

A METHODOLOGY FOR SOFTWARE IMPLEMENTED TRANSIENT ERROR
RECOVERY IN SPACECRAFT COMPUTATION

by

EVERETT NORCROSS MCKAY

SUBMITTED TO THE DEPARTMENT OF
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREES OF

BACHELOR OF SCIENCE IN ELECTRICAL ENGINEERING

and

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
January 18, 1985

© Everett Norcross McKay, 1985

The author hereby grants to M.I.T. and Hughes Aircraft Company permission
to reproduce and distribute copies of this thesis document in whole or in
part.

Signature of Author.....
Department of EECS

Certified by.....
James T. Yonemoto
Thesis Supervisor
Hughes Aircraft Company

Certified by.....
Professor Nancy Lynch
Thesis Supervisor

Accepted by.....
Professor Arthur C. Smith, Chairman
Departmental Graduate Committee
Department of EECS

ARCHIVES

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

APR 01 1985

LIBRARIES

A METHODOLOGY FOR SOFTWARE IMPLEMENTED TRANSIENT ERROR
RECOVERY IN SPACECRAFT COMPUTATION

by

EVERETT NORCROSS MCKAY

Submitted to the Department of Electrical Engineering
and Computer Science on January 18, 1985 in partial fulfillment
of the requirements for the combined Degrees of Bachelor of
Science and Master of Science in Electrical Engineering

ABSTRACT

Software Implemented Transient Error Recovery in spacecraft computation is the ability of a spacecraft to recover from transient errors using software techniques alone. Transient errors, typically caused by high-energy cosmic radiation, are the primary source of error in spacecraft computation.

The objective of this paper is to present a specific methodology for employing the Software Implemented Transient Error Recovery techniques. The methodology has three objectives: to limit the propagation of errors by performing computations on temporary objects, to detect errors by providing redundant information, and to correct errors by determining the appropriate recovery action by interpreting redundant information.

The methodology is an improvement of the approach used on the Intelsat VI attitude control sub-system, and was derived with the assistance of a computer simulation of a processor experiencing single and multiple bit upsets. Various performance metrics are discussed. The metric used to develop the methodology is the least probability of first-order catastrophe. A probabilistic analysis of systems using the methodology is performed. In the analysis, the conditional probability of catastrophe given the occurrence of an upset and the mean time to catastrophe are calculated with parameters from the Intelsat VI attitude control sub-system and with parameters from a possible future spacecraft control sub-system, both with and without Software Implemented Transient Error Recovery.

The proposed methodology provides several advantages over previous approaches. The most important of these advantages are that it is a structured, standardized approach, capable of recovering from multiple bit upsets and under most circumstances, it can recover from transient errors without re-initialization or restarting.

Thesis Supervisor: Prof. Nancy Lynch

Title: Associate Professor of Electrical Engineering
and Computer Science

ACKNOWLEDGEMENT

This project was inspired by a presentation on software error recovery for the Intelsat VI attitude control system made by Ron Obert in the Summer of 1983.

I would like to thank Jim Yonemoto for his guidance and support throughout the project. His careful attention to detail was most helpful. I would like to thank George Hrycenko for his support and concern. I am especially grateful to all the people who took the time to review the many drafts of the thesis, including Jim Yonemoto, Ron Obert, and Prof. Lynch. Lastly, I would like to thank Hughes Aircraft Company and the Engineering Internship Program at M.I.T. for making the opportunity possible.

TABLE OF CONTENTS

	page
ABSTRACT.....	II
ACKNOWLEDGEMENT.....	III
TABLE OF CONTENTS.....	IV
LIST OF FIGURES.....	VII
LIST OF TABLES.....	VII
1. INTRODUCTION.....	1
1.1 Autonomous Spacecraft Maintenance.....	2
1.2 Fault-Tolerance.....	3
1.3 Need for Transient Error Recovery.....	3
1.4 Previous Work.....	5
1.5 Advantages of New Approach.....	12
1.6 Type of System Assumed.....	14
1.7 Overview of Thesis.....	14
2. PROPOSED METHODOLOGY.....	16
2.1 Introduction.....	16
2.2 Discussion of General Techniques.....	16
2.2.1 Error Propagation Control.....	16
2.2.2 Error Detection.....	18
2.3 Discussion of Recovery Method.....	20
2.3.1 Computation Blocks and Idempotent Sections.....	20
2.3.2 Additional Considerations.....	22
2.4 Recovery Software Format.....	22
2.4.1 Computation Block Format.....	23
2.4.2 Recovery Block Format.....	26
2.5 Extending Technique to Real Computations.....	29

3.	SIMULATION ANALYSIS.....	33
3.1	Introduction.....	33
3.2	The System Models.....	36
3.2.1	The Multiple Bit Upset Model.....	36
3.2.2	The Upset Mapping Model.....	36
3.2.3	The High-Level Language/Machine Language Model.....	40
3.2.4	The Structure/Content Model.....	42
3.3	Performance Metrics for Recovery Evaluation.....	44
3.3.1	Coverage and Recovery Profile.....	45
3.3.2	Catastrophes.....	48
3.3.3	Time and Space.....	48
3.4	Discussion of Results.....	49
4.	PROBABALISTIC ANALYSIS.....	51
4.1	Introduction.....	51
4.2	Detailed Probabilistic Analysis.....	52
4.2.1	Definitions.....	52
4.2.2	Failure Classifications.....	54
4.2.3	General Detailed Probabilistic Analysis.....	56
4.2.4	Intelsat VI ACE Example.....	61
4.2.5	Future Spacecraft Example.....	65
4.3	Register Method.....	69
4.3.1	Register Method Procedure.....	70
4.3.2	The Worst Case Methodology.....	73
4.3.3	The Intelsat VI ACE Methodology.....	74
4.3.4	The Proposed Methodology.....	75
4.3.5	The Best Case Methodology.....	76
4.4	Summary of Results.....	77
4.5	Discussion of Results.....	78

5.	CONCLUSION.....	80
APPENDIX		
	ADDITIONAL CONSIDERATIONS FOR TRANSIENT ERROR RECOVERY.....	81
	THE SIMULATION PROGRAM DESCRIPTION.....	87
	THE SIMULATION PROGRAM LISTING.....	92
	GLOSSARY.....	123
	BIBLIOGRAPHY.....	136

LIST OF FIGURES

	page
Sample Computation Block in Error Recovery Format.....	24
Sample Error Recovery Block.....	28
Simulation System Parameters.....	88
Outline of Simulation Program.....	91

LIST OF TABLES

	page
Intelsat VI ACE Memory Parameters.....	72
Future Spacecraft Sub-system Memory Parameters.....	72
Worst Case Methodology Coverage.....	73
Intelsat VI ACE Methodology Coverage.....	74
Proposed Methodology Coverage.....	75
Best Case Methodology Coverage.....	76
Probability Of Catastrophe - Intelsat VI ACE Configuration.....	77
Probability Of Catastrophe - Future Spacecraft Configuration.....	77
Mean Time To Catastrophe - Intelsat VI ACE Configuration.....	78
Mean Time To Catastrophe - Future Spacecraft Configuration.....	78

INTRODUCTION

Software Implemented Transient Error Recovery in spacecraft computation is the ability of a spacecraft to recover from transient errors using software techniques alone. Transient errors, typically caused by high-energy cosmic radiation, are the primary source of error in spacecraft computation.

The objective of this paper is to present a specific methodology for employing the Software Implemented Transient Error Recovery techniques. The methodology has three objectives: to limit the propagation of errors by performing computations on temporary objects, to detect errors by providing redundant information, and to correct errors by determining the appropriate recovery action by interpreting redundant information. It will be assumed throughout this paper that the methodology will be applied to real-time spacecraft control software. Control software lends itself well to the structuring required by the methodology.

The methodology is an improvement of the approach used on the Intelsat VI attitude control sub-system [14], and was derived with the assistance of a computer simulation of a processor experiencing single and multiple bit upsets. Various performance metrics are discussed. The metric used to develop the methodology is the least probability of first-order catastrophe. A probabilistic analysis of systems using the methodology is performed. In the analysis, the conditional probability of catastrophe given the occurrence of an upset and the mean time to catastrophe are calculated with parameters from the Intelsat VI attitude control sub-system and with parameters from a possible future spacecraft control sub-system, both with and without Software Implemented Transient Error Recovery.

The proposed methodology provides several advantages over previous approaches. The most important of these advantages are that it is a structured, standardized approach, capable of recovering from multiple bit upsets and under most circumstances, it can recover from transient errors without re-initialization or restarting.

1.1 Autonomous Spacecraft Maintenance

Software Implemented Transient Error Recovery is part of an overall system goal of autonomous spacecraft maintenance (ASM). ASM is an attribute of a spacecraft system which allows continuous operation without external control, and performance of its specified mission at an established level for a specified period of time, even in the event of failure of one or more of its components. The scope of ASM includes spacecraft hardware maintenance, navigation and stationkeeping, and mission sequencing. ASM has applications in military and commercial satellites, as well as deep-space probes.

In military applications, ASM removes the vulnerability of telemetry and command communication links by the elimination of their continuous dependence upon ground stations for maintenance and control.

ASM is useful in extending the availability of commercial satellites. ASM reduces the operational cost of commercial satellites by minimizing the manpower and support equipment requirements. ASM can be used to correct problems in the critical elements of a spacecraft, such as attitude control and power.

Deep space probes have critical periods (during planet fly-by, for example) during which a system error could result in mission failure. Ground support is not helpful due to the long transmission delays between the ground station and the spacecraft. ASM increases the probability of

mission success.

1.2 Fault-Tolerance

Autonomous spacecraft hardware maintenance requires that a spacecraft must continue to operate in the presence of hardware faults. Fault-tolerance is defined as the ability of a system to perform correctly in the presence of faults. Although there are techniques for providing fault-tolerance capabilities on terrestrial computer systems through redundant hardware, these methods require too much weight, power, and space for practical usage in most current spacecraft applications.

Present spacecraft provide computer hardware redundancy at the sub-system level. For example, if a memory unit on an attitude control computer were to fail, the entire attitude control computer system would be replaced with a spare. Although this method allows only as many failures as spares, very few satellite failures have been attributed to on-board computer failures. However, present satellites have very limited on-board computing power. Future satellites, many of which may do on-board signal processing of received signals, will require much more computing power, which could lead to more computer failures.

1.3 Need for Transient Error Recovery

Autonomous spacecraft hardware maintenance also requires that a spacecraft must continue to operate in the presence of transient errors. Although errors caused by transient sources are usually less serious than errors caused by permanent sources, their greater frequency make them as important. On present spacecraft, the mean time to permanent failure is on the order of years, whereas the mean time to transient error is on the

order of days.

I do not intend to suggest that transient error recovery is more important than fault-tolerance or should be done in lieu of fault-tolerance. Autonomous spacecraft maintenance requires both techniques, and future systems may try to integrate both approaches. However, given the state-of-the-art in fault-tolerant computing, and given current spacecraft reliability requirements, I believe that it is more cost effective to address transient error recovery.

The known causes of transient error in spacecraft computation are:

1) Single Event Upsets

Single event upsets are caused by high-energy cosmic particles resulting in an ionized track approximately one micrometer in diameter for approximately one nanosecond. Because of their small size, cosmic particles can result in at most one bit flip per particle.

2) Electrostatic Discharge

Electrostatic discharge is caused by the discharging of large potential differences generated during a spacecraft eclipse in a magnetic substorm. Such magnetic disturbances can cause a spacecraft in geosynchronous orbit to charge up differentially to a 20-kilovolt range [20]. This problem is capable of causing permanent as well as transient errors. These transient errors can result in multiple bit flips.

3) Thermal Noise

Thermal noise has the most effect on corrupting analog voltage of spacecraft sensors, resulting in multiple bit errors.

4) Intermittent Hardware Failures

Although errors caused by intermittent hardware failures can be masked using software techniques, they cannot be repaired using software techniques alone. Consequently, intermittent hardware failures are not addressed in this paper.

Several studies have tried to determine the single event upset rate for various components [3,11]. The most common figure used is 1 e ^{-4} upsets/(bit-day). At this rate, a spacecraft sub-system such as the Intelsat VI ACE with 13 K bits of main memory will experience over one upset per day on average. No such upset rate data exists for electrostatic discharge and thermal noise.

1.4 Previous Work

Most of the previous work in fault-tolerant computing has not addressed transient error recovery [2,4,5,10,17,18,19]. Some authors have addressed transient errors in the form of intermittent hardware failures [16,21,22]. Intermittent hardware failures are not addressed in this paper, since they cannot be corrected using software techniques alone.

The principle recovery scheme used in this paper is essentially a simplified variation of the program rollback recovery schemes often used in database management operating systems [1,6,15]. However, rollback recovery is used in database management to undo correctly performed actions in order to eliminate deadlock, which is not a concern in transient error recovery. In transient error recovery, program rollback occurs when the correctness of the initial execution of a program segment is doubtful, which is not a concern in database systems. Consequently,

many of the ideas used in database recovery schemes are not relevant to transient error recovery.

All of the error detection schemes used in this paper can be found in other sources [6,13,14,16]. Many of these ideas are currently being employed on the Intelsat VI attitude control sub-system developed by Ron Obert at Hughes Aircraft Company [14], which is described below.

Summary of the Intelsat VI ACE Transient Error Protection

The goal of Intelsat VI ACE transient error protection is to make the ACE operation immune to single event upsets. The ACE hardware is assumed fixed, so only software solutions are considered.

The main idea to the Intelsat VI ACE approach to transient error protection is to take advantage of fact that 1) the attitude control system is greatly oversampled, resulting in a natural immunity to error and 2) the majority of processing time is spent in a wait loop, which can easily be made immune to most upsets.

The important recovery techniques used in the Intelsat VI ACE are:

Jump return to wait loop:

Jump return to wait loop is a method for detecting and recovering from sequencing errors which occur during the execution of the wait loop. The control program of the ACE sub-system will spend the majority of its run-time in a wait loop. Since wait loops are short, the number of mutations of the wait loop instructions that a single bit flip could cause is small. Here is an example of what could be done:

```
; program fragment
104  add r1, r2
105  jmp 107
106  jmp 200
107  continue...
```

```
; wait loop
200  ei
201  jmp 200
202  jmp 200
```

Suppose an SEU changed line 200 from enable interrupt to jmp 106. Under normal operation, it is impossible for the program to execute line 106, since it is intentionally by-passed by line 105. Consequently, embedding a jump return to the wait loop instruction would recover from such an upset.

Since the amount of time spent in the wait loop is large, this method is very effective in recovering from SEUs, but is ineffective in recovering from larger upsets since mutations of only one bit are considered.

Sequence control codes (SCC):

The testing of sequence control codes is used as a method of sequence error detection. A variable is set to a known value before a section is entered. This variable is then checked at the end of the section. If there is a discrepancy, entry into the section must have been at some point other than the proper entry point of that section. If such a sequence error is detected, the computation is aborted, and the program jumps to the wait-loop. No method is used to prevent error propagation. Consequently, the SCC tests had to be used with a high density to be effective.

Error-Correction Mode Algorithm (ECM):

The Error-Correction Mode is an alternate mode of operation which goes into effect when the difference between the actual output and the estimated output is greater than some margin. With ECM enabled, the ACE will use the average of past outputs for 10 seconds while the system re-initializes.

Protected program sections:

By placing important calculations which must be performed with each real-time interrupt at the beginning of the control loop, the protected program sections have an increased probability of successful execution.

Redundant storage and voting of critical parameters:

Redundant storage protects critical parameters from memory upsets.

Hamming error-correction codes with periodic update:

Error correction codes prevent bit flip errors in main memory. Periodic register updating reduces the probability that two upsets occur in the same word.

A simulation analysis was conducted to measure the effectiveness of the transient error protection. The actual flight code was executed on a 2901 and 2910 simulator, which is capable of arbitrarily flipping individual bits. The simulation model contained a total of 243 flip-flops, 223 in the 2901 and 2910, and 20 in the remaining hardware. Each bit was flipped twice at a random time within a 2 second processing interval. The 486 bit flips correspond to several missions' worth of SEUs in the processor. Main memory upsets were not considered, since

error correction codes provide sufficient protection against single event upsets. The criterion for system failure in the simulation was observing an output error of greater than 0.003 degrees.

The results of simulation analysis were 1) The placement of SCC tests within the flight code had little effect in results, 2) Without the ECM, an average of 6.33 pointing errors of greater than 0.003 degrees were observed from 486 trials, giving 1.3 % probability of failure, 3) With the ECM, an average of 2 pointing errors greater than 0.003 degrees were observed from 486 trials, giving 0.41 % probability of failure, 4) Only active processing time was simulated. Results should be factored by $(1 - \text{time_in_wait_loop})$ to take wait-loop immunity into account, and 5) The probability of system failure by upset is dominated by three unprotected flip-flops.

Another system which uses many of the error detection schemes used in this paper is the DC-9-80 Digital Flight Guidance System developed by Sperry Flight Systems [16]. The major difference between the proposed system (as well as the Intelsat VI ACE) and the DC-9-80 is that the DC-9-80 system uses both hardware redundancy and software self-monitoring techniques to detect transient errors and hardware failures, whereas the proposed system uses only software techniques to recover from transient errors.

Summary of the DC-9-80 Digital Flight Guidance System

Transient Error Detection

The principle concepts in the DC-9-80 approach to transient error detection are : 1) use a series of error detecting "screens" to monitor the correctness of program execution. The screens are redundant in the

types of errors they monitor, so if one screen fails to perform correctly for any reason, the remaining screens can insure correct execution.

2) create a hardware and software structure that simulates two separate computers while using only one processor. These "virtual" computers execute the same software, but use different memory blocks, different sets of data, and different sensor inputs.

The important error detection techniques used in the DC-9-80 Digital Flight Guidance System are:

Redundant Storage:

Dual storage of critical parameters is maintained in separate memory banks. Estimates of "correct" values are made by averaging the two values, except when the discrepancy between the values exceeds specified criteria.

Processor Self-Monitoring Program (BITE):

This program executes the entire processor repertoire each iteration of the flight-program. It tests the processor's instruction set using a full range of test numbers, tests all cases of the branching instructions, and uses the internal bus structure and associated registers at their maximum data rates.

Redundant Computation:

The main function of redundant computations is to guard against transient errors. All critical computations are performed twice, using different sets of data stored in entirely different sections of RAM. Estimates of "correct" values are made by averaging the two values, except when the discrepancy between the values exceeds specified criteria. If a transient error were to occur during the performance of one of the calculations,

it would be detected unless a similar transient error occurred during the performance of the other calculation.

Reasonableness Checking:

When a variable is known to have some range of correct values, then the actual value of the variable can be compared to this range to check the reasonableness of the value. If a discrepancy is found, an error has occurred.

External Hardware Monitor:

The external hardware monitor is similar to the watch-dog timer used on the Intelsat VI ACE. At the end of each control loop, a signal to the external monitor is pulsed, which results in a steady stream of pulses during normal operation. An interruption in these pulses indicates computer malfunction. The external monitor itself is tested by the computer during power-up initialization.

Hardware Monitoring:

Redundant sensors are used, in addition to sensor reasonableness checking. The rate-of-change of sensor readings is compared to a predicted maximum rate-of-change.

The A/D and D/A converters are tested by applying the D/A to a test word, and then applying the A/D to the results. Transient A/D errors are detected the same way as sensor errors.

Ticket Checking:

Ticket checking is used as a method of sequence error detection. A "ticket" is a word which contains information indicating the order the subroutines must be executed. Ticket checking is

similar to the sequence control code technique used on the Intelsat VI ACE.

The strength in the DC-9-80 system is its ability to detect both permanent and transient errors. An error would have to by-pass several screens before resulting in a system error. The DC-9-80 system's main shortcoming is that little emphasis is placed upon error recovery. If an error is detected, the sub-system is simply shut down and replaced by a spare. Also, no method is used to prevent error propagation.

1.5 Advantages of New Approach

The approach used by the Intelsat VI ACE addresses single event upsets on a specific system, and takes advantage of specific system characteristics. Consequently, this approach cannot be easily applied to other spacecraft systems. Furthermore, it is largely ineffective in recovering from transient errors other than single event upsets and it is capable of good performance only in small systems (small programs, few critical modules and variables).

The approach used by the DC-9-80 avoids most of the problems of the Intelsat VI ACE recovery technique. However, the DC-9-80 does not have robust recovery procedures. If an error cannot be corrected by redundant computation or redundant storage, the only course for recovery is shutting down. Clearly, this approach cannot be used effectively in autonomous spacecraft applications.

The methodology presented in this paper avoids these problems by requiring a specific structure of the flight software and it uses the attributes of this structure for error recovery. This structure can be applied to a general class of control programs used in spacecraft

computing (see the next section for details.) Furthermore, most of the techniques used in recovery are effective on multiple bit upsets. Some additional advantages of the methodology are 1) it offers a standardized approach to transient error recovery for control programs, 2) it does not require re-initialization to recover from errors in most cases, 3) it has safeguards to limit error propagation, and 4) it establishes objective criteria for evaluating the performance of error recovery.

The proposed methodology borrows many error detection methods from the Intelsat VI ACE error recovery approach. Where the two techniques differ most is in error propagation control and error recovery strategy. The error recovery concept for the Intelsat VI ACE could be described as: "if in doubt, jump to the wait loop". The error recovery concept in the proposed approach is: "if in doubt, re-execute". The advantages of this error recovery technique are : 1) the program will always make progress, or at least never lose ground, in a burst of upsets, and 2) reinitialization is usually not required for recovery. The Intelsat VI ACE does not address error propagation control.

