It must be noted that initial conditions should not be specified for modules that are called from more than one location in the hierarchy, as this will cause problems in the present implementation.

TABLE=tablefile - indicates a file containing the failure rate of the gates involved in the module. If omitted, this defaults to GDAT:FAILURE.DAT. However, its inclusion allows different modules to be described with different failure statistics for each gate contained therein.

BIND modulename,REPEAT=nn,PRINT,INTERMIT,CYCLES=nn,FLT/MACH=nn,
RECDET=nn

This command takes information from previous compiles, and builds the "rest" of the simulator, including the control program, etc. The parameters are:

modulename - the name of the module (module names are given between colons after the word EXT) of the highest level module in the hierarchy.

REPEAT=nn - the number of times to repeat the simulation for each set of faults. GGLOSS allows the user to simulate inputs using a dataset, and, when this option is used, it is desirable to repeat each faulted run with a different input. This parameter tells how many different values are in the dataset for each input. If this parameter is omitted, it defaults to one.

PRINT - prints a hexadecimal list of the system output variables after each cycle for debugging purposes. If omitted, no printout is produced.

INTERMIT - it is desired to run intermittent faults. If omitted, faults are assumed solid. If included, the user is prompted for characteristics of the intermittent faults.

CYCLES=nn - the number of clock cycles to run each simulation. This parameter should not be omitted.

FLT/MACH=nn - the number of faults per machine. This defaults to one, although it is possible to run multiple faults in each machine using this parameter.

RECDET=nn - the number of detections to print (If all fault detections are printed out, the volume of paper may become excessive in large simulations). This parameter defaults to 5.

FLTGEN PIN=nn,RAM=nn,ROM=nn

This command generates random faults for the simulation using Monte-Carlo techniques based on the failure rate statistics of the module. The parameters are:

PIN=nn - the number of pin faults to inject. GGLOSS chooses nn faults randomly, based on the failure rate of each gate, across the entire faulted portion of the simulation.

RAM=nn - the number of RAM faults to inject. GGLOSS chooses nn faults randomly across all words in scratchpad memory which have been indicated as faultable using the MEMORY command. It then chooses a bit randomly

across the word, and also chooses whether that bit should be stuck at one or zero.

ROM=nn - the number of RAM faults to inject. GGLOSS chooses nn faults randomly across all words in read only memory which have been indicated as faultable using the MEMORY command. It then chooses a bit randomly across the word, and inverts that bit to fault it.

MEMORYmodname,TYPE=PROM,TYPE=RAM,ADDBITS=nn,DATABITS=nn,IO=INPUT, IO=OUTPUT,COMMON,INIT=filename,DIFF=nn,STARTADD=nn,LENGTH=nn, FAULT

This command generates a table within GGLOSS to handle the different memory chips that could be used to comprise PROM and RAM. It can also be used to model memory mapped I/O. The parameters are:

modname - the name of the memory chip. This must be the same as the name of the chip given in the partslist description. To facilitate building complex functional memories, multiple MEMORY statements may refer to the same name.

TYPE=PROM \ TYPE=RAM / - indicates whether the chip is PROM or RAM (default is PROM).

ADDBITS=nn - the number of address bits for the chip.

DATABITS=nn - the number of data bits for the chip.

IO=INPUT \ IO=OUTPUT / - for memory mapped IO, indicates whether input or output. Note that the inclusion of one of these two parameters identifies the chip as memory mapped IO.

COMMON - indicates that the memory is shared among more than one simulated processor, using adjacent machines in the VAX parallel simulation word.

INIT=filename - file containing data for PROM or initialization code for RAM

DIFF=nn - for multiprocessor simulation using shared memory (see COMMON above), the number of processors sharing the memory.

STARTADD=nn - the starting address in memory that should be simulated for memory chips too large to simulate the entire chip.

LENGTH=nn - the size of memory that should be simulated for memory chips too large to simulate the entire chip.

FAULT - this memory chip should be faulted when using the ROM or RAM (as appropriate) parameter on the FLTGEN command.

INCLUDE modulenm

This command lets GGLOSS know about previously compiled modules that may be used to build hierarchically larger structures without recompiling that module. It does this by reading in a dataset produced at compilation time containing relevant information about the module that would be needed for linking with other modules. The parameters are:

modulenm - the name of the module. Note that this is not a dataset name but rather a name included in the SDL file EXT field to identify the module.

END

This command terminates the GGLOSS program returning control to VMS.

PROGRAMMING WITH GGLOSS

To successfully build a simulator using GGLOSS, it is first necessary to construct datasets describing the digital system. A separate SDL file describes each module. At the beginning of the SDL file, the name of the module is included in the EXT statement between two colons, i.e.,

EXT:modname: pin1,pin2,etc.

