

ERROR RECOVERY IN MULTI-VERSION SOFTWARE

Kam Sing Tso

**March 1987
CSD-870013**

UNIVERSITY OF CALIFORNIA

Los Angeles

Error Recovery in Multi-Version Software

**A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy
in Computer Science**

by

Kam Sing Tso

(曹錦成)

1987

© Copyright by

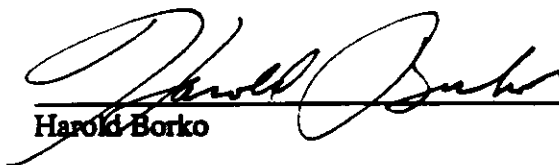
Kam Sing Tso

1987

The dissertation of Kam Sing Tso is approved.



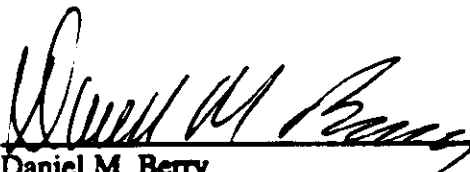
Kirby A. Baker



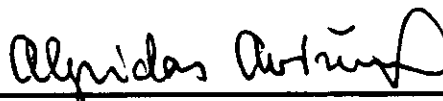
Harold Borko



David A. Rennels



Daniel M. Berry



Algirdas Avizienis, Committee Chair

University of California, Los Angeles

1987

TABLE OF CONTENTS

page

1 INTRODUCTION	1
1.1 The Need for Dependable Software	1
1.2 Approaches to Achieve Dependable Software	1
1.3 Fault-Tolerant Software by Design Diversity	2
1.4 Approaches to Fault-Tolerant Software	3
1.4.1 Multi-Version Software	3
1.4.2 Recovery Blocks	3
1.5 Multi-Version Software in Practice	5
1.6 The Need of Recovery in MVS	8
1.7 Error Recovery Approaches	8
1.7.1 Backward Error Recovery	8
1.7.2 Forward Error Recovery	9
1.7.3 Discussion	10
1.8 Goals of This Research	11
2 COMMUNITY ERROR RECOVERY	12
2.1 Objectives of Error Recovery in MVS	12
2.2 Principles of Community Error Recovery	14
2.2.1 Cross-Check Points	15
2.2.2 Recovery Points	17
2.3 Comparison of Cross-Check Points and Recovery Points	19
2.4 Rationale of Two Levels in CER	21
2.5 Program Structure for CER	23
2.6 Exception Handlers for State Input/Output	30
2.7 Comparison of the Recovery Algorithms of MVS and RB	36
3 DEDIX AND THE IMPLEMENTATION OF CER	39
3.1 The UCLA DEDIX System	39
3.2 The DEDIX Layers	41
3.2.1 The Version Layer	44
3.2.2 The Decision and Executive Layer	45
3.2.2.1 The Decision Function	45
3.2.2.2 The Local Executive	46
3.2.2.3 The Global Executive	48
3.2.3 The Synchronization Layer	50
3.2.4 The Transport Layer	51
3.3 Program Interface	52
3.4 User Interface	57
3.5 Input/Output System	59
3.6 Reconfiguration	60
3.7 Future Improvements of DEDIX	61
4 RELIABILITY MODELS FOR MULTI-VERSION SOFTWARE	63
4.1 Execution Model of MVS on DEDIX	64
4.2 Types of Errors	64
4.3 Basic Assumptions	66
4.4 Reliability Model Without Recovery	67

4.5	Reliability Model With Recovery	67
4.6	ARIES Evaluation	71
4.6.1	State Transition-Rate Matrices of the Models	71
5	EXPERIMENTATION ON MULTI-VERSION SOFTWARE	76
5.1	Previous Experiments on MVS	76
5.1.1	Zero Generation	76
5.1.2	First Generation	77
5.2	The NASA-Four University Multi-Version Software Experiment	77
5.3	Program Generation	79
5.4	Application: The RSDIMU	81
5.5	Specification of CER in RSDIMU	83
5.6	Validation Testing on the Initial Versions	83
5.7	Maintenance and Certification	85
5.8	Characteristics of UCLA Certified Versions	86
6	EXPERIMENTAL EVALUATION OF CER	87
6.1	Testing Strategy for CER	87
6.2	Testing Driver for CC-Point Recovery	88
6.3	Observed Faults in the Versions	91
6.4	Observed Errors in the Versions	94
6.4.1	Failures of the Individual Versions	94
6.4.2	Coincident Failures of the Versions	95
6.4.3	Similar Errors of the Versions	96
6.5	Faults vs. Errors	97
6.6	Classification of Triplet Decisions	99
6.7	Analysis of CC-Point Recovery	101
6.7.1	Results on CC-Point Improvement	101
6.7.2	Results on RSDIMU System Improvement	107
6.7.3	Granularity of Comparison	110
6.8	Analysis of R-Point Recovery	112
6.8.1	Results Based on Previous Test Cases	113
6.8.2	Results Based on Error Seeding	116
6.9	Discussion	117
7	CONCLUSIONS	119
7.1	Practicality of CER	119
7.2	Effectiveness of CER	119
7.3	Related Faults and Similar Errors	120
7.4	Suggestions for Future Research	121
7.4.1	Resolving Multiple Correct Results	121
7.4.2	Enhanced Fault Tolerance Techniques	122
7.4.3	Future Experiments	122
	REFERENCES	127
	Appendix A RSDIMU Defined Constants	134
	Appendix B RSDIMU Defined Types	136
	Appendix B RSDIMU Input and Output Variables	138

LIST OF FIGURES

	page
Figure 1-1: Multi-Version Software	4
Figure 1-2: Recovery Blocks	6
Figure 2-1: Scenario of Error Recovery in MVS	13
Figure 2-2: An Instrumented Avionics Program	16
Figure 2-3: The Actions of a CC-Point	18
Figure 2-4: The Actions of a Recovery Point	20
Figure 2-5: Cyclic Data Dependency in Cross-Check Points	25
Figure 2-6: Solutions For Cyclic Data Dependency	26
Figure 2-7: Structure of a Program to Support CER	28
Figure 2-8: A Program with Two Levels of Control Structure	29
Figure 2-9: The Airport Scheduler Data Base Structure	31
Figure 2-10: The Airport Scheduler Operations	32
Figure 2-11: Data Structure and Exception Handler of Version 1	33
Figure 2-12: Data Structure and Exception Handler of Version 2	34
Figure 2-13: Data Structure and Exception Handler of Version 3	35
Figure 3-1: The N Sites of DEDIX	42
Figure 3-2: The Layers and Entities at One Site of DEDIX	43
Figure 3-3: Flow Diagram of the Local Executive for CER	47
Figure 3-4: Flow Diagram of the Global Executive for CER	49
Figure 3-5: A Sample Program	53
Figure 3-6: The Instrumented Sample Program	54
Figure 3-7: Exception Handlers of the Instrumented Sample Program	56
Figure 4-1: Execution Model of MVS on DEDIX	65
Figure 4-2: Detailed State Diagram of the Reliability Model Without	

Recovery	68
Figure 4-3: State Diagram of the Reliability Model Without Recovery	69
Figure 4-4: State Diagram of the Reliability Model With Recovery	70
Figure 4-5: Reliability vs. Time	73
Figure 4-6: Reliability vs. Version Failure Rate	75
Figure 5-1: Block Diagram of RSDIMU	82
Figure 6-1: Block Diagram of the Test Case Generator	89
Figure 6-2: Testing Driver for CC-Point Recovery Testing	90
Figure 6-3: Decision Changed from GOOD2 to GOOD3	102
Figure 6-4: Decision Changed from NOMAJ to GOOD3	103
Figure 6-5: Decision Changed from BAD2 to BAD3	104

LIST OF TABLES

	Page
Table 2-1: Differences of Cross-Check Points and Recovery Points	21
Table 2-2: Differences of the Recovery Algorithms of MVS and RB	38
Table 5-1: Characteristics of the UCLA Certified Versions	86
Table 6-1: Characteristics of Observed Faults	92
Table 6-2: Failures of Individual Versions	94
Table 6-3: Coincident Failures of the Versions	95
Table 6-4: Similar Errors of the Versions	96
Table 6-5: Faults vs. Errors	97
Table 6-6: Classification of Triplet Decisions	100
Table 6-7: Results on CC-Point Improvement	105
Table 6-8: Decisions of Triplets Without Recovery	107
Table 6-9: Decisions of Triplets With Recovery	109
Table 6-10: Results of RSDIMU System Improvement	109
Table 6-11: Decisions of Triplets With Recovery (comparison by variables)	112
Table 6-12: Possible Outcomes of a DEDIX Run	114
Table 6-13: Results of R-Point Recovery	116

VITA

- April 8, 1954 Born, Hong Kong
- 1979 B.S. in Electronics,
The Chinese University of Hong Kong, Hong Kong
- 1981 M.E. in Electronic Engineering,
Philips International Institute, Eindhoven, The Netherlands
- 1983-1985 Teaching Assistant, Associate, Fellow,
UCLA Computer Science Department
- 1985-1986 Post Graduate Research Engineer,
Dependable Computing and Fault-Tolerant Systems Laboratory,
UCLA Computer Science Department

PUBLICATIONS

A. Avižienis, P. Gunningberg, J.P.J. Kelly, L. Strigini, P.J. Traverse, K.S. Tso, and U. Voges, "The UCLA DEDIX System: A Distributed Testbed for Multiple-Version Software," in *Digest of 15th Annual International Symposium on Fault-Tolerant Computing*, Ann Arbor, Michigan: June 1985, pp. 126-134.

A. Avižienis, P. Gunningberg, J.P.J. Kelly, R.T. Lyu, L. Strigini, P.J. Traverse, K.S. Tso, and U. Voges, "Software Fault-Tolerance by Design Diversity; DEDIX: A Tool for Experiments," in *Proceedings IFAC Workshop SAFECOMP'85*, Como, Italy: October 1985, pp. 173-178.

J.P.J. Kelly, A. Avižienis, B.T. Ulery, B.J. Swain, R.T. Lyu, A.T. Tai, and K.S. Tso, "Multi-Version Software Development," in *Proceedings IFAC Workshop SAFECOMP'86*, Sarlat, France: October 1986, pp. 43-49.

K.S. Tso, A. Avižienis, and J.P.J. Kelly, "Error Recovery In Multi-Version Software," in *Proceedings IFAC Workshop SAFECOMP'86*, Sarlat, France: October 1986, pp. 35-41.

K.S. Tso and A. Avižienis, "Community Error Recovery in N-Version Software: A Design Study with Experimentation," submitted to *17th Annual International Symposium on Fault-Tolerant Computing*, Pittsburgh, Pennsylvania: July 1987.

ACKNOWLEDGEMENTS

I would like to thank my committee members, Professors Algirdas Avižienis, Daniel Berry, David Rennels, Harold Borko, and Kirby Baker for serving on my dissertation committee with such enthusiasm.

I wish to express my sincere thanks to Professor Avižienis for his guidance, advice and encouragement that have contributed significantly to the preparation of this dissertation.

Special thanks to Professor John Kelly for his advice and support of my research that have greatly improved the quality of this work. Thanks to Professor Jean-Claude Laprie for his guidance on developing the reliability models during his visit to UCLA during 1985.

Thanks to Professor Berry for reading the manuscript drafts so carefully and helping me to state my meaning better.

Thanks to members of the UCLA Dependable Computing and Fault-Tolerant Systems Research Group, especially Per Gunningberg, Lorenzo Strigini, Pascal Traverse, and Udo Voges of the DEDIX project, and Rong-Tsung Lyu, Barbara Swain, Ann Tai, and Bradford Ulery of the NASA-Four University Multi-Version Software Experiment for invaluable discussions and support that have made this research possible.

I must give my heartfelt thanks to my wife Ann for her love and dedication.

This work was supported by the Federal Aviation Administration Advanced Computer Science Program (via NSF Grant MCS81-21696), under the direction of A. Avižienis, Principal Investigator, and the National Aeronautics and Space Administration, Contract NAG1-512, under the direction of A. Avižienis, Principal Investigator, and J.P.J. Kelly, Co-Principal Investigator.

ABSTRACT OF THE DISSERTATION

Error Recovery in Multi-Version Software

by

Kam Sing Tso

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1987

Professor Algirdas Avižienis, Chair

In multi-version software (MVS), design faults are masked through the consensus of results from several diverse versions. Since we assume that all versions are likely to contain design faults, it is essential to be able to recover the state of individual versions as they fail.

A Community Error Recovery (CER) algorithm which makes use of the natural redundancy that exists among the diverse versions to recover failed versions has been designed. The CER algorithm is based on two levels of recovery: *Cross-check points*, which provide a consensus result for immediate masking and partial error recovery of the erroneous results, and *recovery points*, which provide a complete recovery of the erroneous states of failed versions. This provision of two levels minimizes both the disturbance to the system and the restrictions to the implementation of diverse versions.

The CER recovery algorithm has been implemented on the UCLA DEDIX distributed MVS testbed. The purpose of DEDIX is to supervise and to observe the execution of N diverse versions of an application program functioning as a fault-

tolerant MVS unit. DEDIX also provides a highly transparent interface to the users, versions, and the input/output system. The author's contributions to the cooperative DEDIX effort include the User Interface, Input/Output System, and the integration of the CER mechanism into the Version Layer, Local Executive, and Global Executive.

Markov models for reliability evaluation of CER have been developed and the results show that recovery may substantially improve the reliability of a MVS system.

A large scale experiment is being conducted at UCLA in coordination with other institutions to determine the effect of fault tolerance techniques under carefully controlled conditions. To assess the effectiveness of the proposed CER algorithm, extensive testing were performed based on the five UCLA independently generated versions. The results of the evaluation are presented and discussed.

CHAPTER 1

INTRODUCTION

1.1 The Need for Dependable Software

Computers are playing an increasingly important role in our society. Nowadays computers have been used in such critical applications as nuclear power plant control, aircraft flight control, etc.; their failures would endanger human life and entail serious economic consequences. In the past decades efforts were concentrated mainly on the tolerance of hardware failures [Siew84]. Computer systems tolerating hardware faults with a predicted failure rate of less than 10^{-9} per hour have been designed and prototyped for aircraft flight control [Hopk78, Wens78]. However, dependable computing requires the correct behavior of both the hardware and software. At present software is much less dependable than hardware, and the gap is growing as software systems become more complex.

1.2 Approaches to Achieve Dependable Software

The conventional approach to achieve dependable software is by the use of *fault avoidance* and *fault removal* techniques. Fault avoidance is the use of structured design and programming methodology that attempt to avoid the introduction of faults. Systematic testing, verification and validation, and proofs of program correctness are

used to detect the occurrence of faults for removal. Regardless of the effort put into this approach, residual design flaws have been discovered in almost all software during operation. This suggests using the alternate approach of *fault tolerance*. This approach attempts to increase reliability by designing software to continue to provide service in spite of the presence of faults [Aviz75].

1.3 Fault-Tolerant Software by Design Diversity

Successful hardware fault detection and recovery techniques have been developed employing redundant copies of hardware, data and programs. However, these techniques cannot be applied to recover from design faults, since such replication of hardware and software elements will only reproduce the faults.

An alternate fault tolerance approach to deal with design faults is offered by *design diversity* [Aviz82, Aviz84]. In this approach, hardware and software elements are not copies, but are independently designed from a very good (preferably verifiable, formal) specification to meet a system's requirements. Different designers and design tools are employed in each effort, and commonalities are systematically avoided. The obvious advantage of design diversity is that dependable software does not require the complete absence of design faults, but only that those faults should not produce a majority of similar errors.

1.4 Approaches to Fault-Tolerant Software

The increasing awareness of the need for design fault tolerance led to the development of multi-version software at UCLA and the Recovery Block method at University of Newcastle upon Tyne since 1975.

1.4.1 Multi-Version Software

Multi-version software (or N -version programming) makes use of the potential advantages of design diversity, uses N ($N \geq 2$) versions of a program which have been independently generated from an initial specification. Programming efforts are carried out by individuals or groups that do not interact with respect to the programming process. Wherever possible, different algorithms, programming languages, environments, and tools are used in each separate effort. The goal of MVS is to minimize the probability of similar errors in the execution so that distinct erroneous results are masked by a majority vote, as shown in Figure 1-1 [Chen78b, Aviz85c].

1.4.2 Recovery Blocks

While MVS is analogous to the hardware N -modular redundancy, the Recovery Block scheme is similar to hardware standby sparing technique. It provides fault tolerance by the execution of alternate modules until a result passes an acceptance test as shown in Figure 1-2 [Rand75]. A recovery block consists of a primary module and some alternates. The primary and the alternates represent different algorithms for producing acceptable results, the primary block representing the preferred algorithm. After the execution of the primary block, an acceptance test is evaluated to check whether the results produced are acceptable. If not, the state of

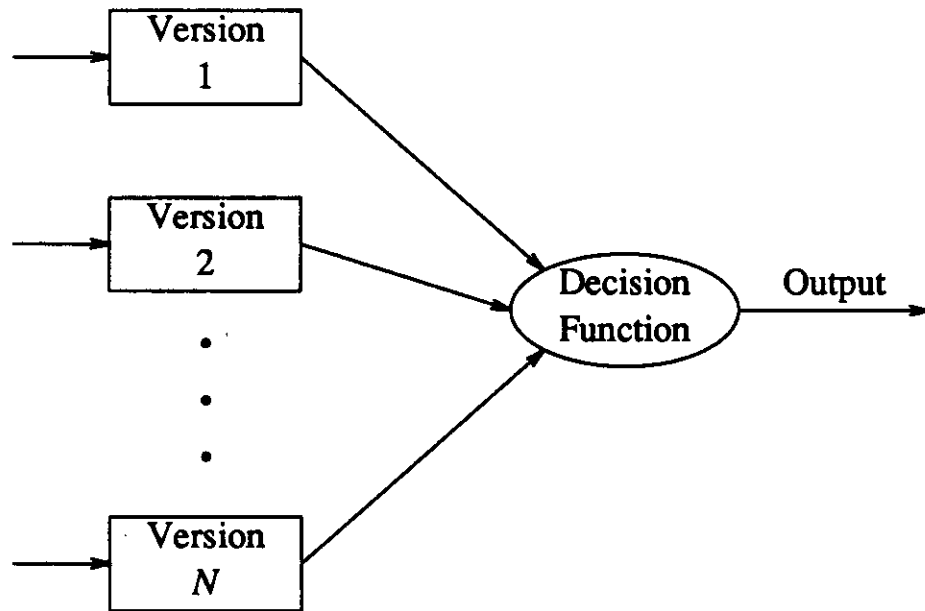


Figure 1-1: Multi-Version Software

the computation is restored to that at entry to the recovery block, and the next alternate is tried, and so on. If the primary and all the alternates fail, then this is regarded as a failure of the recovery block.

1.5 Multi-Version Software in Practice

Since its development, MVS has been applied to develop safety-critical systems to achieve ultrahigh reliability in industry. One of the reasons for its use is motivated by the certification process of the regulatory agencies of the respective industries.

1. Airbus A310

Two versions of hardware and software were used with diverse software requirements documents (from a common system specification), host computers and instruction sets, and programming teams in the Airbus A310 slat and flap control [Mart82, Hill83]. The experience shows that using diverse channels to test each other has been very successful. Crucial software errors were found during the initial testing and were detected by disagreement. Actual operations until 1983 confirmed the original MTBF (Mean Time Between Failures) prediction of > 10,000 hours by achieving an actual MTBF of 12,988 hours.

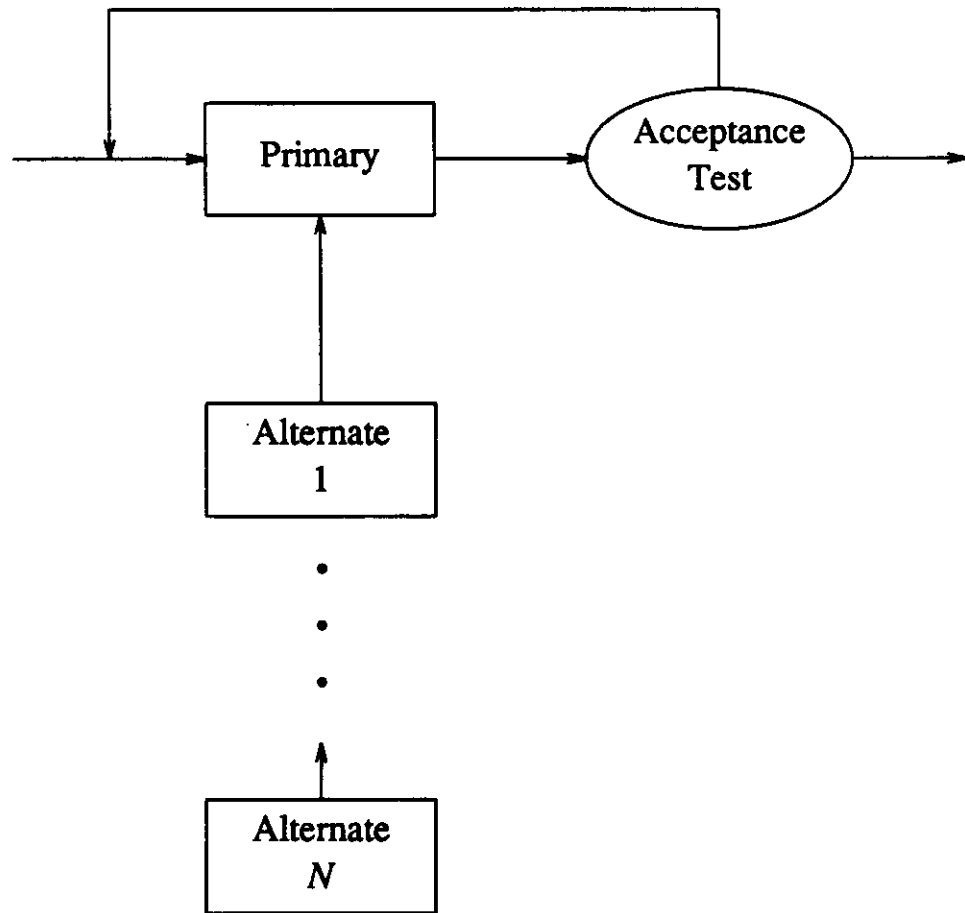


Figure 1-2: Recovery Blocks

2. **Boeing 737/757/767**

Two versions of hardware and software were used with diversity in design, code, and testing for an autopilot flight director and a yaw damper [Youn84]. The development cost of the second version of software was found to be 60% of the first one.

3. **German Nuclear Reactor**

Three versions of software with diversity in teams, programming languages (Pascal, Fortran, and assembly language) and testing, were used for a nuclear reactor safety shut-down protection system [Gmei79]. The results show that MVS proved to be a good technique for detecting ambiguities in the specifications.

4. **PODS**

Three versions of software with diversity in teams, specification languages (formal and informal), and programming languages (Fortran and assembly language) were used for a nuclear reactor protection system [Bish86]. Significantly lower failure rate has been achieved through combining three diverse versions than the failure rate of any individual program. The experiment found the the requirement specification caused most of the residual faults and was the only source of common mode failure.

5. **Swedish State Railways**

Two versions of software with diversity in teams were used in a computerized interlocking system for the Swedish state railways [Tayl81]. Successful operation of the system has been found to date.

1.6 The Need of Recovery in MVS

In multi-version software, erroneous results produced by versions are masked by a majority vote. Since we assume that all program versions (whether in a fault-tolerant system or not) are likely to contain some design faults, it is critical to have a recovery mechanism that is able to recover these versions as they fail, otherwise, the accumulation of failures may eventually become large enough to saturate the fault-masking ability, and the entire system will fail.