It should be emphasised that this is a thesis on a methodology for employing software implemented transient error recovery techniques effectively on a class of real systems, and not a thesis on the techniques themselves. Although most of the recovery techniques used in this paper are well known, any methodology used to employ them in an real system is either non-existent or vague. For example, McCluskey [13] describes redundant computation by stating "Execute a program a second time and compare results." How does one do this on a real system? What about the outputs, the side-effects of the program on the program state, etc? How does one compare the results? Although McCluskey is certainly not attempting to describe a methodology for error detection through

redundant computation, this description is typical in the extent to which any sort of methodology is described.

1.6 Type of System Assumed

It will be assumed throughout this presentation that the methodology will be applied to an interrupt driven, real-time spacecraft control sub-system. Control software lends itself well to the structuring required by the methodology. Additional system characteristics which are helpful but not essential are : 1) constrained program execution flow (to facilitate program sequence monitoring), 2) idempotent outputs (to facilitate error recovery), and 3) a large amount of available execution time (to allow for transient error recovery execution overhead).

1.7 Overview of Thesis

There are four chapters in this presentation. The first chapter describes the specific methodology proposed. It includes a discussion on the approach, the specific techniques used, and the practical application of the techniques. The second chapter describes the simulation analysis made on a program written using the proposed methodology. This chapter includes a discussion on the simulation models and assumptions, the simulation performance metrics, and a discussion of the results. The third chapter is a probabilistic analysis of the results of the simulation. In this chapter, the probability of failing to recover from transient errors is computed for both present and future spacecraft configurations. The final chapter is a summary of the results.

Many terms used in this presentation were originated by the author. Consequently, a glossary is included to assist the reader in understanding this document.

PROPOSED METHODOLOGY

2.1 Introduction

There are three main aspects of Software Implemented Transient Error Recovery: limiting the propagation of errors, detecting errors, and taking appropriate recovery action upon error detection.

Error propagation control is achieved by performing complicated calculations on temporary variables. Once a calculation is complete, an error detection process is performed to verify that the calculation is correct. The calculation is then committed, either by changing the program state or performing an output. Error detection is achieved by imbedding redundant information in the program state. Inconsistent states imply the existence of errors. Error recovery is achieved by structuring the program so that the redundant information can be used to determine the appropriate place to resume program execution. By requiring idempotent program sections, the error recovery routine can re-execute a section whose proper execution is doubtful.

This methodology is designed to recover in the presence of single and multiple bit upsets per word. This methodology is also designed to recover in bursts of upsets. The recovery procedure attempts to make progress in program execution during a burst of upsets, and will restart only as a last resort.

2.2 Discussion of General Techniques

2.2.1 Error Propagation Control

Error propagation control is an important problem in transient error recovery. If a value which has been upset is used in a calculation, all values which depend upon the upset value will be incorrect.

Consequently, one upset value may result in several incorrect values, all of which must be corrected to successfully perform recovery.

A process experiencing transient errors can be considered to be two processes, the actual process and the transient error process. The main objective of error propagation control is to make all actual process transactions appear atomic with respect to the permanent program state.

Atomic actions are traditionally used in data base concurrency control [9]. The objective of atomic actions is to make actions appear indivisible, that is, all other actions appear to have occurred either before or after an atomic transaction. Furthermore, atomic actions appear to have completely happened (commit) or have never happened (abort). By using atomic actions in transient error recovery, it is possible to insure that all transient errors occur either before or after an action but not during.

To limit error propagation, all intermediate and final results of a computation are stored in temporary variables. This technique insures that all errors that occur during a computation have no effect on the permanent program state. Once a computation has completed, its results are checked by various error detection techniques. If the results pass the error checking, they are committed by an atomic action either by assigning a variable which is part of the permanent program state (known as a critical variable) or by performing an output action. Thus, the atomic action is essentially a boundary which errors cannot penetrate. If the results fail the error checking, they are thrown away and the calculation is repeated until the results pass the error checking.

2.2.2 Error Detection

The main concept in error detection is to create redundant information in the program state and check for inconsistencies. If an inconsistency is found, an error has occurred. The error detection techniques used are:

1) Redundant computation:

A single processor experiencing only transient errors can verify its computations by repeating a computation until an agreement is reached. This is a very powerful technique since the majority of active processing time is spent performing computations.

2) Redundant storage:

Replicated storage of important values (critical variables). Voting is used to determine the correct values.

3) Memory coding:

Error-correcting codes to correct single bit errors in RAM storage.

4) Reasonableness checking:

Reasonableness checking is a method of data error detection. When a variable is known to have some range of correct values, then the actual value of the variable can be compared to this range to check the reasonableness of the value. If a discrepancy is found, an error has occurred. Reasonableness checking is extremely powerful when the range of correct values is small compared to the range of possible values.

This technique is especially useful for testing program control variables. For example, suppose we have the following code:

```
index := 0;
while index < 4 do
  begin
    index := index + 1;
    { etc. }
  end;
```

We know that at the end of this block, index must have the value 4. We also know that during the execution of this block, index cannot have a value of less than 0 or greater than 4. Although this is obvious, this example shows that the implementation of reasonableness checking can be made very precise.

Reasonableness checking is similar to a process used in program correctness verification called "assertion checking" [12].

5) Sequence control codes (SCC):

The testing of sequence control codes is used as a method of sequence error detection. A variable is set to a known value before a section is entered. This variable is then checked at the end of the section. If there is a discrepancy, entry into the section must have been at some point other than the proper entry point of that section.

6) Jump return to wait loop:

Jump return to wait loop is a method for detecting and recovering from sequencing errors which occur during the execution of the wait loop. The control program of a typical spacecraft sub-system will spend the majority of its run-time in a wait loop. Since wait loops are short, the number of mutations of the wait loop instructions that a single bit flip

could cause are small. Code is imbedded to cause all jumps out of the loop resulting from a mutated instruction to be followed by a return to the wait-loop.

2.3 Discussion of Recovery Method

Once an error has been detected, appropriate recovery action must be performed. The key idea to error recovery is to structure the code into re-executable sections and provide redundant information to determine where to continue program execution.

2.3.1 Computation Blocks and Idempotent Sections

To apply the transient error recovery technique to a control program, the control program must first be divided into computation blocks, which are similar to procedures. Each block is associated with either a critical variable(s) or an output operation or both. The assignment of the critical variables or the output operation is to be done as an atomic action.

Critical variables are program variables which have a direct effect upon some output of the system. Non-critical variables have effect on the output of the system, but only through their effect upon critical variables. Another way of looking at this distinction is that if there were only a single copy of a critical variable, and if that copy were upset, the only possible course for recovery is re-initialization and restart. If a non-critical variable is upset, recovery can be achieved through re-execution of a section(s), so complete restart is not necessary. Since critical variables are actually stored in triplicate, recovery can be performed by voting on their value. Examples of critical

variables in the Intelsat VI ACE are offset pointing values, system modes, and gains. Examples of non-critical variables are counters, flags, and loop variables.

Each computation block is divided into three sections. The initialization section performs any initialization required to execute the block. At the very minimum, each of the critical variables, which are stored in triplicate, must be voted upon and temporary variables must be initialized. The computation section performs the computation. The computation is performed using temporary variables. The computation is performed at least twice, until two results in a row agree. The action section assigns the calculated temporary variables to the critical variables or performs an output operation using an atomic action.

The intention of the computation block/critical variable relationship is to protect critical variables and outputs from upsets and to allow simple recovery through re-execution of idempotent sections. The protection from error propagation arises from the fact that the initialization section and the computation section do not modify the permanent program state, since both sections modify temporary variables. The voting process performed on critical variables by the initialization section does not change the value of critical variables, it simply removes errors. Consequently, if an error detection mechanism were to find an inconsistency in the initialization section or the computation section, recovery simply involves re-executing the correct section. There is no need to undo a previous action. The action section does modify the permanent program state. However, since the run-time length of atomic actions is very short compared to the run-time length of a computation block, the probability of mishap during the execution of an atomic action is small.

A section is idempotent if the result of multiple applications is the same as the result of one application. Consequently, a simple recovery rule that can be used on idempotent sections is: "if in doubt, re-execute". The initialization block and the calculation block are always idempotent, since their execution does not modify the permanent program state. Unfortunately, not all output action sections are idempotent. The only solution to this problem is to require that spacecraft output be in absolute instead of relative terms. For example, "move the platform to 135 degrees" is an idempotent instruction. The instruction "increment platform position + 15 degrees" is not idempotent.

2.3.2 Additional Considerations

A summary of additional programming rules, many of which are unrelated to the methodology but are important to minimizing the effects of transient errors, is given in the appendix.

2.4 Recovery Software Format

In this section, the specific structure for both the computation block and the recovery block are presented. The code structure presented is the actual code structure determined from the simulation program. The computation block structure embodies the error recovery concepts presented above. The most important observation to make is how the error detection techniques are used to interface the initialization, computation, and action sections. For the methodology to achieve full effectiveness, these section interfaces, with the corresponding recovery block, must be implemented exactly as presented, since the coverage of the error detection mechanism is a primary factor in determining the

system's ability to recover from errors.

2.4.1 Computation Block Format

Figure 1 shows the computation block recovery format developed for Software Implemented Transient Error Recovery. This code segment represents one computation block. The block is divided into an initialization section, a computation section, and an action section. Each section is joined by a sequence control check, where the old sequence control code is verified and the new sequence control code is set. The initialization section simply votes on the critical variables and initializes the local variables. The computation section performs its calculation using temporary variables until two results in a row agree. The result is then checked for reasonableness. The critical variables used in the calculation are rechecked to insure that the result was computed from correct data.

The action section in this example updates several critical variables. The control variables and temporary computation variables are then checked for reasonableness. Since updating variables is an idempotent action, the action section meets the idempotent requirement. The atomicity requirement is also met, but in a subtle manner. The recovery block will re-execute the action section until both the reasonableness check and the SCC check pass. Continuing program execution to the next computation block is the atomic action which commits the action.

Additional recovery measures, such as Jump Return to Wait Loop error detection and NOP buffering to prevent mis-interpretation of machine code (see Appendix) would have to be implemented last. Memory error correction codes are standard equipment in spacecraft hardware.

Sample Computation Block in Error Recovery Format

```
procedure sample_procedure;

  { note : sc = sequence check
          rc = reasonableness check
          scc = sequence control code }

begin
  {section 1 - initialization}
  if scc = {last scc} then
    begin
      scc := {scc 1}
    end
  else
    begin
      error_recovery (sc, scc, block_number, section 1);
    end;

    {vote on critical variables}
    variable_number := 0;
    while variable_number < number_of_critical_variables do
      begin
        variable_number := variable_number + 1;
        { vote on critical variables }
      end;

    {initialize other variables }

    {section 2 - perform calculation}
    if scc = {scc 1} then
      begin
        scc := {scc 2};
      end
    else
      begin
        error_recovery (sc, scc, block_number, section 2);
      end;

  repeat

    {perform computation on temporary variables }

  until { two consecutive results agree }

  {check reasonableness}
  if {temp result is not within a reasonable range of results } or
  {if critical variables no longer agree} then
    begin
      error_recovery (rc, scc, block_number, section 2);
    end;
end;
```

```

    {section 3 - assign critical variables}
if scc = {scc 2} then
    begin
        scc := {scc 3}
    end
else
    begin
        error_recovery (sc, scc, block_number, section 3);
    end;

    {update critical variables}
variable_number := 0;
while variable_number < number_of_critical_variables do
    begin
        variable_number := variable_number + 1;
        copy_number := 0;
        while copy_number < number_of_copies do
            begin
                copy_number := copy_number + 1;

                critical_variable {variable_number} [copy_number]
                    := temp_result [variable_number, 1]
            end;
        end;
    end;

    { reasonableness check }
if (variable_number <> number_of_critical_variables) or
(copy_number <> number_of_copies) then
    begin
        error_recovery (rc, scc, block_number, section 3);
    end;

    { reasonableness check }
if (temp_result [0] <> temp_result [1]) then
    begin
        error_recovery (rc, scc, block_number, section 4);
    end;

    { scc check }
if scc <> {scc 3}
then
    begin
        error_recovery (sc, scc, block_number, section 4);
    end;

end;

```

Figure 1

2.4.2 Recovery Block Format

Once an error has been detected by a reasonableness check or a sequence control check, recovery action is performed by the recovery block. The redundant computation, redundant storage, memory coding, and jump return to wait loop error detection techniques all recover from errors in the detection process. Figure 2 shows the recovery software format.

Although the recovery block could simply reset the system, there is usually enough information to determine the location of the error and redo the appropriate section. The trick is to perform a reasonableness check on the sequence control code. If the sequence control code is a reasonable number, then there is a very high probability that it is correct. Since the sections are idempotent, the program can continue execution at the section which assigned that SCC.

The recovery procedure is divided into seven cases

case 1 : scc check failed and scc is reasonable

RECOVERY ACTION - sequence was upset, continue from old scc.

case 2 : scc check failed and scc is not reasonable

RECOVERY ACTION - SCC register was upset, reset scc and continue.

case 3 : reasonableness check failed at end of computation block and scc is reasonable

RECOVERY ACTION - reset scc to scc of last block, section 3. redo current block.

case 4 : reasonableness check failed at end of computation block and scc is not reasonable

RECOVERY ACTION - re-initialize and restart at block 1, section 1.

case 5 : reasonableness check failed at end of action block and scc is reasonable

RECOVERY ACTION - local variable upset, redo action section.

case 6 : reasonableness check failed at end of action block and scc is reasonable

RECOVERY ACTION - temp variable upset, redo current block.

case 7 : reasonableness check failed at end of action block and scc is not reasonable

RECOVERY ACTION - re-initialize and restart at block 1, section 1.

If the SCC value is not reasonable (cases 2,4,7), the recovery action depends upon whether the error is found by a reasonableness check or an SCC check. If the error was found by a reasonableness check, then the section has performed incorrectly, and there is no redundant information (a correct SCC) to determine where to continue program execution. Consequently, a complete restart is necessary.

If the error was found by an SCC check, it is safe to assume that the SCC register itself was upset and that program execution up to the error detection was correct. This is true because the SCC checks follow reasonableness checks in sections 2 and 3. Consequently, if program execution was incorrect, it would have been detected by the preceding reasonableness check.

Sample Error Recovery Block

```
procedure sample_error_recovery ( type : sc or rc ; current_scc,
    current_block, current_section : integer);

{ note : sc = sequence check
        rc = reasonableness check
        scc = sequence control code }

begin
  if {type = sc} then

    { sequence upset detected }

    begin
      if reasonable_scc (current_scc) then

        {sequence upset - continue from the old scc}

        begin { case 1 }
          goto {beginning of current_block, current_section};
        end
      else

        { scc upset - reset scc and continue at scc }

        begin { case 2 }
          scc := { scc of current_block, current_section }
          goto { end of current_block, current_section }
        end;

      end;

    if {type = rc and section = 2} then

      { reasonableness check at end of computation section }

      begin
        if reasonable_scc (current_scc) then

          {continue from current block, section 1}

          begin { case 3 }
            scc := {scc of previous block, section 3}
            goto {beginning of current_block, section 1}
          end
        else

          {catastrophe}

          begin { case 4 }
            {re-initialize and restart }
          end;

        end;

      end;

    end;
```

```

if {type = rc and section = 3} then
    { reasonableness check at end of action section}
begin
    if reasonable_scc (current_scc) then
        {redo action section}
        begin { case 5 }
            goto {beginning of current_block, section 3}
        end
    else
        {catastrophe}
        begin { case 7 }
            {re-initialize and restart }
        end;
    end;
end;

if {type = rc and section = 4} then
    { reasonableness check at end of action section}
begin
    if reasonable_scc (current_scc) then
        {redo block}
        begin { case 6 }
            scc := {scc of previous block, section 3}
            goto {beginning of current_block, section 1}
        end
    else
        {catastrophe}
        begin { case 7 }
            {re-initialize and restart }
        end;
    end;
end;

end;

```

Figure 2

2.5 Extending Technique to Real Computation

The test program for which the error recovery methodology was developed bears little resemblance to actual flight code. Here are some

additional details that should be addressed concerning the application of the methodology to real spacecraft computations.

Structure of Software

As stated previously, it has been assumed throughout the discussion that the methodology would be applied to real-time, interrupt driven spacecraft control sub-system. It is not clear if the methodology can be applied to arbitrary program structures. For example, it is assumed that an interrupt driven system has inherent protection from infinite loops, in that as long as interrupts are enabled, there can be no infinite loops. For another example, it is assumed that program flow is deterministic, so the previous sequence control code is always known. This assumption is not true for arbitrary program structures.

The Idempotence Requirement

It has also been assumed that all output actions can be made idempotent, that is, the result of multiple applications of an output is the same as the result of one application. What is the impact if this requirement cannot be achieved?

The problem is that it is impossible to perform recovery on non-idempotent action sections. However, action sections compose only a small percentage of the total run-time (this is the major reason for using atomic actions), so the probability of needing to recover during the execution of an action section is low. Consequently, Software Implemented Transient Error Recovery still works, but at slightly degraded performance. The amount of performance degradation depends upon the run-time duration of the non-idempotent action sections.

Space Considerations

Although the recovery format is rather large, the size of a block is invariant with the actual computation code. The same basic recovery format can be used with any computation. Consequently, the memory requirement for code using Software Implemented Transient Error Recovery should be comparable to the memory requirement for code not using any error recovery.

Time Considerations

The real execution time of code using Software Implemented Transient Error Recovery should be about twice the execution time for code without any error recovery in the absence of upsets. In the presence of upsets, the execution time increases even more. Since most spacecraft control programs spend the majority of their execution time in a wait loop, the additional time requirement should not be important.

Real-Time Considerations

It is difficult to perform redundant computation error detection on time varying computations, since the correct results change with time. To apply the redundant computation error detection technique to time variant computations, the difference between computations would have to be compared to some predetermined margin. For example, it may be reasonable to say that the results of two consecutive time-varying calculations should agree within a 5% margin. Even in a time-varying environment, redundant computation is still a very powerful error detection technique. To see why, lets examine the possible outcomes:

- 1) Two correct calculations agree
- 2) Two correct calculations disagree
- 3) One correct calculation and one incorrect calculation agree
- 4) One correct calculation and one incorrect calculation disagree
- 5) Two incorrect calculations agree
- 6) Two incorrect calculations disagree

Case 2, although not desirable, is not a problem assuming that further correct calculations will agree within the margin. Case 3 is also not a problem for control applications, since the result is correct (within the margin). The only case that is a problem is case 5. However, the probability of two incorrect results being within 5% of each other is very slight if the cause of incorrectness is transient errors. All other cases perform correctly.

SIMULATION ANALYSIS

3.1 Introduction

This chapter is an overview of the simulation program used to help develop the methodology for Software Implemented Transient Error Recovery. This chapter describes the models used, the performance metrics used, and the simulation results. For a detailed description of the simulation program implementation, see the appendix.

Objective of Simulation

The original objective of the simulation was to perform an efficacy and trade-off analysis of the various recovery techniques. The concept was that a programming methodology could be developed by considering each recovery technique individually, without strong consideration of the relationship between the methods. There seemed to be potential overlap between their recovery capabilities, and using all the techniques in a haphazard manner would be wasteful.

However, three important observations were made during the design and development of the simulation:

- 1) All of the techniques under consideration are needed, and each, when properly used, provides information required for error recovery.
- 2) To say that the non-redundant information provided by a specific technique is unnecessary requires detailed knowledge of the system's behavior, which is not known in general.

3) Trade-off metrics are difficult to measure and are often misleading, making it difficult to draw conclusions from numeric simulation results.

Experience with the simulation has shown that the error recovery techniques are best used as a system. Modifying the usage of one technique may have important consequences in the usage of another. Therefore, the trade-off study that was initially desired is inappropriate.

However, a simulation analysis is still very important. The most important benefit is the capability to immediately evaluate the performance of a recovery structure. The emphasis has shifted from a quasi-quantitative trade-off analysis to a qualitative analysis. The qualitative analysis allows system performance to be evaluated in terms of specific failure events. It will be shown that with the exception of these specific failure events, called "catastrophes", the system will always recover.

The new objectives of the simulation analysis are to 1) provide "real-time" experience with a transient error environment, 2) demonstrate the methodology and evaluate its performance, 3) provide a tractable framework for the problem through system modeling, and 4) to establish suitable metrics for transient error recovery performance.

What to Simulate

It is important to realize that all outcomes of all possible upsets cannot be accurately simulated. The objective of this simulation, or any simulation, is to perform an analysis on models which are abstractions of the real world that capture the essence, but not all the detail, of the real world [7].

The proposed transient error recovery methodology is capable of detecting high-level errors. More precisely, if an error cannot be detected by a high-level language, it cannot be corrected by the methodology. An example of a high-level error is an upset to the processor program counter. An upset to the processor program counter can be detected by testing a sequence control code. Another example of a high-level error is an upset to an internal ALU register. An ALU upset can be detected by redundant computations. An example of a low-level error is an upset to the system reset flip-flop. Such an upset destroys the program state, and cannot be detected by a high-level language. Consequently, if we simply assume that recovery will fail with all low-level errors, then there is no motivation to do a detailed simulation of the processor hardware and the specific flight code in order to simulate the outcome of low-level errors.