Other modules refer to it by this name. In addition to the SDL files, the following files are necessary to assemble the simulator:

- a file containing the failure rates of each of the gates and submodules in the module. This file is in 'free' format (one does not have to adhere to columns, space or commas are delimiters) and uses the following line for gate primitives:

gatenm, failurerate

where

gatenm is the generic name of the gate as indicated in the SDL file TYPE statements, and

failurerate is the failure rate of the gate multiplied by 10**10,

and for submodules, the file line is:

modnm

where

modnm is the name of the module.

The failure rate of a submodule is obtained from previous assemblies of that module if either the module was compiled in this GGLOSS execution, or an INCLUDE statement for the module is given.

If the failure rate file is called GDAT:FAILURE.DAT, its name does not have to be included in each module compilation as this is the default name.

- It is suggested that when a new simulator is built, a VMS command file be constructed with the necessary GGLOSS statements. Certain options obtain their data by prompting the user, and when this data is sizeable, it is easier to place it in a command file. The data that must be entered by prompting is:

o initial conditions,

o intermittent fault data,

o print titles, and

o system inputs.

Note that system inputs may no longer be entered using a separate dataset, as was allowable under previous versions.

RUNNING THE GGLOSS PROGRAM

A GGLOSS simulation is built using a two-step process:

1 - The GGLOSS program is run as outlined in the above sections. The COMPILE commands produce the required BLISS modules representing modules in the digital system. The BIND command produces other required BLISS and FORTRAN modules, and also a command file named BLDSIM.COM. BLDSIM.COM contains VMS commands to compile and link the language modules into an executable simulator.

2 - The command file BLSDIM.COM is edited, if desired, and then executed. BLDSIM.COM will always contain enough commands to recompile every module in the simulation, link them, and then run the resulting simulator. BLISS compilation time is lengthy, and there will be cases when BLISS compiled versions of the modules are already available. Editing can result in a time efficient solution to this problem.

As an example of using GGLOSS, the following command file is used to create a simulation of the toy microprocessor:

```
$R GLOSS
COMPILE GDAT:JEANX1.SDL,FAULT
COMPILE GDAT:JEANX2.SDL,FAULT
```

```
COMPILE GDAT:JEANX3.SDL,FAULT
COMPILE GDAT:JEANXT.SDL,FAULT,TABLE=GDAT:TOYFAIL.DAT
MEMORY RAM,TYPE=PROM,ADDBITS=6,DATABITS=8,INIT=GDAT:TOY.MEM,FAULT
BIND TOY,CYCLES=100
*
*
*
*
*
*
1
0
0
1
Y
2
1
1
1
0
FLTGEN PIN=200,RAM=20
END
```

The first three COMPILE statements create three partitions of the toy processor called TOYALU,TOYADR,TOYCNT. The last compile is the highest level description of the toy, called TOY, that properly connects and invokes the previous three. Note that it uses a different failure rate file, which contains the names of the modules which TOY invokes.

The MEMORY statement gives the name of the chip used in the SDL description as RAM and describes its address and data length. It also names a dataset containing the program, and says the memory can be faulted.

The BIND statement indicates that the name of the main module is TOY (this is the module name, not the dataset name). It also indicates that the program will run for 100 clock cycles. Following lines indicate answers to questions about column headings and input values, as per their respective prompts.

The FLTGEN generates 200 pin level faults at random, and 20 RAM faults at random.

If after running the simulator, one were interested in injecting another 300 faults, it is not necessary to reconstruct the simulator. Rather, a file of the form:

```
$R GLOSS
INCLUDE TOY
FLTGEN PIN=300
END
```

would generate a new fault file. The old simulator could be run

again by entering

```
$R gsim:EXE
```

and it would use the new fault file.

GGLOSS datasets

GGLOSS requires many datasets to obtain all the information required to put together the simulation. Some of the required datasets have been listed in detail in previous reports. An overview of the datasets required is listed here:

LIBRARY.DAT A description of the gates used in GGLOSS

RTNES.R32 BLISS macros corresponding to the gates used in GGLOSS

PARTSLIST Description of the circuit

MEMORY.DAT Data to be entered in ROM or RAM. The first line of the memory dataset has been changed to be:

LODMEM,HIDMEM,LRWMEM,HIWMEM,ENDPC,WMEMBS,LOIOOT,HIIOOT,LOIOIN, HIIOIN,LOCOM,HICOM,NODM

where

LODMEM - Low address of Read Only Memory
HIDMEM - High address of Read Only Memory
LRWMEM - Low address of writable memory (RAM)
HIWMEM - High address of writable memory (RAM)
ENDPC - Program ending address (not presently used)
WMEMBS - Address of stack segment (not presently used)
LOIOOT - Low address of memory mapped output (within RAM)
HIIOOT - High address of memory mapped output (within RAM)
LOIOIN - Low address of memory mapped input (within RAM)
HIIOIN - High address of memory mapped input (within RAM)
LOCOM - Low address of common area (within RAM)
HICOM - High address of common area (within RAM)
NODM - Number of different processors in complex

The remaining lines contain the data to be stored in ROM, repeated NODM times. Each set of ROM data ends with an END statement.