1.7 Error Recovery Approaches

Error recovery is the elimination of errors from an erroneous state to return the system to a proper state. Two approaches to error recovery have been employed: *backward error recovery* restores a prior saved state; and *forward error recovery* manipulates the current state to generate another state. The aim of both approaches is to obtain a state which is free from errors, or proper [Ande81].

1.7.1 Backward Error Recovery

Backward error recovery restores a prior saved state of the system without regard to the current state. It involves the establishment of *recovery points*, that is, points in time during the execution of a process at which the state of the system is saved for future restoration if required. A recovery point is discarded once the process is committed, or restored if an error is detected.

There are various mechanisms that can be used to establish recovery points. A *checkpoint* or *rollback point* [Aviz71] simply saves a complete copy of the state when a recovery point is established. An *audit trail* [Bjor75] records all changes made to the state of the process. A *recovery cache* [Horn74] records only the original states of those objects which are changed after the most recent recovery point.

The advantage of backward error recovery is that it provides a mechanism for error recovery without assessment of the faults. However, it has some drawbacks. Not all objects are recoverable: for instance, output on a printer is not recoverable. Also, significant memory and time overhead is incurred in the recovery point establishment, regardless of whether a failure actually occurs; when a failure does occur, the need for re-execution of the program module may make it unsuitable for critical, real-time applications.

1.7.2 Forward Error Recovery

Forward error recovery manipulates the current state of the system to obtain a new error-free state. This requires full knowledge of the nature of the fault involved and its exact consequences. Hence, forward error recovery techniques are usually designed specifically for each system. Nevertheless, forward error recovery can provide simple and efficient recovery from anticipated faults [Mili85].

Common techniques include exception handling and compensation. In exception handling, an operation raises an exception when standard service cannot be provided. The invoker of the operation calls for an exception handler to take care of the exception [Cris82]. An underflow exception raised by the ALU is an example which can be handled by setting the result to zero. Compensation is an act by which

one component provides to another component supplementary information intended to correct the effects of information that it had previously sent [Bjor72]. For example, an erroneous transaction that reserves an airline seat may be compensated for by another transaction that cancels the seat.

When applicable, forward error recovery is cost-effective; however, its effectiveness depends on knowledge of the faults.

1.7.3 Discussion

Backward error recovery has been successfully used to recover from transient faults in applications such as database management systems [Gray79]. For applications in which a default state exists, errors due to design faults that are triggered only by some particular input patterns may also be recovered by restoring the default state to handle other inputs. However, in MVS, backward error recovery will not be effective, since re-executing the same version will reproduce the error. Although design faults such as division by zero and overflow can be anticipated and handled by forward error recovery, the class is limited and the recovery techniques to handle them have to be designed specifically. Therefore, conventional backward and forward error recovery techniques are not powerful enough to handle the general class of design faults.

1.8 Goals of This Research

The following goals are pursued in this research:

1. To develop an error recovery algorithm which is effective in MVS, easy to use by designers and programmers, and efficient in its execution.
2. To integrate the recovery algorithm into the distributed MVS testbed DEDIX to ensure its practicality in implementation and provide an experimental vehicle for future fault-tolerant design investigations.
3. Evaluate the effectiveness of the recovery algorithm through mathematical models.
4. Study the recovery algorithm using programs created in a large scale experiment to assess its feasibility in practical use, and to obtain quantitative empirical results about its effectiveness from the execution of the programs.

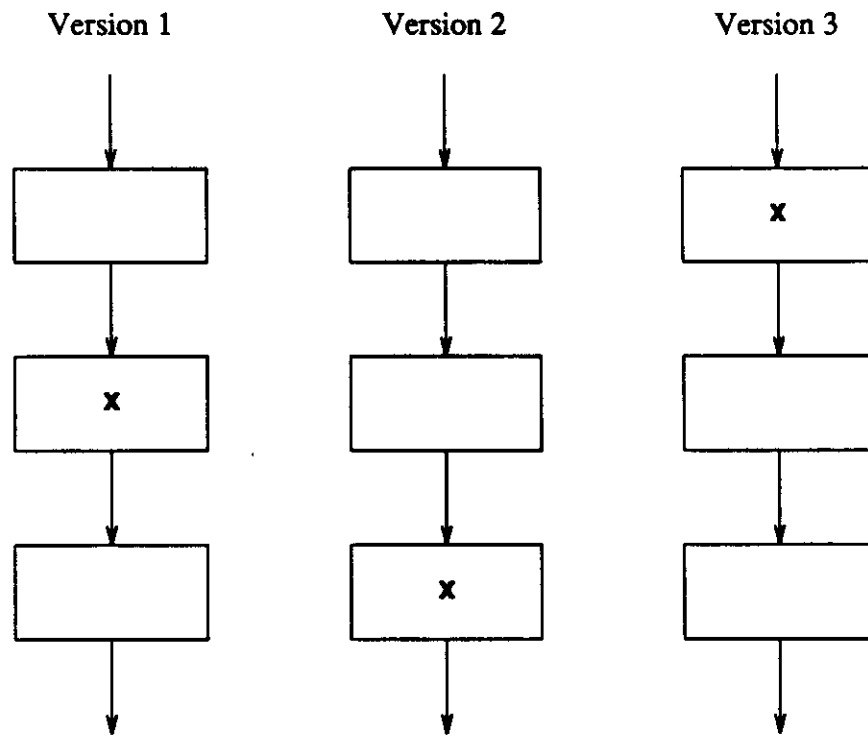
CHAPTER 2

COMMUNITY ERROR RECOVERY

2.1 Objectives of Error Recovery in MVS

The following objectives were set forth in the development of an error recovery algorithm for multi-version software:

1. *Recovery from design faults.* It is the intent of MVS to handle those design faults in a system, which are unanticipated. The error recovery algorithm should be able to recover from errors caused by this class of faults. The scenario depicted in Figure 2-1 shows how the recovery algorithm will recover from errors in a 3-version MVS system. Each of the three versions has a fault in a different program module. Without recovery, the error caused by the fault will most likely be propagated to the subsequent modules in each version. At the end of the execution, all the three versions will produce erroneous results, making the MVS system fail. The error recovery algorithm attempts to recover a version from its error right after the faulty module so that *normal* execution can continue. If successful, the 3-version MVS system will have a correct majority result at the end of each module.
2. *Minimal limitation on design diversity.* Diversity in the design and implementation of the independent versions is crucial to the success of MVS. The recovery algorithm should not adversely limit the programmers in their



x - a fault in the module

Figure 2-1: Scenario of Error Recovery in MVS

choices of design and implementation. We regard the inevitable limitation in design diversity resulting from the inclusion of the recovery provisions in the versions to be the cost paid for the increased reliability made possible by the recovery algorithm.

3. *Simplicity in implementation.* The required additional programming effort for the recovery algorithm should be simple, not only because we do not wish to increase the workload of the application programmers, also, we do wish to ensure that the recovery algorithm can be easily observed by the programmers and verified for its correctness.

2.2 Principles of Community Error Recovery

In multi-version software, the method of Community Error Recovery† (CER) makes use of the assumption that at any given time during execution there exists a majority of good versions which can supply information to recover the failed versions. It uses the *natural* redundancy that exists among the program versions in an MVS system.

The CER method is based on two levels of recovery: *Cross-check points* (cc-points) which provide a consensus result for immediate masking and partial error recovery of the erroneous results, and *recovery points* (r-points) which are inserted between program modules (usually containing several cc-points) for the complete recovery of the erroneous states of failed versions.

† The idea of Community Error Recovery is analogous to a community of people helping individuals with difficulties.

Figure 2-2 shows an instrumented avionics control program consisting of four modules and the major computations within the Acceleration Estimation module that was implemented 20 times for the NASA-Four University Multi-Version Software Experiment. In this discussion, a *computation* consists of one or more functions that compute some result, and a *module* consists of a sequence of such computations. We also distinguish between two types of design faults in software which have different effects on the execution of a program. These are *computation faults*, which cause the assignment of erroneous values to the state variables, and *control flow faults*, which cause erroneous branching in a program, resulting a cc-point or r-point being skipped or incorrectly called.

2.2.1 Cross-Check Points

A standard *N*-version cross-check point is a decision point at which the redundant versions output their results after a computation for comparison [Chen78b]. Associated with each cc-point are a cc-point id (*ccp-id*), which uniquely identifies the cc-point; a *format* string, which indicates the types and number of state variables to be compared; and a set of pointers to the state variables (*cc-vector*) which allows results to be passed to the MVS Supervisor for comparison. The standard cc-point is extended beyond its fault masking capability in CER to include functions of error detection and error recovery of the failed versions. Hence the cc-points serve the following functions in MVS with CER:

- *fault masking* - a decision result is obtained through comparison of the cc-vectors, thus masking faults in the versions.
- *fault detection* - faults in failed versions will manifest themselves as either

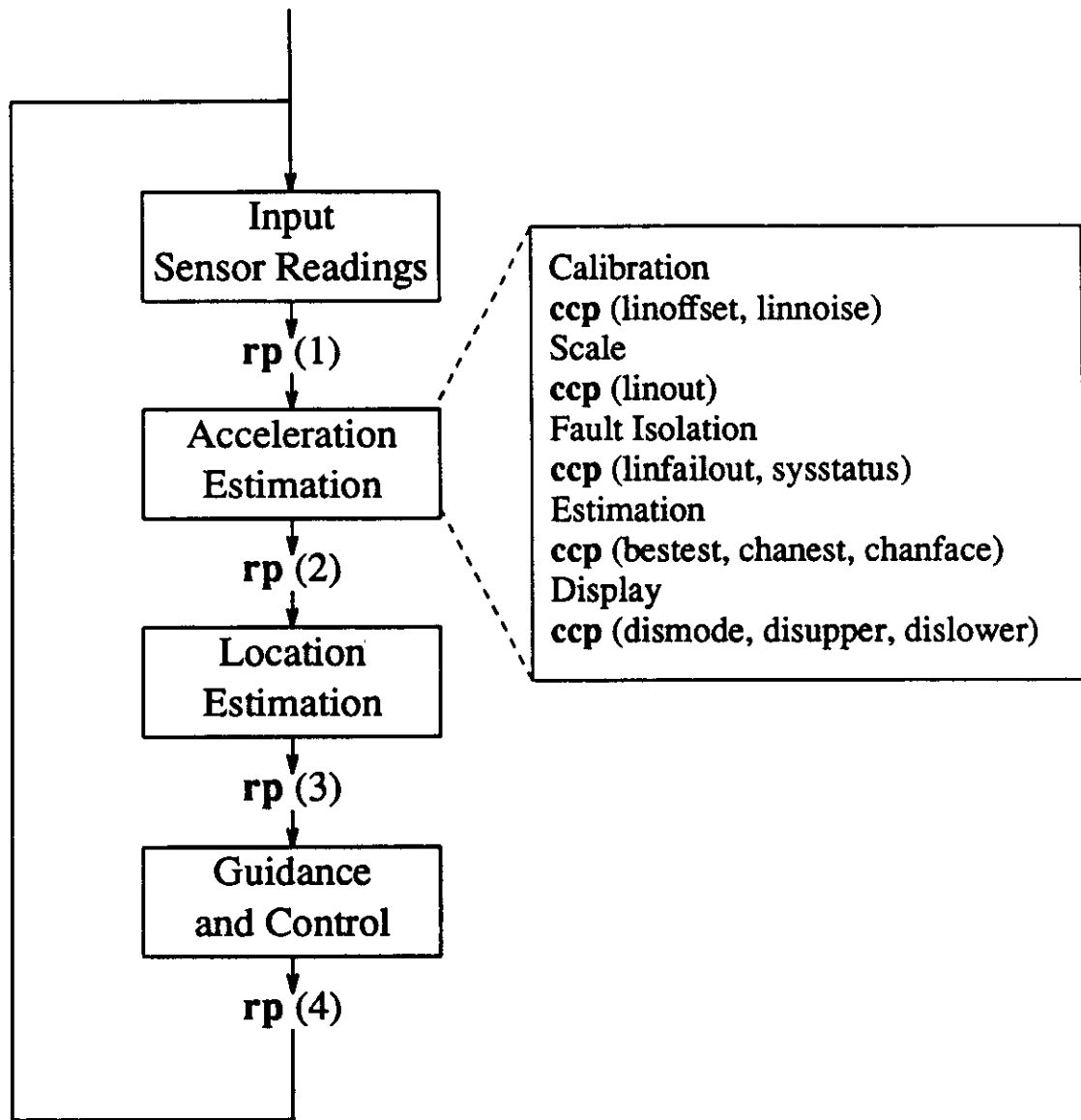


Figure 2-2: An Instrumented Avionics Control Program

erroneous or missing cc-vectors, which are detectable. This information is accumulated for the second level of recovery.

- *error recovery* - the decision result is returned to the failed versions for future computation, in order to attempt a recovery from the effects of errors.

Figure 2-3 shows the actions of a cc-point in a 3-version MVS system with version 3 failed at the cc-point following computation *i*. The MVS Supervisor, which will be discussed in the next chapter, compares the results with a *decision function* and sends the *decision result* to the faulty version for recovery. If the result is an output of the program, the Supervisor will output the decision result so that faults due to a minority of versions will be masked.

Error recovery at the cc-point level is fully effective only if a minority of versions fail between two successive cc-points, if the failed versions have executed the right cc-point on time, and if all other state variables that are not in the cc-point have not been corrupted. If there is no consensus among the versions, recovery will not be performed and the system will be shut down safely. If, in the worst case, a majority of versions produces similar errors at the cc-point, then the minority of good versions will be forced to an erroneous state.

2.2.2 Recovery Points

Complete error recovery of failed versions is performed at recovery points. Associated with each r-point in each version, the following are specified: a unique recovery point id (*rp-id*), which uniquely identifies the r-point, and two exception handlers, the *state-input exception handler* and the *state-output exception handler*, which are required to input and to output, respectively, the internal state of the version

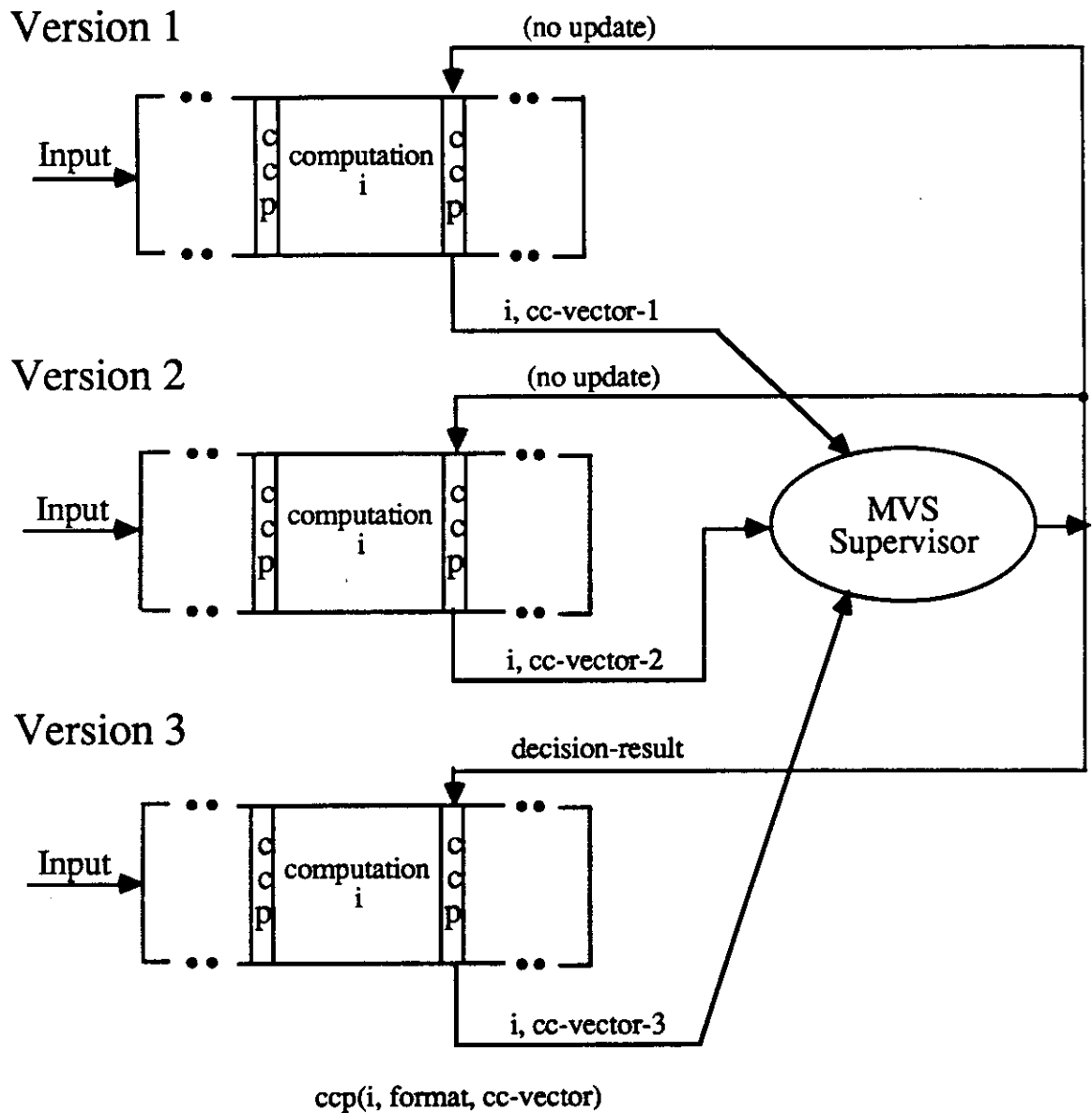


Figure 2-3: Cross-Check Point Actions With Failed Version 3
(cc-vector-3 differs from cc-vector-1 and cc-vector-2)

(*version state*) in a specified format. Every version is required to have the same version state at the recovery point. At a recovery point, the rp-ids of the versions are submitted to the MVS Supervisor and compared, as shown in Figure 2-4. Failed versions are identified by missing or incorrect rp-ids, and by errors that have been detected at cc-points. Exception handlers are invoked by the MVS Supervisor upon any failed versions thus detected. The state-output exception handler in every good version is then instructed to output its version state. The version states are compared by the decision function of the MVS Supervisor to produce a *decision state* which will be input by the state-input exception handler in every failed version. Versions that have control flow faults are detected by missing or incorrect rp-ids. These versions are restarted at the current recovery point by means of the decision rp-id. Before their execution is resumed, the MVS Supervisor invokes their state-input exception handlers to input the decision state. In this way, failed versions will proceed to the next program module with a correct internal state after the recovery point.

2.3 Comparison of Cross-Check Points and Recovery Points

The functions of cc-points and r-points in CER complement each other in their recovery efforts. We will gain a better understanding of CER by further contrasting these mechanisms.

A cc-point cross-checks the result of a computation from each version (which most likely consists of only a few state variables), while a r-point involves the comparison of the complete version states at that decision point. Hence, cc-points are more efficient both in terms of inter-site communication and in the comparison of the values than are r-points. However, a cc-point can only achieve partial recovery, since

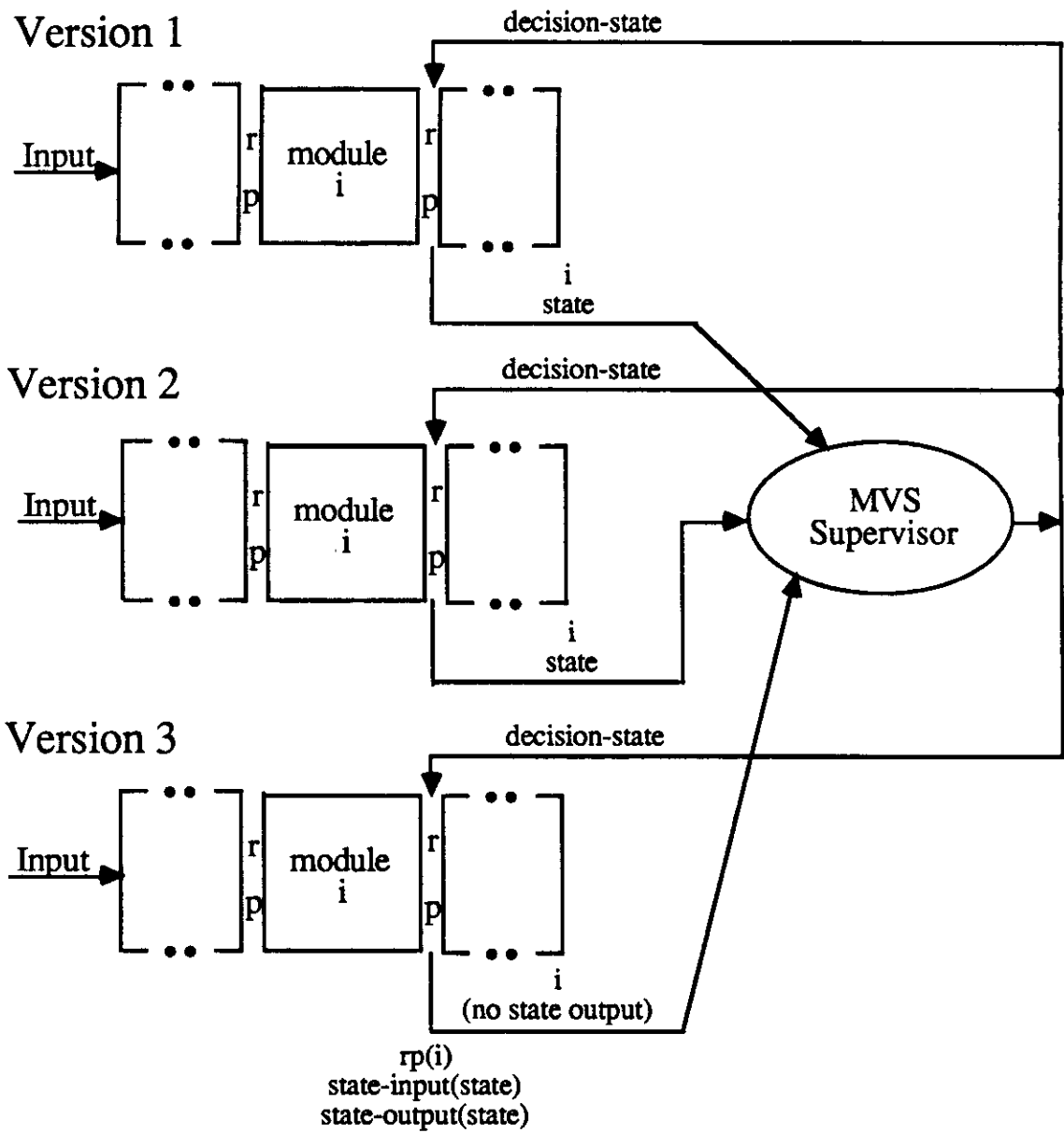


Figure 2-4: Recovery Point Actions with Failed Version 3

it recovers only the cross-checked state variables and does not handle control flow faults. Recovery at the r-point level may limit diversity in the design and implementation of program versions because it requires them to be structured into modules with identical specified states.

The differences between these two recovery mechanisms are summarized in Table 2-1.