The Multiple Bit Upset Model, the Upset Mapping Model, the High-Level Language/Machine Language Model, and the Structure/Content Model, strive to simplify the system to capture the essence of most high-level errors. These models work together to simplify the system by abstracting various aspects of the system. The Multiple Bit Upset Model is an abstraction of the transient error environment, the Upset Mapping Model is an abstraction of the system hardware, the High-Level Language/Machine Language Model is an abstraction of the hardware/software interface, and the Structure/Content Model is an abstraction of the spacecraft software. The simulation program applies these models to a specific "benchmark" computation block written in the recovery format.

3.2 The System Models

3.2.1 The Multiple Bit Upset (MBU) Model

Definition: The Multiple Bit Upset Model abstracts transient errors as events in which one or more bits per word may be upset simultaneously.

The MBU Model is an abstraction of the transient error processes which spacecraft experience. The MBU model is a generalization of the known causes of transient errors: single event upsets, electro-static discharges, and thermal noise. Intermittent hardware errors are not modeled.

3.2.2 The Upset Mapping Model

Definition: The Upset Mapping Model abstracts the outcome of multiple bit upsets as either main memory errors, processor memory errors, or processor sequence errors. Any upset outcome not modeled directly by the above outcomes can either be modeled indirectly as a combination of the above errors, or must be considered individually.

Definition: Main memory refers to the main bank of volatile RAM memory.

Definition: Processor memory is the volatile memory used by the processor, whether internal or external to the physical processor. In the context of the simulation program, a "processor memory error" refers to a processor memory error which can be observed by inspecting the processor registers.

To determine the reasonableness of this model, it is first necessary to enumerate the locations of volatile memory in spacecraft hardware. Here is a list of volatile memory in a typical spacecraft sub-system:

- 1) Main RAM memory
- 2) Addressable processor registers
- 3) Internal processor registers
- 4) The program counter
- 5) The stack pointer
- 6) The memory control hardware
 - Hamming encoder/decoder, MMU, DMA
- 7) Misc. processor hardware
 - interrupt enable register, etc.
- 8) Misc. sub-system hardware
 - I/O chips, clocks, control flip-flops, etc.

It is clear that upsets to the main memory and the program counter are modeled well by main memory errors and processor sequence errors, respectively. Upsets to the stack pointer, memory control hardware, and miscellaneous processor and sub-system hardware are not modeled by the upset mapping model, but are addressed in the probabilistic analysis. These memory cells are not modeled because it is very difficult to model the outcome of, say, an upset to a system clock flip-flop.

The outcome of an upset to an addressable processor register or an internal processor register must be examined more closely. Some possible outcomes from a processor register upset are:

- 1) an incorrect memory address is read.
- 2) an incorrect memory address is written to.
- 3) a correct memory address is read, but the data is incorrect.
- 4) a correct memory address is written to with incorrect data.

5) an incorrect program sequence is executed.

6) an incorrect program instruction is executed.

Reading the wrong main memory location and reading incorrect data are both modeled by processor memory upsets. Writing to the wrong main memory location and writing incorrect data to the correct address are both modeled by main memory upsets. An incorrect program instruction execution is not modeled by any of the above outcomes. An incorrect program sequence execution is modeled directly by a sequence upset. It will be shown that an incorrect program instruction execution can be modeled as a combination of sequence and memory errors.

To understand the effect of executing an incorrect program instruction, it is necessary to categorize the various types of processor instructions. Processor instructions can be divided into the following categories: 1) data movement, 2) arithmetic operations, 3) program control, and 4) status control. Any instruction transformation caused by an upset can be modeled as a combination of sequence and memory errors. The following transformation examples provide an illustration:

Example 1

MBU(data movement) -> incorrect data movement

e.g. MBU(mov r1,ADDR1) -> mov ADDR2,r6

= MBU(ADDR1) and MBU(r6)

- processor memory error and main memory error

Example 2

MBU(data movement) -> incorrect arithmetic operation

e.g. MBU(mov r1,r2) -> add r1,r2

= MBU(r2)

- processor memory error

Example 3

MBU(data movement) -> incorrect program control

e.g. MBU(mov r1,r2) -> jmp label

= MBU(r2) and MBU(program counter)

- processor memory error and sequence error

Example 4

MBU(arithmetic operation) -> incorrect data movement

e.g. MBU(add r1,r2) -> mov r1,r6

= MBU(r2) and MBU(r6)

- two processor memory errors

Example 5

MBU(arithmetic operation) -> incorrect arithmetic operation

e.g. MBU(add r1,r2) -> mul r1,r6

= MBU(r2) and MBU(r6)

- two processor memory errors

Example 6

MBU(arithmetic operation) -> incorrect sequence control

e.g. MBU(add r1,r2) -> jmp label

= MBU(r2) and MBU(program counter)

- processor memory error and sequence error

Example 7

MBU(sequence control) -> incorrect data movement

e.g. MBU(jmp label) -> mov r1,ADDR1

= MBU(ADDR1) and MBU(program counter)

- main memory error and sequence error

Example 8

MBU(sequence control) -> incorrect arithmetic operation

e.g. MBU(jmp label) -> add r1,r2

= MBU(r2) and MBU(program counter)

- processor memory error and sequence error

Example 9

MBU(sequence control) -> incorrect sequence control

e.g. MBU(jmp label1) -> jmp label2

= MBU(program counter)

- sequence error

Example 10

MBU(status control) -> any other operation

= potential sequence error and result of incorrect operation

A distinction is made between processor memory errors and main memory errors. The reason for making this distinction is that processor memory errors can occur only to data currently being used in a computation and the probability of a particular variable being upset is independent how many copies are stored in main memory. For example, critical variables, which are stored in triplicate, are three times more likely of being upset in main memory than other variables. In processor memory, critical variables are as likely of being upset as other variables currently being processed.

3.2.3 The High-Level Language/Machine Language Model

Definition: The High-Level Language/Machine Language Model

abstracts errors from the Upset Mapping Model on bit level code

(machine language) and data as bit upsets on high-level language

and data.

The Upset Mapping Model is based on the observation that all results of the Upset Mapping Model can be produced using a high-level language and that Software Implemented Transient Error Recovery is only capable of recovering from errors that can be detected from a high-level. Assuming the Upset Mapping Model is reasonable, then the High-Level Language/Machine Language Model is a natural consequence.

The High-Level Language/Machine Language Model is an imperfect abstraction. It suffers four problems 1) arbitrary machine-level sequence errors are impossible, 2) machine code mis-interpretation is not modeled, 3) upsets are atomic with respect to high-level language statements, and 4) run-time checking prevents arbitrary data upsets.

A high-level language is simply not capable of jumping to any equivalent individual machine language step. Using the GOTO statement in high-level language allows one to jump to any line of high-level code. The problem is that a line of high-level code may compile into possibly several lines of machine language, making it impossible to make arbitrary jumps on the machine instruction level

Machine code mis-interpretation is caused from the ambiguity of stored programs, since machine language is context sensitive. For example, the machine code 10110001 may be interpreted as the instruction "inc R1", or as the data B1 hex. Consequently, if a sequence error occurs, the code intended to detect this error may be misinterpreted. Clearly, this process cannot be modeled with a high-level language. However, as noted in the appendix, this problem is easily solved with a process called "NOP buffering".

A high-level language is not capable of injecting an upset in the middle of a high-level statement. The real upset process is atomic with

respect to machine instructions, that is, all upsets can be considered to occur either before or after an machine instruction, but not during. The modeled process is atomic with respect to high-level instructions. All errors occur either before or after a high-level statement, but not during.

High-level language run-time checking interferes with a high-level languages' ability to model upsets to various control variables. For example, an array subscript cannot be set to an arbitrary value without causing a run-time error. The result of this constraint is that it forces many unlikely recovery failures to become more likely, since many control variables will always be upset to reasonable values in the simulation.

The importance of these shortcomings is not clear. There is no reason to believe any of these problems are serious. However, this model is very important, for it, in conjunction with the Structure/Content Model allow an abstract study of the upset process without having to consider the implications of the specific flight software or the specific hardware system.

3.2.4 The Structure/Content Model

Definition: The Structure/Content Model abstracts software as having a recovery structure without computational content.

The Structure/Content Model embodies the idea that the ability of a system to recover from transient errors does not depend upon what computation is being performed, but on how it is being performed. It is the structure of a computation, and not its content per se, which dictates the performance of Software Implemented Transient Error Recovery.

More specifically, the ability to perform error propagation control, error detection, and error recovery upon the initialization, computation, and action sections is independent of the specific action performed in each section as long as the requirements, such as the idempotence and atomic action requirements, are met.

A common exception to this model is a non-idempotent action section. The fact that an action is non-idempotent does not give the recovery routine the liberty to retry an action if an error is found. However, any section which is idempotent, which includes all initialization and computation sections, and most action sections, can be repeated if necessary, so consequently, the specific content of the section is irrelevant to recovery.

Although specific content of a section does not modify the ability to recover, it does modify the outcome of upsets. Specifically, the run-time characteristics of a program determine the result of a processor memory upset. This fact is accounted for in the simulation by specifying the run-time length of the program sections in the absence of errors. The simulation upset rate is then modified in each section so that the simulated run-time error density is equivalent to the specified run-time error density.

The result of the Structure/Content Model is that any "benchmark" block of code can be used in the simulation, rather than the actual flight code. A good benchmark is one which allows a simple evaluation of recovery. Specifically, the values of the critical variables should be deterministic and conform to some pattern.

3.3 Performance Metrics for Recovery Evaluation

In order to evaluate the effectiveness of Software Implemented Transient Error Recovery, it is necessary to develop appropriate performance measures for error recovery. This is not a simple task. I propose the use of three metrics: coverage, recovery profile, and probability of catastrophe. Since it is difficult to precisely define the meaning of coverage and since recovery profile only applies to error detection performance, I use the probability of catastrophe as the principle metric of recovery.

First, some definitions:

Coverage:

Coverage is defined in reliability theory [8] as the conditional probability that a failure of a unit will be detected and appropriate recovery action will be performed given the occurrence of a fault and sufficient resources for recovery. Since the addressed phenomenon are transient in nature, no resources are required for transient error recovery. For transient errors, coverage is an aggregate measure of performance for error propagation control, error detection, and error recovery.

Recovery Profile:

The recovery profile is a histogram of the number of detected errors for each error detection technique. It is not capable of measuring the effectiveness of error propagation control or error recovery. The intention of the recovery profile is to determine the relative value of error detection techniques.

Catastrophes:

A catastrophe is an event which is the result of an upset, from which a transient error recovery technique may not recover without re-initialization and restart, if at all. In terms of ASM, a catastrophic upset implies that autonomy may be compromised. In terms of mission success, the result of a catastrophic upset is undefined. The probability of catastrophe indicates recovery performance in terms of the probability of events which a system cannot recover. The probability of catastrophe is an aggregate measure of performance for error propagation control, error detection, and error recovery.

3.3.1 Coverage and Recovery Profile

Problems with Coverage

There are two problems with calculating coverage: single events can result in multiple consequences, and the meaning of "appropriate recovery action" is not well defined.

Multiplicity of effects - need for both coverage and recovery profile

One upset can result in several errors (error propagation), each of which must be individually detected and corrected. Also, several errors can be corrected by one action. Furthermore, some errors need no correction at all.

It is difficult to decide what constitutes "appropriate recovery action". There are two possible approaches to defining recovery. The first is to consider recovery as an individual condition with respect to upsets. That is, if an upset propagates into several errors, the system

cannot be considered to have recovered unless all the errors caused by that upset have been corrected. The problem with this approach is that it is impossible to determine which error detection technique is responsible for recovery, since many different techniques may share the credit for detection.

The second approach to defining recovery is to consider recovery as an individual condition with respect to errors. Whenever an error is corrected, the system is said to recover. The problem with this approach is that there may be more recoveries than upsets, so a ratio of recoveries to upsets would be misleading as a metric of recovery. The number of errors resulting from an upset is usually not known, so a ratio of recoveries to errors cannot be computed.

The solution to this problem is to use both performance measures, and give each a distinct meaning. The first performance measure is called "coverage", which measures the effectiveness of recovery once an upset has occurred. The second performance measure is called the "recovery profile", which is a histogram of the error detection techniques responsible for recovery. There is only a weak relationship between the two measures. If error propagation occurs, then there is no function which can determine the coverage from the recovery profile.

Meaning of Recovery

Another problem with defining "appropriate recovery action" is that some errors do not have to be corrected at all. For example, if an upset occurs to a memory register not in use, and this error is not corrected, has the system failed to recover? Certainly not. A distinction must be made between "critical" and "non-critical" variables. As defined in the methodology chapter, critical variables are variables whose value has a

direct effect upon some output of the system. Non-critical variables have effect on the output of the system, but only through their effect upon critical variables. If all relevant critical variables are correct at the time of any output, then that output is correct, regardless of the state of the non-critical variables.

Definition: For a system to "recover" from any number of upsets, at least two out of three copies of the relevant (pertaining to the current output) critical variables must have the correct value at the time of any output action.

Definition: A "correct" value is the value that would be determined by a computation in the absence of upsets (assuming time invariant computations).

Meaning of Coverage

With a specific definition of recovery, it is possible to have a specific definition of coverage. The definition should have the form:

$$(\text{Number of recovered units}) / (\text{Total units executed})$$

The only thing left is to define the unit of execution. The two possibilities are blocks and control loops. Either could be used.

As defined, it is difficult to understand coverage values in an absolute sense. The value of coverage depends as much on the unit of execution used and the number of critical variables as the recovery technique used. Also, most software systems do not have critical variables or computation blocks, so any definition of coverage may have meaning to only a small subset of systems. Consequently, it is difficult to precisely define coverage, or even understand its meaning in absolute

terms for all systems.

3.3.2 Catastrophes

Problem with Probability of Catastrophe

The major problem with using the probability of catastrophe as a figure of merit is that it is impossible to calculate when the number of catastrophes is large. However, when the number of catastrophes is small, it is an objective measure of recovery performance.

Although using the probability of catastrophe as a metric also requires a precise definition of recovery (since catastrophes are defined in terms of recovery), it does not have the numeric ambiguity that coverage does. It should be pointed out that coverage and the probability of catastrophe measure essentially the same thing; that is, $P(\text{catastrophe} \mid \text{upset}) = 1 - P(\text{coverage} \mid \text{upset})$, except that coverage is more difficult to define precisely. Since the number of catastrophes detected by the simulation is small, the probability of catastrophe is the best metric to measure recovery performance.

3.3.3 Time and Space

Additional execution time required for recovery and additional memory required for recovery software are also important performance metrics.

Additional execution time is used by error recovery for redundant computations, redundant critical variable storage, and aborted computations. It is clear that a real-time control program cannot spend an arbitrary amount of time correcting errors, so the timeliness of recovery is an important factor. The proposed methodology tries to

minimize the additional execution time needed for recovery by reducing the number of aborted computations through the "if in doubt, re-execute" recovery strategy.

The additional memory requirement for more complicated flight software is also important, but difficult to estimate for real flight code.

3.4 Discussion of Results

Here is a list of the catastrophes detected by the simulation program:

- 1) a critical control variable is upset, and the result is reasonable
- 2) a sequence upset jumps to an SCC assignment
- 3) a sequence upset jumps to a non-idempotent action section
- 4) both SCC and reasonableness checks fail within the computation or action sections.
- 5) two or three copies of a critical variable are upset
- 6) two incorrect calculations agree and are reasonable
- 7) both temporary copies of the calculation block are upset to the same incorrect value
- 8) sequence control register is upset, and a sequence error occurs.

Performance in Bursts of Upsets

The proposed recovery methodology is designed to recover from bursts of upsets. The error recovery concept used is: "if in doubt, re-execute". The advantages to this error recovery technique are :

- 1) the program will always make progress, or at least never lose ground, in a burst of upsets, and
- 2) re-initialization is not required for

recovery unless a catastrophe occurs.

The simulation program was run at a very high upset rate. With the exception of the occurrence of catastrophes, the simulated system recovered from bursts of upsets. With the error rate very high (approximately one error per 10 lines of high-level code), catastrophe 4, which requires restart, occurred often enough that the simulated control loop could not terminate.

The simulation program does not simulate upsets during execution of the recovery block. The problem of recovering during recovery, which is a key factor for determining recovery performance during a burst of upsets, is not addressed in this paper.

PROBABILISTIC ANALYSIS

4.1 Introduction

The objective of the probabilistic analysis is to evaluate the performance of the proposed transient error recovery methodology in comparison to alternative transient error methodologies in light of future spacecraft autonomy requirements.

This performance analysis is accomplished by estimating the conditional probability of catastrophe given an upset, and the mean time to catastrophe (MTTC). The conditional probability of catastrophe given an upset is used in lieu of the probability of catastrophe because the probability of upset is determined by the hardware, and cannot be modified by software techniques. It is the conditional probability of catastrophe given an upset which Software Implemented Transient Error Recovery has direct influence.

The first performance analysis performed is a detailed probabilistic analysis based upon the simulation results. In this analysis, the conditional probability of catastrophe given the occurrence of an upset and the mean time to catastrophe are calculated with parameters from the Intelsat VI ACE and with parameters from a possible future spacecraft configuration, for both SEU and MBU cases.

The results of the first analysis suggests that an alternative form of analysis gives reasonable results. This method approximates the probability of catastrophe given an upset by simply determining which flip-flops and registers in the sub-system are not covered by the recovery methodology, and dividing by the total number of flip-flops in the sub-system. Although the latter approach is certainly less accurate than the former, it offers an advantage in that it can be applied to any

system. In this analysis, the conditional probability of catastrophe given the occurrence of an upset and the mean time to catastrophe are calculated for four methodologies: a worst cases methodology, the Intelsat VI ACE methodology, the proposed methodology, and the best case methodology. Each case examines present and future spacecraft sub-system configurations, for both SEU and MBU cases.

4.2 Detailed Probabilistic Analysis

The results of the simulation indicate that there is a small class of upsets from which the system may not recover. This fact makes a simple, but accurate probabilistic analysis of transient error recovery possible. The class of upsets from which the system may not recover is called catastrophes. In this analysis, the conditional probability of catastrophe given the occurrence of an upset and the mean time to catastrophe (MTTC) are calculated with parameters from the Intelsat VI ACE and with parameters from a possible future spacecraft configuration, for both SEU and MBU cases.

4.2.1 Definitions

Before preceeding with the probabilistic analysis, a few terms require definition or clarification:

Modified Definition of MBU:

The previous (correct) definition of a multiple bit upset was an event in which one or more bits per word may be upset simultaneously. Since the probabilistic analysis is simplified by looking at two cases, the single bit upset case and the multiple

(> 1) bit upset case, it will be assumed that the term MBU does not refer to the single bit case within this section.

Definition of Critical Registers:

Critical registers are registers within the processor and its supporting logic which can result in a catastrophe if upset, and are not covered by Software Implemented Transient Error Recovery. Since they are referred to as "registers" (as opposed to "flip-flops"), the implication is that they typically hold value information as opposed to control information, their criticalness has a low duty-cycle. Thus, an upset during a non-critical period, such as the execution of a wait loop, is not catastrophic.

Definition of Critical Flip-Flops:

Critical flip-flops are flip-flops within the processor and its supporting logic which can result in a catastrophe if upset, and are not covered by Software Implemented Transient Error Recovery. Since they are referred to as "flip-flops" (as opposed to "registers"), the implication is that they typically hold control information as opposed to value information, their criticalness has a high duty-cycle. Thus, most upsets to critical flip-flops are catastrophic.

Definition of SEU Upset Rate:

The SEU upset rate refers to the frequency of upsets caused by cosmic radiation, based upon tests conducted upon actual devices [3,12]. The upsets are assumed to occur in a constant stream, which accurately models the real phenomenon. The rate used is

0.0001 upsets/(bit-day).

Definition of MBU Upset Rate:

The MBU upset rate refers to the frequency of upsets caused primarily by the electrostatic discharge problem. There is no accurate data available on the frequency characteristics of electrostatic discharges. Since it is known that SEUs are the dominant source of transient error in present spacecraft systems, the MBU rate was chosen so that the frequency of catastrophic upsets caused by MBUs is the same order of magnitude as the frequency of catastrophic upsets caused by SEUs. The rate used is 0.000001 upsets/(bit-day).

4.2.2 Failure Classifications

Definition of Catastrophe:

An event which is the result of an upset, which a transient error recovery technique may not recover without re-initialization and restart, if at all. In terms of ASM, a catastrophic upset implies that autonomy may be compromised. In terms of mission success, the result of a catastrophic upset is undefined.

Catastrophes are divided into two categories, first-order catastrophes, which result from one upset, and second-order catastrophes, which result from two upsets. The list of first-order catastrophes is intended to be comprehensive. The list of second-order catastrophes may not be comprehensive. However, it will be shown that the probability of occurrence of a second-order catastrophe is small compared to the probability of occurrence of first-order catastrophes so they can be

ignored.

Catastrophes:

First-Order Catastrophes:

First-Order Catastrophes are events that are either not covered by any of the recovery techniques, or events that bypass the recovery techniques.