INPUTS.DAT A list of input values for memory mapped input encoded as one on a line in the order they will be requested by the simulated processor's program. No check is made to insure that the proper memory mapped address was entered; this mechanism works solely on the order of request at the present time.

FAILURE.DAT Contains the failure rates of each component for use in randomly choosing failures. The format of each line of the dataset is:

TYPE,RATE,NUMBER

where

TYPE - Type of gate as listed in the partslist
RATE - Failure rate of the gate scaled times 10^{10}
NUMBER - Number of gates of this type in the circuit

ROM FAULT FILE A file containing the addresses ranges of ROM to generate random ROM faults in. It contains one line with two numbers separated by commas listing the starting and ending addresses.

RAM FAULT FILE A file containing a list of RAM faults. The lines are in the following format:

ADDRESS, BIT, 1 OR 0

BIBLIOGRAPHY

1. Hopkins, A. L., Smith, T. B., and Lala, J. H.,"FTMP - A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft," Proceedings of the IEEE, vol. 66, no. 10, October, 1978, pp 1221-1239.

2. Ramamoorthy, C. V. and Han, Y,"Reliability Analysis of Systems with Concurrent Error Detection'" IEEE Transactions on Computers, vol C24, no. 9, September, 1975, pp 868-878.

3. Hopkins, A. L., and Smith, T. B.,"The Architectural Elements of a Symmetric Fault-Tolerant Multiprocessor'" IEEE Transactions on Computers, vol C24, no. 5, May, 1975, pp 498-505.

4. Bavuso, S. J.,"Impact of Coverage on the Reliability of a Fault Tolerant Computer," NASA TN D-7938, September, 1975.

5. Nagel, P.,"Modeling of a Latent Fault Detector in a Digital System," NASA Contractor Report 145371, NASA Langley Research Center, August 1978.

6. McGough, J. G., and Swern, F. L.,"Measurement of Fault Latency in a Digital Avionic Miniprocessor," NASA Contractor Report 3462, NASA Langley Research Center, October, 1981.

7. McGough, J. G.,"Effects of Near-Coincident Faults in Multiprocessor Systems," in Proc. AIAA/IEEE Digital Avionics System Conference, 1983, pp 16.6.1-16.6.7.

8. McGough, J. G., Smotherman, M., and Trivedi, K. S.,"The Conservativeness of Reliability Estimates Based on Instantaneous Coverage," IEEE Transactions on Computers, vol C34, no. 7, July, 1985, pp 602-609.

9. Meyer, J. F., and Sundstrom, R. J.,"On-Line Diagnosis of Unrestricted Faults," IEEE Transactions on Computers, vol. C24, no. 5, May, 1975, pp 468-475.

10. Meyer, J. F., Furchtgott, D. G., and Wu, L. T,"Performability Evaluation of the SIFT Computer," IEEE Transactions on Computers, vol. C29, no. 6, June, 1980, pp 501-509.

11. Geist, R. M., and Trivedi, K. S.,"Ultrahigh Reliability Prediction for Fault-Tolerant Computer Systems," IEEE Transactions on Computers, vol. C32, no. 12, December, 1983, pp 1118-1127.

12. Smotherman, M., Geist, R. M., and Trivedi, K.,"Provably Conservative Approximations to Complex Reliability Models," IEEE Transactions on Computers, vol. C35, no. 4, April, 1986,

pp 333-338.

13. Shin, K. G., Krishna, C. M., and Lee, Y-H,"A Unified Method for Evaluating Real-Time Computer Controllers and Its Applications," IEEE Transactions on Automatic Control, vol. AC30, no. 4, April 1985, pp 357-366.

14. Stiffler, J. J., and Bryant, L. A.,"CARE III Phase II Report - Mathematical Description," NASA Contractor Report 3566, NASA Langley Research Center, November, 1982.

15. McGough, J. G.,"Feasibility Study for a Generalized Gate Level Logic Software Simulator," NASA Contractor Report 172159, April, 1983.