Table 2-1: Differences of Cross-Check points and Recovery points

<i>Cross-Check Points</i>	<i>Recovery Points</i>
Involve only a few variables (computed results)	Involve complete internal state
Partial recovery	Complete recovery
Lower overhead	Higher overhead
Easy to specify and implement	May limit design diversity

2.4 Rationale of Two Levels in CER

The addition of the second level of recovery by means of CER has the following advantages:

1. *To minimize the adverse effects on program design and implementation diversity.* Cc-points that compare the version results after a computation can easily be specified and observed by the programmers. The complete internal states have to be specified at the recovery points may constrain the design diversity of the program versions.

2. *To match the type and severity of the errors.* At the first level, cc-points are placed after major computations in a module. If a version fails to compute correct results, it is immediately corrected in the hope that further computations that are based on those results will use the correct values and normal execution can continue.
3. *To minimally disturb the system.* In some systems, recovery actions may be more disruptive than the error they are intended to correct. The cc-point level makes sure that local errors are corrected without disruption of other parts of the system. If the local recovery action is inadequate, the versions will be recovered by the use of recovery points at the second level. At a recovery point, the complete internal state of a failed version is recovered at the expense of the time and effort needed to communicate and compare states of the versions. For applications with a large system state, the exception handler can be designed so that the communication and comparison of version states is done progressively in two or more recovery points.

Cross-check points can be used alone for error recovery in an MVS system without the need for a recovery point. However, it can only be a partial recovery if not all the state variables are cross-checked or control flow faults are not handled. It might be sufficient for applications which have small internal states and low probability of control flow faults. If cc-points do cross-check complete internal states, the following problems may still arise: 1) limited design diversity may result if the computations between the cc-points are relatively small, 2) there is an increased chance of coincident errors if the computations between cc-points are large, and 3) high communication and comparison overheads may result because complete states have to be sent and compared even in the absence of errors.

A hybrid scheme may be devised by combining the fault masking and error detection functions of cc-points by cross-checking only a few state variables, and the error recovery functions of recovery points by invoking exception handlers upon detected failure. It may be called a cross-check recovery point (*cr-point*). The *cr-point* has the merit of less overhead when there is no failure but it is still able to achieve complete recovery. Nevertheless, it cannot be used in place of the cc-points because complete states have to be specified, and this puts a severe limitation on design diversity if they are placed too close together. It would be convenient to use a *cr-point* to replace the cc-point of the last computation and the recovery point of a module.

2.5 Program Structure for CER

With CER implemented in MVS, the initial specification not only specifies the functional requirements of the application, it is also required to specify the locations of cc-points and r-points, and their corresponding associated cc-vectors and states. A great deal of research has focused on the system structuring and placement of recovery points in order to avoid the so-called *domino effect* [Rand78] in recovery of asynchronous systems [Merl78, Russ80, Bari83, Camp86, Koo87]. Since the versions in MVS are synchronous at the cc-points and r-points, we do not need to deal with this complex problem.

A cc-point can be placed anywhere in a version once all of the state variables of its cc-vector have been computed and before any one of them is used. If some state variables in a cc-vector have been used before they are cross-checked, recovery will not be effective, since errors in those variables would have been propagated before

they are recovered.

Care must be taken in the specification of cc-points to make sure that there is no *cyclic data dependency* between any of them. Otherwise, no implementation can satisfy the recovery requirements. Cyclic data dependency arises when one or more state variables in a cc-point depend on the result of the next computation which, in turn, must use some of the state variables of the cc-point. Figure 2-5 shows a program fragment consisting of two computations and their cc-points. Suppose the sequence of the cc-points and their associated cc-vectors have been specified as in Figure 2-5. If the function f_c is placed in the first computation, the value of the state variable a will be used before it is cross-checked. If f_c is placed in the second computation, f_b cannot be computed in the first computation. Fortunately, we can always avoid the cyclic data dependency problem by splitting a cc-point into two or more cc-points which, together, cross-check the same state variables as the unsplit one, or by rearranging the state variables in the cc-points. For this example, we can either split the two cc-points as in Figure 2-6(a), or cross-check the state variable c in the first cc-point so that f_c can be placed in the first computation, as shown in Figure 2-6(b).

Recovery points require the versions be structured into modules. The *intended service* of a module can be specified by a relation between its initial internal state and final internal state. Modular programming has been advocated by modern programming methodology. A module is intended to be a unit of work assignment. It is intended to be a unit which can be constructed with no knowledge of the internal structure of other modules. It is also intended that most design decisions in a system can be changed by altering a single module [Parn72a, Parn72b]. The purpose of modular decomposition in CER is to minimize the size of the internal state between modules, and to be able to restart a version from any module by the use of a rp-id.

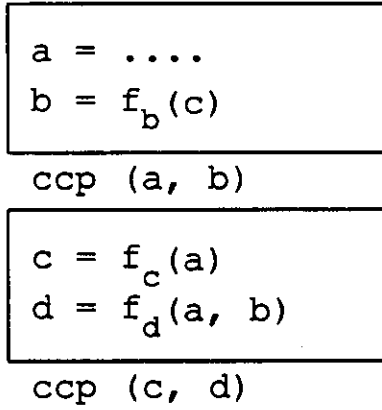
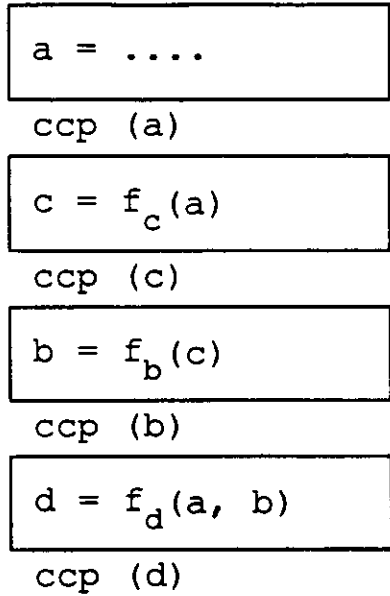
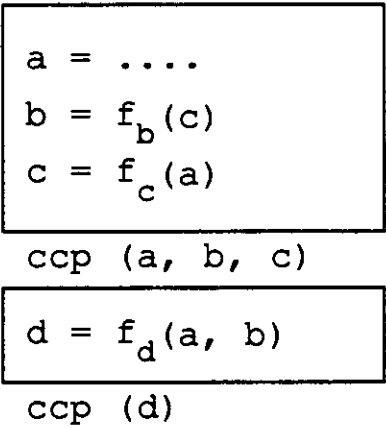


Figure 2-5: Cyclic Data Dependency in Cross-Check Points



(a) Splitting CC-Points



(b) Rearranging CC-Vectors

Figure 2-6: Solutions of Cyclic Data Dependency

Many modern programming languages which support exception handling have been devised to help designers program the specified exceptional responses correctly [Lisk79, Luck80]. In this research, we intend to use common programming languages for the versions. Since these languages do not have a global goto or language constructs for restarting a version anywhere in the program, versions are structured as shown in Figure 2-7 in order to make restart possible. In this scheme, a *switch* is inserted at the beginning of a version which determines the module from which the execution starts using the argument *rp-id*. The MVS Supervisor executes all the versions with a *rp-id* equal to 0 at the beginning. If a version fails due to a control flow fault at *r-point r*, that version will be aborted by the MVS Supervisor and restarted with an argument *r*, hence from the module after *r-point r*.

It may appear that *r-points* restrict the structure of the versions to a sequence of modules. In fact, other control structures such as branching and looping are possible, as shown in Figure 2-7. In the case of branching, both *r-points* in the two branches go to the next module after the branching. With loops, the *r-point* is called in each iteration and the *rp-id* goes back to the loop itself.

Some programs may have large modules, each in turn consisting of a number of nested modules in which *r-points* are needed for recovery. For these programs, the single *rp-id* can be extended to a list of *ids*, such that the first one identifies the modules in the first level, and the second one identifies the nested modules, and so on. Figure 2-8 shows the structure of a program with two levels of control. Note that the *r-point* requires two arguments: the *rp-ids* for the first and second level of control. This scheme of restarting a version requires no special linguistic support and can be applied with any common programming language.

```

version(rp-id)

begin
  case (rp-id) of
    0 : goto M0
    1 : goto M1
    2 : goto M23
    3 : goto M4
    4 : goto M4
    5 : goto M4
    .
    .
  end

  M0:
    Module0
    rp (1)
  M1:
    Module1
    rp (2)
  M23:
    if (c) then
      Module2
      rp (3)
    else
      Module3
      rp (4)
    endif
  M4:
    while (c) do
      Module4
      rp (5)
    end

    .
    .
end

```

Figure 2-7: Structure of a Program to Support Recovery Points

```

version(rp-id)
begin
  case (rp-id[1]) of
    0 : goto M0
    1 : goto M1
    .
    .
  end

  M0:
    Module0
    rp (1, 0)
  M1:
    Module1
    rp (2, 0)
  .
  .
end

```

```

Module1(rp-id)
begin
  case (rp-id[2]) of
    0 : goto M10
    1 : goto M11
    .
    .
  end

  M10:
    Module10
    rp (1, 1)
  M11:
    Module11
    rp (1, 2)
  .
  .
end

```

Figure 2-8: A Program With Two Levels of Control Structure

2.6 Exception Handlers for State Input/Output

The use of exception handlers for input and output of the version states allows the programmers flexibility in the choice of data structures for their programs. For state recovery at the r-points, it is necessary to specify the state variables comprising the internal states at those r-points, their formats, and their order of exchange. Different data structures can be used to represent a state which meets the specification. The following example demonstrates how this can be accomplished.

The application used in the example is a data base problem which concerns the operation of an airport in which flights are scheduled to depart for other airports, and seats are reserved on those flights. The problem was discussed originally in [Ehri78] and was used by Kelly in a specification-oriented experiment for multi-version software [Kell82]. The data base structure is shown in Figure 2-9. The airport scheduler application has seven operations, defined as possible inputs to the data base of the system, as shown in Figure 2-10. Three different data structures have been used to implement the data base in Pascal, as shown in Figures 2-11, 2-12, and 2-13.

The first data structure uses three tables in a way similar to the specification. The second uses one single table to represent the data base structure. The last one uses a linked list, with each node representing a record. The respective exception handlers for state output are also shown in the figures. The function `stateout` used in the exception handlers is supplied by the MVS system for collecting state values from the versions. Since the internal state of each version is the data base itself, the exception handlers do not depend on the `rp-id`.

This example shows that by using exception handlers for input and output of the internal states, the programmers are not restricted in their choice of data structures.

1	APS			Airport Scheduler
2	FS			Flight Scheduler
	3	F#		Flight Number
	3	DEST		Destination
	3	TIME		Departure Time
		4	HOURS	
		4	MINS	
2	P			Planes
	3	P#		Plane Number
	3	TYPE		Plane Type
	3	NOSEATS		Number of Available Seats
2	PF			Planes and Flights
	3	F#		Flight Number
	3	P#		Plane Number

Figure 2-9: The Airport Scheduler Data Base Structure

CREATE

- Initialize data base

SCHEDULE F#, DEST, TIME, P#, TYPE

- Schedule a flight,
- Add a data base record

CHANGE_TIME F#, TIME

- Change a departure time,
- Modify a data base record

RESERVE_SEAT F#

- Reserve a seat on a flight,
- Modify a data base record

CANCEL F#

- Cancel a flight,
- Remove a data base record

USED? F#

- See if a flight is scheduled,
- Make a data base query

LIST

- Show all flights,
- List data base contents

Figure 2-10: The Airport Scheduler Operations

```

type
  HrMin = record
    Hr: integer;
    Min: integer;
  end;
  FSrec = record
    FN: integer;
    Dest: string;
    Time: HrMin;
  end;
  Prec = record
    PN: integer;
    Types: string;
    Seat: integer;
  end;
  PFrec = record
    FN: integer;
    PN: integer;
  end;
  TableIndex = 1 .. TABLELENGTH;
var
  FStable : array [TableIndex] of FSrec;
  Ptable : array [TableIndex] of Prec;
  PFtable : array [TableIndex] of PFrec;
  TablePtr : TableIndex;

```

(a) Data Structures: Using Three Tables

```

procedure stateoutput (rpid : integer);
  var i : TableIndex;
  begin
    for i := 1 to TablePtr do begin
      stateout (rpid, "%d%s%d%d%d%s%d",
        FStable[i].FN, FStable[i].Dest,
        FStable[i].Time.Hr, FStable[i].Time.Min,
        Ptable[i].PN, Ptable[i].Type, Ptable[i].Seats);
    end;
  end;

```

(b) Exception Handler for State Output

Figure 2-11: Data Structures and Exception Handler of Version 1

```

type
  HrMin = record
    Hr:    integer;
    Min:   integer;
  end;
  FSPrec = record
    FN:    integer;
    Dest:  string;
    Time:  HrMin;
    PN:    integer;
    Types: string;
    Seat:  integer;
  end;
  TableIndex = 1 .. TABLELENGTH;
var
  FSPTable : array [TableIndex] of FSPrec;
  TablePtr : TableIndex;

```

(a) Data Structures: Using One Table

```

procedure stateoutput (rpid : integer);
  var i : TableIndex;
  begin
    for i := 1 to TablePtr do begin
      stateout (rpid, "%d%s%d%d%d%d%s%d",
        FSPTable[i].FN, FSPTable[i].Dest,
        FSPTable[i].Time.Hr, FSPTable[i].Time.Min,
        FSPTable[i].PN, FSPTable[i].Type, FSPTable[i].Seats);
    end;
  end;

```

(b) Exception Handler for State Output

Figure 2-12: Data Structures and Exception Handler of Version 2


```

type
  HrMin = record
    Hr:    integer;
    Min:   integer;
  end;
  FSPrec = record
    FN:     integer;
    Dest:   string;
    Time:   HrMin;
    PN:     integer;
    Types:  string;
    Seat:   integer;
    NextRec: ^FSPrec;
  end;
var
  FSPlist  : ^FSPrec;

```

(a) Data Structures: Using List of Records

```

procedure stateoutput (rpid : integer);
begin
  while FSPlist <> nil do begin
    stateout (rpid, "%d%s%d%d%d%s%d",
      FSPlist^.FN, FSPlist^.Dest,
      FSPlist^.Time.Hr, FSPlist^.Time.Min,
      FSPlist^.PN, FSPlist^.Type, FSPlist^.Seats);
    FSPlist := FSPlist^.NextRec;
  end
end;

```

(b) Exception Handler for State Output

Figure 2-13: Data Structures and Exception Handler of Version 3

Of course, they can use any appropriate algorithm in the module for the data structure they have chosen. If the types of all the state variables are defined in the specification, the exception handlers will be the same for each version and can be supplied to the programmers. This will relieve most of the work required to implement CER in the versions.

2.7 Comparison of the Recovery Algorithms of MVS and RB

Both the Community Error Recovery algorithm in MVS and the Recovery Block method attempt to tolerate design faults in software. They both require the internal state be specified between program modules. The RB method saves the state before the execution of a module in order to be able to roll-back if the acceptance test rejects the results. CER uses the states of the majority (good) versions to recover the erroneous states of the faulty versions. Hence, the RB scheme is a backward error recovery algorithm, while the CER scheme is a forward error recovery algorithm.

One major difference between the two schemes is their mechanisms to detect errors. The RB scheme uses an acceptance test after each module to detect errors. Some experiments have shown that sometimes the acceptance test can be as complicated as the original algorithm [Glas80, Ande84], and it may make sense to replace it with a re-execution of an alternate module and compare their results for error detection [Kim84]. The generic decision algorithm of MVS is simple and efficient.

The overheads of both algorithms are compared in two aspects: 1) overhead incurred during error-free execution, and 2) overhead needed to recover from errors. During normal execution, the RB method incurs time and storage overhead on saving

the state before the execution of each module. It is also required to execute an acceptance test after each module. For the CER scheme, time overhead is incurred on the inter-site communication and the comparison of cc-vectors for each cc-point, and rp-ids for each r-point. During recovery actions, the RB scheme is required to reinstate the previous saved state and to execute an alternate module and the acceptance test. The time overhead on the multiple executions sometimes would be too high for time-critical applications. The CER scheme requires that the good versions output their internal states, compare the states, and input the decision states into the faulty versions. This inter-site communication and comparison overhead is much lower than the re-execution of alternate modules.

Another important difference is that a Recovery Block has only one version of state for the primary and alternate modules, while in MVS different versions may implement the specified state in different ways. This has been shown in a previous section, which allows more diversity among versions.

The differences between the recovery algorithms of MVS and RB are summarized in Table 2-2.

Table 2-2: Differences of the Recovery Algorithms of MVS and RB

<i>Multi-Version Software</i>	<i>Recovery Blocks</i>
Forward error recovery	Backward error recovery
Error detection by cross-check points	Error detection by acceptance tests
No internal state needs to be saved	Internal state saved at every recovery point
Cc-vector compared at every cc-point and rp-id compared at every r-point	Acceptance test executed after every module
Time overhead for communication and comparison of states	Time overhead for state restoration and alternate modules execution
One version of state in a Recovery Block	Possible diversity in the implementation of a state in different versions

CHAPTER 3

DEDIX AND THE IMPLEMENTATION OF CER

3.1 The UCLA DEDIX System

In order to facilitate experimental investigations into the design and evaluation of multi-version software as a means of achieving fault-tolerant systems, a distributed MVS supervisor and testbed, called the DEsign DIversity eXperiment (DEDIX) system, has been designed and implemented by the UCLA Dependable Computing and Fault-Tolerant Systems research group, at the UCLA Center for Experimental Computer Science [Aviz85a, Aviz85b, Aviz85c]. DEDIX currently executes above LOCUS, a distributed, network-transparent UNIX-like operating system running on the Center's Olympus Net local network operating a set of about 20 VAX minicomputers connected by Ethernet [Pope81]. The purpose of DEDIX is to supervise and to observe the execution of N diverse versions of an application program functioning as a fault-tolerant multi-version software unit. DEDIX also provides a highly transparent interface to the users, versions, and the input/output system. DEDIX is the cooperative effort of several individuals. The contributions of the author include the User Interface, Input/Output System, and the integration of the CER mechanism into the Version Layer, Local Executive, and Global Executive.

The general functional requirements for DEDIX are the followings:

1. *Distribution*: The versions should be able to execute on separate physical sites in order to take advantage of the physical isolation between sites, to benefit from parallel execution, and to survive a crash of a minority of sites.
2. *Transparency*: The interface for the application programmers has to be simple, and the interface must be independent of the number of actual versions used.
3. *Environment*: DEDIX is designed to run on the distributed LOCUS environment at UCLA, and should be portable to other UNIX† systems. DEDIX must be able to run concurrently with all other normal activities of the local network.

In order to fulfill these requirements, DEDIX was developed as a modular redundant system. Its structure can be considered as a network-based generalization of SIFT [Gold80] that is able to tolerate both hardware and software faults. Both have similar partitioning, with a local executive and a decision function at each site that processes broadcast results, and a copy of the global executive at each site that is responsible for error recovery and reconfiguration decisions by majority vote. DEDIX is extended to allow some diversity in results and in version execution times.

DEDIX itself is written in C and makes use of several LOCUS features, e. g. the distributed execution of the version processes and the linking of these processes via remote pipes. The DEDIX system can be located either in a single computer that executes all versions sequentially, or in a multicomputer system running one or more versions at each site. If DEDIX were to run on a single computer, it would be vulnerable to hardware and software faults that affect the host computer, and the execution of MVS units would be slower. In a computer network environment, the

† UNIX is a trademark of AT&T Bell Laboratories

system is partitioned to protect it against most hardware faults. This has been done by providing each site with its own copy of DEDIX software. The DEDIX design is suitable for any specified number $N \geq 2$ of sites and versions, and the current implementation allows up to 20 versions due to the limited computing resources, such as memory size, the maximum number of communication links and running processes. DEDIX has been ported to other UNIX systems running on a single site. It should be able to port DEDIX to a network of computers running UNIX if mechanism for communication between different sites is provided, such as the socket mechanism found in the Berkeley UNIX BSD4.2 or BSD4.3 systems [Leff86].

A global view of the DEDIX system supporting N versions is given in Figure 3-1. The versions communicate with its local DEDIX, which in turn makes use of the LOCUS operating system, and the different sites are interconnected with each other via Ethernet.