- 1) a critical processor register is upset
(from Upset Mapping Model)
- 2) a critical processor control flip-flop is upset
(from Upset Mapping Model)
- 3) a critical control variable is upset, and the result is reasonable
(bypasses reasonableness checking)
- 4) a sequence upset jumps to an SCC assignment
(bypasses SCC checking)
- 5) a sequence upset jumps to a non-idempotent action section
(from Structure/Content Model)

Second-Order Catastrophes:

All error recovery techniques work by comparing redundant information. Second-order catastrophes are events which destroy enough redundant information to require restart or result in incorrect recovery decisions. It is impossible to insure that there is enough redundant information for correct recovery; consequently, it is impossible to eliminate second-order catastrophes. However, it is possible to make the probability of second-order catastrophes arbitrarily small by increasing the amount of redundant information. This might be a good strategy if one is concerned about bursts of upsets.

- 6) both SCC and reasonableness checks fail within computation or action sections.
(requires restart)
- 7) two or three copies of a critical variable are upset
(incorrect recovery)
- 8) two incorrect calculations agree and are reasonable
(incorrect recovery)
- 9) both temporary copies of the calculation block are upset to the same incorrect value
(incorrect recovery)
- 10) sequence control register is upset, and a sequence error occurs.
(incorrect recovery)

4.2.3 General Detailed Probabilistic Analysis

System Constants

- 1) frequency of upsets to system
(in SEUs/(bit-day)) := SEUFREQ
- 2) frequency of upsets to system
(in MBUs/(bit-day)) := MBUFREQ
- 3) probability of main memory upset
given an upset to system := MS
- 4) probability of processor memory upset
given an upset to system := PS
- 5) probability of processor sequence upset
given an upset to system := PSS
- 6) size of main memory (in bytes) := MMS
- 7) size of processor memory (in bits) := PMS
- 8) action section size (in instructions) := COM
- 9) time in the wait loop := WL
- 10) word size (in bits) := WS
- 11) address space (in bytes) := AS
- 12) number of blocks := NOB
- 13) number of sections := NOS

- 14) number of critical processor registers := NOCPR
- 15) number of critical control variables := NOCV
(not including SCC)
- 16) number of reasonable values := NORV
for control variables
- 17) number of critical processor control := NOFF
flip-flops (in bits)
- 18) number of control loops executed := CLFREQ
per second

Given that an upset has occurred...

First-Order Catastrophes:

- 1) the probability a critical processor register is upset

$$= (WS)(NOCPR)(PS)/(PMS)$$

PS is the probability that any bit in the processor is upset, given an upset has occurred. Dividing PS by the number of processor bits PMS, gives the probability any specific bit is upset. This number is then multiplied by the number of critical processor register bits (WS)(NOCPR).

- 2) the probability a critical processor control flip-flop is upset

$$= (NOFF)(PS)/(PMS)$$

Same as 1, except NOFF is in terms of flip-flops, consequently it does not need to be scaled by WS.

- 3) the probability a control variable is upset, and its results are reasonable

$$= (MS/MMS + PS/PMS)(NOS)(NOB)/(2^{**}WS) \\ + (MS/MMS + PS/PMS)(NOCV)(NORV)/(2^{**}WS)$$

The first term is the probability that a sequence control register is upset to a reasonable value, which is equal to the probability a sequence control register is upset in main memory (MS/MMS) or a sequence control register is upset in the processor memory (PS/PMS) multiplied by the number of reasonable values (NOS*NOB) divided by the total number of values possible (2**WS).

The second factor is the same as the first, except it is scaled by the number of critical variables other than the SCC, and the number of reasonable values NORV, which is the average number of reasonable values for all the critical variables.

- 4) the probability a sequence upset jumps to an SCC assignment

$$= (PSS)(NOS)(NOB)/(AS)$$

PSS is the probability a sequence upset occurs, given an upset has occurred. Multiplying PSS by the number of sequence assignments (NOS)(NOB) and dividing by the total number of possible addresses gives the probability a sequence upset jumps to an SCC assignment.

- 5) the probability a sequence upset jumps to a non-idempotent action section

$$= (PSS)(NOB)(COM)/(AS)$$

PSS is the probability a sequence upset occurs, given an upset

has occurred. Multiplying PSS by the number of action statements (NOB)(COM) and dividing by the total number of possible addresses gives the probability a sequence upset jumps to a action section. This value is an upper bound, since an erroneous execution of an action section is only a catastrophe if the section is not idempotent.

Further explanation of calculations 4 and 5:

In calculations 4 and 5, the probability of a sequence catastrophe is given as the number of bad addresses divided by the address space. This calculation assumes that the probability of jumping from any random address to a given bad address is equally likely for all addresses independent of N, the number of bits flipped by the upset. I wish to show that for random bad addresses and large address spaces, the calculation is independent of N.

let

WS = word size

AS = address space

N = upper bound of number of bits flipped

M = number of bad addresses

$f(N,M)$ = number of original addresses which can permute to a bad address

$g(N)$ = number of N bit permutations

Suppose the word size $WS = 8$, $AS = 256$, and $N = 4$ and I am given a bad address. The first question is how many original addresses can map to the bad address with N or less bit flips ? Since this calculation is like a Bernoulli process,

N	=	0	1	2	3	4	5	6	7	8
number of addresses mapped by N bit flips	=	1	8	28	56	70	56	28	8	1
number of addresses mapped by N or less bit flips = f(N,M=1)	=	1	9	37	93	163	219	247	255	256

So the probability of being on an address which can map to a bad address with $N = 4$ bit flips is $f(N,M=1)/AS$, or in this case, $163/256$. The next question is, given an address which can be mapped to a bad address, what is the probability that it will be mapped to a bad address? This calculation is the same Bernoulli process as above, so this probability is $1/g(N) = 1/f(N,M=1)$.

For one bad address, the probability of jumping to that bad address is

$$\begin{aligned} &\text{Prob of jumping to a bad address} \\ &= (f(N,M=1) / AS) * (1 / g(N)) = 1/AS \end{aligned}$$

For many bad addresses, the upper bound for $f(N,M>1)$ is $M*f(N, M=1)$. However, several bad addresses can have the same original address, so $f(N,M) \leq M*f(N,M=1)$ and of course, $f(N,M) \leq AS$. But for large AS , the probability of overlap is small. Furthermore, if the bad addresses are random, the overlap of original address should be negligible. Consequently,

$$\begin{aligned} &\text{Prob of jumping to a bad address (M)} \\ &= M/AS, \text{ where } M \ll AS \end{aligned}$$

which is independent of N .

4.2.4 Intelsat VI ACE Example

In this analysis, the conditional probability of catastrophe given the occurrence of an upset and the mean time to catastrophe are calculated with parameters from the Intelsat VI ACE for both the SEU and MBU cases. The system constants were taken from Obert [15].

Example 1 - Intelsat VI ACE with Software Implemented Transient Error Recovery

System Constants

- 1) frequency of upsets to system
(in SEUs/(bit-day)) := 0.0001
- 2) frequency of upsets to system
(in MBUs/(bit-day)) := 0.000001
- 3) probability of main memory upset
given an upset to system := 0.96
- 4) probability of processor memory upset
given an upset to system := 0.035
- 5) probability of processor sequence upset
given an upset to system := 0.005
- 6) size of main memory (in bits) := 13 K
- 7) size of processor memory (in bits) := 539
- 8) action section size (in instructions) := 5
- 9) time in the wait loop := 0.80
- 10) word size (in bits) := 8
- 11) address space (in bytes) := 2^{16}
- 12) number of blocks := 10
- 13) number of sections := 3
- 14) number of critical processor registers := 3
- 15) number of critical control variables := 3

- 16) number of reasonable values
for control variables := 10
- 17) number of critical processor control
flip-flops (in bits) := 30
- 18) number of control loops executed
per second := 42

Given that an upset has occurred...

First-Order Catastrophes:

- 1) the probability a critical processor register is upset

$$= (WS)(NOCPR)(PS)/(PMS) = (8)(3)(0.035)/(539) = 0.00156$$
- 2) the probability a critical processor control flip-flop is upset

$$= (NOFF)(PS)/(PMS) = (30)(0.035)/(539) = 0.00195$$
- 3) the probability a control variable is upset, and the results are reasonable

$$= (MS/MMS + PS/PMS)(NOS)(NOB)/(2^{**}WS)$$

$$+ (MS/MMS + PS/PMS)(NOCV)(NORV)/(2^{**}WS)$$

$$= (0.96/13*2^{**}10 + 0.035/539)(3)(10)/(2^{**}8)$$

$$+ (0.96/13*2^{**}10 + 0.035/539)(3)(10)/(2^{**}8)$$

$$= (0.000137)(60)/(2^{**}8) = 0.0000321$$
- 4) the probability a sequence upset jumps to an SCC assignment

$$= (PSS)(NOS)(NOB)/(AS)$$

$$= (0.005)(3)(10)/(2^{**}16)$$

$$= 0.00000229$$
- 5) the probability a sequence upset jumps to a non-idempotent action section

$$= (PSS)(NOB)(COM)/(AS)$$

$$= (0.005)(10)(5)/(2^{**}16)$$

$$= 0.00000381$$

Second-Order Catastrophes:

Since second-order catastrophes require two upsets to occur during the execution of one control loop, it is first necessary to calculate the probability of two upsets within one loop, given that one upset has already occurred.

Active time during one control loop

$$\begin{aligned} &= T1 = (1 - WL) / CLFREQ = (1 - 0.80) / 42 \\ &= 0.00476 \text{ sec} \end{aligned}$$

Memory size

$$= MMS + PMS = 13 \times 2^{10} + 539 = 13851 \text{ bits}$$

Frequency of upsets

$$\begin{aligned} &= 1e-4 \text{ SEUs}/(\text{bit-day}) * 13851 \text{ bits} * 1 \text{ day} / 86400 \text{ sec} \\ &= 0.0000160 \text{ SEUs/sec} \end{aligned}$$

$$\text{Mean time between upsets (MTBU)} = 1/0.0000160 = 62378 \text{ sec}$$

Probability that another upset occurs within T1

$$= T1 / \text{MTBU} = 0.00476 / 62378 = 0.0000000763$$

Since the probability of two upsets occurring during one control loop is several orders of magnitude less than the probability of a first-order catastrophe occurring, it is safe to ignore higher order catastrophes.

TOTAL PROBABILITY OF CATASTROPHE given an upset has occurred

$$\begin{aligned} &= 0.00156 + 0.00195 + 0.0000321 + 0.00000229 \\ &\quad + 0.00000381 \\ &= 0.00354 \end{aligned}$$

MEAN TIME TO CATASTROPHE

For MBUs,

$$\begin{aligned} P &= \text{probability of catastrophe given an upset} \\ &= 0.00354 \end{aligned}$$

$$\begin{aligned} \text{MTC} &= \text{mean time to catastrophe} \\ &= 1 / (\text{Mbufreq} * \text{PMS} * P) \\ &= 1 / (1e-6 \text{ MBUs}/(\text{bit-day}) * (13 \text{ K} + 539) \text{ bits} * \\ &\quad 0.00354 \text{ catastrophe/upset}) \\ &= 20400 \text{ days} \end{aligned}$$

For SEUs, all registers, except critical control flip-flops, are vulnerable only during active processing time. Any other single event upsets that occur during the execution of the wait loop are covered.

For SEUs, the TOTAL PROBABILITY OF CATASTROPHE given an upset has occurred

$$\begin{aligned} &= 0.00195 + (1 - 0.80)(0.00156 + 0.0000321 \\ &\quad + 0.00000229 + 0.00000381) \\ &= 0.00226 \end{aligned}$$

MEAN TIME TO CATASTROPHE

For SEUs,

$$\begin{aligned} P &= \text{probability of catastrophe given an upset} \\ &= 0.00226 \end{aligned}$$

$$\begin{aligned} \text{MTC} &= \text{mean time to catastrophe} \\ &= 1 / (\text{SEUFREQ} * \text{PMS} * P) \end{aligned}$$

$$= 1 / (1e-4 \text{ SEUs}/(\text{bit-day}) * (13 \text{ K} + 539) \text{ bits} * 0.00226 \text{ catastrophe/upset})$$

$$= 318 \text{ days}$$

4.2.5 Future Spacecraft Example

In this analysis, the conditional probability of catastrophe given the occurrence of an upset and the mean time to catastrophe are calculated with parameters from a hypothetical future spacecraft for both the SEU and MBU cases.

The system constants used in the future spacecraft example are intended to give a conservative lower bound in the performance of Software Implemented Transient Error Recovery. The hypothetical system has a 16 bit data word and 60 K bits main memory. The processor and software systems have about four times the complexity of their Intelsat VI counterparts. The upset rates are assumed to be the same as the Intelsat VI ACE upset rates.

Example 2 - Hypothetical future spacecraft with Software Implemented Transient Error Recovery

System Constants

- 1) frequency of upsets to system
(in SEUs/(bit-day)) := 0.0001
- 2) frequency of upsets to system
(in MBUs/(bit-day)) := 0.000001
- 3) probability of main memory upset
given an upset to system := 0.960
- 4) probability of processor memory upset
given an upset to system := 0.035
- 5) probability of processor sequence upset
given an upset to system := 0.005

- 6) size of main memory (in bits) := 60 K
- 7) size of processor memory (in bits) := 2 K
- 8) action section size (in instructions) := 10
- 9) time in the wait loop := 0.50
- 10) word size (in bits) := 16
- 11) address space (in bytes) := 2^{32}
- 12) number of blocks := 20
- 13) number of sections := 3
- 14) number of critical processor registers := 10
- 15) number of critical control variables := 10
- 16) number of reasonable values
for control variables := 100
- 17) number of critical processor control
flip-flops (in bits) := 100
- 18) number of control loops executed
per second := 50

Given that an upset has occurred...

First-Order Catastrophes:

- 1) the probability a critical processor register is upset

$$= (WS)(NOCPR)(PS)/(PMS) = (16)(10)(0.035)/(2^{2*10})$$

$$= 0.00273$$
- 2) the probability a critical processor control flip-flop is upset

$$= (NOFF)(PS)/(PMS) = (100)(0.035)/(2^{2*10})$$

$$= 0.00170$$
- 3) the probability a control variable is upset, and the results are reasonable

$$= (MS/MMS + PS/PMS)(NOS)(NOB)/(2^{*WS})$$

$$+ (MS/MMS + PS/PMS)(NOCV)(NORV)/(2^{*WS})$$

$$\begin{aligned}
&= (0.996/60 \cdot 2^{10} + 0.035/2 \cdot 2^{10})(3)(20)/(2^{16}) \\
&\quad + (0.996/60 \cdot 2^{10} + 0.035/2 \cdot 2^{10})(10)(100)/(2^{16}) \\
&= (0.0000333)(1060)/(2^{16}) = 0.000000539
\end{aligned}$$

4) the probability a sequence upset jumps to an SCC assignment

$$\begin{aligned}
&= (PSS)(NOS)(NOB)/(AS) \\
&= (0.005)(3)(20)/(2^{32}) \\
&= 6.98 \text{ e }^{-11}
\end{aligned}$$

5) the probability a sequence upset jumps to a non-idempotent action section

$$\begin{aligned}
&= (PSS)(NOB)(COM)/(AS) \\
&= (0.005)(20)(10)/(2^{32}) \\
&= 2.32 \text{ e }^{-10}
\end{aligned}$$

Second-Order Catastrophes:

Again, since second-order catastrophes require two upsets to occur during the execution of one control loop, it is first necessary to calculate the probability of two upsets within one loop, given that one upset has occurred.

Active time during one control loop

$$= T_1 = (1 - WL) / CLFREQ = (1 - 0.50) / 50 = 0.01 \text{ sec}$$

Memory size

$$= MMS + PMS = 60 \cdot 2^{10} + 2 \cdot 2^{10} = 63488 \text{ bits}$$

Frequency of upsets

$$\begin{aligned}
&= 1e-4 \text{ SEUs}/(\text{bit-day}) \cdot 63488 \text{ bits} \cdot 1 \text{ day} / 86400 \text{ sec} \\
&= 0.0000735 \text{ SEUs}/\text{sec}
\end{aligned}$$

$$\text{Mean time between upsets (MTBU)} = 1/0.0000735 = 13608 \text{ sec}$$

Probability that another upset occurs in T1

$$= T1 / \text{MTBU} = 0.01 / 13608 = 0.000000735$$

Since the probability of two upsets occurring during one control loop is several orders of magnitude less than the probability of a first-order catastrophe occurring, it is safe to ignore higher order catastrophes.

TOTAL PROBABILITY OF CATASTROPHE given an upset has occurred

$$\begin{aligned} &= 0.00273 + 0.00170 + 0.000000539 + 6.98 \text{ e }^{-11} \\ &\quad + 2.32 \text{ e }^{-10} \\ &= 0.00444 \end{aligned}$$

MEAN TIME TO CATASTROPHE

For MBUs,

$$\begin{aligned} P &= \text{probability of catastrophe given an upset} \\ &= 0.00444 \end{aligned}$$

$$\begin{aligned} \text{MTTC} &= \text{mean time to catastrophe} \\ &= 1 / (\text{MUFREQ} * \text{PMS} * P) \\ &= 1 / (1\text{e-}6 \text{ MBUs}/(\text{bit-day}) * (60 \text{ K} + 2 \text{ K}) \text{ bits} * \\ &\quad 0.00444 \text{ catastrophe/upset}) \\ &= 3540 \text{ days} \end{aligned}$$

For SEUs, all registers, except processor control flip-flops, are vulnerable only during active processing time. Any other single event upsets that occur during the execution of the wait loop are covered.

For N = 1, TOTAL PROBABILITY OF CATASTROPHE given an upset has occurred

$$\begin{aligned} &= 0.00170 + (1 - 0.50) (0.00273 + 0.000000539 \\ &\quad + 6.98 e^{-11} + 2.32 e^{-10}) \\ &= 0.00307 \end{aligned}$$

MEAN TIME TO CATASTROPHE

For SEUs,

$$\begin{aligned} P &= \text{probability of catastrophe given an upset} \\ &= 0.00307 \end{aligned}$$

$$\begin{aligned} \text{MTTC} &= \text{mean time to catastrophe} \\ &= 1 / (\text{SEUFREQ} * \text{PMS} * P) \\ &= 1 / (1e^{-4} \text{ SEUs}/(\text{bit-day}) * (60 \text{ K} + 2 \text{ K}) \text{ bits} * \\ &\quad 0.00307 \text{ catastrophe/upset}) \\ &= 51.2 \text{ days} \end{aligned}$$

4.3 Register Method

The results of the detailed probabilistic analysis suggests that an alternative form of analysis can give reasonable results. Since the probability of a catastrophic upset in a register which is not covered by the methodology is far greater than a catastrophic upset to a register which is covered, a close approximation to the probability of catastrophe given an upset is the ratio of the number uncovered registers to the total number of registers. Although this new approach using coverage is certainly less accurate than the original, it offers an advantage in that it can be applied to any system using any recovery methodology. In this analysis, the conditional probability of catastrophe given the occurrence

of an upset and the mean time to catastrophe are calculated for four methodologies: a worst cases methodology, the Intelsat VI ACE methodology, the proposed methodology, and the best case methodology. Each case examines present and future spacecraft sub-system configurations, for both SEU and MBU cases.

As noted in section 3.3, there are problems associated with using coverage as a performance metric, specifically, coverage is difficult to define precisely. However, since this technique is used to give a rough approximation, the difficulty in making a precise definition of coverage are not important.

Note on accuracy:

It should be noted that it is very difficult to estimate various key parameters for this analysis, such as the number of critical flip-flops, and the number of critical registers, as well as the upset rates. Since the result of any calculation can be no more accurate than the numbers it uses, more accurate calculations (than presented in this section) are of little use without more accurate parameters. However, since the objective of the calculations is to make relative comparisons between different recovery configurations, rather large errors can be tolerated.

4.3.1 Register Method Procedure

The procedure to determine the conditional probability of catastrophe given an upset using the register method is quite simple. For a given sub-system and a given recovery methodology, there are four cases : SEU computation case, SEU wait loop case, MBU computation case, and MBU wait loop case. For each case, one must categorize all the

flip-flops in the sub-system as either covered by a methodology, or not covered by a methodology. As a rule, a flip-flop should be covered at least 80% of the time to be considered covered by a methodology. For each case, the probability of catastrophe given an upset is the ratio of the number of uncovered flip-flops to total number of flip-flops. Then the computation case and wait loop case are combined as a weighted sum.

Four flip-flop categories are used : main memory, internal memory, critical registers, and critical flip-flops. Main memory is the volatile RAM, including any RAM used for error correcting purposes. The critical registers are registers within the processor and its supporting logic which can result in a catastrophe if upset, and are not covered by Software Implemented Transient Error Recovery. Since they are referred to as "registers" (as opposed to "flip-flops"), the implication is that they typically hold value information as opposed to control information. Critical flip-flops are flip-flops within the processor and its supporting logic which can result in a catastrophe if upset, and are not covered by Software Implemented Transient Error Recovery. Since they are referred to as "flip-flops" (as opposed to "registers"), the implication is that they typically hold control information as opposed to value information. Critical flip-flops cannot be covered by any software implemented recovery technique. Internal Memory is the remaining memory in the processor and its supporting logic that is not considered critical.