16. Lamport, L., Shostak, R., and Pease, M.,"The Byzantine Generals Problem," ACM Transactions on Programming Languages and System, vol. 4, no. 3, pp. 382-401, July, 1982.

17. Swern, F. L, Bavuso, S. J., Martensen, A. L., and Miner, P. S.,"The Effects of Latent Faults on Highly Reliable Computer Systems" IEEE Transactions on Computers, vol C-36, No. 8, August, 1987, pp 1000-1005.

18. Swern, F. L., Bavuso, S. J., Martensen, A. L., and Miner, P. S.,"A Latent Fault Markov Model for a Highly Reliable Triplex Computer System," in Proceedings of the AIAA Guidance, Navigation and Control Conference, Monterey, Ca. August, 1987, pp 1554-1559.

19. Shin, K. G., and Lee, Y-H,"Measurement and Application of Fault Latency," IEEE Transactions on Computers, vol. C35, no. 4, April, 1986, pp 370-375.

20. Huang, J. C.,"Program Instrumentation and Software Testing," Computer, vol. 11, no. 4, April, 1978, pp25-32.

21. Musa, J. D.,"A Theory of Software Reliability and Its Application," IEEE Transactions on Software Engineering, vol. SE-1, no. 3, Sept. 1975, pp312-327.

22. Scott, R. K., Gault, J. W., and McAllister, D. F.,"Fault-Tolerant Software Reliability Modeling," IEEE Transactions on Software Engineering, vol SE-13, no. 5, May, 1987, pp 582-592.

23. A. L. Goel,"Software Reliability Models: Assumptions, Limitations and Applicability," IEEE Transactions on Software Engineering, vol. SE-11, no. 12, Dec. 1985, pp. 1411-1423.

24. J. D. Musa, A. Iannino, and K. Okumoto, Software Reliability, New York: McGraw Hill, 1987.

25. F.-W. Scholz,"Software Reliability Modeling and Analysis," IEEE Transactions on Software Engineering, vol. SE-12, no. 1,

Jan., 1986, pp. 25-31.

26. E. Nelson,"Estimating Software Reliability from Test Data," in Microelectronics and Reliability, vol. 17. New York: Pergamon, 1978, pp. 67-74.

27. C. V. Ramamoorthy and F. B. Bastani,"Software Reliability - Status and Perspectives," IEEE Transactions on Software Engineering, vol. SE-8, no. 4, July, 1982, pp. 354-370.

28. A. Avizienis,"The N-Version Approach to Fault-Tolerant Software," IEEE Transactions on Software Engineering, vol. SE-11, no. 12, Dec., 1985, pp. 1491-1501.

29. Federal Aviation Administration, Criteria for Approving Category I and Category II Landing Minima for FAR 121 Operators, AC 120-29, September 25, 1970.

30. Federal Aviation Administration, Automatic Landing Systems, AC 20-57A, January 12, 1971.

31. R. Harboe-Sorensen, et al.,"The SEU Risk Assessment of Z80A, 8086, and 80C86 Microprocessor Intended for use in a Low Altitude Polar Orbit," IEEE Trans. Nuc. Sci., vol. NS-33, no. 6, Dec. 1986, pp. 1626-1631.

32. E. Petersen and J. Langworthy,"Suggested Single Event Upset Figure of Merit," IEEE Trans. Nuc. Sci., vol. NS-30, no. 6, Dec. 1983, pp. 4533-4539.

33. A. M. Finn,"System Effects of Single Event Upsets," in AIAA Computers in Aerospace Conference VII, Monterey, Ca., Oct. 3-5, 1989, pp. 994 - 1001.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 1994	Contractor Report

4. TITLE AND SUBTITLE
Hardware and Software Reliability Estimation Using Simulations

5. FUNDING NUMBERS
G NAG1-587

WU 505-66-21

6. AUTHOR(S)
Frederic L. Swern

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Stevens Institute of Technology
Castle Point
Hoboken, NJ 07030

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
505-66-21

NASA CR-187588

11. SUPPLEMENTARY NOTES
NASA Langley Senior Project Engineer: Salvatore J. Bavuso
Replaces NASA CR-186637 (90N25580).

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Subject Category - 61
Uhclassified - unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

The simulation technique is used to explore the validation of both hardware and software. It was concluded that simulation is a viable means for validating both hardware and software and associating a reliability number with each. This is useful in determining the overall probability of system failure of an embedded processor unit, and improving both the code and the hardware where necessary to meet reliability requirements. The methodologies were proved using some simple programs, and simple hardware models.

14. SUBJECT TERMS
Simulation, GGLOSS hardware, software, validation

15. NUMBER OF PAGES
159

16. PRICE CODE
A08

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	