3.2 The DEDIX Layers

DEDIX has been designed as a set of hierarchically structured layers as shown in Figure 3-2. The layered structure reduces implementation complexity and facilitates the inevitable modifications. The purpose of each layer is to offer services to the higher layers, shielding them from details on how those services actually are implemented. Each of the sites running DEDIX has an identical set of layers and entities, providing services to its version and the external user. These layers, from top to bottom, are:

- the Version Layer,
- the Decision and Executive Layer,

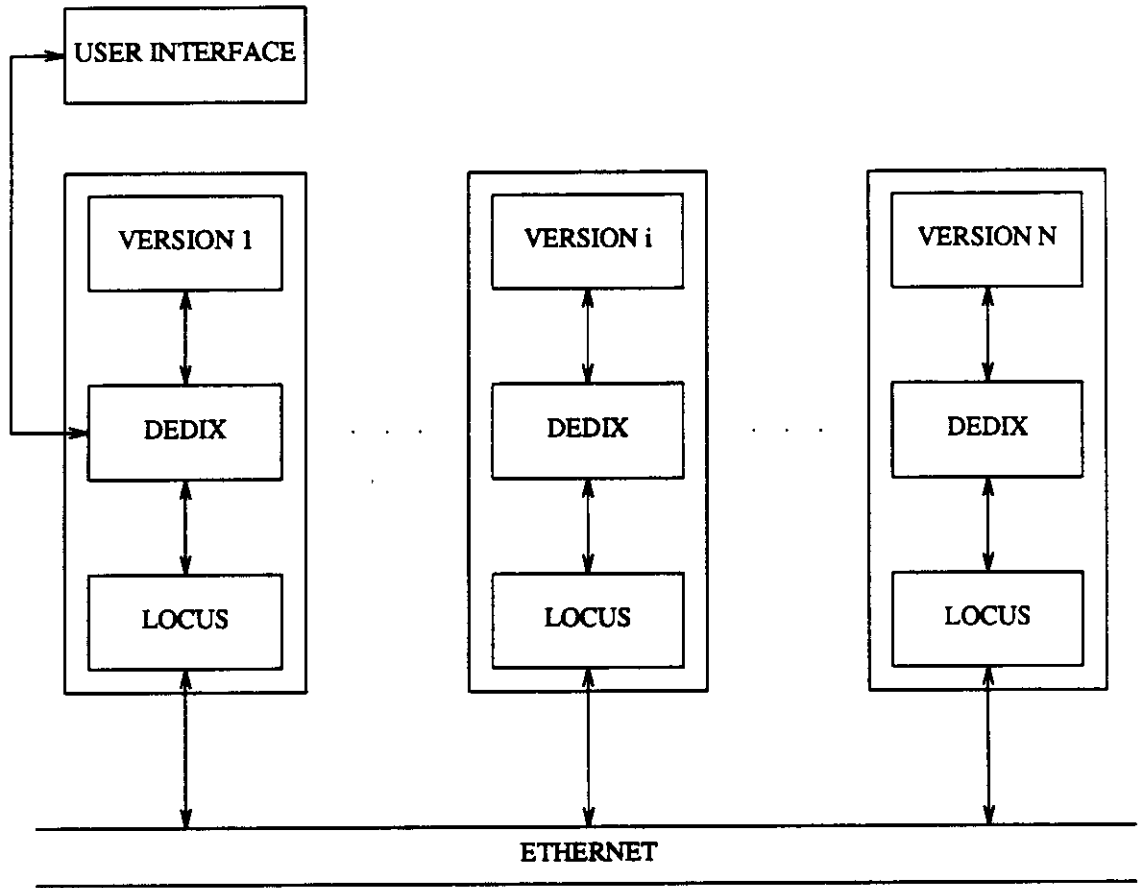


Figure 3-1: The N sites of DEDIX

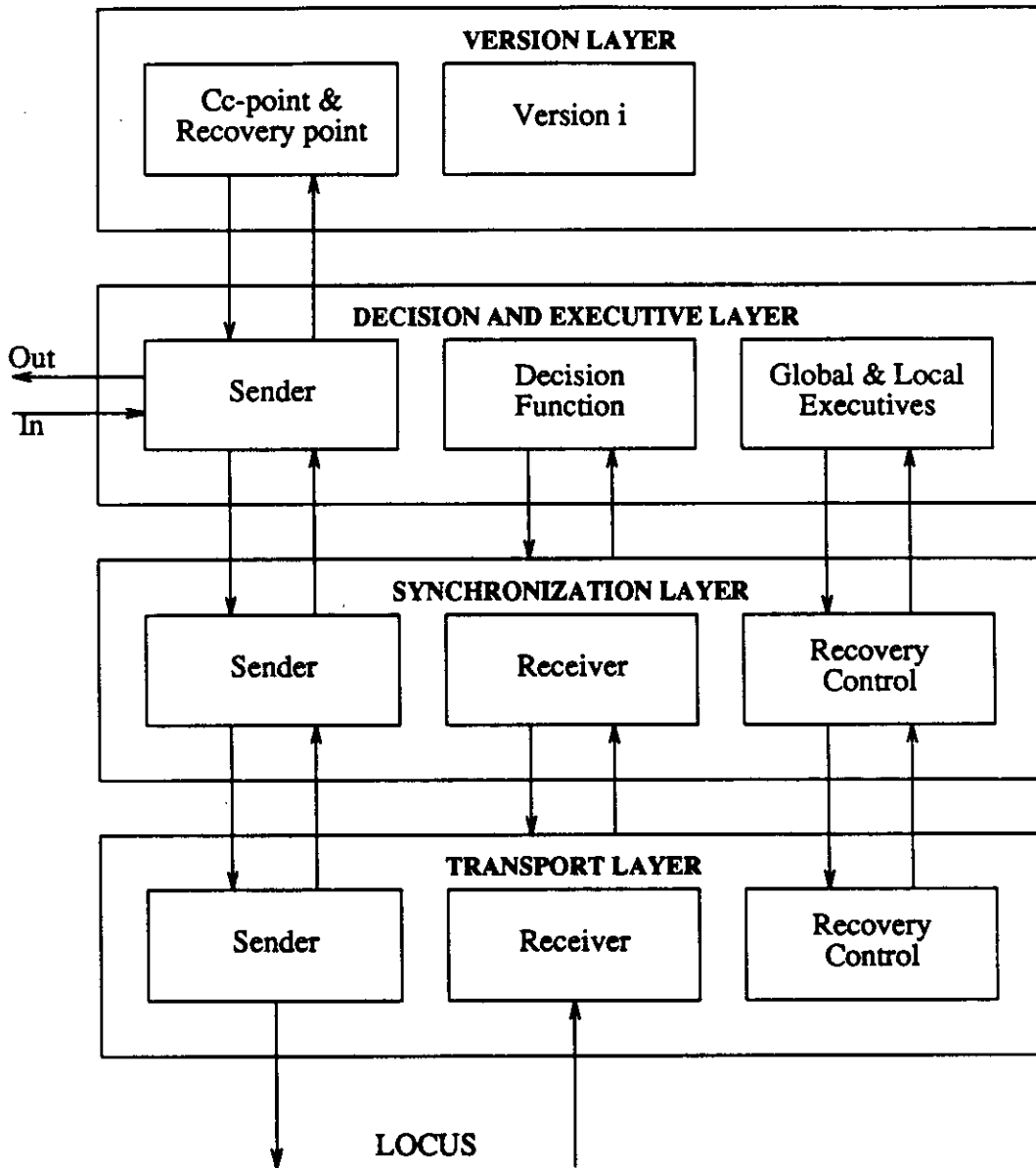


Figure 3-2: Layers and Entities at One Site of DEDIX

- the Synchronization Layer,
- the Transport Layer.

3.2.1 The Version Layer

The purposes of this layer are to interface the i -th version with the DEDIX system, to correct state variables and to restart failed versions. The interface is supported through calls to cc-points and r-points. Three different types of cc-points are currently implemented:

1. *ccinput*, which is used by DEDIX for centralized data input and broadcast to the versions;
2. *ccoutput*, which is used for comparing the output values collected from the versions and then uses the decision result as output;
3. *ccpoint*, which is used for cross-checking state variables for fault detection and error recovery.

Each cc-point has a cc-point id (ccp-id), which uniquely identifies the cc-point; a format string, which indicates the number of state variables to be compared and their types; and a set of pointers to the state variables (cc-vector), which allows results to be passed to DEDIX for comparison and passed back for recovery.

Each r-point has one parameter, a r-point id (rp-id). Control flow errors of failed versions are detected by missing or incorrect rp-ids. It is also used to identify the exception handlers needed for global recovery, and to determine the point at which a failed version should be restarted.

To run on DEDIX a version must be instrumented. That is, the version must call DEDIX at each occurrence of a cc-point or a r-point, and pass its results to DEDIX. We will show how this is done in a subsequent section. Currently, the available application languages are C and Pascal. Other languages could be used for the versions, if an interface between this language and C is available.

3.2.2 The Decision and Executive Layer

This layer receives cc-vectors and internal states from its local version and from the other versions through the Transport Layer and Synchronization Layer, produces decision results, determines faulty versions, and performs recovery and reconfiguration actions. It also controls the input/output of versions and handles exceptions. This layer has four entities: the Sender, the Decision Function, the Local Executive, and the Global Executive. The Sender receives requests from the local version and broadcasts the requests to other sites. It also attaches an *occurrence* number to a ccp-id or rp-id. The occurrence number is used to uniquely identify a cc-point or r-point, since the same ccp-id or rp-id will appear in loops and other repeated program sequences. The Decision Function is used to determine a single decision from the multiple results. The Local Executive and Global Executive are responsible for fault detection, error recovery, and reconfiguration.

3.2.2.1 The Decision Function

The Decision Function is used to determine a single decision result from the N-version results (cc-vectors or version states). In the case that a decision result cannot be determined, a higher level recovery procedure may be invoked.

In DEDIX we have implemented a generic Decision Function which may be replaced by user written routines provided that the interfaces are preserved. This allows application-specific decision algorithms to be incorporated in those cases where the default mechanisms are inappropriate; for example, this may occur because of lack of sensitivity, or unnecessary elimination of program versions.

The generic Decision Function is hierarchical in nature. The algorithm attempts to determine a decision by applying the following major decision classes sequentially:

1. *bit by bit* - identical match only;
2. *cosmetic* - detecting character string differences caused by misspelling or character substitution;
3. *numeric* - integer and real number decisions.

All numeric decisions use the median value of the version results as the decision result; It can be proved that, so long as the majority of versions are not faulty, the median of all results is acceptably close to a supposed ideal value [Aviz85b]. Numeric values are allowed to be different within some "skew interval," thus allowing results to be non-identical but still similar.

3.2.2.2 The Local Executive

The Local Executive is activated when the Decision Function indicates that a cc-point decision is not unanimous, or when some unrecoverable exception is signaled from the local version or some other layer. The actions to handle a cc-point by the Local Executive are shown in Figure 3-3. If the cc-point decision is no majority, then

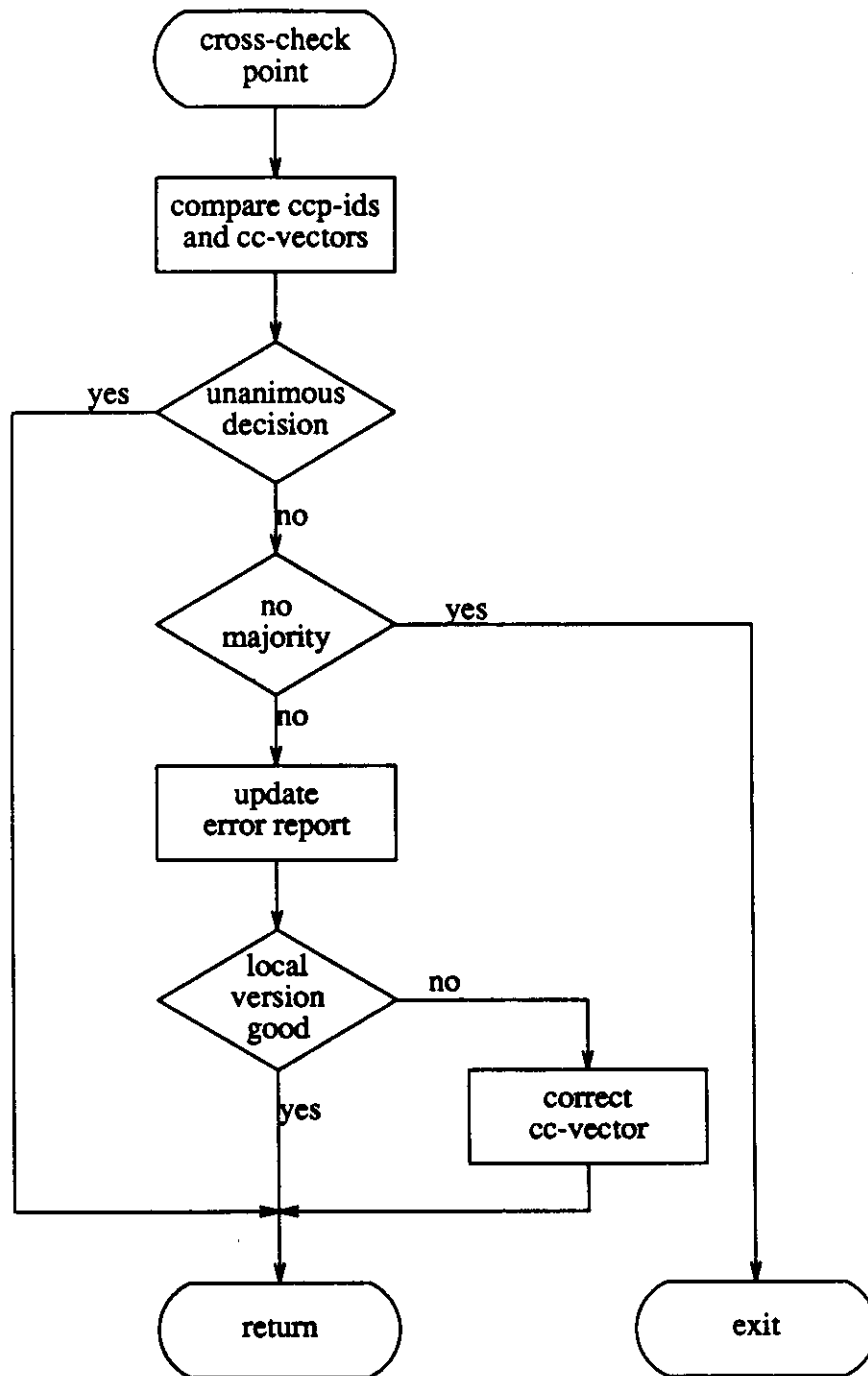


Figure 3-3: Flow Diagram of the Local Executive for CER

DEDIX will terminate without further execution. Otherwise if there is any failed versions, each Local Executive updates its error report. Each Local Executive has an error report table, with one entry per site. Each entry is an error counter of a site. The Local Executive increments the counter when that site has either a disagreeing or missing cc-vector. Local error recovery is accomplished by passing back the decision result to the failed versions.

3.2.2.3 The Global Executive

The Global Executive is activated when a r-point is executed. As shown in Figure 3-4, it performs the following actions to determine if global recovery is necessary: 1) compares the rp-ids delivered by the versions, 2) collects error reports from the Local Executives, 3) exchanges error reports with other Global Executives, and 4) determines failed versions. Since each site might get different numbers of cc-vectors due to varying communication delays, the sites may have somewhat different error reports. The exchange and comparison of error reports ensure a consensus among the sites on failed versions.

If no failed version is detected, the Global Executive merely resets the error report table and the versions continue their execution. Otherwise, global error recovery is initiated.

Two types of failed versions are distinguished: 1) those have errors detected at the cc-points, and 2) those have incorrect or missing rp-ids. Each Global Executive of the good versions signals the *state-output exception handler* of its local version to output the internal state at that rp-id. These states are compared by the Decision Function to obtain the decision state. Each failed version of the first type is recovered

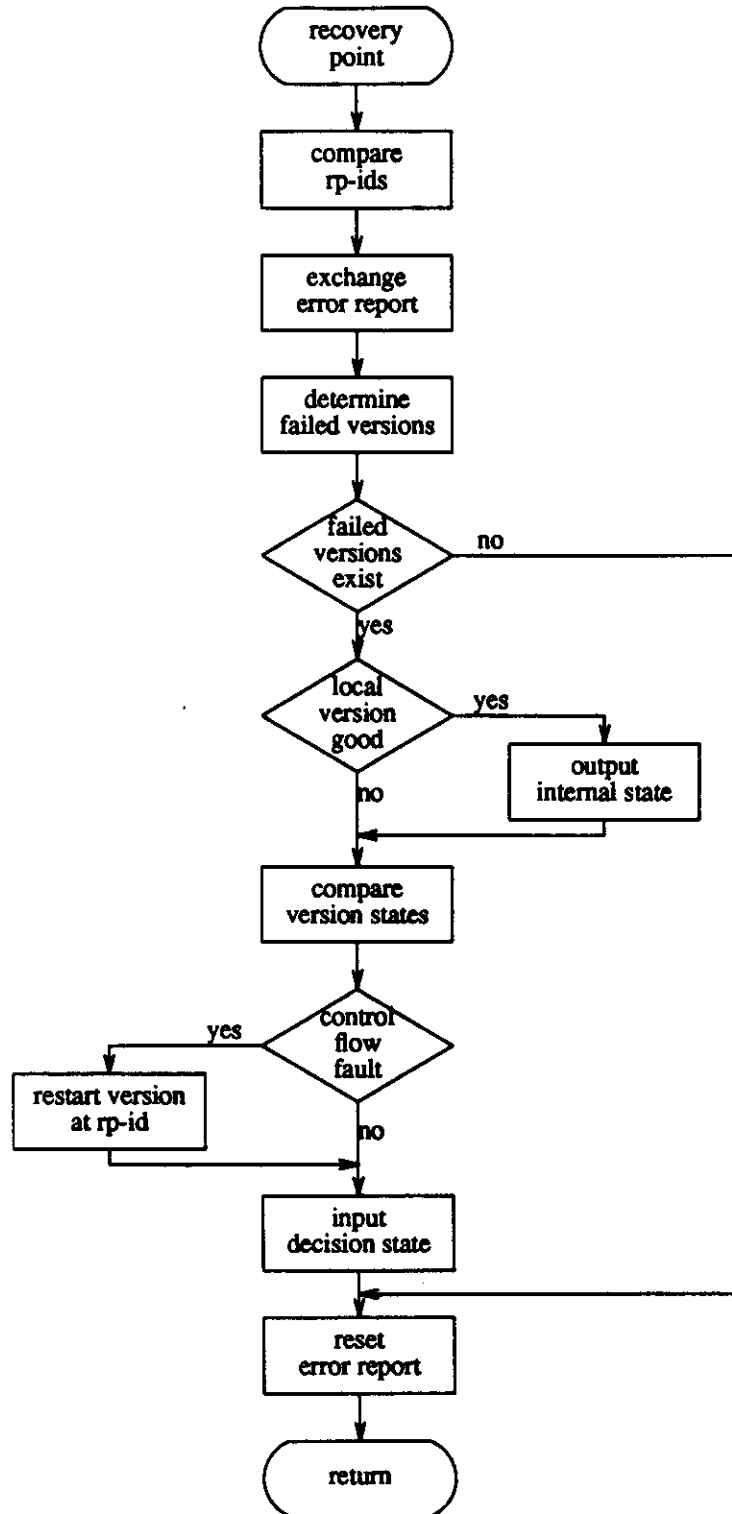


Figure 3-4: Flow Diagram of the Global Executive for CER

by invoking its *state-input exception handlers* to input the decision state. After the exchange of internal states, actions of the global error recovery are completed and execution of the versions is resumed. A failed version of the second type is first aborted by its Global Executive. In the current implementation this is accomplished by sending a kill signal (SIGKILL) to the version process. The version is then restarted by its Global Executive at r-point with the decision rp-id. The restarted version also inputs the decision state before its execution is resumed by invoking the state-input exception handler.

3.2.3 The Synchronization Layer

For each physically distributed site, this layer broadcasts results (using the Transport service) and collects messages with the results from all other sites. The layer only accepts results that are both broadcast within a certain time interval and that will arrive within the same time interval. The collected results are delivered to the Decision Function for comparison. The layer accepts a new set of results when every site has confirmed that all or a majority of the previous results have been delivered.

The processors need to be event-synchronized in order to ensure that results from corresponding cc-points or r-points are compared. Otherwise, if two sets of results from two different points are compared, the Decision Function might wrongly conclude that some of the processors are faulty. Traditionally, this synchronization has been obtained by referring to a common clock or set of clocks. The SIFT system is one example of such a clock synchronous system. In SIFT it is predicted *when* the results should be available for a comparison.

The underlying distributed system and the versions have the following characteristics which make the clock synchronous technique difficult to use or impractical in DEDIX:

- the versions have different *execution times* between the cc-points;
- the versions will run *concurrently* with other network activities, which means that processors temporarily can be heavily loaded, and hence prolong the time to execute some versions;
- the communication network has inherently varying transport delays of messages.

A synchronization protocol is designed to provide the service. It ensures that the results that are compared by the Decision Function are from the same cc-point or r-point in each version. The versions are stopped until all of them have reached the same cc-point or r-point, and they are not started again until the results are exchanged and a decision is made. To be able to detect versions that are in an infinite loop and to allow slow versions to catch up, a time-out mechanism is used by the protocol. Details of the synchronization protocol can be found in [Gunn85].

3.2.4 The Transport Layer

This layer controls the communication of messages between the sites. Messages are broadcast to all active sites. The layer makes sure that no message is lost, duplicated, damaged, or misaddressed, and it preserves the ordering of sent messages. A disconnection is reported to the layer above.

Currently, this layer is implemented as a simple ring of point to point links by LOCUS inter-processor pipes. The decision of the initial ring implementation was made due to the limitation of the number of pipes per process in the LOCUS operation system. Nevertheless, it provides us with some determinism in the system which made it much easier to observe and debug the Transport Layer.

3.3 Program Interface

In MVS the versions of an application program are all written according to the same functional specification. The specification must dictate not only the overall input-output transformation the program has to perform, but also which intermediate results must be compared, and at which points in the execution. The difference between a non-redundant program and the corresponding MVS program version running on DEDIX is minimized for programmers. Figure 3-5 and 3-6 shows a sample program written in Pascal [Joy86] and its corresponding instrumented version.

The program consists of two modules: 1) RSDIMU which consists of five computations (specified in the file `rsdimu.h`) for estimating the aircraft acceleration, is also shown in the figure; 2) MODULE2 for computations specified in `module2.h`. The declarations of constants, types, and variables of the program is in the files `consts.h`, `types.h`, and `vars.h` respectively.

The differences between the program and the version are as follows:

1. A cc-point function is inserted after each computation in the RSDIMU module to cross-check results of the computation and to recover errors in the results. The first argument of each cc-point function specifies the cc-point id. The

```

program avionics (input, output);

#include "consts.h"
#include "types.h"
#include "vars.h"
#include "rsdimu.h"
#include "module2.h"

procedure RSDIMU;
begin
    CALIBRATION;
    SCALE;
    FAULTDETECTION;
    ESTIMATION;
    DISPLAY;
    writeln ('Display Outputs = ', dismode,
            disupper[1], disupper[2], disupper[3],
            dislower[1], dislower[2], dislower[3]);
end;

begin
    RSDIMU;
    MODULE2;
end;

```

Figure 3-5: A Sample Program

```

#include "consts.h"
#include "types.h"
#include "vars.h"
#include "rsdimu.h"
#include "module2.h"
#include "dedix.h"

procedure RSDIMU;
begin
  CALIBRATION;
  ccpoint (1, '%8K%8E%8c', skew, linoffset, linnoise);
  SCALE;
  ccpoint (2, '%8K%8E', skew, linout);
  FAULTDETECTION;
  ccpoint (3, '%d%8c', sysstatus, linfailout);
  ESTIMATION;
  ccpoint (4, '%c%3K%3E%c%3K%3E%c%3K%3E%c%3K%3E%c%3K%3E%4c',
    bestest.status, skew, bestest.acceleration,
    chanest[0].status, skew, chanest[0].acceleration,
    chanest[1].status, skew, chanest[1].acceleration,
    chanest[2].status, skew, chanest[2].acceleration,
    chanest[3].status, skew, chanest[3].acceleration,
    chanface);
  DISPLAY;
  ccoutput (5, '%S%d%3d%3d', 'Display Outputs = ',
    dismode, disupper, dislower);
end;

procedure version (rpid: integer);
label 100, 101;
begin
  case (rpid) of
    0:
      goto 100;
    1:
      goto 101;
  end;
100:
  RSDIMU;
  rpoint (1);
101:
  MODULE2;
end;

```

Figure 3-6: An Instrumented Sample Program

second is the format string which specifies the sizes and formats of the variables participated in the voting, and the decision algorithms to be used. The *format specifiers* %8K%8E specify that the array of eight double precision real numbers are compared with the corresponding supplied skew values. The format specifiers %d and %c are used for voting integers and Booleans respectively.

2. Instead of using `writeln` function for output, the `ccoutput` function is used which first votes on the output values and then outputs them. The format specifier %S specifies that the string can tolerate "cosmetic" error (described in Section 3.2.2.1).
3. The main body of the program has become a version procedure with an argument `rpId`.
4. A `r-point` with an argument `rpId` is inserted between the two modules for global recovery. The `rpId` will be used to determine the module to be restarted in a failed version, and used by the exception handlers to input and output the current internal state.
5. At the beginning the of the version procedure is a `case` statment which decides which module is to be executed according to `rpId`. This is used to restart a version after a control flow fault.

The state-input and state-output exception handlers of the instrumented sample program are shown in Figure 3-7. Inside each exception handler is a `case` statment which determines the internal state to be input or output according to `rpId`. Since a state may consist of many variables, functions `sinput` and `soutput` are supplied by DEDIX to help the programmers to assemble a complete internal state. The format

```

procedure stateinput (rpid: integer);
var
    i: integer;
begin
    case (rpid) of
    1:
        begin
            sinput (rpid, '%A%8K%8E%8c', skew, linoffset, linnoise);
            sinput (rpid, '%8K%8E', skew, linout);
            sinput (rpid, '%d%8c', sysstatus, linfailout);
            sinput (rpid, '%c%3K%3E',
                bestest.status, skew, bestest.acceleration);
            for i := 1 to 4 do
                sinput (rpid, '%c%3K%3E',
                    chanest[i].status, skew, chanest[i].acceleration);
            sinput (rpid, '%4c', chanface);
            sinput (rpid, '%d%3d%3d', dismode, disupper, dislower);
            sinput (rpid, '%4c%3K%3E%Z', statebadface, skew, statefhat);
            end
        end
    end;

procedure stateoutput (rpid: integer);
var
    i: integer;
begin
    case (rpid) of
    1:
        begin
            soutput (rpid, '%A%8K%8E%8c', skew, linoffset, linnoise);
            soutput (rpid, '%8K%8E', skew, linout);
            soutput (rpid, '%d%8c', sysstatus, linfailout);
            soutput (rpid, '%c%3K%3E',
                bestest.status, skew, bestest.acceleration);
            for i := 1 to 4 do
                soutput (rpid, '%c%3K%3E',
                    chanest[i].status, skew, chanest[i].acceleration);
            soutput (rpid, '%4c', chanface);
            soutput (rpid, '%d%3d%3d', dismode, disupper, dislower);
            soutput (rpid, '%4c%3K%3E%Z', statebadface, skew, statefhat);
            end
        end
    end;
end;

```

Figure 3-7: Exception Handlers of the Instrumented Sample Program

specifiers %A and %Z respectively designate the beginning and end of the state. Other format specifiers have the same meaning as those in cc-point functions.