The flip-flops for the Intelsat VI ACE configuration and the future spacecraft configuration break down as follows:

Intelsat VI ACE Memory Parameters

Main Memory Size	13312	bits
Internal Memory Size	485	bits
Critical Register Size	24	bits
Critical Flip-Flop Size	30	bits

Table 1

Future Spacecraft Sub-system Memory Parameters

Main Memory Size	61440	bits
Internal Memory Size	1788	bits
Critical Register Size	160	bits
Critical Flip-Flop Size	100	bits

Table 2

This break down is in agreement with the values previously used in the detailed probabilistic analysis.

The above procedure is applied to the data contained in tables 1 through 6, and is summarized in section 4.4.

4.3.2 The Worst Case Methodology

Definition :

The worst case methodology provides the minimum coverage. It uses error-correcting codes to recover from SEUs in main memory.

General Spacecraft Sub-System Worst Case Methodology Coverage

SEU Case	Computation	Wait Loop
Main Memory	C	C
Internal Registers	NC	NC
Critical Registers	NC	NC
Critical Flip-Flops	NC	NC

MBU Case	Computation	Wait Loop
Main Memory	NC	NC
Internal Registers	NC	NC
Critical Registers	NC	NC
Critical Flip-Flops	NC	NC

C : Covered by Methodology
NC : Not Covered by Methodology

Table 3

4.3.3 The Intelsat VI ACE Methodology

Definition :

The Intelsat VI ACE methodology is the approach proposed by Obert. It protects the main memory from SEUs, and the internal and critical registers from SEUs while in the wait loop. It provides very little coverage of internal and critical registers during computation. It offers very little coverage against MBUs.

General Spacecraft Sub-System Intelsat VI ACE Methodology Coverage

SEU Case	Computation	Wait Loop
Main Memory	C	C
Internal Registers	NC	C
Critical Registers	NC	C
Critical Flip-Flops	NC	NC

MBU Case	Computation	Wait Loop
Main Memory	NC	NC
Internal Registers	NC	NC
Critical Registers	NC	NC
Critical Flip-Flops	NC	NC

C : Covered by Methodology
NC : Not Covered by Methodology

Table 4

4.3.4 The Proposed Methodology

Definition :

The proposed methodology covers all of the processor and control memory except for the critical registers and flip-flops. Critical registers are covered during the wait loop for SEUs.

**General Spacecraft Sub-System
Proposed Methodology Coverage**

SEU Case	Computation	Wait Loop
Main Memory	C	C
Internal Registers	C	C
Critical Registers	NC	C
Critical Flip-Flops	NC	NC

MBU Case	Computation	Wait Loop
Main Memory	C	C
Internal Registers	C	C
Critical Registers	NC	NC
Critical Flip-Flops	NC	NC

C : Covered by Methodology
NC : Not Covered by Methodology

Table 5

4.3.5 The Best Case Methodology

Definition :

The best case methodology provides the maximum protection against SEUs and MBUs possible using software techniques alone. Only upsets to the critical flip-flops can result in catastrophe.

General Spacecraft Sub-System Best Case Methodology Coverage

SEU Case	Computation	Wait Loop
Main Memory	C	C
Internal Registers	C	C
Critical Registers	C	C
Critical Flip-Flops	NC	NC

MBU Case	Computation	Wait Loop
Main Memory	C	C
Internal Registers	C	C
Critical Registers	C	C
Critical Flip-Flops	NC	NC

C : Covered by Methodology
NC : Not Covered by Methodology

Table 6

4.4 Summary of Results

PROBABILITY OF CATASTROPHE GIVEN AN UPSET

Intelsat VI ACE Configuration

	SEU	MBU (N > 1)
Worst Case	0.0389	1.0
ACE Methodology	0.00952	1.0
Proposed Methodology	0.00251	0.00390
Proposed Methodology (from detailed analysis)	0.00226	0.00355
Best Case	0.00217	0.00217

Table 7

Future Spacecraft Configuration

	SEU	MBU (N > 1)
Worst Case	0.0323	1.00
ACE Methodology	0.0169	1.00
Proposed Methodology	0.00284	0.00410
Proposed Methodology (from detailed analysis)	0.00307	0.00444
Best Case	0.00158	0.00158

Table 8

MEAN TIME TO CATASTROPHE
(in days)

Intelsat VI ACE Configuration

	SEU	MBU (N > 1)
Worst Case	18.6	72.2
ACE Methodology	75.9	72.2
Proposed Methodology	287.	18500.
Proposed Methodology (from detailed analysis)	318.	20400.
Best Case	333.	33300.

Table 9

Future Spacecraft Configuration

	SEU	MBU (N > 1)
Worst Case	4.88	15.8
ACE Methodology	9.31	15.8
Proposed Methodology	55.6	3850.
Proposed Methodology (from detailed analysis)	51.2	3540.
Best Case	100.	10000.

Table 10

4.5 Discussion of Results

As indicated in the introduction, the primary motivation for examining the transient error recovery problem is future autonomous spacecraft maintenance (ASM) requirements. Although transient error recovery has some impact upon other system requirements such as reliability, the principle consideration is autonomy. Although catastrophic upsets can result in spacecraft failure, the most likely outcome is loss of autonomy.

Autonomy is defined as the attribute of a spacecraft system that

allows it to operate without external control, and to perform its specified mission at an established performance level for a specified period of time. Typically, the period specified in autonomy requirements is one month.

By examining figures 1 - 4, it is clear that the transient error recovery methodology proposed for the Intelsat VI ACE is sufficient for that system, with a MTTC for SEUs of over 2 months. However, if the same transient error recovery methodology is applied to the future spacecraft configuration, the results are unacceptable, with a MTTC of only 6 days for SEUs. The proposed transient error recovery has much better performance, with a MTTC of over 2 months for SEUs, which should be acceptable. Although MBUs are not a problem with the Intelsat VI ACE configuration, they become an important factor with the future spacecraft configuration, since the Intelsat VI ACE methodology provides almost no protection from MBUs.

For the present spacecraft configuration, the proposed methodology offers approximately a factor of 4 improvement over the Intelsat VI ACE methodology, and a factor of 16 improvement over the worst case. For the future spacecraft configuration, the proposed methodology offers approximately a factor of 10 improvement over the Intelsat VI ACE methodology, and a factor of 12 improvement over the worst case.

The most important observation to be made is that several orders of magnitude improvement over the worst case methodology is simply not possible using software techniques alone. The limiting factor is the inability to recover from upsets to critical control flip-flops using software techniques. To make a major break-through in transient error recovery, both hardware and software redundancy techniques must be used.

CONCLUSION

As stated in the introduction, the proposed methodology for Software Implemented Transient Error Recovery has the following advantages over previous efforts: 1) it offers a structured, standardized approach to transient error recovery, 2) it has safeguards to limit error propagation, 3) it is effective on multiple event upsets, and 4) it does not require re-initialization to recover from upsets in most cases.

The transient error recovery methodology used by on Intelsat VI ACE has its merits. Specifically, it is relatively simple to implement and it is very effective in recovering from single event upsets. With present spacecraft requirements and configurations, this approach is probably the method of choice, since single event upsets are the predominant source of transient errors and since the present control systems are greatly oversampled, giving natural immunity to transient errors. However, future spacecraft systems and requirements may not have the tolerance that present systems have. As shown in the probabilistic analysis, it seems unlikely that the future spacecraft configuration using the recovery methodology used on the Intelsat VI ACE could achieve a one month autonomy requirement. It was also shown that the proposed transient error recovery methodology could make a one month autonomy requirement.

An important conclusion which must be drawn is that if extremely lengthy autonomy periods are required, or very complex control systems are used, then software implemented techniques are not sufficient. Such systems would have to integrate both hardware and software recovery techniques. The proposed Software Implemented Transient Error Recovery methodology is compatible with most hardware oriented fault-tolerant techniques, and could be used with them.

APPENDIX

ADDITIONAL CONSIDERATIONS FOR TRANSIENT ERROR RECOVERY

This is a summary of additional programming rules, many of which are unrelated to the methodology, but are important to minimizing the effects of transient errors.

1) No FOR statements:

A FOR statement which is improperly entered by a sequence upset will never terminate. A simple solution is to use a WHILE statement, which will always terminate.

2) Reasonableness checking of loop variables during WHILE loops:

Although WHILE loops will always terminate in the presence of upsets, it may take a very long time. For example, a WHILE loop which iterates from one to four, if upset to -10000, will eventually terminate, but could take a very long time to do so.

3) Computation loop should have an upper bound on the number of iterations:

If for some reason the computation block never obtains two equal results, the computation section will never terminate. The computation section should call a recovery block after the execution of some maximum number of iterations.

4) Carefully choose sequence control codes:

There are two considerations in choosing sequence control codes:

a) The larger the Hamming distance between codes, the more effective the error recovery. The error recovery routine checks sequence control codes for reasonableness. Large Hamming

distances between codes will prevent upset codes from appearing correct.

b) The simpler the codes, the easier it is to check the code for reasonableness in the recovery block. Use codes which conform to some pattern which is easy to check.

5) Use specific constants instead of variables when possible:

Variables which influence the permanent program state that are upset eventually have to be corrected. Constants, which are stored in program ROM, do not have to be corrected if upset since their values are re-read during program execution.

6) Clear stack on RTI entry:

Protects against stack pointer upsets.

7) Periodically re-enable interrupts:

To protect against spurious disable interrupt instructions, re-enable interrupt instructions should be placed throughout the code.

8) Run-time considerations in calculation of reasonableness checks:

The results of time varying computations cannot be directly compared, since the correct results change with time. The difference between time variant computations would have to be compared to some predetermined margin.

9) Give priority to the most important routines:

Since the first blocks in a control program have a higher probability of completion than the last blocks, the most important functions, such as the despin function on the Intelsat

VI ACE, should be executed first.

10) Something should be done in hardware to call a error recovery routine when the processor attempts to address memory outside the physical address space.

11) Protect against machine code mis-interpretation with NOPs:

Machine code mis-interpretation is caused by the ambiguity of stored programs, since machine language is context sensitive. For example, the machine code 10110001 may be interpreted as the instruction "inc R1", or as the data B1 hex. Consequently, if a sequence error occurs, the code intended to detect this error may be misinterpreted. This problem is solved with a process called "NOP buffering".

Example:

```
instead of compiling

    x := x + 1;
    if scc <> 5 then
        goto error_recovery;

as

    mov R0, x
    inc R0
    mov x, R0

    mov R0, scc
    mov R1, 5
    cmp
    jne error_recovery
```

use

```
mov R0, x
inc R0
mov x, R0
nop
nop
mov R0, scc
mov R1, 5
cmp
jne error_recovery
```

To see why NOP buffering is a good idea, we have to examine the machine code. To do this, I will use a hypothetical instruction set.

With NOPs:

```
1 -> 1: 10101000      mov R0, x
      2: 11110010
2 -> 3: 00010010
      4: 11001000      inc R0
      5: 10100000      mov x, R0
3 -> 6: 11110010
      7: 00010010
      8: 00000000      nop
4 -> 9: 00000000      nop
      10: 10101000     mov R0, scc
      11: 11000101
      12: 00101010
      13: 10001001     mov R1, 5
      14: 00000101
      15: 11010101     cmp
      16: 10101101     jne error_recovery
      17: 01010100
      18: 01001001
```

Assume a sequence error has occurred. The program counter may be set to any location in the address space. Lets examine how the machine code is interpreted in the above cases.

Case 1:

Sequence error coincides with the beginning of an instruction.

The machine code is interpreted correctly.

Case 2:

Sequence error does not coincide with the beginning of an instruction. In the worst case, the machine code could be interpreted as:

```
3:    00010010          add R2, ADDR1
4:    11001000
5:    10100000
6:    11110010          rot R2
7:    00010010          mul R2, ADDR2
8:    00000000
9:    00000000
10:   10101000          mov R0, scc
11:   11000101
12:   00101010
      etc..
```

Lines 3 through 9 are mis-interpreted, so they executed incorrectly. However, correct execution begins on line 10, so the sequence error is detected by testing the sequence control code. Had there not been NOPs, the instructions may have been interpreted as:

```
3:    00010010          add R2, ADDR1
4:    11001000
5:    10100000
6:    11110010          rot R2
7:    00010010          mul R2, ADDR2
10:   10101000
11:   11000101
12:   00101010          dec R2
      etc..
```

Without the NOPs, the code intended to detect sequence errors is completely ineffective, since it is not correctly executed.

Case 3:

Sequence error does not coincide with the beginning of an instruction. Same outcome as case 2.

Case 4:

Sequence error coincides with the beginning of an instruction.

Machine code is interpreted correctly.

As shown in this example, a sequence error which causes machine code mis-interpretation can make error recovery ineffective.

Inserting extra NOPs before recovery code can correct this problem.

THE SIMULATION PROGRAM DESCRIPTION

The objectives of the simulation program are to provide real-time experience with a transient error environment, to demonstrate the methodology, and to evaluate its performance. The Multiple Bit Upset Model, the Upset Mapping Model, the High-Level Language/Machine Language Model, and the Structure/Content Model strive to simplify the system to capture the essence of most high-level errors. These models work together to simplify the system by abstracting various aspects of the system. The Multiple Bit Upset Model is an abstraction of the transient error environment, the Upset Mapping Model is an abstraction of the system hardware, the High-Level Language/Machine Language Model is an abstraction of the hardware/software interface, and the Structure/Content Model is an abstraction of the spacecraft software. The simulation program applies these models to a specific "benchmark" computation block written in the recovery format.

The computation block used in the program is a very simple routine which performs a simple transformation on critical variables. The routine takes the average of four critical variables and writes the average back to the four critical variables. The full error recovery format is implemented for the computation block. However, the simulation program does not simulate errors during error recovery.

The main functions of the simulation program are to execute the recovery software, to monitor the software execution, to inject errors into the program state, and to monitor recovery performance.

The recovery software is written as it would normally be written, except that each program statement begins with a label (needed to simulate sequence errors) and ends with a call to an error simulation routine. In the absence of errors, the error simulation routine simply

returns.

There are three types of errors injected: program sequence errors, main memory errors, and processor memory errors. When the error simulation routine is called, it first determines from the upset rate if it is time for an upset to occur. If so, the simulation routine then determines the type and location of the error from the system parameters (see figure 3), as defined by the Upset Mapping Model and the Structure/Content Model. These parameters are determined by hardware and software specifications.

Simulation System Parameters

- 1) probability of main memory upset
 given an upset to system (Upset Mapping Model)
- 2) probability of processor memory upset
 given an upset to system (Upset Mapping Model)
- 3) probability of processor sequence upset
 given an upset to system (Upset Mapping Model)
- 4) number of critical variables / total variables
 (Upset Mapping Model)
- 5) initialization section size (in instructions)
 (Structure/Content Model)
- 6) computation section size (in instructions)
 (Structure/Content Model)
- 7) action section size (in instructions)
 (Structure/Content Model)
- 8) time in the wait loop (for jump return to wait loop)
 (software dependent)
- 9) frequency of upsets to system
 (hardware dependent)
- 10) word size (in bits)
 (hardware dependent)
- 11) n (size of upset) (in bits)

Figure 3

Memory errors are simulated by simply changing variable values to random numbers. An exception to this procedure is an error to an array subscript. Array subscripts must be within a given range, otherwise program execution is halted. Program sequence errors are simulated by jumping to a random computation block statement.

The recovery metrics used are: number of catastrophes, coverage, and recovery profile. The number of catastrophes requires the classification of recovery failures. Since this task would be very difficult to automate, it is done manually through various display options. The display options used are:

- 1) display program code
- 2) display program data
- 3) display errors injected
- 4) display errors detected
- 5) display current section
- 6) display program data at injection
- 7) display program data at detection
- 8) display program data at beginning of block
- 9) display program data at end of block

Miscellaneous Assumptions

Here is a list of miscellaneous assumptions concerning the implementation of the simulation program.

- 1) The number of bits upset is irrelevant if greater than one.
This assumption simplifies analysis by breaking the upsets into two categories: the SEU case (number of bits upset equals one) and the MBU case (number of bits upset is greater than one).

This is not a perfect assumption, since the performance of reasonableness checking increases as the number of bits upset increases.

- 2) Although control programs are always time varying, it is assumed that the effectiveness of recovery is not influenced by time variance. Consequently, the simulated control program is time invariant.
- 3) The simulation program does not simulate upsets during the execution of the recovery block. It is assumed that upsets which occur during recovery are important only during bursts of upsets. The problem of recovering during recovery is not addressed.

Outline of Simulation Program

A summary of the simulation program is given in Figure 4.

```
program simulate;
  { initialization procedures }
  procedure simulation;
    { initialization procedures }
    { display option procedures }
    { statistics gathering procedures }
    procedure recovery_example (block_number : integer);
      { manual upset injection procedures }
      { automatic upset injection procedures }
      { error simulation driver }
    begin
      { software under test in recovery format
        with calls to error injection routine }
      { recovery evaluation }
      { error recovery block }
    end;
  begin
    while true do
      begin
        initialize;
        for block_number := 1 to number_of_blocks do
          recovery_example (block_number);
        end;
      end;
    end;

    { main program body }
  begin
    initialize_parameters;
    initialize_stats;
    simulation;
  end.
```

Figure 4

THE SIMULATION PROGRAM LISTING

```

program simulate;

type
    { invariant - the probability of all outcomes should add to one }

mapping =          { used by parameters }
record
    memory : real;   { the probability that a MBU affects a processor
                     memory location }
    sequence : real; { the probability that a MBU affects a program
                     sequence }
end;

display_options =
record
    lines          : boolean;
    data           : boolean;
    errors_detected : boolean;
    errors_injected : boolean;
    section        : boolean;
    data_at_injection : boolean;
    data_at_detection : boolean;
    data_at_start  : boolean;
    data_at_end    : boolean;
end;

{ parameter invariants -
  processor_sus + memory_sus = 1
  section1_size > 0
  section2_size > 0
  section3_size > 0
  0 <= critical_variables <= 1
  0 <= wait_loop <= 1
  word_size > 0
  0 <= n <= word_size
  error_period > 1 }

{ note : an error period of 133 is approximately one upset per
  block}

parameters =
record
    processor_sus : real;   { the susceptibility of the processor to MBU's }
    memory_sus    : real;   { the susceptibility of the memory to MBU's }
    neu_map       : mapping; { the mapping of MBU's outcomes }
    section1_size : integer; { the size of a typical initialization section}
                     { in number of instructions executed }
    section2_size : integer; { the size of a typical computation section}
                     { in number of instructions executed }
    section3_size : integer; { the size of a typical action section}
                     { in number of instructions executed }
    critical_var  : real;   { the percentage of critical variables }
    wait_loop     : real;   { the percentage of time in the wait loop }
    n             : integer; { number of bits upset }
    word_size     : integer; { the word size of the processor }
    error_period  : integer32; { the mean period, in instructions, between MBU's }
    test_mode     : boolean; { enables change in display options }
    display       : display_options;
    factor1       : integer; { rate of upset in initialization section }
    factor2       : integer; { rate of upset in computation section }
    factor3       : integer; { rate of upset in action section }

```

```

end;

{ recovery profile }
profile =
  record
    redundant_comp      : integer32;
    redundant_storage  : integer32;
    memory_coding      : integer32;
    scc                 : integer32;
    wait_loop          : integer32;
    reasonableness_check : integer32;
  end;

upset_type =
  record
    wait_loop      : integer32;
    sequence       : integer32;
    processor_memory : integer32;
    memory         : integer32;
  end;

stats =
  record
    total_errors      : integer32; { the total number of errors injected }
    total_errors_corrected : integer32;
    total_coverage    : real;      { the coverage of the whole system }
    recovery          : profile;
    upset             : upset_type;
  end;

var
  p : parameters;          { the parameters of the simulation }
  s : stats;               { the recovery data }
  seed, seed1 : integer;   { used to determine location and type of error }

procedure init_factors (var p : parameters);

  { this procedure calculates the local upset rate as specified by the
    form/content model }

  const
    { these numbers are the number of lines executed per section }
    init_size   = 30;
    calc_size   = 30;
    commit_size = 54;

  var
    temp_factor1, temp_factor2, temp_factor3 , average : real;

begin
  with p do
    begin
      temp_factor1 := init_size / section1_size;
      temp_factor2 := calc_size / section2_size;
      temp_factor3 := commit_size / section3_size;

      average := temp_factor1*init_size + temp_factor2*calc_size +
        temp_factor3*commit_size;
      average := average/(init_size + calc_size + commit_size);

      factor1 := trunc(temp_factor1 * error_period / average);
      factor2 := trunc(temp_factor2 * error_period / average);
      factor3 := trunc(temp_factor3 * error_period / average);

    end;
  end;
end;