3.4 User Interface

As an experimental testbed, DEDIX provides a flexible interface for the users to design, implement, and evaluate multi-version software. It allows users to debug the system as well as the versions, monitor the operations of the system, apply stimuli to the system, and to collect empirical data during experimentation. A number of commands are available to the user for controlling the execution and defining additional output.

1. Breakpoint

The *break* command enables the user to set breakpoints. At a breakpoint, DEDIX stops executing and goes into the user interface where the user can enter commands to examine the current system states, examine past execution history, or inject stimuli to the system. Some examples of using the break commands are as follows: The command

```
break no-majority
```

instructs the user interface to stop when there is no majority in a cc-point decision. The command

```
break cc-point-id = 3
```

stops DEDIX when the third cc-point is executed.

2. Monitoring

The user can examine the current contents of the message passing through the transport layer by using the *display* command. Since every message is logged, the user may also specify conditions in the *display* command to examine any message logged in the past. The user can also examine the internal system states by using the *show* command, e.g., to examine the breakpoints which have been set, the results of voting, etc. Examples are:

```
display sender = 1, cc-point-id = 2
```

displays the message sent by version 1 at the second cc-point on the screen.

The command

```
show decision-vector
```

displays the result of comparison.

3. Stimuli Injection

The user is allowed to inject faults to the system by changing the system states, e.g., the cc-vector, by using the *modify* command. An example of using the command is

```
modify cc-vector
```

which changes the values of the cc-vector.

4. Statistics Collection

The user interface gathers empirical data and collects statistics of the experiments. Every message passing the transport layer is logged into a file with a time-stamp. This enables the user to do post-execution analysis or even replay the experiment. Statistics like elapsed time, system time, number of cc-points executed, and their results of decision are also collected.

3.5 Input/Output System

The input/output system for the versions is designed to be replicated as well. However, in the current implementation a centralized terminal connection is used for all input/output. The site designated for input/output is called the *primary site*. The primary site collects results to be printed from all versions. These version results are run through the Decision Function and the decision result is output only if there is majority in the comparison. The request to read data is also run through the Decision Function to ensure consistency. The primary site compares the cc-ids and the format strings sent from all versions. If majority exists, the data will be input according to the decision format string and the data are then distributed to all versions. The interface between DEDIX and the input/output system is similar to the interface between DEDIX and the local version. For example, a read from a terminal might be timed-out if it does not respond in phase with the other terminals. If errors are detected in the input/output system of the primary site, another site will be selected to be the primary site for input and output.

3.6 Reconfiguration

CER can recover versions from transient and design faults. If the error is caused by a permanent physical fault, CER will not be useful since the version will continue to produce errors due to the physical fault. After a version has produced errors at several consecutive r-points, a physical fault is likely to have occurred, and reconfiguration is used to repair the system.

Reconfiguration techniques have been used to repair systems from physical faults successfully in a number of systems [Toy78, Hopk78, Wens78]. The techniques basically involves disconnecting the faulty channel from the system, replacing it with a standby spare or rescheduling its tasks to the remaining components.

The state of the system after the reconfiguration may be classified as [Aviz78]:

1. *Fully recovered* - the system returns to the state that existed before the fault occurred.
2. *Degraded* - it returns the system to a fault-free state but with a reduced computing capacity.
3. *Safely shutdown* - the system is degraded to a state with no computing capacity.

If a multi-version software system has more channels than the number of independent versions, reconfiguration is done by migrating the version in the faulty channel to a redundant one for full recovery. The migrated version is brought to the current state obtained from the decision state of the good versions. If there is no redundant channel, depending on the situation, the system can be degraded in one of

the following three ways:

1. The system is degraded such that there is one less version. If the system was running with more than three versions, majority decisions can still be obtained from the remaining versions.
2. The system is degraded such that there is one less channel, but the version of the disconnected channel is migrated to another functioning channel. Of course, a channel that runs two versions will take longer to produce both results.
3. This is similar to the first one, but instead of throwing away a version, that version is kept as a standby. Its state is updated at r-points with the decision state so that it can be brought into use whenever the two remaining versions disagree.

The first approach lowers the reliability of the system, since fewer versions are used in the comparison. The second reduces the computing capacity of the system. The choice depends on the overall system reliability requirement, real-time constraint, and available computing resources.

3.7 Future Improvements of DEDIX

DEDIX can be improved in the following ways in order to make it able to tolerate hardware faults, more flexible in comparison of results, and to improve performance for extensive testing:

1. The simple ring structure currently implemented in the Transport Layer does not allow a site crash. A redundant interconnection structure should be introduced so that reconfiguration can be possible when physical faults occur on the links or sites.
2. The current Decision Function treats a cc-vector as a single result in the comparison. Under this implementation, a 3-version MVS system reaches no consensus if two versions fail on different variables, despite the fact that there is a consensus on each variable. DEDIX should provides options for the users to choose the granularity of the comparison. The Decision Function should also allow user-specific decision algorithms being used, such as adaptive voting [Broe75], or utilize only a subset of all N results for a decision, for example, the first result that passes an acceptance test.
3. DEDIX is intended to provide two distinct research and utilization opportunities. First, it serves as an experimental vehicle for fault-tolerant design investigations; and second, it will be a very high-reliability and continuous availability system for users who have a need for exceptionally reliable computing. Experiences on the use of DEDIX showed that these two goals are not compatible. The current DEDIX has too much overhead on the Transport and Synchronization layers because of the fault-tolerant provisions, making it too slow for extensive testing on the versions. Other configuration of DEDIX should be investigated for a highly efficient testbed.

CHAPTER 4

RELIABILITY MODELS FOR MULTI-VERSION SOFTWARE

A number of analytic models have been proposed for evaluating reliability of fault-tolerant software. In [Gma80], the performances of both the multi-version software and the Recovery Blocks software fault-tolerant strategies were evaluated using queueing models. The results showed that the models can be useful for system designers in deciding which of the strategies should be used, depending on system parameters. Laprie [Lapr84] has developed a Markov model which enables one to account for the failures due to design faults in evaluating the reliability of a system. The model has been applied to evaluating the reliability of the recovery block scheme as an example. A probabilistic model has been developed by Eckhardt and Lee for analyzing the effectiveness of multi-version software subjected to coincident errors [Eckh85].

The reliability models developed in this chapter for the multi-version software scheme follow the principles of Laprie's work. The models account for the failures of the underlying system that executes the diverse versions and the CER scheme for the error recovery. Two reliability models are developed according to the operation of DEDIX, one without recovery and the other with recovery. The models are based on the execution of three program versions on DEDIX.

4.1 Execution Model of MVS on DEDIX

The model for the execution of the diverse versions is shown in Figure 4-1. Each version communicates with its local copy of DEDIX, which exchanges information with the other DEDIXs through the services provided by the underlying computer system. Without CER, a version sends the computed result to its local DEDIX at cc-points. The results are exchanged and compared. Faults in minority versions are masked through the comparison. However, without CER, the failed versions are not recovered. With CER, a version sends the computed result to its local DEDIX at cc-points. The results are exchanged and compared. The decision result is sent back to the failed versions for recovery. At recovery points, if failures are detected in any versions at the cc-points, the internal states of the good versions are exchanged and compared. Then the decision state is sent to the failed versions for recovery.

4.2 Types of Errors

In multi-version software, the functional requirements of an application are determined by a specification from which the independent programming teams implement the program versions. Hence an error in the specification, labeled as $CE(V_1, V_2, V_3)$ for a *similar error* of the three versions, may lead to system failure. Each programming team may make independent errors, labeled as $IE(V_i)$ for an *independent error* in version i , during the design and implementation of the version. During the execution of the versions, DEDIX may have an error, $CE(D)$, for common-mode failure caused by DEDIX, which leads to failure of the system. The recovery mechanism may not be successful, so that a failed version is not recovered.

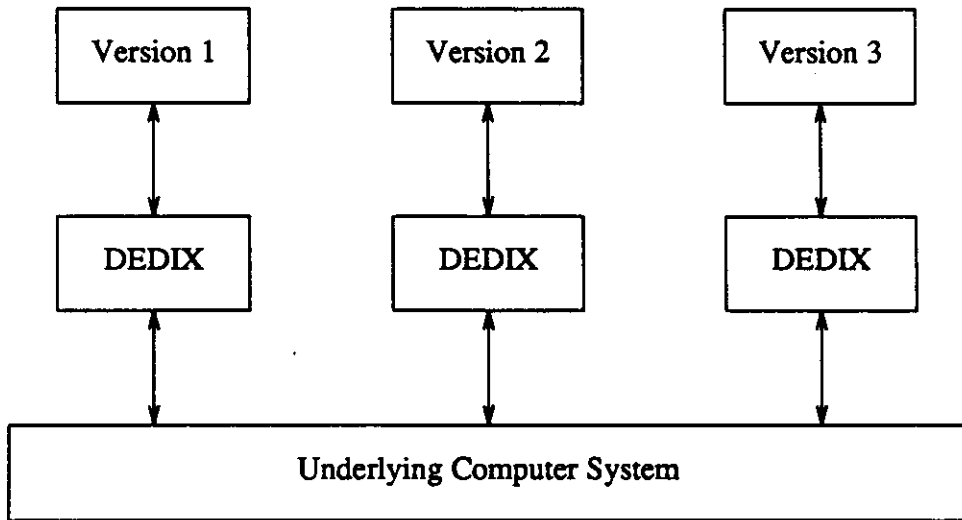


Figure 4-1: Execution Model of MVS on DEDIX

This failure is labeled as IE(R). It should be noted that independent errors of DEDIX, or IE(D), may occur in the system if 1) each site has an independently implemented DEDIX system, or 2) Heisenbugs [Gray86] occur, which by their nature appear to be randomly distributed across sites. However, this type of error is not included in this analysis.

4.3 Basic Assumptions

The following is assumed in developing the models.

1. The errors caused by the underlying system (hardware, operating system, etc.) on which DEDIX and the versions are executed are not included in this analysis.
2. Versions are implemented in modules with cross-check points or recovery points placed between them. To simplify the analysis, both the execution durations and the failure rates of each module of the versions are the same.
3. Versions interact with DEDIX through cross-check points and recovery points. We do not distinguish between cross-check points and recovery points, i.e., they have the same execution durations and failure rates.
4. If recovery is unsuccessful, the failed version is assumed to produce erroneous results in subsequent computations. This is a pessimistic assumption since versions that fail at one cc-point may produce correct results at later cc-points.

4.4 Reliability Model Without Recovery

In this model, DEDIX is assumed to provide only the communication and decision functions for the three versions to run in parallel and to compare results for masking faults. Figure 4-2 shows the detailed state diagram of the reliability model. Initially, the system is in the idle state (I) with zero failure rate and idle duration $1/\eta$. The versions are executed (V) with an execution duration $1/\gamma_V$ until a cc-point is reached. Their results are compared by DEDIX (D) with a duration $1/\gamma_D$. The versions either continue the computation or go to the idle state after the comparison. The activation ratio q is the probability that the versions will be resumed after a cc-point. During the execution, independent errors may occur in the versions at a rate λ_V . If there is only one version that has an error at a cc-point, the system will be degraded to two versions (the 1 IE(V) states). Independent errors in more than one version, common-mode errors of the versions with a rate λ_C , or failure of DEDIX with a rate λ_D , all lead to failure of the system (CF).

By merging common states and considering the fact that the execution rates are much larger than the failure rates, the state diagram is simplified to Figure 4-3.

4.5 Reliability Model With Recovery

In this model DEDIX is assumed to provide recovery to failed versions through the cc-points and recovery points. A detailed state diagram of the reliability model of three versions running on DEDIX with recovery is not shown here. Figure 4-4 shows the simplified state diagram. It is similar to the one without recovery except that here DEDIX attempts to recover a failed version if an error occurs. Unsuccessful recovery of one version will leave the system with two good versions.

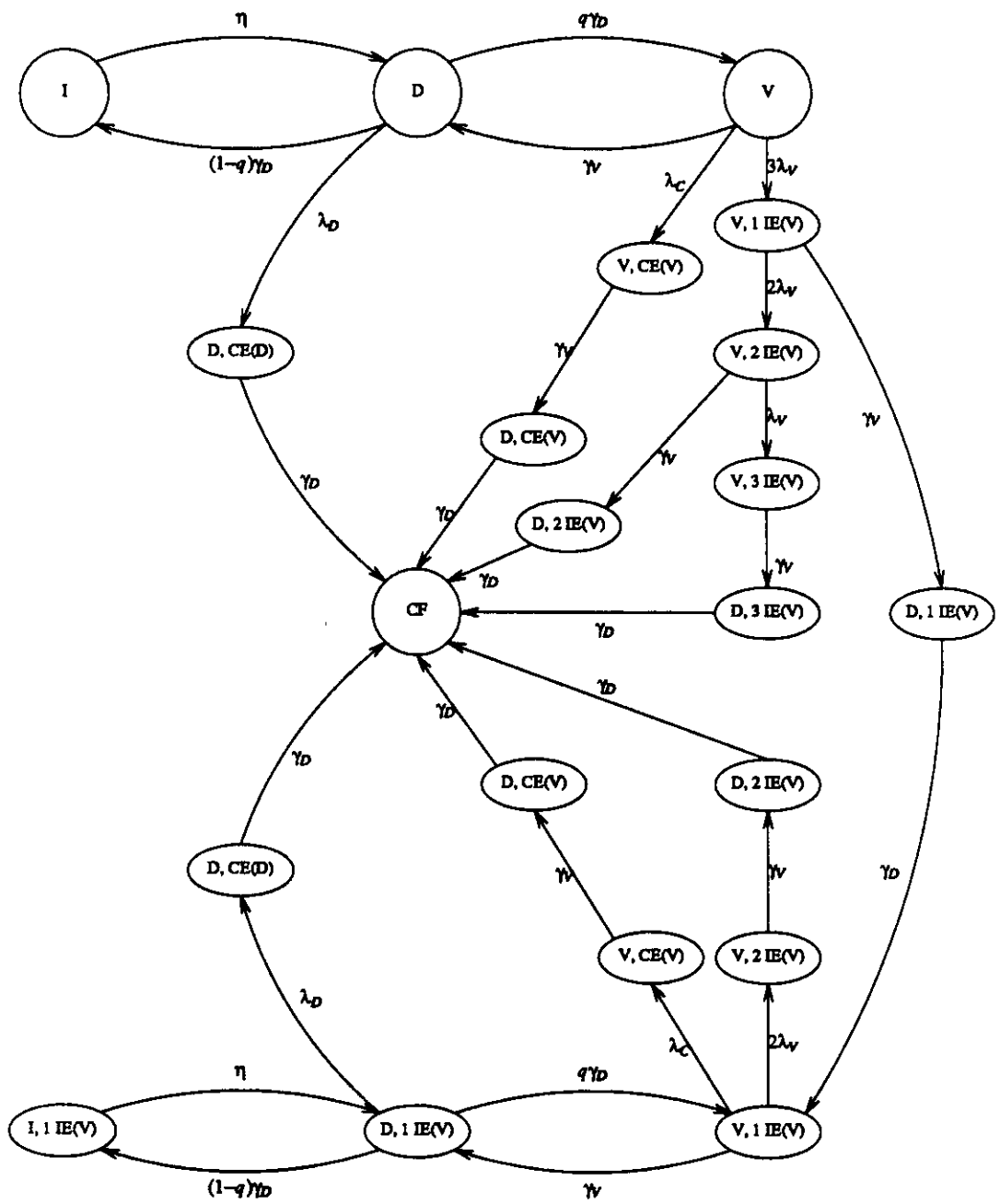


Figure 4-2: Detailed State Diagram of the Reliability Model Without Recovery

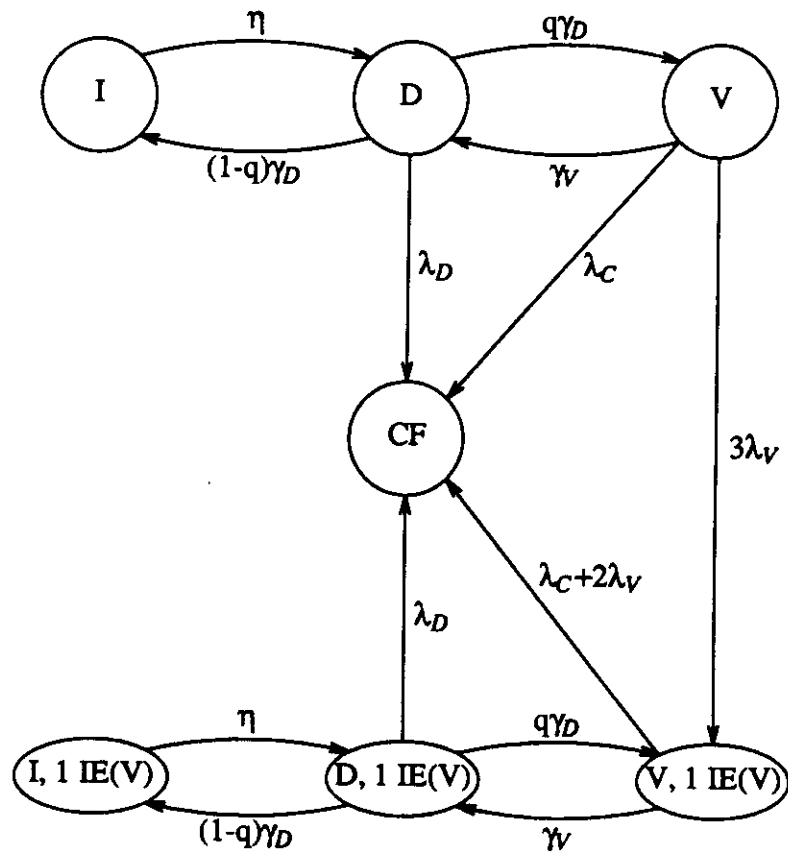


Figure 4-3: State Diagram of the Reliability Model Without Recovery

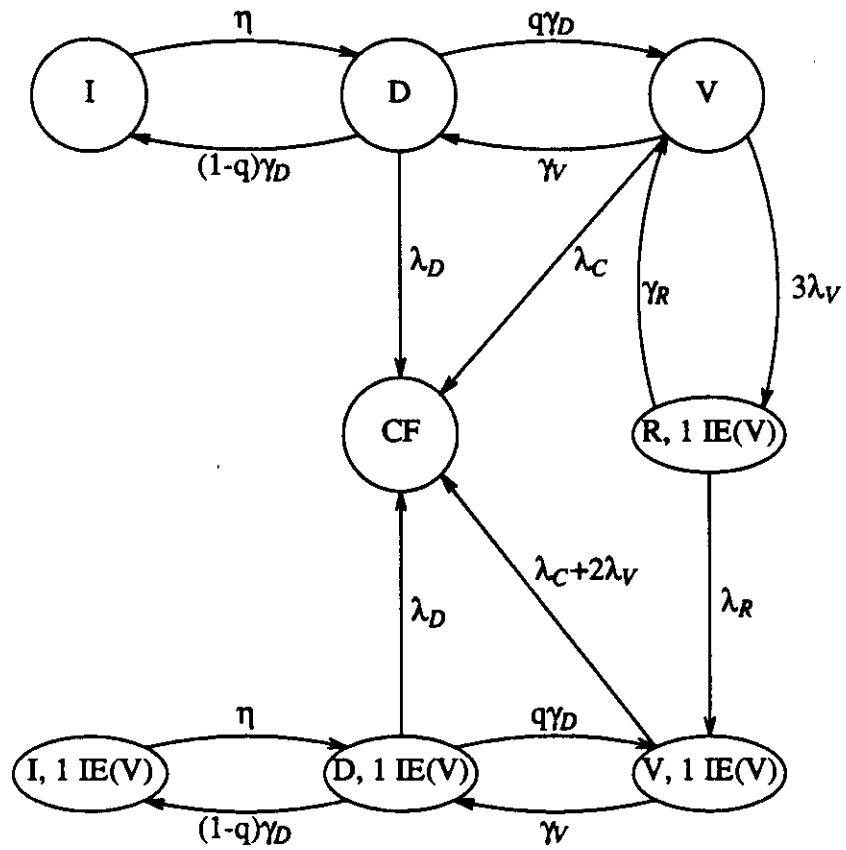


Figure 4-4: State Diagram of the Reliability Model With Recovery

4.6 ARIES Evaluation

The models developed in Figure 4-3 and Figure 4-4 are evaluated using ARIES 82 (Automated Reliability Interactive Estimation System) [Maka82], a reliability estimation tool for fault-tolerant systems developed at UCLA. ARIES 82 supports the evaluation of models for four specific types of fault-tolerant systems, namely, closed systems, repairable systems, systems with transient fault recovery, and periodically renewed closed systems. In addition, it allows users to evaluate more complex systems which can be modeled under finite-state homogeneous Markov processes. Our evaluation makes use of this additional feature.

4.6.1 State Transition-Rate Matrices of the Models

The state transition-rate matrix for the model without recovery is as follow:

$$\begin{pmatrix} -\eta & \eta & 0 & 0 & 0 & 0 & 0 \\ \gamma_D(1-q) & -\lambda_D-\gamma_D & \gamma_Dq & 0 & 0 & 0 & \lambda_D \\ 0 & \gamma_V & -\gamma_V-\gamma_C-3\lambda_V & 3\lambda_V & 0 & 0 & \lambda_C \\ 0 & 0 & 0 & -\gamma_V-\lambda_C-2\lambda_V & \gamma_V & 0 & \lambda_C+2\lambda_V \\ 0 & 0 & 0 & \gamma_Dq & -\lambda_D-\gamma_D & \gamma_D(1-q) & \lambda_D \\ 0 & 0 & 0 & 0 & \gamma_I & -\gamma_I & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The state transition-rate matrix for the model with recovery is as follow:

$$\begin{pmatrix} -\eta & \eta & 0 & 0 & 0 & 0 & 0 & 0 \\ \gamma_D(1-q) & -\gamma_D-\lambda_D & \gamma_Dq & 0 & 0 & 0 & \lambda_D & 0 \\ 0 & \gamma_V & -\gamma_V-\lambda_C-3\lambda_V & 3\lambda_V & 0 & 0 & 0 & \lambda_C \\ 0 & 0 & \gamma_R & -\gamma_R-\lambda_R & \lambda_R & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\gamma_V-\lambda_C-2\lambda_V & \lambda_V & 0 & \lambda_C+2\lambda_V \\ 0 & 0 & 0 & 0 & \gamma_Dq & -\gamma_D-\lambda_D & \gamma_D(1-q) & \lambda_D \\ 0 & 0 & 0 & 0 & 0 & \gamma_I & -\gamma_I & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The following parameter values are used in the evaluation:

Version Failure Rate	$\lambda_V = 1 / \text{hour};$
Correlated Failure Rate	$\lambda_C = 0.01;$
DEDIX Failure Rate	$\lambda_D = 0.01;$
Recovery Failure Rate	$\lambda_R = 0.01 \text{ and } 1;$
Version Execution Duration	$1/\gamma_V = 0.01 \text{ hour};$
DEDIX Execution Duration	$1/\gamma_D = 1/\gamma_V;$
Recovery Execution Duration	$1/\gamma_R = 1/\gamma_D;$
Idle Duration	$1/\eta = 1/\gamma_V * 10;$
Activation Ratio	$q = 0.9.$

Figure 4-5 plots reliability against time for the models with and without recovery. The graphs show that three versions running on DEDIX in a multi-version software configuration have a higher reliability than a single version. With recovery,

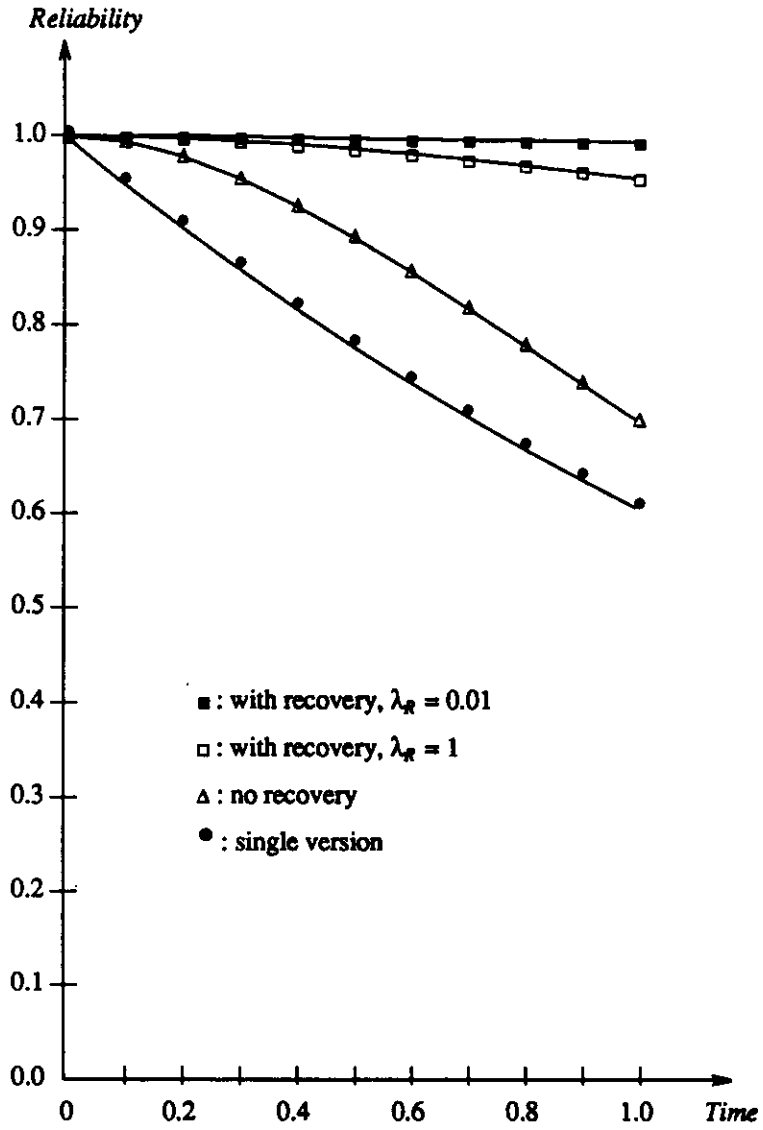


Figure 4-5: Reliability vs. Time with $\lambda_Y = 1$

even with a relatively high recovery failure rate, the reliability is much higher than without recovery. When the failure rate of the versions, λ_V , increases, as shown in Figure 4-6, the reliability of a three version multi-version software system without recovery becomes lower than a single version; however, the reliability is still good with recovery.

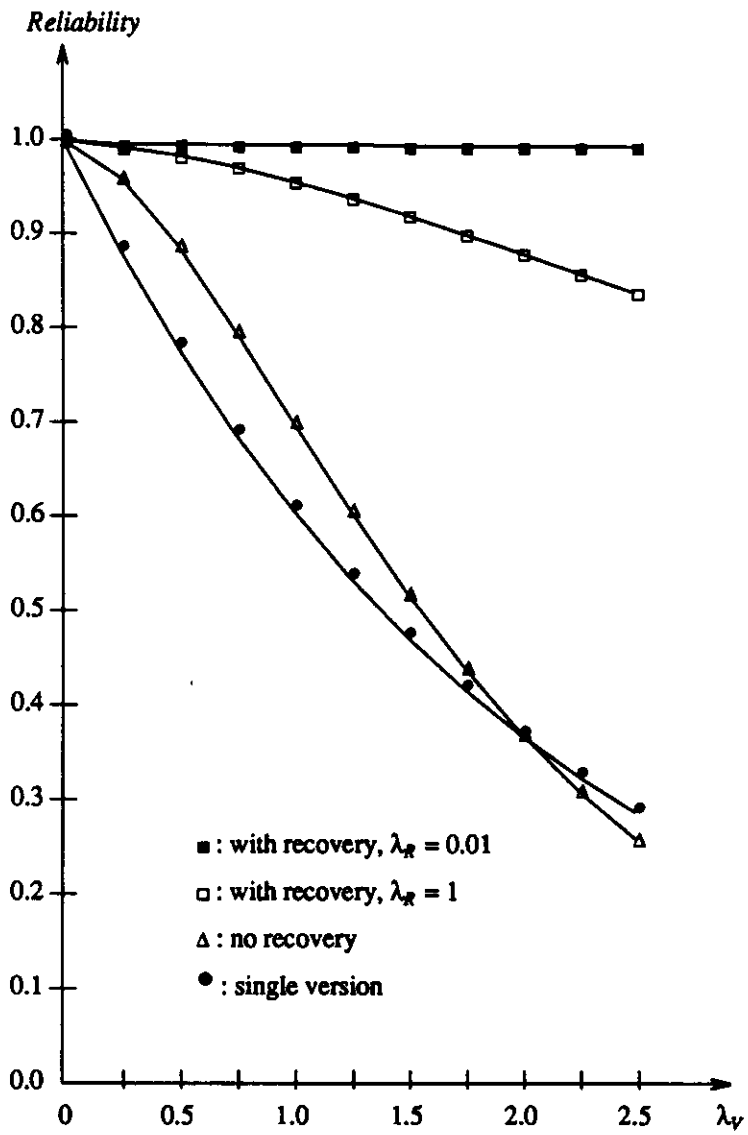


Figure 4-6: Reliability vs. λ_V with Time = 1

CHAPTER 5

EXPERIMENTATION ON MULTI-VERSION SOFTWARE

5.1 Previous Experiments on MVS

A number of experiments to investigate the role of multi-version software in the production of fault-tolerant software have been performed at UCLA since 1975.

5.1.1 Zero Generation

The objectives of this initial MVS research effort at UCLA (1975-1978) were: 1) to study the feasibility and effectiveness, and 2) to identify problems or difficulties in using MVS as a means to tolerate design faults. Two experiments were performed. The first involved 27 diverse programs of a text editor and the second used three different algorithms to enforce another level of diversity among the resulting 16 programs of a scientific application. Both applications were specified in English and programmed in PL/1. These exploratory research demonstrated the practicality of the MVS approach and the need for high quality software specifications [Chen78b, Chen78a].

5.1.2 First Generation

The objectives of the next phase of UCLA research (1979-1982) concentrated on the investigation of the relative applicability of various software specification techniques. Three specification languages were used to compare the effects of a formal specification written in OBJ [Gogu79], a non-formal specification written in PDL [Cain75], and a "control" specification written in English. Eighteen programs of an "airport scheduler" application were developed with an average length of about 500 lines of PL/1 code. The research concluded that the MVS approach is a viable supplement to fault avoidance and removal techniques for producing highly dependable software. It was also found that specification errors are the most serious because they can lead to similar errors in the final program versions [Kell82, Kell83].

5.2 The NASA-Four University Multi-Version Software Experiment

Encouraged by the results from the previous experiments in fault tolerant software, the NASA Langley Research Center began funding the NASA-Four University Multi-Version Software Experiment in 1984 [Kell86]. This large-scale experiment was designed to evaluate the performance of the MVS approach to fault-tolerant software in a realistic aerospace application developed under a controlled software development process which is indicative of industry practice. The goals include:

1. *Reliability assessment.* To obtain empirical estimates of the reliability of the programs both executed individually and in multi-version configurations in order to make quantitative assessment of the effects of multi-version systems on overall reliability.

2. *Characterization of faults.* To determine what faults the programs contain in order to facilitate their avoidance during development and avoidance during testing.
3. *The role of error recovery.* To evaluate the effectiveness of the CER method to recover errors of the failed versions.

The NASA-Four University Multi-Version Software Experiment involves four universities, University of California at Los Angeles, the University of Illinois at Urbana-Champaign, North Carolina State University, and the University of Virginia, as well as the Research Triangle Institute (RTI), and Charles River Analytics (CRA). The specifications were written by RTI and CRA. Tools, the preliminary acceptance test and experimental coordination were provided by RTI. CRA has now assumed the role of customer and specification arbiter, and is providing flight simulation test data. Each university employed ten graduate students to generate five program versions. In order to ensure the versions were independently generated, people involved in the design and coordination of the experiment did not participate in the programming effort.

The experiment proceeded in several phases:

1. *Choice of a suitable application.* The application was chosen that is both realistic in industry and appropriately sized for the experiment.
2. *Specification of the problem.* The specification not only specifies the functionality of the application, it includes a description on the CER requirements.
3. *Recruiting Programmers.* All the programmers recruited are graduate students

with a few years experience in program development.

4. *Generation of redundant software versions.* Ten weeks were allowed to the programmers to design, code, and debug their programs, and concluded with the pass of a preliminary acceptance test.
5. *Validation testing.* The resulting versions were subjected to extensive preliminary testing. During validation, many errors and ambiguities in the specification were revealed and subsequently the specification was improved.
6. *Maintenance and Certification.* The initial versions were maintained according to the revised specification and passed a much more stringent certification test.
7. *Final analysis.* The final versions will be analyzed according to the objectives of this experiment.

5.3 Program Generation

Five independent programming teams at each of the four universities generated software from a common specification. A controlled software development process, uniform across all four universities, was designed to reflect standard industry practice. Additionally, programmers were not permitted to discuss any aspect of their work with members of other teams. Work-related communications between programmers and a central project coordinator (specification expert) were conducted via electronic mail. Copies of each question and answer pair were locally rebroadcast to all programming teams.

The experiment included ten weeks for software generation. These were organized into five phases:

1. *Training.* The programmers attended a brief training meeting. An introductory presentation was made summarizing the experiment's goals, requirements and the multiple version software techniques. At this meeting, the programmers were given written specifications and documentation on system tools.
2. *Design.* At the end of this four-week phase, each team delivered a design document following guidelines provided at the training meeting. Each team delivered a design walk-through report after conducting a walk-through which was attended by silent observers including the site's principal investigator.
3. *Coding.* By the end of this 2-week phase, programmers had finished coding, conducted a code walk-through and delivered a code walk-through report.
4. *Testing phase.* Each team was provided four sample test data sets. No two teams received the same test cases. Two weeks were allotted to this phase.
5. *Preliminary acceptance test.* Programmers formally submitted their programs. Each program was run in a test harness. When a program failed a test it was returned to the programmers with the input case on which it failed, for debugging and resubmission. By the end of this two week phase, all twenty programs had passed this preliminary acceptance test.

5.4 Application: The RSDIMU

A Redundant Strapped Down Inertial Measurement Unit (RSDIMU) is part of an integrated avionics system [CRA85]. This unit contains eight linear sensors mounted on the four triangular faces of a semioctahedron. Each sensor measures the component of Estimated Acceleration along its axis. This fault tolerant configuration requires a special component to manage sensor redundancy and to reconfigure the system in the event of sensor failures. It uses the *Edge Vector Test* to detect a failed face during the flight and then uses the *Least Square Algorithm* to isolate the failed sensor on that face.

A significant amount of linear algebra, particularly matrix transformations, is involved. There are eleven reference frames of interest (coordinate systems corresponding to the earth, vehicle, sensors, etc.), four of which are non-orthogonal. Input to the programs consists of geometric data of the system, calibration data, flags indicating which sensors are already known to have failed, raw data measurements from the eight sensors, etc. They are required first to identify faulty sensors and then to compute a statistical estimate of vehicle acceleration based on the redundant set of operational sensors. Final system status and acceleration estimates are reported by a digital display panel as specified by an input parameter. A block diagram of the major computations is shown in Figure 5-1, which also includes cc-points and the variables of the cc-vectors. The defined Pascal constants, types, and variables of RSDIMU which were provided to the programmers are included in Appendices 1, 2, and 3.

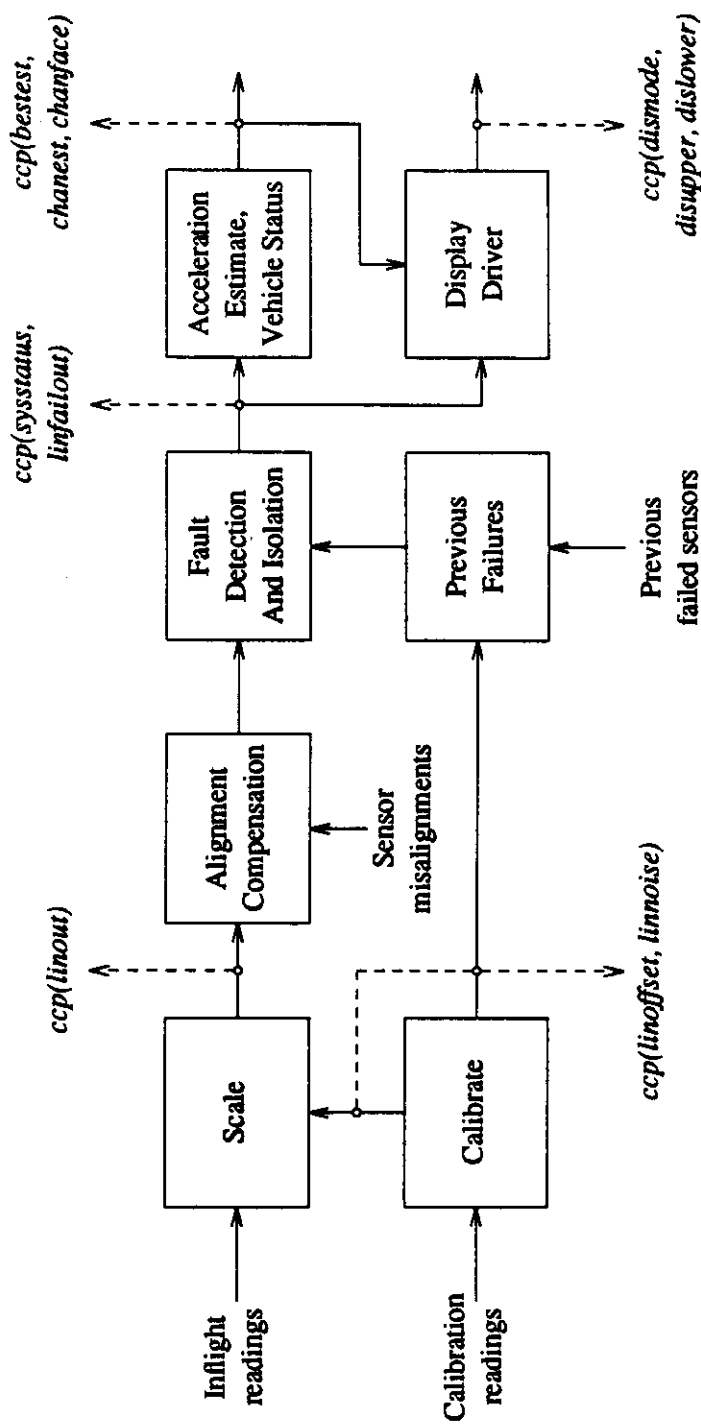


Figure 5-1: Block Diagram of RSDIMU

5.5 Specification of CER in RSDIMU

Four cross-check points were specified in the original RSDIMU specification, as shown in Figure 5-1. The additional cc-point on LINOUT shown in the figure was later added to the revised specification. No recovery points were specified. To avoid restricting design diversity, the decision was made not to tell programmers where to place the cc-points in their programs. The sequence in which the cc-points occurred and the variables involved were specified, and it was required that the variables of each cc-point be computed but not yet used when the cc-point is reached. The programmers were also required to use the (possibly modified) values returned by the MVS Supervisor in all subsequent computations.

5.6 Validation Testing on the Initial Versions

A long and careful validation phase including extensive preliminary testing of the versions followed the 10-week program generation phase. The validation showed that the initial versions obtained were in very poor quality. The results of the preliminary evaluation of the initial versions can be found in [Dora86, Swai86]. The inadequacy of the programs stems largely from the inadequacy of the original specification and from the inadequacy of the preliminary acceptance test.

During program development, the specification was unstable. Questions posed by the programmers were answered by RTI and broadcast to all programming teams as the formal protocol for clarifying the specification. The number of questions (over 250) posed by the 40 programmers was overwhelming. Although most questions derived from only a handful of errors and ambiguities in the original specifications, each was phrased differently so that simple affirmative or negative responses were

interpreted to have extraneous and contradictory ramifications. In the end, the specifications had grown unwieldy and imprecise.

The preliminary acceptance test provided by RTI was grossly inadequate for the following reasons: 1) the number of test cases was insufficient, 2) the test cases represented only a small subset of the entire input domain, and 3) only two out of the eleven output variables were checked for correctness.

The preliminary acceptance test did not test recovery. It ensured that the cc-points were placed in the right sequence, but output values were checked at the end of the execution of each version. Due to this inadequacy, most versions which passed the acceptance test still have faults in the recovery implementation.

The design faults related to cc-points can be classified into two categories:

1. *Incorrectly located cc-points.* Some teams inserted cc-points at the point where a value was first calculated. Under some circumstances, such as the detection of a sensor failure, values would be later revised to reflect the failure. The corresponding cc-points were located too early in the procedure. Some versions were found to use computed values before passing them to the decision function. These cc-points occurred too late.
2. *Unused returned values.* This fault occurred when a version used a local variable in some computation and its value was assigned to the state variable of the cc-vector before the cc-point is called, but subsequent computations still based on the value of the local variable.

5.7 Maintenance and Certification

The errors and ambiguities of the original specification found in the Validation Testing has been eliminated. It has now been restored to a single document, a document that has benefited from the scrutiny of more than 50 motivated programmers and researchers. Furthermore, an attempt has been underway at UCLA to formalize the RSDIMU specification using the LARCH specification languages [Gutt85, Tai86]. Although the formal RSDIMU specification was not used in the Certification, it contributed to the detection and correction of errors and ambiguities in the original English specification. One more cc-point on the variable LINOUT has been added. This will make all the eleven output variables of the application be cross checked and recovery be done better.

Benefited from the validation process, a better acceptance test has been developed for the certification of the versions. The new acceptance test has the following features:

- Versions had to pass 200 test cases randomly selected from a pool of 1000 prepared cases. The test cases sampled a much larger input space.
- 55 hand-made test cases which included Extremal Test Data and Special Test Data [Adri82].
- All output variables were checked and a much tighter skew interval for real numbers was used.
- The output values were checked at the cc-points, which detected the incorrect placement of cc-points. Also, special tests were included that deliberately returned new values to some cc-points. The results of the next cc-point are

then checked to verify the returned values were actually used.

The five UCLA versions were brought in accordance by the original programmers with the revised specification and certified by the final acceptance test.

5.8 Characteristics of UCLA Certified Versions

Table 5-1 shows, for each UCLA certified version, the size of the source program in number of lines of Pascal code, the size of the compiled object in bytes, the compile time in seconds, and the execution time in milliseconds.

Table 5-1: Characteristics of the UCLA Certified Versions

Versions	Source Program (lines of code)	Object Code (bytes)	Compile Time (seconds)	Execution Time (milliseconds)
ucla1	2016	34827	95	340
ucla2	1685	38028	103	290
ucla3	1962	38003	105	300
ucla4	2794	67614	107	360
ucla5	1677	34411	90	220
range	1.67 : 1	1.96 : 1	1.19 : 1	1.64 : 1

The compile times and the execution times were measured on a lightly loaded VAX-11/780 machine. The execution times were the average of 100 runs of test cases identical to each version. It can be seen the size of the largest program is about two third of the smallest one, and the object size varies by the ratio almost 2 to 1. A close look at the implementation revealed that ucla5 is the cleanest and most well structured program. Version ucla4 uses a lot of global variables and lack of structure.

CHAPTER 6

EXPERIMENTAL EVALUATION OF CER

6.1 Testing Strategy for CER

The evaluation of CER with the five UCLA certified versions generated in the NASA-Four University Multi-Version Software Experiment is divided into two parts:

1. *Extensive random testing of cc-points.* A special testing driver was used to test the cc-point recovery implemented in the RSDIMU modules using randomly generated test cases. In total 200,000 test cases were run through the versions with an estimated execution time of 800 hours of a VAX-11/780 computer.
2. *Systematic testing of r-points.* DEDIX was used to evaluate the r-point recovery of instrumented versions each containing a RSDIMU module. Test cases which found one or more failed RSDIMU modules during the extensive random testing were used. Also, verification of the r-point recovery through systematic error seeding was carried out.

The main reason to divide the evaluation into two parts is the use of different tools in the two testing processes. Extensive testing of the versions requires the use of an efficient testing harness. However, DEDIX was designed to be a general purpose MVS supervisor which is too slow for this purpose. Hence a testing driver specially designed for the testing of the RSDIMU modules has been used. DEDIX was used in the testing of r-point recovery because the testing requires a sophisticated MVS

supervisor for state recovery and version restart.

6.2 Testing Driver for CC-Point Recovery

A Test Case Generator (TCG) is used throughout the extensive cc-point recovery testing to generate random test cases. The TCG uses a reverse algorithm which begins by assuming some fixed geometry angles, temperature readings, scales, offsets of the sensors, and the display mode. It randomly selects which sensors will be noisy, which will have failed before the flight, and which will fail during the flight. It then projects a vector representing gravity on each sensor. These projections are converted into values in "counts" with the introduction of noise according to the array of noisy sensors, creating the raw sensor readings at calibration. It also chooses an acceleration vector and projects it onto each sensor, modified with noise according to the selected array of sensors that failed during the flight, creating the in-flight raw sensor readings. Figure 6-1 shows the block diagram of the TCG with the input and output variables indicated. Italics indicate which of the values are randomly selected by the TCG. The TCG has been designed to sample the input space as thoroughly as possible. Discrete variables are selected with all possible combinations, while continuous variables are sampled evenly on their largest possible ranges. With this reverse algorithm, the TCG not only generates the input data, but some *known* output values of the test case are also provided since these values were used by the reverse algorithm to generate the input data.