```

```

procedure initialize_parameters (var p : parameters);
label
  100;
var
  ch : char;
begin
  { assign default values to parameters }

  p.processor_sus := 0.0;
  p.memory_sus := 1.0;
  p.neu_map.memory := 0.5;
  p.neu_map.sequence := 0.5;
  p.section1_size := 30;
  p.section2_size := 30;
  p.section3_size := 54;
  p.critical_var := 0.5;
  p.wait_loop := 0.0;
  p.word_size := 16;
  p.n := 4;
  p.error_period := 20;
  p.test_mode := true;
  seed := 5;
  seed1 := 10;

  { verify resulting parameters }

  writeln('here are the default parameters: ');
  writeln('processor susceptibility : ', p.processor_sus:4:4);
  writeln('memory susceptibility : ', p.memory_sus:4:4);
  writeln('NEU -> memory mapping : ', p.neu_map.memory:4:4);
  writeln('NEU -> sequence mapping : ', p.neu_map.sequence:4:4);
  writeln('initialization section size : ', p.section1_size);
  writeln('computation section size : ', p.section2_size);
  writeln('action section size : ', p.section3_size);
  writeln('percentage of critical variables : ', p.critical_var:4:4);
  writeln('percentage of time in the wait loop : ', p.wait_loop:4:4);
  writeln('word size : ', p.word_size);
  writeln('n (size of upset) : ', p.n);
  writeln('mean period between upsets : ',
    p.error_period);
  writeln('test mode : ', p.test_mode);

  { make changes to default parameters }

  write('change parameters ? (y/n) ');
  readln(ch);
  if ch = 'y' then
    repeat
      write('change processor susceptibility ? (y/n/q) ');
      readln(ch);
      if ch = 'q' then
        goto 100;
      if ch = 'y' then
        begin
          repeat
            write('processor susceptibility (real : 0 - 1) : ');
            readln(p.processor_sus);
            until (p.processor_sus >= 0) and (p.processor_sus <= 1);
            p.memory_sus := 1 - p.processor_sus;
          end;

          write('change memory susceptibility ? (y/n/q) ');
          readln(ch);
          if ch = 'q' then
            goto 100;
        end;
    end;
  goto 100;
end;

```

```

if ch = 'y' then
begin
repeat
write('memory susceptibility (real : 0 - 1) : ');
readln(p.memory_sus);
until (p.memory_sus >= 0) and (p.memory_sus <= 1);
p.processor_sus := 1 - p.memory_sus;
end;

write('change NEU outcome mapping ? (y/n/q) ');
readln(ch);
if ch = 'q' then
goto 100;
if ch = 'y' then
repeat
writeln('probabilities must add to 1');
repeat
write('memory upset probability (real : 0 - 1) : ');
readln(p.neu_map.memory);
until (p.neu_map.memory >= 0) and (p.neu_map.memory <= 1);
repeat
write('sequence upset probability (real : 0 - 1) : ');
readln(p.neu_map.sequence);
until (p.neu_map.sequence >= 0) and (p.neu_map.sequence <= 1);
until (p.neu_map.memory + p.neu_map.sequence) = 1;

write('change initialization section size ? (y/n/q) ');
readln(ch);
if ch = 'q' then
goto 100;
if ch = 'y' then
repeat
write('initialization section size in executed instructions (integer > 0) : ');
readln(p.section1_size);
until p.section1_size > 0;

write('change computation section size ? (y/n/q) ');
readln(ch);
if ch = 'q' then
goto 100;
if ch = 'y' then
repeat
write('computation section size in executed instructions (integer > 0) : ');
readln(p.section2_size);
until p.section2_size > 0;

write('change action section size ? (y/n/q) ');
readln(ch);
if ch = 'q' then
goto 100;
if ch = 'y' then
repeat
write('action section size in instructions (integer >= 1) : ');
readln(p.section3_size);
until p.section3_size > 0;

write('change percentage of critical variables ? (y/n/q) ');
readln(ch);
if ch = 'q' then
goto 100;
if ch = 'y' then
repeat
write('percentage of critical variables (real : 0 - 1) : ');
readln(p.critical_var);
until (p.critical_var >= 0) and (p.critical_var <= 1);

write('change wait loop time ? (y/n/q) ');
readln(ch);

```

```

if ch = 'q' then
    goto 100;
if ch = 'y' then
    repeat
        write('percentage of time in wait loop (real : 0 - 1) : ');
        readln(p.wait_loop);
        until (p.wait_loop >= 0) and (p.wait_loop <= 1);

write('change word size ?                               (y/n/q) ');
readln(ch);
if ch = 'q' then
    goto 100;
if ch = 'y' then
    repeat
        write('word size (integer > 0) : ');
        readln(p.word_size);
        until p.word_size > 0;

write('change n (size of upset) ?                       (y/n/q) ');
readln(ch);
if ch = 'q' then
    goto 100;
if ch = 'y' then
    repeat
        write('n (integer : 1 <= n <= word size) : ');
        readln(p.n);
        until (p.n >= 1) and (p.n <= p.word_size);

write('change error period ?                             (y/n/q) ');
readln(ch);
if ch = 'q' then
    goto 100;
if ch = 'y' then
    repeat
        write('error period in instructions (integer >> 1) : ');
        readln(p.error_period);
        until p.error_period > 1;

write('change test mode ?                               (y/n/q) ');
readln(ch);
if ch = 'q' then
    goto 100;
if ch = 'y' then
    begin
        write('test mode (boolean) : ');
        readln(p.test_mode);
    end;

    { verify resulting parameters }

100:
writeln('here are the resulting parameters: ');
writeln('processor susceptibility :                    ', p.processor_sus:4:4);
writeln('memory susceptibility :                        ', p.memory_sus:4:4);
writeln('NEU -> memory mapping :                          ', p.neu_map.memory:4:4);
writeln('NEU -> sequence mapping :                          ', p.neu_map.sequence:4:4);
writeln('initialization section size :                      ', p.section1_size);
writeln('computation section size :                          ', p.section2_size);
writeln('action section size :                              ', p.section3_size);
writeln('percentage of critical variables :                  ', p.critical_var:4:4);
writeln('percentage of time in the wait loop :              ', p.wait_loop:4:4);
writeln('word size :                                         ', p.word_size);
writeln('n (size of upset) :                                 ', p.n);
writeln('mean period between upsets :                       ',
    p.error_period);
writeln('test mode :                                         ', p.test_mode);

write('are these the desired parameters ? (y/n) ');
readln(ch);

```

```

    until not (ch = 'n');

init_factors(p);

if p.test_mode then
begin
    p.display.lines           := true;
    p.display.data           := false;
    p.display.errors_detected := true;
    p.display.errors_injected := true;
    p.display.section        := false;
    p.display.data_at_injection := true;
    p.display.data_at_detection := true;
    p.display.data_at_start   := true;
    p.display.data_at_end    := true;
end
else
begin
    p.display.lines           := false;
    p.display.data           := false;
    p.display.errors_detected := false;
    p.display.errors_injected := false;
    p.display.section        := false;
    p.display.data_at_injection := false;
    p.display.data_at_detection := false;
    p.display.data_at_start   := false;
    p.display.data_at_end    := false;
end;
end;

procedure initialize_stats (var s : stats);
begin
    with s do
        begin
            total_errors           := 0;
            total_errors_corrected := 0;

            recovery.redundant_comp := 0;
            recovery.redundant_storage := 0;
            recovery.memory_coding := 0;
            recovery.scc := 0;
            recovery.wait_loop := 0;
            recovery.reasonableness_check := 0;
        end;
    end;
end;

procedure simulation(var p : parameters;var s : stats);

type
    control =
        record
            {simulation control data}
            free_run : boolean;           { free run mode }
            line_break : boolean;         { break point mode }
            break_point_line : integer;   { break point line number }
            error : boolean;              { has an error been detected ? }
            skip_block : boolean;         { sequence error to another block }
            skip_to_line : integer;       { skip to line number }
            error_message : string;       { the cause of the last discovered error }
            number_of_computations : integer; { the actual number of computations }
            recovery_line : integer;      { the line number where error was detected }
        end;

{ values - the domain and range of average
  e.g. average(values) -> values

```



```

        two copies of values are needed to allow atomic actions
        three copies of the above are kept for redundant storage
temp    - the extra copy of temp is used for repeating computations
block_number, value_number, copy_number
        - the variables for iteration
temp_version
        - toggles the temp variables
number_of_computations
        - the cumulative number of computations
scc     - sequence control code }

const
number_of_blocks = 10;
number_of_lines  = 54;
number_of_values = 4;
number_of_copies = 3;

var
values : array [0..1,1..number_of_copies,
               1..number_of_values,1..number_of_blocks] of integer;
temp   : array [0..9] of integer;
block_number, value_number, copy_number, scc : integer;
number_of_computations, temp_version : integer;
sim_data : control;
il,i2,i3,i4 : integer;

procedure init;
begin
    { initialize program data }
    scc := 003;
    value_number := 0;
    copy_number := 0;
    { initialize values [0] to block_number*100 }
    for i2 := 1 to number_of_copies do
        for i3 := 1 to number_of_values do
            for i4 := 1 to number_of_blocks do
                begin
                    values [1,i2,i3,i4] := i4 * 100;
                    values [0,i2,i3,i4] := 0;
                end;
            end;
        end;
    for il := 0 to 9 do
        temp [il] := 0;
    number_of_computations := 0;
    temp_version := 0;
    { initialize control data }
    with sim_data do
        begin
            free_run := false;
            line_break := false;
            break_point_line := 100 + number_of_lines;
            error := false;
            skip_block := false;
            error_message := '';
            number_of_computations := 0;
            recovery_line := 0;
        end;
    end;
end;

function random(var seed : integer) : real;
begin
    random := seed/65535 + 0.5;
    seed := (25173 * seed + 13849) mod 65536;
end;

```

```

function random_offset(var seed : integer; amplitude : integer) : integer;
{ if the amplitude is less than 0, both positive and negative
  offsets are allowed. If the amplitude is greater than 0,
  only positive offsets are allowed. }

begin
  if amplitude < 0 then
    begin
      random_offset := trunc((seed/65535)*2*amplitude);
      seed := (25173 * seed + 13849) mod 65536;
    end
  else
    begin
      random_offset := trunc((seed/65535 + 0.5)*amplitude);
      seed := (25173 * seed + 13849) mod 65536;
    end
  end;
end;

function reasonable_line_number (line_number : integer) : boolean;
{ this boolean function tests the reasonableness of a given line_number
  for manual control input }
begin
  if ((line_number mod 100) >= 1) and ((line_number mod 100) <= number_of_lines) and
    ((line_number div 100) >= 1) and ((line_number div 100) <= number_of_blocks) then

    reasonable_line_number := true ;
  else
    reasonable_line_number := false;
  end;
end;

procedure display_section(block_number, line_number : integer);
{ this procedure displays the section currently being executed }

begin
  case line_number of
    1: begin
      writeln;
      writeln;
      writeln('block number is ', block_number);
      writeln;
      writeln('*****');
      writeln('          section 1');
      writeln('*****');
      writeln;
    end;
    22: begin
      writeln;
      writeln;
      writeln('block number is ', block_number);
      writeln;
      writeln('*****');
      writeln('          section 2');
      writeln('*****');
      writeln;
    end;
    41: begin
      writeln;
      writeln;
      writeln('block number is ', block_number);
      writeln;
      writeln('*****');
      writeln('          section 3');
      writeln('*****');
    end;
  end;
end;

```

```

        writeln;
    end;
end; {case}
end;

```

```

procedure display_computation(block_number, line_number : integer);
{ this procedure displays the PASCAL program line currently being executed }
var
    temp, templ : integer;

begin
    writeln;
    writeln;
    writeln('block number is ', block_number);
    writeln('line number is ', line_number);
    writeln;
    case line_number of
        1: writeln(' {initialize} ');
        2: begin
            temp := (block_number - 1)*100 + 3;
            writeln(' if scc = ',temp,' then');
            end;
        3: begin
            temp := block_number*100 + 1;
            writeln(' scc := ',temp,'');
            end;
        4: begin
            writeln('else');
            writeln(' goto 100;');
            end;
        5: begin
            writeln(' {vote on values}');
            writeln('value_number := 0;');
            end;
        6: begin
            writeln('while value_number < number_of_values do');
            writeln(' begin');
            end;
        7: writeln(' value_number := value_number + 1;');
        8: begin
            write(' if values [1,copy 1,value_number] = ');
            writeln('values [1,copy 2, value_number] ');
            writeln(' then');
            writeln(' begin');
            end;
        9: begin
            write(' values [0,copy 1,value_number] := ');
            writeln('values [1,copy 1,value_number];');
            end;
        10: begin
            write(' values [0,copy 2,value_number] := ');
            writeln('values [1,copy 1,value_number];');
            end;
        11: begin
            write(' values [0,copy 3,value_number] := ');
            writeln('values [1,copy 1,value_number];');
            writeln(' next;');
            writeln('end;');
            end;
        12: begin
            write(' if values [1,copy 2,value_number] = ');
            writeln('values [1,copy 3,value_number]');
            writeln(' then ');
            writeln(' begin');
            end;
        13: begin

```

```

        write('      values [0,copy 1,value_number] := ');
        writeln('values [1,copy 2,value_number];');
    end;
14: begin
        write('      values [0,copy 2,value_number] := ');
        writeln('values [1,copy 2,value_number];');
    end;
15: begin
        write('      values [0,copy 3,value_number] := ');
        writeln('values [1,copy 2,value_number];');
        writeln('      next;');
        writeln('    end;');
    end;
16: begin
        write('      if values [1,copy 1,value_number] = ');
        writeln('values [1,copy 3,value_number] ');
        writeln('      then');
        writeln('        begin');
    end;
17: begin
        write('      values [0,copy 1,value_number] := ');
        writeln('values [1,copy 1,value_number];');
    end;
18: begin
        write('      values [0,copy 2,value_number] := ');
        writeln('values [1,copy 1,value_number];');
    end;
19: begin
        write('      values [0,copy 3,value_number] := ');
        writeln('values [1,copy 1,value_number];');
        writeln('      next;');
        writeln('    end;');
        writeln('end;');
    end;
20: writeln('  temp_version := 0;');
21: writeln('  number_of_computations := 0;');
22: begin
        writeln('    {calculate values}');
        temp := block_number*100 + 1;
        writeln('if scc := ',temp,' then');
    end;
23: begin
        temp := block_number*100 + 2;
        writeln('scc := ',temp,'');
    end;
24: begin
        writeln('else');
        writeln('  goto 100');
    end;
25: writeln('  repeat');
26: begin
        temp := (temp_version + 1) mod 2;
        writeln('    temp_version := ',temp,' ');
    end;
27: writeln('  temp [temp_version] := 0;');
28: writeln('  number_of_computations := number_of_computations + 1;');
29: writeln('  value_number := 0;');
30: writeln('  while value_number < number_of_values do');
31: begin
        writeln('    begin');
        writeln('      value_number := value_number + 1;');
    end;
32: begin
        write('  temp [temp_version] := temp [temp_version] ');
        writeln('+ values [0,copy 1,value_number];');
        writeln('end;');
    end;
33: begin

```

```

        write('  temp [temp_version] := temp [temp_version] ');
        writeln('div number_of_values;');
    end;
34: writeln('  temp [temp_version] := temp [temp_version] - 10;');
35: writeln('  temp [temp_version] := temp [temp_version] - 5;');
36: writeln('  temp [temp_version] := temp [temp_version] + block_number;');
37: writeln('  temp [temp_version] := temp [temp_version] + 15;');
38: begin
        write('    until (number_of_computations > 1) and ');
        write('(temp [0] = temp [1]);');
    end;
39: begin
        temp := block_number * 101 - 5;
        temp1 := block_number * 101 + 5;
        writeln('  {check resonableness} ');
        write('if (temp [temp_version] < ',temp,',) or ');
        writeln('(temp [temp_version] > ',temp1,',) then ');
    end;
40: writeln('  goto 100;');
41: begin
        writeln('  {assign values} ');
        temp := block_number*100 + 2;
        writeln('if scc = ', temp, ' then');
    end;
42: begin;
        temp := block_number*100 + 3;
        writeln('scc := ',temp,');
    end;
43: begin
        writeln('else');
        writeln('  goto 100;');
    end;
44: writeln('  value_number := 0;');
45: begin
        writeln('  while value_number < number_of_values do');
        writeln('    begin ');
    end;
46: writeln('  value_number := value_number + 1;');
47: writeln('  copy_number := 0; ');
48: begin
        writeln('  while copy_number < number_of_copies do ');
        writeln('    begin ');
    end;
49: writeln('  copy_number := copy_number + 1;');
50: begin
        writeln('    values [1,copy_number,value_number] := temp [1];');
        writeln('  end;');
        writeln('end;');
    end;
51: begin
        writeln('  {scc check}');
        temp := block_number*100 + 3;
        writeln('(if scc <> ',temp,',) then');
    end;
52: writeln('  goto 100; ');
53: begin
        writeln('  { reasonableness check }');
        write('if (value_number <> number_of_values) or ');
        writeln('(copy_number <> number_of_copies) or ');
        writeln('(temp [0] <> temp [1]) then');
    end;
54: writeln('  goto 100; ');

    otherwise writeln('ERROR : unhandled case, display computation');
end; {case}
writeln;
writeln;
end; {display computation}

```

```

procedure display_data;
{ this procedure displays all the data values of the simulated
  recovery block }
begin
  writeln;
  writeln;
  for i3 := 1 to number_of_values do
    for i2 := 1 to number_of_copies do
      for il := 0 to 1 do
        writeln('values [',il:2,', ',i2:2,', ',i3:2,'] = ',
          values [il,i2,i3,block_number]);
        writeln('temp [0] = ', temp [0]);
        writeln('temp [1] = ', temp [1]);
        writeln('scc = ', scc);
        writeln('number_of_computations = ', number_of_computations);
        writeln('temp_version = ', temp_version);
        writeln('block_number = ', block_number);
        writeln('value_number = ', value_number);
        writeln('copy_number = ', copy_number);
      end;
    end;
  end;

procedure change_data;
{ this procedure allows the user to manually change any data used by the
  recovery block }
label
  200;
var
  ch : char;
  temp_value : integer;

begin
  write('change values ? (y/n/q) ');
  readln(ch);
  if ch = 'q' then
    goto 200;
  if ch = 'y' then
    for i3 := 1 to number_of_values do
      for i2 := 1 to number_of_copies do
        for il := 0 to 1 do
          begin
            write('change values [',il,', ',i2,', ',i3,'] ? (y/n/q) ');
            readln(ch);
            if ch = 'q' then
              goto 200;
            if ch = 'y' then
              begin
                write('values [',il,', ',i2,', ',i3,'] = ');
                readln(temp_value);
                values [il,i2,i3,block_number] := temp_value;
              end;
            end;
          end;
        end;
      end;
    end;

  write('change temp [0] ? (y/n/q) ');
  readln(ch);
  if ch = 'q' then
    goto 200;
  if ch = 'y' then
    begin
      write('temp [0] = ');
      readln(temp_value);
      temp [0] := temp_value;
    end;

  write('change temp [1] ? (y/n/q) ');

```

```

readln(ch);
if ch = 'q' then
  goto 200;
if ch = 'y' then
  begin
    write('temp [1] = ');
    readln(temp_value);
    temp [1] := temp_value;
  end;

write('change scc ? (y/n/q) ');
readln(ch);
if ch = 'q' then
  goto 200;
if ch = 'y' then
  begin
    write('scc = ');
    readln(scc);
  end;

write('change number_of_computations ? (y/n/q) ');
readln(ch);
if ch = 'q' then
  goto 200;
if ch = 'y' then
  begin
    write('number_of_computations = ');
    readln(number_of_computations);
  end;

write('change temp_version ? (y/n/q) ');
readln(ch);
if ch = 'q' then
  goto 200;
if ch = 'y' then
  begin
    write('temp_version = ');
    readln(temp_version);
  end;

write('change block_number ? (y/n/q) ');
readln(ch);
if ch = 'q' then
  goto 200;
if ch = 'y' then
  begin
    write('block_number = ');
    readln(block_number);
  end;

write('change value_number ? (y/n/q) ');
readln(ch);
if ch = 'q' then
  goto 200;
if ch = 'y' then
  begin
    write('value_number = ');
    readln(value_number);
  end;

write('change copy_number ? (y/n/q) ');
readln(ch);
if ch = 'q' then
  goto 200;
if ch = 'y' then
  begin
    write('copy_number = ');
    readln(copy_number);
  end;

```

```

        end;
    200:
end;

procedure change_display_options;
{ this procedure allows the user to change the display options }
label
    200;
var
    ch : char;
begin
    write('display lines ? (y/n/q) ');
    readln(ch);
    if ch = 'q' then
        goto 200;
    if ch = 'y' then
        p.display.lines := true
    else
        p.display.lines := false;

    write('display data ? (y/n/q) ');
    readln(ch);
    if ch = 'q' then
        goto 200;
    if ch = 'y' then
        p.display.data := true
    else
        p.display.data := false;

    write('display errors detected ? (y/n/q) ');
    readln(ch);
    if ch = 'q' then
        goto 200;
    if ch = 'y' then
        p.display.errors_detected := true
    else
        p.display.errors_detected := false;

    write('display errors injected ? (y/n/q) ');
    readln(ch);
    if ch = 'q' then
        goto 200;
    if ch = 'y' then
        p.display.errors_injected := true
    else
        p.display.errors_injected := false;

    write('display section ? (y/n/q) ');
    readln(ch);
    if ch = 'q' then
        goto 200;
    if ch = 'y' then
        p.display.section := true
    else
        p.display.section := false;

    write('display data after upset injection ? (y/n/q) ');
    readln(ch);
    if ch = 'q' then
        goto 200;
    if ch = 'y' then
        p.display.data_at_injection := true
    else
        p.display.data_at_injection := false;