After the TCG has generated the data for a test case, individual versions are executed consecutively, using the same input data. If a majority of similar results exists, they are used to decide the *reference* output which is further checked by the

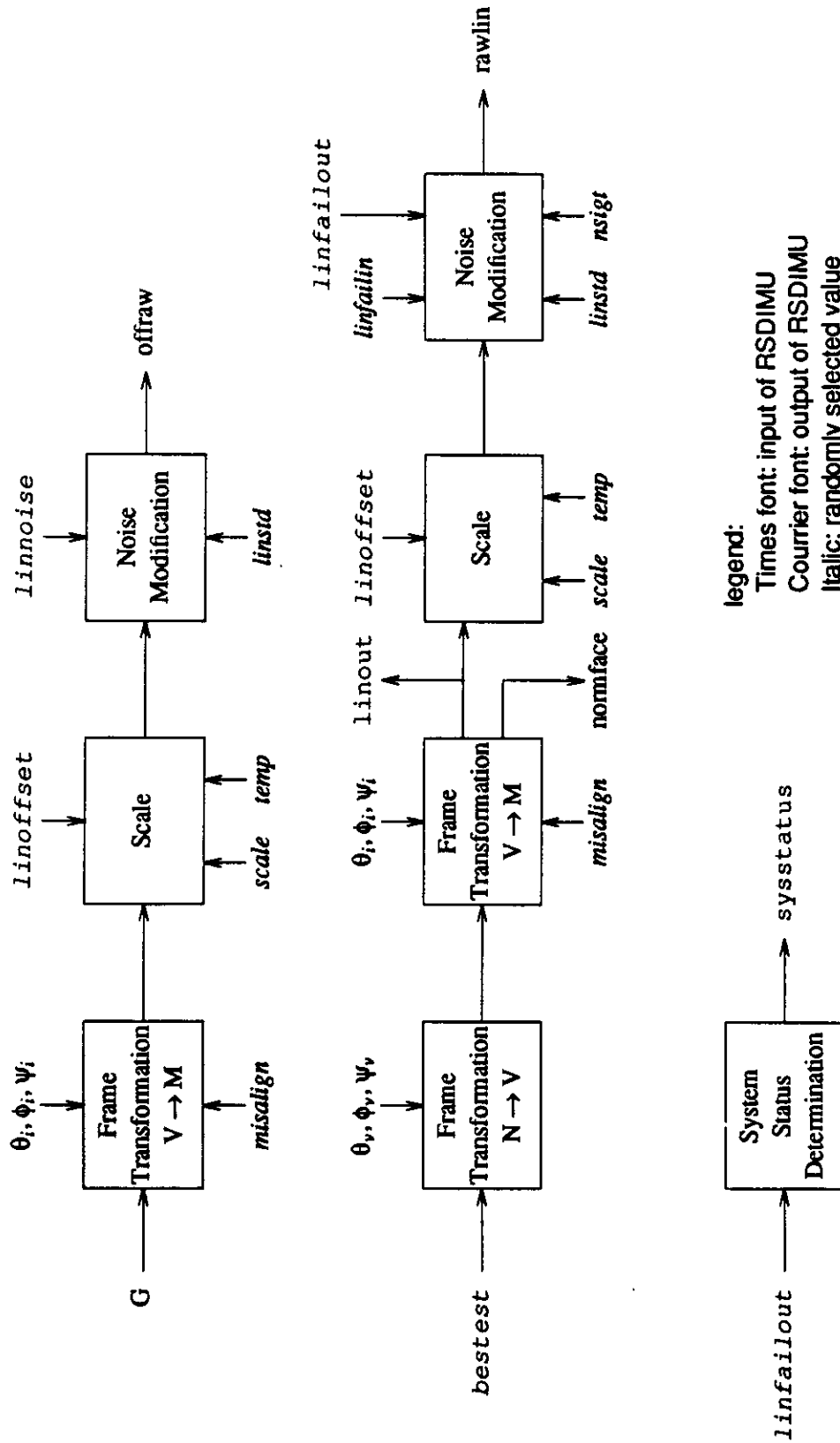


Figure 6-1: Block Diagram of the Test Case Generator

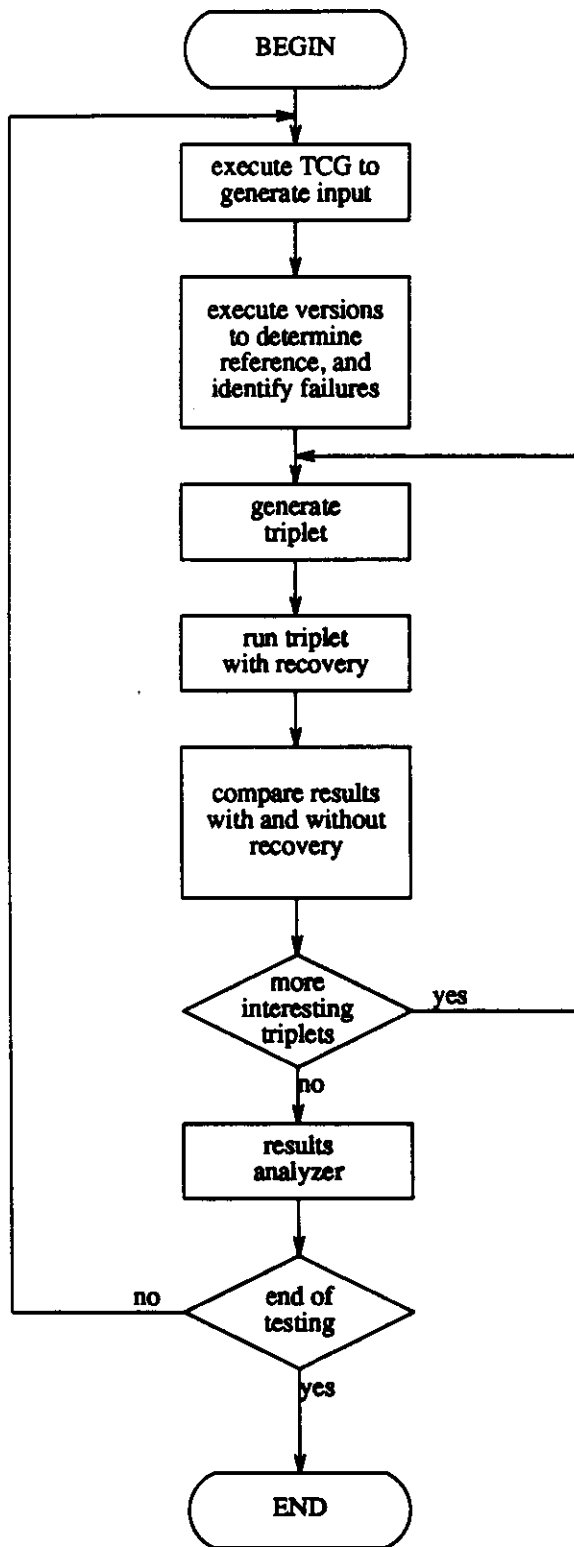


Figure 6-2: Flow Diagram of the Testing Driver

known TCG output values to ensure its consistency. At the same time, individual version failures are identified. This failure information is used to generate "interesting" 3-version combinations (triplets) using the assumption that all majority versions are identical. That means triplets of three different good versions, such as (G1, G2, G3) will be eliminated; triplets of two good versions, such as (G1, G2, B), (G1, G3, B), and etc., will be treated as one, i.e. (G, G, B). In this way, many triplets can be eliminated from testing. The interesting triplets are then executed in a MVS configuration. The decision results are passed back to the failed versions to attempt recovery at the cc-point level. Decision results of the triplets without recovery are obtained simply by comparing individual version outputs of the combinations. The decision results, both a) without recovery and b) with recovery, are then used to determine the results of the recovery. The process is then repeated for further test cases as shown in Figure 6-2.

6.3 Observed Faults in the Versions

We define a *fault* to be any instance of program text that causes the program to produce erroneous outputs when that program text is executed on some test cases. During the CER evaluation process several faults have been found in the five UCLA certified versions. Table 6-1 summarizes the observed faults and their effects on the outputs.

The fault ucla1-1 manifested itself in the testing because of the use of a Pascal compiler in the testing harness, while a Pascal interpreter was used in the program development and certification processes. Obviously the interpreter initializes variables in a Pascal procedure while the compiler does not. Since this fault fails the

Table 6-1: Characteristics of Observed Faults

Label	Class	Fault	Effect
ucla1-1	incorrect algorithm	uninitialized variable	incorrectly set sensor failure
ucla1-2	incorrect algorithm	incorrect display rounding	incorrect display
ucla1-3	incorrect algorithm	handle overflow incorrectly	incorrect display
ucla2	no known fault		
ucla3-1	spec mis-interpretation	use individual vs. average slope values	incorrect sensor status
ucla3-2	spec mis-interpretation	use wrong frame of reference	incorrect sensor status
ucla3-3	spec ambiguity	set system failure if inconsistency in fault detection and fault isolation	set system failure
ucla3-4	incorrect algorithm	display rounding	incorrect display
ucla3-5	incorrect algorithm	overflow not handled	incorrect display
ucla4-1	spec ambiguity	set system failure wrongly in fault isolation	set system failure
ucla5	no known fault		

version more than half of the time, it is taken out in our evaluation. One of the display functions is to display the five most significant digits and the decimal point of the Estimated Acceleration which is a floating point number. Two versions fail to round the numbers correctly, although in different ways: ucla1 does not round up when the last four digits are a run of 9's (fault ucla1-2), while ucla3 separates the rounding of the integral and mantissa parts, so the carry from the mantissa is dropped (fault ucla3-4). When the Acceleration has value larger than 99999, only 99999 should be displayed. Version ucla1 uses 100000 as the threshold instead of 99999 (fault ucla1-3), so any value between them will be displayed incorrectly. The programmers of ucla3 missed overflow handling (fault ucla3-5).

Both versions ucla3 and ucla4 incorrectly identify failures of the RSDIMU system[†] on some test cases for different reasons but would be triggered by the same input. The RSDIMU specification requires that after a face of the semioctahedron has been found to have failed, the program isolates the failed sensor on that face. In some cases the isolation algorithm does not find any sensor failure on the detected failed face. Version ucla3 sets the RSDIMU system to failure in these cases while the others just continue the execution (fault ucla3-3). Version ucla4 sets the RSDIMU system to failure when four of the six Edge Relations in the Edge Vector Test for fault detection are violated (fault ucla4-1). Although the original specification is ambiguous and implicitly implies that this situation would not happen, the revised specification clearly indicates that this is not an abnormal situation. Another fault of ucla3 is the incorrect use of frame of reference in the Fault Isolation algorithm (fault ucla3-2). Since the misalignment angles between the two frames of reference are very small, it is difficult to detect the fault in the acceptance tests.

[†] The RSDIMU system fails when there are not enough working sensors in the system to compute the Estimated Acceleration.

6.4 Observed Errors in the Versions

The five UCLA certified versions were subjected to an extensive testing using 200,000 random test cases. The following are the failure statistics of the versions collected during the testing.

6.4.1 Failures of the Individual Versions

The result of a version running a test case is defined erroneous if one or more of its output values (out of a total of 64) differs from the reference values as defined in section 7.2. We also say that the version fails on that test case. Table 6-2 shows the observed failures for the *individual* program versions in the 200,000 test cases.

Table 6-2: Failures of Individual Versions

Version	Number of Failures	Failure Probability
ucla1	1	0.0000050
ucla2	0	0.0000000
ucla3	702	0.0003510
ucla4	283	0.0001415
ucla5	0	0.0000000

It must be noted that the failure probability depends very much on the test case generator, and the range of variation (skew) that is allowed in comparing results. We consider that the versions tested in this evaluation were under stress because the test cases were sampled randomly from the largest possible input space. In actual flight, extremal input data are much less likely to happen.

6.4.2 Coincident Failures of the Versions

Two versions are said to fail coincidentally if they both fail (produce erroneous outputs) on the same test case. Table 6-3 shows the observed *coincident failures* of the versions. It has been observed that no more than two versions have failed at the same test case during the 200,000 test runs. Errors may be similar or distinct.

Table 6-3: Coincident Failures of the Versions

Version	ucla2	ucla3	ucla4	ucla5
ucla1	0	1	0	0
ucla2		0	0	0
ucla3			110	0
ucla4				0

6.4.3 Similar Errors of the Versions

It should be noted that the results of the versions which fail coincidentally may not be similar. *Similar results* are defined to be two or more results (good or erroneous) that are within the range of variation that is allowed by the decision algorithm. When two or more similar results are erroneous, they are called *similar errors* [Aviz86].

Table 6-4: Similar Errors of the Versions

Version	ucla2	ucla3	ucla4	ucla5
ucla1	0	0	0	0
ucla2		0	0	0
ucla3			96	0
ucla4				0

6.5 Faults vs. Errors

In order to measure the number of errors caused by each of the detected faults, the faults found during the extensive testing were all removed and were put back to the versions one by one. These versions were then run through the test cases that had caused failures of the respective version. Table 6-5 shows the results.

Table 6-5: Faults vs. Errors

Fault	Number of Failures
ucla1-1	numerous
ucla1-2	1
ucla1-3	0
ucla3-1	320
ucla3-2	404
ucla3-3	54
ucla3-4	65
ucla3-5	0
ucla4-1	283

The faults ucla1-3 and ucla3-5 were not triggered in the 200,000 test cases. They were uncovered through code inspection. Some test cases triggered more than one fault in a version, hence the sum of all errors of version ucla3 in Table 6-5 is larger than the total number of failures shown in Table 6-2. With the presence of other faults in the version, a fault may or may not be triggered due to the erroneous state caused by the other faults.

Faults in the multiple versions affecting their decision results may be divided into two classes: independent, and related [Aviz86]. Faults in the multiple versions

are considered to be *related* if they are attributed to some common cause, such as a common link between the separate design efforts, otherwise they are considered to be *independent*. The faults ucla3-3 and ucla4-1 are related because they are caused by the ambiguity of the specification. Although the faults ucla1-2 and ucla3-4 both cause rounding errors and sometimes produce similar erroneous results, they are independent because they do not have a common cause. The faults ucla1-3 and ucla3-5 may also produce similar erroneous results, nonetheless the chance is slight.

The approach that is employed by MVS to avoid related faults in the multiple versions is the maximal independence of design and implementation efforts. Obviously, the single specification used in this experiment becomes the most likely source of related faults. Independent faults producing similar errors should also be systematically avoided by enforcing diverse algorithms, design tools, and testing methods. None of these were employed in this experiment. In fact, the design tools used by each programming team were the same and each version was accepted based on the same acceptance tests. It is those parts of the program texts that had not been exercised in the two acceptance tests that caused most of the faults that produced similar errors.

6.6 Classification of Triplet Decisions

In the analysis of recovery the individual versions are combined into 3-version MVS systems (triplets). MVS systems of two versions cannot perform recovery if no extra information is supplied to determine which one of the two versions is good. MVS systems with four versions and five versions are not analyzed in this research because we found at most two versions failed at the same test case in all the test runs. That means combining the versions into 4-version MVS system will at worst have no majority in the comparison, and a 5-version MVS system will always have a majority of good versions. Analysis of these results will not be fruitful. Besides this, a 3-version MVS system is most practical in practice because of the high cost in generating software versions.

We wish to introduce the concept of *decision* before we discuss the results of recovery. As has been mentioned previously, a *decision result* is the single consensus result obtained from the comparison of the individual version results using a decision algorithm. The decision result may be a consensus of all the versions, or of a majority of the versions. Sometimes there is no decision result because no consensus can be obtained. The number of versions that are in consensus gives the decision result the *confidence level*. Also a decision result may be good or erroneous depending on the correctness of the majority similar results. A *decision* summarizes the confidence level and the correctness of the decision result. Our analysis is interested in the decisions made by the decision algorithm, not the decision results per se. The decisions of a triplet can be classified into five categories as shown in Table 6-6. In the figure "G" represents a good result, while "B" represents an erroneous result. We do not distinguish similar results from each other, whether they are good or erroneous. "B1", "B2", and "B3" represent distinct erroneous results.

Table 6-6: Classification of Triplet Decisions

Decision	Individual Results			Comment
GOOD3	G	G	G	All the three versions produce good results (G). The triplet not only has the good decision result, but also shows high confidence for its result.
GOOD2	G	G	B	Only two versions are good. The error (B) of the failed version is masked by the majority.
NOMAJ	B1	B2	G	All the results are different from each other. We do not distinguish whether there exists a good version or not. This decision is a fail-safe result.
	B1	B2	B3	
BAD2	B	B	G	A similar error occurred in two versions. Again we do not distinguish whether the third version is good or bad.
	B	B	B1	
BAD3	B	B	B	A similar error in all three versions. It is seriously bad because the triplet not only produces an erroneous result, but also shows high confidence for the result.

6.7 Analysis of CC-Point Recovery

Two approaches are used for estimating the effectiveness of cc-point recovery. The first one is based on the comparison of cc-point decisions of a triplet executed *without* the recovery provision and the cc-point decisions of the same triplet executed *with* the recovery provision. Instead of looking at the improvement of cc-point decisions, the second approach examines the difference in decision results produced by the RSDIMU modules in triplet if recovery provision is in place. This approach is more related to the application itself.

6.7.1 Results on CC-Point Improvement

The objective of cc-point recovery in a triplet is to recover a failed version from errors that has occurred at some cc-point so that normal execution can continue and good results can be produced at subsequent cc-points. Figures 6-3, 6-4, and 6-5 show some of the possible changes in cc-point decisions after an attempted recovery. Figure 6-3 is a successful recovery of a single error. The decision of the subsequent cc-points have been changed from GOOD2 to GOOD3. Figure 6-4 shows a successful recovery of two failures on different modules in two versions. The decision of the third cc-point has been improved from NOMAJ to GOOD3. An unsuccessful recovery is shown in Figure 6-5. The similar error of the two failed versions produces an erroneous decision result and an attempted recovery forces the only good version to fail in the same way. Hence one way to evaluate the effectiveness of cc-point recovery is by examining the improvements on the decisions of the cc-points after an attempted recovery. Table 6-7 summarizes the results of the 200,000 test runs.

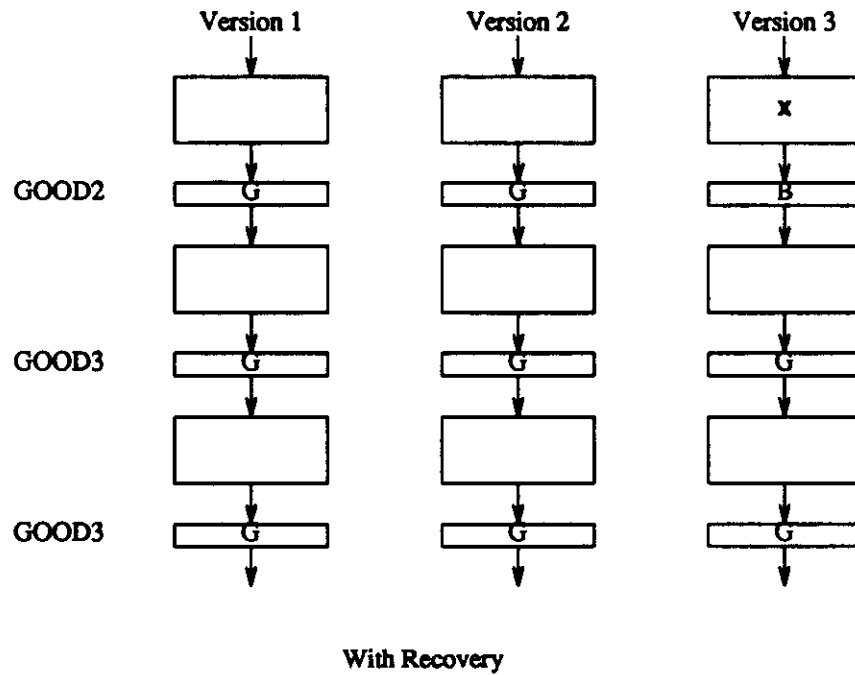
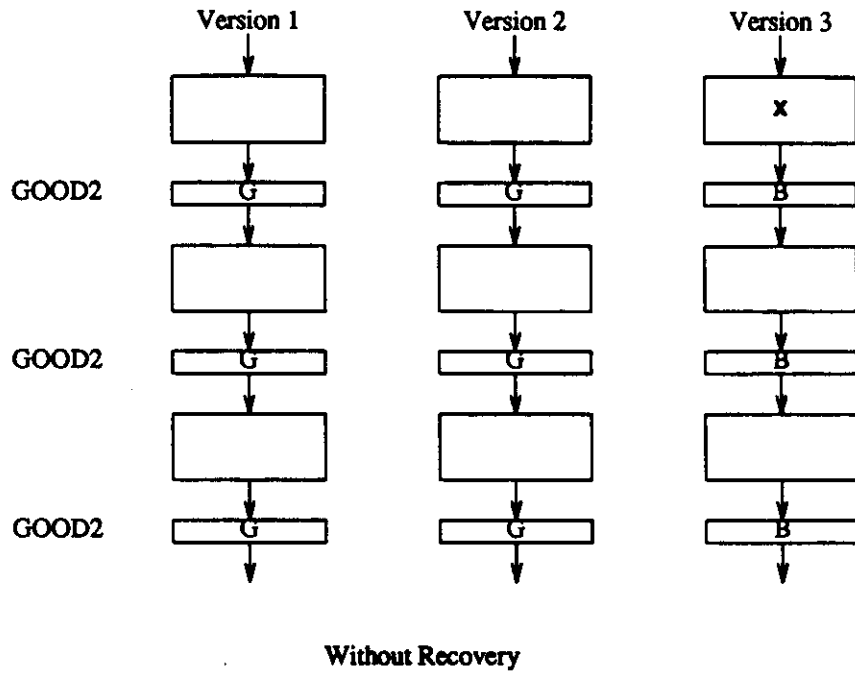
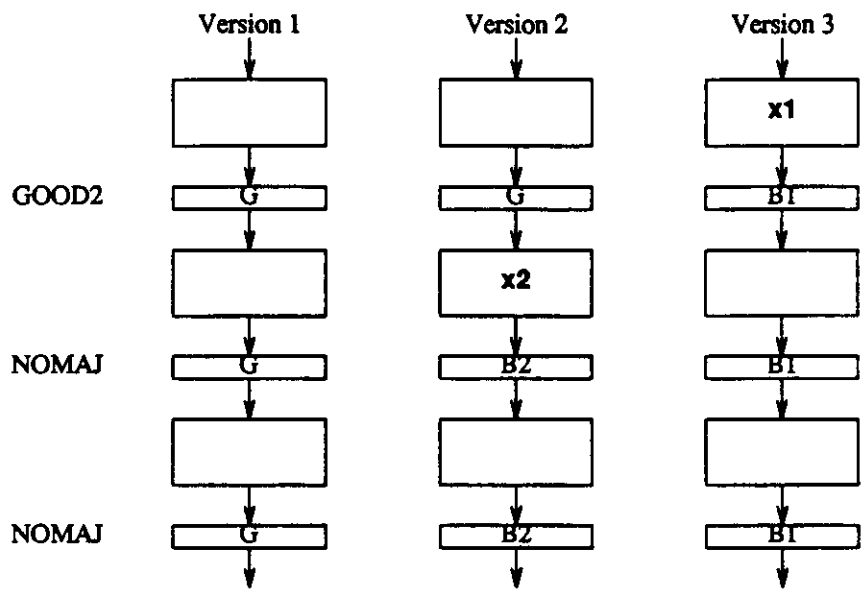
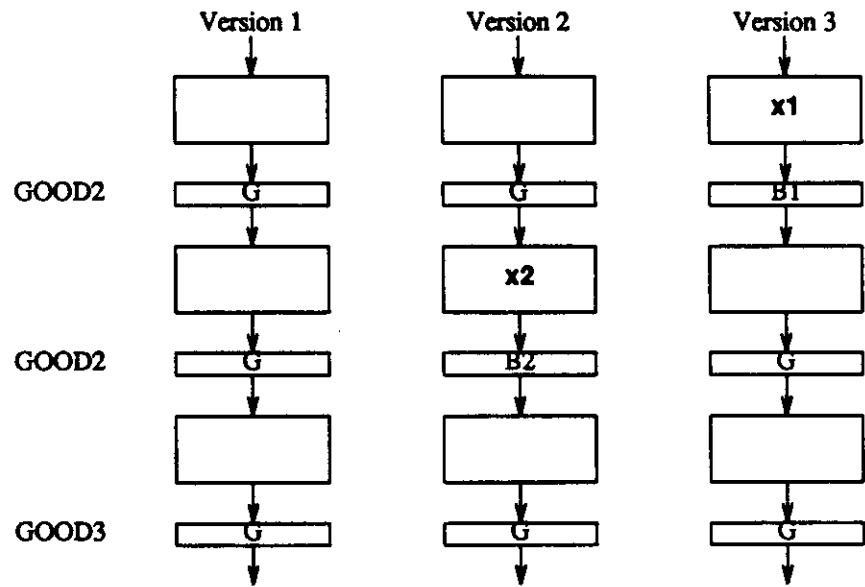