```



```

write('display data after upset detection ? (y/n/q) ');
readln(ch);
if ch = 'q' then
  goto 200;
if ch = 'y' then
  p.display.data_at_detection := true
else
  p.display.data_at_detection := false;

write('display data at start of block ? (y/n/q) ');
readln(ch);
if ch = 'q' then
  goto 200;
if ch = 'y' then
  p.display.data_at_start := true
else
  p.display.data_at_start := false;

write('display data at end of block ? (y/n/q) ');
readln(ch);
if ch = 'q' then
  goto 200;
if ch = 'y' then
  p.display.data_at_end := true
else
  p.display.data_at_end := false;

200:
end;

```

```

procedure free_run;
{ this procedure allows the user to specify a free run (otherwise,
  the program is halted after each instruction). There are two
  free run modes : break on line and break on error }
var
  ch : char;
begin
  sim_data.free_run := true;
  write('break on line ? (y/n) ');
  readln(ch);
  if ch = 'y' then
    begin
      sim_data.line_break := true;
      writeln('sequence numbers have the form ');
      writeln('  (block number) * 100 + line number');
      writeln('for example, to jump to line number 5 in block 5,');
      writeln('use sequence number 505');
      writeln;
      repeat
        write('break at which sequence number ? ');
        write('( integer : sequence number > 101 ) : ');
        readln(sim_data.break_point_line);
      until reasonable_line_number(sim_data.break_point_line);
    end
  else
    begin
      write('break on error ? (y/n) ');
      readln(ch);
      if ch = 'y' then
        sim_data.line_break := false;
    end;
  end;
end;

```

```

procedure display_error (var sim_data : control);
{ this procedure displays the recovery profile }
begin
  with s.recovery do
    begin
      {display error}
      writeln;
      writeln;
      writeln(sim_data.error_message);
      {display profile}
      writeln;
      writeln;
      writeln(
        error recovery profile
      );
      writeln(
        redundant computation :
        ,redundant_comp);
      writeln(
        redundant storage :
        ,redundant_storage);
      writeln(
        memory coding :
        , memory_coding);
      writeln(
        sequence control codes :
        , scc);
      writeln(
        wait loop check :
        , wait_loop);
      writeln(
        reasonableness check :
        , reasonableness_check);
    end;
  end;
end;

procedure recovery_profile (var s : stats;var sim_data : control; line : integer);
{ this procedure maintains the recovery profile statistics, and displays errors
  detected, if opted }
begin
  sim_data.error := false;
  with s.recovery do
    case line of
      4: begin
          sim_data.error := true;
          sim_data.error_message :=
            sequence error, found on line 4, by sequence control codes;
          scc := scc + 1;
        end;

      8: if values [1,3,value_number,block_number] <>
          values [1,1,value_number,block_number] then
          begin
            sim_data.error := true;
            sim_data.error_message :=
              data error, found on line 8, by redundant storage;
            redundant_storage := redundant_storage + 1;
          end;

      12: if values [1,1,value_number,block_number] <>
          values [1,2,value_number,block_number] then
          begin
            sim_data.error := true;
            sim_data.error_message :=
              data error, found on line 12, by redundant storage;
            redundant_storage := redundant_storage + 1;
          end;

      16: if values [1,1,value_number,block_number] <>
          values [1,2,value_number,block_number] then
          begin
            sim_data.error := true;
            sim_data.error_message :=
              data error, found on line 16, by redundant storage;
            redundant_storage := redundant_storage + 1;
          end;

      21: sim_data.number_of_computations := 0;

      24: begin

```

```

        sim_data.error := true;
        sim_data.error_message :=
            'sequence error, found on line 24, by sequence control codes';
        scc := scc + 1;
    end;

28: sim_data.number_of_computations := sim_data.number_of_computations + 1;

38: if sim_data.number_of_computations > 2 then
    begin
        sim_data.error := true;
        sim_data.error_message :=
            'data error, found on line 38, by redundant computations';
        redundant_comp := redundant_comp +
            (sim_data.number_of_computations - 1) div 2;
    end;

40: begin
    sim_data.error := true;
    sim_data.error_message :=
        'data error, found on line 40, by reasonableness checks';
    reasonableness_check := reasonableness_check + 1;
end;

43: begin
    sim_data.error := true;
    sim_data.error_message :=
        'sequence error, found on line 43, by sequence control codes';
    scc := scc + 1;
end;

52: begin
    sim_data.error := true;
    sim_data.error_message :=
        'sequence error, found on line 52, by sequence control codes';
    scc := scc + 1;
end;

54: begin
    sim_data.error := true;
    sim_data.error_message :=
        'data error, found on line 54, by reasonableness checks';
    reasonableness_check := reasonableness_check + 1;
end;

end; {case}

if sim_data.error then
    sim_data.recovery_line := line;

if sim_data.error and p.display.errors_detected then
    display_error(sim_data);

if sim_data.error and p.display.data_at_detection then
    display_data;

end;

procedure recovery_example(var block_number : integer);

label
    1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
    21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,
    38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,99,100;

```

```

const
  scc1 = 4;
  scc2 = 24;
  scc3 = 43;
  scc4 = 52;
  real = 40;
  rea2 = 54;

procedure jump (jump_to : integer);
  { this procedure allows PASCAL to make arbitrary program jumps. Note that
    x := 5;
    goto x;
    is illegal }

begin
  case jump_to of
    1: goto 1;
    2: goto 2;
    3: goto 3;
    4: goto 4;
    5: goto 5;
    6: goto 6;
    7: goto 7;
    8: goto 8;
    9: goto 9;
    10: goto 10;
    11: goto 11;
    12: goto 12;
    13: goto 13;
    14: goto 14;
    15: goto 15;
    16: goto 16;
    17: goto 17;
    18: goto 18;
    19: goto 19;
    20: goto 20;
    21: goto 21;
    22: goto 22;
    23: goto 23;
    24: goto 24;
    25: goto 25;
    26: goto 26;
    27: goto 27;
    28: goto 28;
    29: goto 29;
    30: goto 30;
    31: goto 31;
    32: goto 32;
    33: goto 33;
    34: goto 34;
    35: goto 35;
    36: goto 36;
    37: goto 37;
    38: goto 38;
    39: goto 39;
    40: goto 40;
    41: goto 41;
    42: goto 42;
    43: goto 43;
    44: goto 44;
    45: goto 45;
    46: goto 46;
    47: goto 47;
    48: goto 48;
    49: goto 49;
    50: goto 50;
    51: goto 51;
  end case;
end;

```

```

52: goto 52;
53: goto 53;
54: goto 54;
99: goto 99;
end;
writeln("ERROR : illegal jump ");
end;

```

```

procedure processor_memory_upset;
{ this procedure is used to automatically inject processor memory errors }

var
  offset_amplitude : integer;
  upset_version, upset_copy, upset_value : integer;

begin
  { processor memory upset }

  if p.display.errors_injected then
  begin
    writeln;
    writeln;
    writeln("*****");
    writeln("      processor memory upset");
    writeln("*****");
    sim_data.error := true;
    writeln;
    writeln;
  end;

  s.upset.processor_memory := s.upset.processor_memory + 1;

  case trunc(random(seed)*9 + 1) of

    1 : begin
        { values upset }
        offset_amplitude := -10000;
        upset_version := random_offset(seed1,1);
        upset_copy := random_offset(seed1,number_of_copies) + 1;
        upset_value := random_offset(seed1,number_of_values) + 1;
        values[upset_version,upset_copy,upset_value,block_number] :=
          values[upset_version,upset_copy,upset_value,block_number] +
            random_offset(seed1,offset_amplitude);
        if p.display.errors_injected then
          writeln(" values[",upset_version,"",upset_copy,"",
            upset_value,"] = ",
            values[upset_version,upset_copy,upset_value,block_number] );
        end;

    2 : begin
        { values upset }
        offset_amplitude := -10000;
        upset_version := random_offset(seed1,1);
        upset_copy := random_offset(seed1,number_of_copies) + 1;
        upset_value := random_offset(seed1,number_of_values) + 1;
        values[upset_version,upset_copy,upset_value,block_number] :=
          values[upset_version,upset_copy,upset_value,block_number] +
            random_offset(seed1,offset_amplitude);
        if p.display.errors_injected then
          writeln(" values[",upset_version,"",upset_copy,"",
            upset_value,"] = ",
            values[upset_version,upset_copy,upset_value,block_number] );
        end;

    { upset non-critical variable }
  end;

```

```

3 : begin
    offset_amplitude := -10000;
    temp[0] := temp[0] + random_offset(seed1,offset_amplitude);
    if p.display.errors_injected then
        writeln(' temp[0] = ', temp[0]);
    end;
4 : begin
    offset_amplitude := -10000;
    temp[1] := temp[1] + random_offset(seed1,offset_amplitude);
    if p.display.errors_injected then
        writeln(' temp[1] = ', temp[1]);
    end;
5 : begin
    value_number := random_offset(seed1,number_of_values) + 1;
    if p.display.errors_injected then
        writeln(' value_number = ', value_number);
    end;
6 : begin
    copy_number := random_offset(seed1,number_of_copies) + 1;
    if p.display.errors_injected then
        writeln(' copy_number = ', copy_number);
    end;
7 : begin
    offset_amplitude := -10000;
    scc := scc + random_offset(seed1,offset_amplitude);
    if p.display.errors_injected then
        writeln(' scc = ', scc);
    end;
8 : begin
    offset_amplitude := -10000;
    number_of_computations := number_of_computations
        + random_offset(seed1,offset_amplitude);
    if p.display.errors_injected then
        writeln('number_of_computations = ',number_of_computations);
    end;
9 : begin
    temp_version := random_offset(seed1,number_of_blocks);
    if p.display.errors_injected then
        writeln('temp_version = ',temp_version);
    end;
    otherwise writeln('ERROR : illegal non-critical variable upset');
end; {case}

if p.display.data_at_injection then
    display_data;

end;

```

```

procedure sequence_upset;
{ this procedure is used to automatically inject sequence upsets }
var
    upset_block, upset_line : integer;

begin
    {sequence error}

    if p.display.errors_injected then
        begin
            writeln;
            writeln;
            writeln('*****');
            writeln('      sequence upset upset');
            writeln('*****');
            sim_data.error := true;
            writeln;
            writeln;
        end;
    end;

```

```

end;

s.upset.sequence := s.upset.sequence + 1;
upset_block := random_offset(seed1,number_of_blocks) + 1;
upset_line := random_offset(seed1,number_of_lines) + 1;
sim_data.skip_to_line := 100*upset_block + upset_line;

if p.display.errors_injected then
  writeln(' sequence skiped to line ',sim_data.skip_to_line);

if upset_block <> block_number then
  begin
    sim_data.skip_block := true;
    goto 99;
  end
else
  jump(upset_line);
end;

```

```

procedure memory_upset;
{ this procedure is used to automatically inject main memory upsets }
var
  offset_amplitude, upset_block : integer;
  upset_version, upset_copy, upset_value : integer;

begin
  {memory error}
  if p.display.errors_injected then
    begin
      writeln;
      writeln;
      writeln('*****');
      writeln('      memory upset');
      writeln('*****');
      sim_data.error := true;
      writeln;
      writeln;
    end;

  s.upset.memory := s.upset.memory + 1;
  if random(seed) < p.critical_var then
    begin
      { upset critical variable }
      { chose upset block }
      upset_block := random_offset(seed1,number_of_blocks) + 1;

      { values upset}
      offset_amplitude := -10000;
      upset_version := random_offset(seed1,2);
      upset_copy := random_offset(seed1,number_of_copies) + 1;
      upset_value := random_offset(seed1,number_of_values) + 1;
      values[upset_version,upset_copy,upset_value,upset_block] :=
        values[upset_version,upset_copy,upset_value,upset_block] +
        random_offset(seed1,offset_amplitude);
      if p.display.errors_injected then
        writeln(' values[',upset_version,',',upset_copy,',',
          upset_value,',',upset_block,'] = ',
          values[upset_version,upset_copy,upset_value,upset_block] );
    end
  else
    begin
      { upset non-critical variable }
      case trunc(random(seed)*7 + 1) of
      1: begin
          offset_amplitude := -10000;
          temp[0] := temp[0] + random_offset(seed1,offset_amplitude);
        end;
      end;
    end;
  end;

```

```

        if p.display.errors_injected then
            writeln(' temp[0] = ', temp[0]);
        end;
    2: begin
        offset_amplitude := -10000;
        temp[1] := temp[1] + random_offset(seed1,offset_amplitude);
        if p.display.errors_injected then
            writeln(' temp[1] = ', temp[1]);
        end;
    3: begin
        value_number := random_offset(seed1,number_of_values) + 1;
        if p.display.errors_injected then
            writeln(' value_number = ', value_number);
        end;
    4: begin
        copy_number := random_offset(seed1,number_of_copies) + 1;
        if p.display.errors_injected then
            writeln(' copy_number = ', copy_number);
        end;
    5: begin
        offset_amplitude := -10000;
        scc := scc + random_offset(seed1,offset_amplitude);
        if p.display.errors_injected then
            writeln(' scc = ', scc);
        end;
    6: begin
        offset_amplitude := -10000;
        number_of_computations := number_of_computations
            + random_offset(seed1,offset_amplitude);
        if p.display.errors_injected then
            writeln('number_of_computations = ',number_of_computations);
        end;
    7: begin
        temp_version := random_offset(seed1,number_of_blocks);
        if p.display.errors_injected then
            writeln('temp_version = ',temp_version);
        end;
        otherwise writeln('ERROR : illegal non-critical variable upset');
    end; {case}
end;

if p.display.data_at_injection then
    display_data;

end;

```

```

procedure upset(var p : parameters);

```

```

    { given that an upset has occurred, this procedure detirmines where
      it has occurred }

```

```

begin
    {inject error}
    if (p.n = 1) and (random(seed) < p.wait_loop) then
        begin
            {upset in wait loop}
            s.upset.wait_loop := s.upset.wait_loop +1;
        end
    else
        if random(seed) < p.processor_sus then
            begin
                { processor error }
                if random(seed) < p.neu_map.memory then
                    processor_memory_upset
                else
                    sequence_upset;
            end
        end
    end
end

```



```

        else
            memory_upset;
        end;

procedure inject_upset(var p : parameters; line : integer);

    { this procedure determines when an upset occurs, using the
      different upset rates }

    const
        start_of_init = 1;
        end_of_init = 21;
        start_of_calc = 22;
        end_of_calc = 38;
        start_of_commit = 39;
        end_of_commit = 54;

    begin
        if (line >= start_of_init) and (line <= end_of_init) then
            begin
                if trunc(random(seed) * p.factor1) = 1 then
                    upset(p);
                end
            end
        else
            if (line >= start_of_calc) and (line <= end_of_calc) then
                begin
                    if trunc(random(seed) * p.factor2) = 1 then
                        upset(p);
                    end
                end
            else
                if (line >= start_of_commit) and (line <= end_of_commit) then
                    begin
                        if trunc(random(seed) * p.factor3) = 1 then
                            upset(p);
                        end
                    end
                end
            end;

procedure free_run_mode(line : integer);
begin
    inject_upset(p,line);
    if sim_data.line_break then
        begin
            if block_number * 100 + line =
                sim_data.break_point_line then
                begin
                    sim_data.line_break := false;
                    sim_data.free_run := false
                end;
            end
        end
    else
        if sim_data.error then
            begin
                sim_data.free_run := false;
                sim_data.error := false;
            end;

        if p.display.section then
            display_section(block_number,line);

        if p.display.lines then
            display_computation(block_number,line);
    end;

```

```

procedure change_sequence;
begin
  writeln('sequence numnber have the form ');
  writeln(' (block number) * 100 + line number');
  writeln('for example, to jump to line number 5 in block 5,');
  writeln('use sequence number 505');
  writeln;
  write('input next sequence number (integer) : ');
  readln(sim_data.skip_to_line);
  if (sim_data.skip_to_line div 100) <> block_number then
    begin
      sim_data.skip_block := true;
      goto 99;
    end
  else
    begin
      jump(sim_data.skip_to_line mod 100);
    end;
end;

procedure recovery_evaluation;
var
  b,v : integer;

begin
  { the test - at least 2 out of 3 critical values must be correct }
  b := block_number;
  v := block_number * 101;
  if
    ((values[1,1,1,b] = v ) and
     (values[1,2,1,b] = v )) or
    ((values[1,1,1,b] = v ) and
     (values[1,3,1,b] = v )) or
    ((values[1,2,1,b] = v ) and
     (values[1,3,1,b] = v)))
    and
    ((values[1,1,2,b] = v ) and
     (values[1,2,2,b] = v )) or
    ((values[1,1,2,b] = v ) and
     (values[1,3,2,b] = v )) or
    ((values[1,2,2,b] = v ) and
     (values[1,3,2,b] = v)))
    and
    ((values[1,1,3,b] = v ) and
     (values[1,2,3,b] = v )) or
    ((values[1,1,3,b] = v ) and
     (values[1,3,3,b] = v )) or
    ((values[1,2,3,b] = v ) and
     (values[1,3,3,b] = v)))
  then
    begin
      writeln;
      writeln;
      writeln('recovery is successful');
    end
  else
    begin
      writeln;
      writeln;
      writeln('recovery has failed');
    end;
end;
end;

```

```

procedure error_sim( line : integer);
label
  200;
var
  ch : char;
  automated : boolean;

begin
  automated := false;
  recovery_profile(s,sim_data,line);
  if sim_data.skip_block then
    begin
      if line = 1 then
        begin
          sim_data.skip_block := false;
          jump( sim_data.skip_to_line mod 100);
        end
      else
        writeln('ERROR : illegal skip_block in procedure error_sim');
      end;
    if automated then
      begin
        inject_upset(p,line);
        if sim_data.error then
          display_error(sim_data);
        end
      else { manual error simulation }
        if sim_data.free_run then
          free_run_mode(line)
        else
          begin
            if p.display.lines then
              display_computation(block_number,line);

            if p.display.data then
              display_data;

            write('examine variables ? (y/n/q) ');
            readln(ch);
            if ch = 'q' then
              goto 200;
            if ch = 'y' then
              display_data;

            write('change variables ? (y/n/q) ');
            readln(ch);
            if ch = 'q' then
              goto 200;
            if ch = 'y' then
              change_data;

            write('change sequence ? (y/n/q) ');
            readln(ch);
            if ch = 'q' then
              goto 200;
            if ch = 'y' then
              change_sequence;

            write('free run ? (y/n/q) ');
            readln(ch);
            if ch = 'q' then
              goto 200;
            if ch = 'y' then
              free_run;

            if p.test_mode then
              begin
                write('change display options ? (y/n/q) ');

```

```

        readln(ch);
        if ch = 'q' then
            goto 200;
        if ch = 'y' then
            change_display_options;
        end;

    200:
        end
    end;

begin
    if p.display.data_at_start then
        display_data;

1:      {initialize}
2:      if scc = (block_number-1)*100 + 3 then
3:          begin
                scc := block_number*100 + 1;
            end
        else
            begin
4:          goto 100;
            end;

            {vote on values}
5:      value_number := 0;
6:      while value_number < number_of_values do
7:          begin
                value_number := value_number + 1;
8:          if values [1,1,value_number,block_number] =
                values [1,2,value_number,block_number]
            then
9:              begin
                    values [0,1,value_number,block_number] :=
                        values [1,1,value_number,block_number];
10:             values [0,2,value_number,block_number] :=
                values [1,1,value_number,block_number];
11:             values [0,3,value_number,block_number] :=
                values [1,1,value_number,block_number];
                next;
            end;
12:         if values [1,2,value_number,block_number] =
                values [1,3,value_number,block_number]
            then
13:             begin
                    values [0,1,value_number,block_number] :=
                        values [1,2,value_number,block_number];
14:             values [0,2,value_number,block_number] :=
                values [1,2,value_number,block_number];
15:             values [0,3,value_number,block_number] :=
                values [1,2,value_number,block_number];
                next;
            end;
16:         if values [1,1,value_number,block_number] =
                values [1,3,value_number,block_number]
            then
17:             begin
                    values [0,1,value_number,block_number] :=

```

```

                values [1,1,value_number,block_number];
18:                values [0,2,value_number,block_number] :=
                    values [1,1,value_number,block_number];
19:                values [0,3,value_number,block_number] :=
                    values [1,1,value_number,block_number];
                next;
            end;
        end;
20:    temp_version := 0;
21:    number_of_computations := 0;
        {initialization complete}
        {calculate values}
22:    if scc = block_number*100 + 1 then
        begin
23:        scc := block_number*100 + 2;
        end
    else
        begin
24:            goto 100;
        end;
25:    repeat
26:        temp_version := (temp_version + 1) mod 2;
27:        temp [temp_version] := 0;
28:        number_of_computations := number_of_computations + 1;
29:        value_number := 0;
30:        while value_number < number_of_values do
        begin
31:            value_number := value_number + 1;
32:            temp [temp_version] := temp [temp_version]
                + values [0,1,value_number,block_number];
            error_sim(32)
        end;
33:        temp [temp_version] := temp [temp_version] div number_of_values;
34:        temp [temp_version] := temp [temp_version] - 10;
35:        temp [temp_version] := temp [temp_version] - 5;
36:        temp [temp_version] := temp [temp_version] + block_number;
37:        temp [temp_version] := temp [temp_version] + 15;
38:    until (number_of_computations > 1) and (temp [0] = temp [1]);
        {check resonableness}
39:    if (temp [temp_version] < block_number * 101 - 5)
        or (temp [temp_version] > block_number * 101 + 5) then
        begin
40:            goto 100;
        end;
        {assign values}
41:    if scc = block_number*100 + 2 then
        begin
42:        scc := block_number*100 + 3;
        end
    else
        begin
43:            goto 100;
        end;
44:    value_number := 0;
45:    while value_number < number_of_values do

```

```

begin
46:     value_number := value_number + 1;          error_sim(45)
47:     copy_number := 0;                          error_sim(46)
48:     while copy_number < number_of_copies do    error_sim(47)
        begin
49:         copy_number := copy_number + 1;        error_sim(48)
50:         values [1,copy_number,value_number,block_number] := temp [1]; error_sim(49)
                                                error_sim(50)
        end;
    end;

    { scc check }
51: if (scc <> block_number*100 + 3)
    then
        begin
                                                error_sim(51)
52:         goto 100;                               error_sim(52)
        end;

    { reasonableness check }
53: if (value_number <> number_of_values) or (copy_number <> number_of_copies)
    or (temp [0] <> temp [1]) then
        begin
                                                error_sim(53)
54:         goto 100;                               error_sim(54)
        end;

    if p.display.data_at_end then
        display_data;

    recovery_evaluation;

    goto 99;

100: writeln('error detected, restarting computation');

    if (sim_data.recovery_line = scc1) or (sim_data.recovery_line = scc2) or
    (sim_data.recovery_line = scc3) or (sim_data.recovery_line = scc4) then
        { sequence upset detected }

        begin
            if reasonable_line_number(scc) then
                begin
                    {restart at the old scc}
                    block_number := scc div 100 ;
                    case (scc mod 100) of
                        1: sim_data.skip_to_line := scc1 + 1;
                        2: sim_data.skip_to_line := scc2 + 1;
                        3: sim_data.skip_to_line := scc3 + 1;
                    end; {case}

                    writeln('*****');
                    writeln('  recovery block 1 - goto ',sim_data.skip_to_line);
                    writeln('          scc = ',scc);
                    writeln('*****');

                    jump (sim_data.skip_to_line);
                end
            else
                begin
                    { scc upset - reset scc and restart at scc }
                    if (sim_data.recovery_line = scc1) then
                        begin
                            scc := (block_number - 1) * 100 + 3;

                            writeln('*****');
                            writeln('  recovery block 2 - goto ',scc1 - 2);

```

```

        writeln('          scc = ',scc);
        writeln('*****');

        jump (scc1 - 2);
    end;

    if (sim_data.recovery_line = scc2) then
    begin
        scc := block_number * 100 + 1;

        writeln('*****');
        writeln(' recovery block 3 - goto ',scc2 - 2);
        writeln('          scc = ',scc);
        writeln('*****');

        jump (scc2 - 2);
    end;

    if (sim_data.recovery_line = scc3) then
    begin
        scc := block_number * 100 + 2;

        writeln('*****');
        writeln(' recovery block 4 - goto ',scc3 - 2);
        writeln('          scc = ',scc);
        writeln('*****');

        jump (scc3 - 2);
    end;

    if (sim_data.recovery_line = scc4) then
    begin
        scc := block_number * 100 + 3;

        writeln('*****');
        writeln(' recovery block 5 - goto ',scc4 - 1);
        writeln('          scc = ',scc);
        writeln('*****');

        jump (scc4 - 1);
    end;
end;

else

if (sim_data.recovery_line = real) then
{ reasonableness check }

begin
    if reasonable_line_number(scc) then
    begin
        {restart at the old block, line 1}
        block_number := scc div 100 ;
        sim_data.skip_to_line := scc1 + 1;

        writeln('*****');
        writeln(' recovery block 6 - goto ',sim_data.skip_to_line);
        writeln('          scc = ',scc);
        writeln('*****');

        jump (sim_data.skip_to_line);
    end
    else
    begin
        {restart at block 1, line 1}
        block_number := 0;
        initialize_stats(s);
    end;
end;

```

```

        init;

        writeln('*****');
        writeln(' recovery block 7 - restart');
        writeln('          scc = ',scc);
        writeln('*****');

    end;

end

else

if (sim_data.recovery_line = rea2) then
    { reasonableness check }

begin
    if reasonable_line_number(scc) then
        begin
            if (temp [0] <> temp [1]) then
                begin
                    {restart at current block, line 1}
                    scc := (block_number)*100 + 1;

                    writeln('*****');
                    writeln(' recovery block 8 - goto ',scc2 - 4);
                    writeln('          scc = ',scc);
                    writeln('*****');

                    jump (scc2 - 4);
                end
            else
                begin
                    {redo update}

                    writeln('*****');
                    writeln(' recovery block 9 - goto ',scc3 + 1);
                    writeln('          scc = ',scc);
                    writeln('*****');

                    jump (scc3 + 1);
                end
            end
        end
    else
        begin
            {restart at block 1, line 1}
            block_number := 0;
            initialize_stats(s);
            init;

            writeln('*****');
            writeln(' recovery block 10 - restart');
            writeln('          scc = ',scc);
            writeln('*****');

        end
    end;

end

else

if reasonable_line_number(scc) then
    begin
        {restart at the old scc}
        block_number := scc div 100 ;
        case (scc mod 100) of
            1: sim_data.skip_to_line := scc1 + 1;
            2: sim_data.skip_to_line := scc2 + 1;
            3: sim_data.skip_to_line := scc3 + 1;
        end; {case}
    end;
end;

```



```

        writeln('*****');
        writeln(' recovery block ll - goto ',sim_data.skip_to_line);
        writeln('      scc = ',scc);
        writeln('*****');

        jump (sim_data.skip_to_line);
    end;

```

99:

```

    end;

    {main body - simulation }

    begin
        while true do
            begin
                init;
                for block_number := 1 to number_of_blocks do
                    begin
                        recovery_example(block_number);
                        if sim_data.skip_block then
                            block_number := (sim_data.skip_to_line div 100) - 1;
                        end;
                    end;
                end;
            end;
        end;

        { main program body }
    begin
        initialize_parameters(p);
        initialize_stats(s);
        simulation(p, s);
    end.

```

GLOSSARY

This is a glossary of words used throughout this document

autonomous spacecraft maintenance (ASM):

The goal of ASM is to provide spacecraft the ability to function for a specified period of time without ground support. ASM requires autonomous health and maintenance, navigation and stationkeeping, mission sequencing, as well as autonomous hardware fault recovery.

action section:

A section of code where any action is committed and performed. In this section, critical variables are assigned or output operations are performed. The objective is to keep the action section as short as possible in order to minimize the probability of upset during the execution of this section.

atomic actions:

Atomic actions are traditionally used in data base concurrency control [9]. The objective of atomic actions is to make actions appear indivisible, that is, all other actions appear to have occurred either before or after an atomic transaction. Furthermore, atomic actions appear to completely happened (commit) or never happen (abort).

A process experiencing transient errors can be considered to be two processes, the actual process and the transient error process. The objective of using atomic actions in transient error recovery is to insure that all transient errors occur either before or

after an action but not during.

block:

To apply the proposed transient error recovery technique to a program, it must be divided into blocks, which are similar to a procedures. Each block is associated with either a critical variable or an output operation or both. Each block is divided into three idempotent sections : an initial section, a computation section, and an action section.

catastrophe:

An event which is the result of an upset, which a transient error recovery technique may not recover without re-initialization and restart, if at all. In terms of ASM, a catastrophic upset implies that autonomy may be compromised. In terms of mission success, the result of a catastrophic upset is undefined.

Catastrophes are divided into two categories : first-order catastrophes, which result from one upset and second-order catastrophes, which result from two upsets.

computation block:

To apply the proposed transient error recovery technique to a program, it must be divided into blocks, which are similar to a procedures. Each block is associated with either a critical variable or an output operation or both. Each block is divided into three idempotent sections : an initial section, a computation section, and an action section.

computation section:

The section of code where the actual computation takes place. All computations are performed using temporary variables to insure the idempotence of the section and to limit the possibility of error propagation. The computation is performed at least twice, until two results in a row agree. For time variant systems, results must agree within a pre-defined margin.

control variables:

Control variables are program variables used to control data access and program flow. Control variables include subscripts, counters, flags, sequence control codes, etc.

coverage:

Coverage is defined in reliability theory [8] as the conditional probability that a failure of a unit will be detected and appropriate recovery action will be performed given the occurrence of a fault and sufficient resources for recovery. Since the addressed phenomenon are transient in nature, no resources are required for transient error recovery. For transient errors, coverage is a metric of a systems' error detection and recovery action capability.

critical flip-flops:

Critical flip-flops are flip-flops within the processor and its supporting logic which can result in a catastrophe if upset, and are not covered by Software Implemented Transient Error Recovery. Since they are referred to as "flip-flops" (as opposed to "registers"), the implication is that they typically hold control information as opposed to value information, their criticalness has

a high duty-cycle. Thus, most upsets to critical flip-flops are catastrophic.

critical registers:

Critical registers are registers within the processor and its supporting logic which can result in a catastrophe if upset, and are not covered by Software Implemented Transient Error Recovery. Since they are referred to as "registers" (as opposed to "flip-flops"), the implication is that they typically hold value information as opposed to control information, their criticalness has a low duty-cycle. Thus, an upset during a non-critical period, such as the execution of a wait loop, is not catastrophic.

critical variables:

Critical variables are program variables which have a direct effect upon some output of the system. Non-critical variables have effect on the output of the system, but only through their effect upon critical variables. Another way of looking at this distinction is that if there were only a single copy of a critical variable, and if that copy were upset, the only possible course for recovery is re-initialization and restart. If a non-critical variable is upset, recovery can be achieved through re-execution of a section, so restart is not necessary. Since critical variables are actually stored in triplicate, recovery can be performed by voting on their value in the initialization section. Examples of critical variables in the Intelsat VI ACE are offset pointing values, system modes, and gains. Examples of non-critical variables are counters, flags, loop variables, etc.

error propagation control:

Error propagation control is an important problem in transient error recovery. If a value which has been upset is used in a calculation, all values which depend upon the upset value will be incorrect. Consequently, one upset value may result in several incorrect values, all of which must be corrected to successfully perform recovery.

fault-tolerance:

Fault-tolerance is the ability of a system to perform correctly in the presence of one or more hardware failures. Fault-tolerance techniques use some form of hardware redundancy to detect and replace failed units.

high-level errors:

The proposed transient error recovery methodology is capable of detecting high-level errors. If an error cannot be detected by redundancy at the high-level language level, it most likely cannot be corrected by the methodology. An example of a high-level error is an upset to a register in an ALU of a processor. Such an upset can be observed by examining the processor registers. An example of a low-level error, which the methodology cannot recover, is an upset to the master reset flip-flop. A high-level language could not observe such an error, since a master reset destroys all program state.

High-Level Language/Machine Language Model:

The High-Level Language/Machine Language Model abstracts errors from the Upset Mapping Model on bit level code (machine language) and data as bit upsets on high-level language and data. The Upset

Mapping Model is based on the observation that all results of the Upset Mapping Model can be produced using a high-level language and that Software Implemented Transient Error Recovery is only capable of recovering from errors that can be detected from a high-level.

idempotence:

A procedure is idempotent if the result of multiple applications is the same as the result of one application. An example of an idempotent procedure is : move platform to 135 degrees. An example of a non-idempotent procedure is : increment platform position + 15 degrees.

initialization section:

The section of code where the block initialization takes place. At the very least, critical variables are voted upon and local variables are initialized.

Intelsat VI ACE:

The Intelsat VI attitude control subsystem. This is the first major satellite project to address the issue of recovering from single event upsets by software techniques.

jump return to wait loop:

Jump return to wait loop is a method for detecting and recovering from sequencing errors which occur during the execution of a wait loop. The control program of a typical spacecraft sub-system will spend the majority of its run-time in a wait loop. Since wait loops are short, the number of mutations of the wait loop instructions that a single bit flip could cause is small. Here is an example of what could be done:

```

; program fragment
104  add r1, r2
105  jmp 107
106  jmp 200
107  continue...

; wait loop
200  ei
201  jmp 200
202  jmp 200

```

Suppose an SEU changed line 200 from enable interrupt to jmp 106. Under normal operation, it is impossible for the program to execute line 106, since it is intentionally by-passed by line 105. Consequently, embedding a jump return to the wait loop instruction would recover from such an upset.

Since the amount of time spent in the wait loop is large, this method is very effective in recovering from SEUs, but is ineffective in recovering from larger upsets since mutations of only one bit are considered.

machine code mis-interpretation:

Machine code mis-interpretation is caused from the ambiguity of stored programs, since machine language is context sensitive. For example, the machine code 10110001 may be interpreted as the instruction "inc R1", or as the data B1 hex. Consequently, if a sequence error occurs, the code intended to detect this error may be misinterpreted. This problem is solved with a process called "NOP buffering".

Example: instead of compiling

```

If scc <> 5 then
    goto error_recovery;

```

as


```

mov R1, scc
mov R1, 5
cmp
jne error_recovery

use

nop
nop
mov R1, scc
mov R1, 5
cmp
jne error_recovery

```

The extra NOPs with put the program on the right track.

main memory:

Main bank of volatile RAM memory.

multiple bit upsets (MBU) :

The Multiple Bit Upset Model abstracts transient errors as events in which one or more bits per word may be upset simultaneously. A sub-class of MBUs are single event upsets (SEUs), which occur, by definition, when only one bit per word is upset.

MBU Upset Rate:

The MBU upset rate refers to the frequency of upsets caused primarily by the electrostatic discharge problem. There is no accurate data available on the frequency characteristics of electrostatic discharges. Since it is known that SEUs are the dominant source of transient error in present spacecraft systems, the MBU rate was chosen so that the frequency of catastrophic upsets caused by MBUs is the same order of magnitude as the frequency of catastrophic upsets caused by SEUs. The rate used is 0.000001 upsets/(bit-day).

processor memory:

Volatile memory used by the processor, whether internal or external to the physical processor. In the context of the simulation program, a "processor memory error" refers to a processor memory error which can be observed by inspecting processor registers or the program counter.

reasonableness checking:

Reasonableness checking is a method of error detection. When a variable is known to have some range of correct values, then the actual value of the variable can be compared to this range to check the reasonableness of the value. If a discrepancy is found, an error has occurred. Reasonableness checking is extremely powerful when the range of correct values is small compared to the range of possible values.

This technique is especially useful for testing program control variables. For example, suppose we have the following code:

```
index := 0;
while index < 4 do
  begin
    index := index + 1;
    { etc. }
  end;
```

We know that at the end of this block, index must have the value 4. We also know that during the execution of this block, index cannot have a value of less than 0 or greater than 4. Although this is obvious, this example shows that the implementation of reasonableness checking can be made very precise. Reasonableness checking is similar to a process used in program correctness verification called "assertion checking" [12].

recovery block:

Once an error has been detected by a reasonableness check or a sequence control check, appropriate recovery action is performed by the recovery block. Although the recovery block could simply reset the system, there is usually enough information to determine the cause of the error and redo the appropriate section.

recovery format:

Recovery format refers to the software structure used for transient error recovery.

recovery profile:

Recovery profile is one metric of recovery used in the simulation program. The recovery profile is a histogram of the number of detected errors for each error detection technique. The intention of the recovery profile is to determine the relative value of error detection techniques.

sections:

A section is the most basic unit of structure in the proposed transient error recovery technique. An initialization section, a computation section, and an action section together form a computation block.

single event upsets (SEU):

A cause of error in digital electronics in spacecraft resulting from exposure to high-energy cosmic particles. Because of their small size, cosmic particles can result in at most one bit flip per particle.

SEU Upset Rate:

The SEU upset rate refers to the frequency of upsets caused by cosmic radiation, based upon tests conducted upon actual devices [3,11]. The upsets are assumed to occur in a constant stream, which accurately models the real phenomenon. The rate used is 0.0001 upsets/(bit-day).

sequence control codes (SCC):

Sequence control codes are a method of sequence error detection. A variable is set to a known value before a section is entered. This variable is then checked at the end of the section. If there is a discrepancy, entry into the section must have been at some point other than the proper entry point of that section. SCCs are a sub-class of reasonableness checking.

Structure/Content Model:

The Structure/Content Model abstracts software as having a recovery structure without computational content. The Structure/Content Model embodies the idea that the ability of a system to recover from transient errors does not depend upon what computation is being performed, but on how it is being performed. It is the structure of a computation, and not its content per se, which dictates the performance of Software Implemented Transient Error Recovery. More specifically, the ability to perform error propagation control, error detection, and error recovery upon the initialization, computation, and action sections is independent of the specific action performed in each section as long as the idempotence and atomic action requirements are met.

An exception to this model is a non-idempotent action section. The fact that an action is non-idempotent does not give one the liberty to retry an action if a discrepancy is found. However, any section which is idempotent, which includes all initialization and computation sections, and most action sections, can be repeated if necessary, so consequently, the specific content of the section is irrelevant to recovery.

temporary variables:

All intermediate and final results of a computation section are stored in temporary variables. This technique insures that all computation sections are idempotent.

transient error:

Transient errors (as used in this paper) are errors that are caused by phenomenon which are transient in nature, which occur randomly, and are not caused by hardware failure. The system does not have to prevent future occurrences of the same transient error to recover. Consequently, transient hardware failures and "transient" software failures do not cause transient errors as defined above, because such errors do not usually occur randomly. Example sources of transient error in spacecraft computers are high-energy cosmic particles, electrostatic discharges, and thermal noise.

upset:

A undesired bit flip occurring in volatile memory.

Upset Mapping Model:

The Upset Mapping Model abstracts the outcome of multiple bit upsets as either main memory errors, processor memory errors, or processor sequence errors. Any upset outcome not modeled directly by the above outcomes can either be modeled indirectly as a combination of the above errors, or must be considered individually.

An example of a transient error which can be modeled as a combination of the above events is an upset to an internal register of the ALU of the processor, which results in either a processor memory upset or a processor sequence upset. An example of an upset which cannot be modeled by the above outcomes is an upset to the master reset flip-flop.

BIBLIOGRAPHY

- [1] Anderson, T., and Randell, B., Computer Systems Reliability, Cambridge University Press, Cambridge, 1979
- [2] Avizienis, A., "Fault-Tolerance: The Survival Attribute of Digital Systems", Proceedings of the IEEE, Vol. 66, No.10, October 1978, pp. 1109-1125
- [3] Binder, D., "Status of the Single Event (Cosmic Ray) Problem", Hughes Aircraft Co., Interdepartmental Correspondence, 1982
- [4] Bouricius, W.G. et al., "Reliability Modeling for Fault-Tolerant Computers", IEEE Transactions on Computers, C-20, 11, November 1971, pp. 1306 - 1311.
- [5] Carter, W., et. al., "Cost Effectiveness of Self Checking Computer Design", 1977 International Symposium on Fault-Tolerant Computing, Los Angeles, June 1977, pp. 117-123
- [6] Gray, J., "Notes on Data Base Operating Systems", IBM Research Laboratory, San Jose, February 1978
- [7] Gray, P., Student Guide to IFPS, McGraw-Hill, New York, 1983
- [8] Hopkins, A.L., "Fault-Tolerant System Design: Broad Brush and Fine Print", Computer, Vol. 13, No. 3, March 1980.
- [9] Lampson, B.W., and Sturgis, H.E., "Crash Recovery in a Distributed Data Storage System", Xerox PARC internal document, June 1979
- [10] Lyons, R.E., and Vanderkulk, W., "The Use of Triple-Modular Redundancy to Improve Computer Reliability", IBM Journal of Research and Development, April 1962, pp. 200 - 209.
- [11] MacPherson, D., "SEU in Spacecraft in Synchronous Earth Orbit", Hughes Aircraft Co., Interdepartmental Correspondence, 1982
- [12] Mahmood, A., et al., "Concurrent Fault Detection Using a Watchdog Processor and Assertions", Center for Reliable Computing, Stanford University, November 1983

- [13] McCluskey, E.J., "Fault Tolerant Systems", Center for Reliable Computing, Stanford University, April 1982
- [14] Obert, R., "Intelsat VI ACE Fault Tolerance", Hughes Aircraft Co., View-graph presentation, June 1983
- [15] O'Brien, F.J., "Rollback Point Insertion Strategies", 1976 International Symposium on Fault-Tolerant Computing, June 1976, pp. 138 - 142.
- [16] Osder, S., "DC-9-80 Digital Flight Guidance System Monitoring Techniques", AIAA Journal of Guidance and Control, Vol.4, No. 1, January 1981
- [17] Randell, B., "System Structure for Software Fault-Tolerance", IEEE Transactions on Software Engineering, SE-1, 2, June 1975, pp. 220 - 232.
- [18] Rennels, D.A., "Distributed Fault-Tolerant Computer Systems", Computer, March 1980, pp. 55 - 65.
- [19] Rennels, D.A., "On Implementing Self-Checking Microprocessors", Mini and Microcomputers in Control and Measurement, Acta Press, 1981
- [20] Rosen, A., "Large Discharges and Arcs on Spacecraft", Astronautics and Aeronautics, June 1975, pp. 36 - 44.
- [21] Spillman, R.J., "A Markov Model of Intermittent Faults in Digital Systems", Proceedings of FTCS-7, June 1977, pp. 157-161
- [22] Stiffler, J.J., "Robust Detection of Intermittent Faults", Proceedings of FTCS-10, June 1980, pp. 216-218