Figure 6-3: Decision Changed from GOOD2 to GOOD3

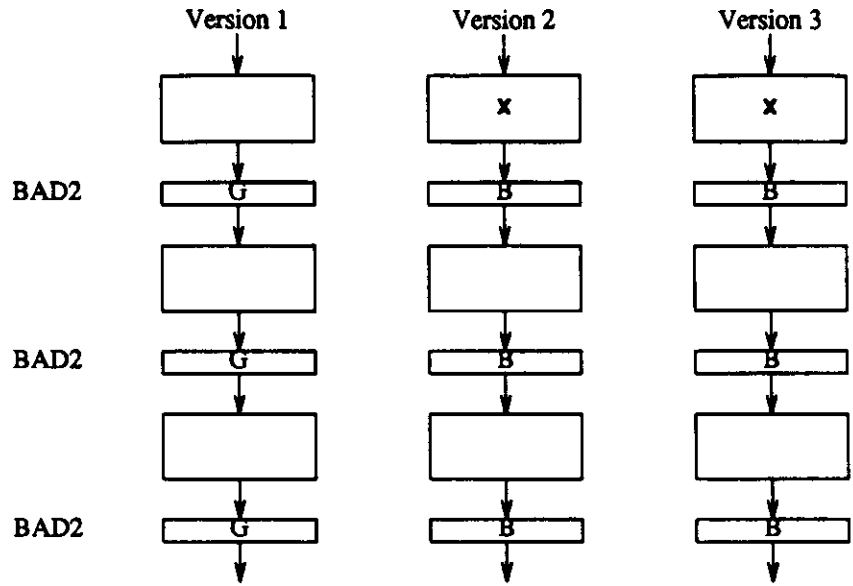


Without Recovery

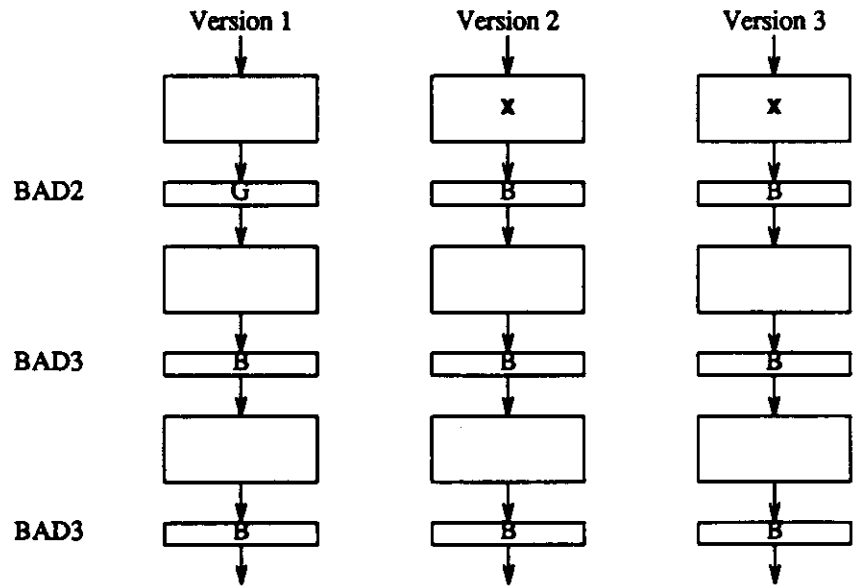


With Recovery

Figure 6-4: Decision Changed from NOMAJ to GOOD3



Without Recovery



With Recovery

Figure 6-5: Decision Changed from BAD2 to BAD3

Table 6-7: Results on CC-point Improvement

Without recovery	With recovery	Triplets of 2 good and 1 bad versions	Triplets of 1 good and 2 bad versions	Total
GOOD2	GOOD3	1259	3	1262
NOMAJ	GOOD3	0	7	7
NOMAJ	GOOD2	0	3	3
NOMAJ	BAD3	0	15	15
BAD2	BAD3	0	127	127
Total number of changed cc-points				1414

Most decisions remained the same whether or not there was recovery provision and their number is not shown in the table. The main reason is that triplets of three good versions always produce GOOD3 decisions and recovery is not activated.

Almost 90% of the changed cc-point decisions are from GOOD2 to GOOD3, meaning that errors occurred in a single version of the triplets had been recovered successfully by the recovery provision. The improvement of the decisions from GOOD2 to GOOD3 should not be diminished by the fact that the decision results of the two decisions are the same so that the change is only on the confidence level. It is this improvement that makes the 3-version MVS system fully recovered and ready to tolerate another fault if that would happen in the subsequent computations. The improvement of the decision from NOMAJ to GOOD2 or GOOD3 demonstrates this effect. Since the RSDIMU module has only five computations and all the observed errors occurred after the second one, the situation that a fully recovered triplet tolerates a second fault does not happen very often.

The 127 cc-points that have their decisions changed from BAD2 to BAD3 show that the good version in those cases was forced to fail in the same way after a

recovery attempt. However, with similar errors in majority of versions, the MVS system is assumed to fail.

The 15 cc-points that have their decisions changed from NOMAJ to BAD2 are bad because the triplicated RSDIMU MVS system has been changed from a fail-safe state to an unsafe state. Let us investigate how such decisions had happened by examining the details of a sample test case. A coincident error occurred in ucla3 and ucla4 at the third cc-point comparing the LINFAILOUT - an array of final sensor status. Suppose version ucla2 produced a good output, the results of the versions will be as follows:

	LINFAILOUT
ucla2	(0 0 0 0 0 0 0 0)
ucla4	(0 0 0 0 0 0 1 0)
ucla3	(1 1 1 1 1 1 1 1)
decision result	(0 0 0 0 0 0 1 0)

Without recovery, the next computation for the Estimated Acceleration will be a NOMAJ decision because the versions have a different set of working sensors. With recovery, the erroneous decision result will be used by all the three versions and they produce a similar erroneous result on the Estimated Acceleration based on the same set of working sensors. In the later section we will discuss how this erroneous recovery can be avoided.

6.7.2 Results on RSDIMU System Improvement

The second approach to evaluate the effectiveness of cc-point recovery is by comparing the decision results of a triplicated RSDIMU system executed without the recovery provision and the one executed with the recovery provision. A RSDIMU system is considered to be working properly if it computes the final System Status and Estimated Acceleration in agreement with the majority-derived reference result. This decision is made at the fourth cc-point. We expect that if there are errors occurred at the first three cc-points, and if cc-point recovery is effective, a failed version will be recovered and produce a good final result. We do not account for the output of the Display Driver because in many cases the Driver is requested to display merely a test pattern. In these cases the Display Driver can produce a correct result even though the RSDIMU module fails to compute the Estimated Acceleration correctly.

Table 6-8 shows the number of decisions in each category of different triplet combinations without the recovery provision over 200,000 test cases. There are ten distinct triplets available with five versions. Therefore there are a total of 2,000,000 decisions.

Table 6-8: Decisions of Triplets Without Recovery

Decision	Triples of 2 good and 1 bad versions	Triples of 1 good and 2 bad versions	All possible triples (2000000 possible)
GOOD3	63	1	1998967
GOOD2	923	3	926
NOMAJ	0	11	11
BAD2	0	96	96
BAD3	0	0	0
Total	986	111	2000000

There is no triplet composed of three failed versions because no test case failed more than two versions was found in the testing. Triplets of three good versions which produce GOOD3 decisions are added to the last column of the table. From Table 6-8 we can observe that triplets consisting two good versions and one failed version invariably have masked the error in the failed version successfully. For triplets with two failed versions and one good version, there are 96 decisions in the BAD2 category which show the presence of a similar error in the two versions. There are 67 decisions in the GOOD3 category even the triplets consist of one or more failed versions because our analysis considers results of the System Status and Estimated Acceleration only, and these failed versions were able to compute them correctly but failed in the Display Driver.

Table 6-9 shows the results obtained from the triplets executed with the recovery provision. Compared to Table 6-8, all the 926 GOOD2 decisions have been improved to GOOD3 for those triplets with only one failed version, that means the failed version had been successfully recovered at the cc-point at which it failed and was able to produce a correct result.

Table 6-9: Decisions of Triplets With Recovery

Decision	Triplets of 2 good and 1 bad versions	Triplets of 1 good and 2 bad versions	All possible triplets (2000000 possible)
GOOD3	986	13	1999902
GOOD2	0	0	0
NOMAJ	0	0	0
BAD2	0	0	0
BAD3	0	98	98
Total	986	111	2000000

Results of the cc-point recovery are not as good for triplets with two failed versions. Although it is found that 9 decisions have been improved from NOMAJ to GOOD3, in 98 triplets similar errors in two failed versions have forced the good version into failure by attempting recovery. Table 6-10 summarizes the results of the cc-point recovery based on the RSDIMU system improvement over the 200,000 test cases.

Table 6-10: Results of RSDIMU System Improvement

Without recovery	With recovery	Triplets of 2 good and 1 bad versions	Triplets of 1 good and 2 bad versions	Total
GOOD2	GOOD3	926	3	926
NOMAJ	GOOD3	0	9	9
NOMAJ	BAD3	0	2	2
BAD2	BAD3	0	96	96
Total number of changed triplicated RSDIMU results				1033

The most common similar errors observed during the testing are due to situations that both versions ucla3 and ucla4 declare the RSDIMU system failed,

which will set all sensor status to non-operative and the estimated acceleration to zero. The program faults (ucla3-3 and ucla4-1 in Table 6-1) are due to extra checks on conditions that should not happen in the original specifications, but which were changed during the course of program development and certification. However, it should be noted that such outputs lead to a fail-safe response of shutting down the system in the RSDIMU application.

6.7.3 Granularity of Comparison

In order to determine the decision result from several versions, the Decision Function must compare and classify computed values. It is found that the granularity of comparison has a profound influence on the decisions of the triplets. The five cc-points specified in the RSDIMU application compare eleven variables. Most of them are of complex type, e.g., an array of 8 Booleans for sensor status, and a structure of 3 real numbers for acceleration components. A total of 64 elements is found in the eleven variables. We can compare their computed values based on 1) elements, 2) variables of complex types, and 3) cc-points of several variables. For comparisons based on variables and on cc-points, two versions agree only if each of their corresponding elements agree.

Comparison based on elements has a potential benefit that consensus may be reached in the face of coincident errors in a majority of versions. For example, in a particular test case there should be no sensor failure determined by the computation, but ucla3 incorrectly detects sensor 2 failed and ucla4 incorrectly detects sensor 5 failed. Their outputs on sensor status are as follows:

	LINFAILOUT
ucla2	(0 0 0 0 0 0 0 0)
ucla3	(0 1 0 0 0 0 0 0)
ucla4	(0 0 0 0 1 0 0 0)
decision result	(0 0 0 0 0 0 0 0)

The consensus is a correct decision with the error masked and recovery on ucla3 and ucla4 will enable them to compute the acceleration correctly based on data from eight sensors. However, if the comparison is based on the single variable of "eight sensor status," there will be no majority decision, and a safe shutdown should follow.

However, comparison by elements also has a potential danger of making a wrong decision. In the above example, if ucla3 incorrectly identifies system failure and sets all sensors to non-operative, the decision based on comparison by elements on the following outputs

	LINFAILOUT
ucla2	(0 0 0 0 0 0 0 0)
ucla3	(1 1 1 1 1 1 1 1)
ucla4	(0 0 0 0 1 0 0 0)
decision result	(0 0 0 0 1 0 0 0)

will lead to an incorrect decision result. After cc-point recovery, all the versions will be forced to have their sensor status as these erroneous values, and most likely it will lead to a BAD3 decision on the Estimated Acceleration.

The results in Tables 6-7, 6-8, and 6-9 are based on comparison by elements. Table 6-11 shows the results of triplets with recovery based on comparison by variables.

**Table 6-11: Decisions of Triplets With Recovery
(comparison by variables)**

Decision	Triplets of 2 good and 1 bad versions	Triplets of 1 good and 2 bad versions	All possible triplets (2000000 possible)
GOOD3	986	5	1999894
GOOD2	0	0	0
NOMAJ	0	10	10
BAD2	0	0	0
BAD3	0	96	96
Total	986	111	2000000

Compared with Table 6-9, the results show exactly the consequences of using a larger granularity: it lost the potential to have more agreement, but also reduced the risk of making erroneous decisions. We have found that the results based on comparison by a complete cc-vector at a cc-point are the same as by individual variables within the cc-vector.

6.8 Analysis of R-Point Recovery

Recovery points were not specified in the RSDIMU specifications. However a test program can be easily composed such that the RSDIMU module is the first module with an auxiliary (AUX) module added and a recovery point inserted between them. The AUX module contains nothing but a new cc-point (ccp6) used to check if the AUX module is indeed executed and started with a correct version state. The skeleton of the instrumented program has been shown on Figure 4-6. The version state at the beginning of the AUX module for our purpose has been defined as a collection of all the eleven output variables and two other variables in the RSDIMU

module found to be common to all the versions. With the version state defined, two exception handlers similar to those illustrated in Figure 4-7 have been implemented for state input and output and used by all the versions.

DEDIX was used for the testing because recovery at the recovery point level requires a sophisticated MVS supervisor for keeping track of errors detected at the cc-points, invoking the exception handlers, and restarting an aborted version.

The evaluation of r-point recovery was performed in two steps. The first one used the test cases which were found to cause failure of some versions during the extensive cc-point testing. The second one used systematic error seeding into the versions for verifying the effectiveness of the r-point recovery mechanism which was not tested completely using the random test cases since there were no control flow errors observed in the versions.

6.8.1 Results Based on Previous Test Cases

All the test cases which failed some versions found in the extensive cc-point recovery testing were used to test triplets of the instrumented programs for r-point recovery. The evaluation is similar to the previous one on RSDIMU system improvement with cc-point recovery. The previous evaluation examines the result (System Status and Estimated Acceleration) of a triplicated RSDIMU system. In this evaluation we examine the version state after the recovery point.

All possible outcomes of a DEDIX test run executing a triplet of instrumented versions each consisting the RSDIMU module, a r-point, and the AUX module are shown in Table 6-12 (not all actually occurred).

Table 6-12: Possible Outcomes of a DEDIX Test Run

Outcome	Comment
CCP-NM	Test run terminated because of no majority occurred at the cc-points in the RSDIMU module.
RPID-NM	Test run terminated because of no majority in comparing the r-point ids; this will occur when some version has control flow fault.
STATE-NM	Test run terminated because of no majority in comparing the version states at r-point.
CCP6-G3	GOOD3 decision at the cc-point in the AUX module after the r-point recovery.
CCP6-G2	GOOD2 decision at the cc-point in the AUX module after the r-point recovery.
CCP6-NM	NOMAJ decision at the cc-point in the AUX module after the r-point recovery.
CCP6-B2	BAD2 decision at the cc-point in the AUX module after the r-point recovery.
CCP6-B3	BAD3 decision at the cc-point in the AUX module after the r-point recovery.
DEDIX-E	DEDIX itself terminated because of its internal errors.

The results, which are summarized in Table 6-13, show that for triplets with only one failed version, 983 of the 986 version states of the failed versions were recovered correctly after the recovery point. There are three cases in which DEDIX terminated because of disagreement in comparing version states. In the testing, the good versions we used for the triplet were the first two good versions in the order according to their version id, so they are different versions. It was found that in those test runs although uclal produced good outputs at the cc-points in the RSDIMU module, in fact it had an erroneous internal state. The two additional variables included in the defined state revealed the errors. The 96 triplets that produced BAD3 decisions shown in Table 6-11 had no majority in comparing the version states. This is because the two versions both incorrectly identified that the RSDIMU system failed and produced similar errors that were due to different faults. The two variables included in the internal state, one is the id number of the failed face, the other is the threshold determining a sensor failure, revealed the differences. This shows that r-point checking is more effective than cc-point only since all the 111 BAD2 decisions were properly detected.

Table 6-13: Results of R-Point Recovery

Result	Triples of 2 good and 1 bad versions	Triples of 1 good and 2 bad versions	Total
CCP-NM	0	12	12
RPID-NM	0	0	0
STATE-NM	3	96	99
CCP6-G3	983	3	986
CCP6-G2	0	0	0
CCP6-NM	0	0	0
CCP6-B2	0	0	0
CCP6-B3	0	0	0
DEDIX-E	0	0	0
Total	986	111	1097

There are three test runs with triplets of 2 bad versions produced GOOD3 at the last cc-point because the errors of the two failed versions occurred in different cc-points and were successfully recovered by cc-point recovery.

6.8.2 Results Based on Error Seeding

As control flow errors were not observed in the 200,000 test cases, we conducted more testing through error seeding specifically to verify the effectiveness of the restart mechanism of the r-point. Faults of the following two categories were seeded:

1. Program exceptions such as division by zero and index out of range,
2. Control flow faults such as infinite loop and incorrect branching that lead to some cc-point incorrectly called or skipped.

Most of the faults we seeded into the versions are those that were eliminated during the certification process in order to make them realistic. The testing was conducted with triplets consisting of two good versions combined with a version with a seeded fault. It is found that in all the hundred different test runs we had performed, the failed versions were restarted with a correct version state after the recovery point.

6.9 Discussion

The empirical results obtained in the experiment show that for 3-version MVS systems Community Error Recovery is highly effective to correct erroneous computations at cc-points and to restore correct states at recovery points. This successful result may be attributed to the computational nature of the RSDIMU application, which is a highly sequential program module in which each computation depends mostly on results of the previous one. This fortunately is typical of a large class of real-time control computations. Although values which passed from one computation to the next without being cross-checked are found when the software versions are examined, these unchecked values have not affected the cc-point recovery. For 3-version systems with similar errors in two versions, in most cases the good version was forced to fail in the same way after a recovery attempt. However, with similar errors in majority of versions, the MVS system is assumed to fail.

Results of the experiment show that comparison of version outputs where each output consists of a single Boolean value is dangerous. One can always find a majority decision in the comparison for three Boolean outputs, but it is hard to place any confidence on the majority decision. In this experiment, ucla3 and ucla4 set System Status to failure incorrectly due to different faults, but the outputs were similar

and erroneous. It will be much better if we compare the values which are used to compute these single bit outputs. For example, if the RSDIMU system decides that a sensor has failed when its noise level exceeds 10.0, ucla2 computed the noise level of the sensor to be 8.5, and ucla3 and ucla4 incorrectly computed 12.0 and 118.5 respectively. Comparison based on Boolean outputs will lead to similar errors, while no majority can be detected if noise levels are compared.

Related to the comparison on Booleans, the granularity of comparison has found to have a profound influence on the results of MVS systems. The smaller the granularity, the higher is the potential to obtain consensus in the comparison. However, the chance of erroneous decisions will be larger. For safety-critical applications, comparison by variables or by cc-points will be more appropriate.

The results in this experiment has shown that a version may produce good cc-vectors even though there are errors in its internal state. This kind of error is known as a *latent error*. The experimental results also show that the r-point is a powerful check on undetected errors in the version states. Reliability of a MVS system would conceivably be improved if at some r-points all the versions are signaled to output their internal states which are then compared, and the resulting decision state is input by the disagreed versions to flush out latent errors.

CHAPTER 7

CONCLUSIONS

7.1 Practicality of CER

The proposed CER recovery mechanism, which makes use of the natural redundant information in the versions, is simple and efficient. It recovers from errors in two levels; these 1) match the type and severity of the faults, 2) minimally disturb the system, and 3) impose minimum restrictions on the implementation of the programs. It has been implemented in the Local and Global Executives of the DEDIX distributed MVS testbed, and has been tested using programs of the NASA-Four University Multi-Version Software Experiment. In this experiment, any given cross-check point was specified to be placed at the point where the cc-vector had been computed and before it would be used. Experience shows that the CER method is easily observed by the programmers and easily verified for its correctness by the experimenters.

7.2 Effectiveness of CER

Evaluation of the reliability models developed in Chapter 5 has shown that recovery may substantially improve the reliability of a multi-version software system. The prediction was validated by the results obtained with programs from the NASA-

Four University Multi-Version Software Experiment. Experimental results show that cc-points are highly effective in recovering independent errors in a 3-version MVS system, and r-points which compare version states are effective to detect erroneous states in the versions even though they have similar errors in their results. It should be noted that the results are obtained from this experiment for the chosen application, and may not extend to other applications. The RSDIMU module is a highly sequential program in which each computation depends mostly on the results of the previous one. This is typical of a large class of real-time control applications.

7.3 Related Faults and Similar Errors

The average failure rate of the versions during the testing was less than 0.0001 which indicates that the UCLA certified versions are of high quality. However the observed faults and errors show an annoying result: of the eight observed faults (not counting the uninitialized fault in ucla1), every one of them may produce similar errors with some of the other faults; of 111 observed coincident failures, 96 of them are similar. Does this imply a warning that residual design faults in high quality software are prone to produce similar errors? Detailed analysis indicates that all the 96 similar errors are due to the incorrect identification of system failure of the RSDIMU module which entails setting the subsequent output values to zero. The Fault Detection and Isolation computation is the largest and most complex one in the RSDIMU module. Independent faults in that computation may cause system failure because of different reasons, but the output depends only on the Boolean result, causing serious similar errors. To detect these errors, we should compare the values which determine a Boolean output, not the Boolean output itself. The results on r-point recovery which detected all the similar errors (since the version states were

different) support this conclusion.

7.4 Suggestions for Future Research

7.4.1 Resolving Multiple Correct Results

In this application, one way to identify a faulty sensor is to observe that its calibration readings are noisy, i.e., that the standard deviation over a set of values exceeds some prescribed threshold. Should the noise level be precisely at this threshold, two algorithms could conceivably produce different results, one indicating that the sensor is noisy, the other indicating that it is operational. The effect of slight numerical differences is seen by the MVS Supervisor as totally contradictory responses. This situation has, in fact, happened during the extensive testing. In three out of the 20,000 test cases, version ucla1 has different noisy sensor values from the other versions. When the values of the internal variables are examined, it is surprising that their difference is really so small:

```
ucla1 standard deviation of noise level = 2.9999999999999999
ucla2 standard deviation of noise level = 3.0000000000000001
threshold = 3
```

This problem may be avoided by introducing an additional decision point that uniformly returns to the versions the decision value of the standard deviation. The versions may then compare this value to the threshold and obtain a consistent result. The generality of this approach has not yet been determined.

7.4.2 Enhanced Fault Tolerance Techniques

The simple ring structure currently implemented in DEDIX does not allow the implementation of the reconfiguration scheme proposed in Chapter 4. A redundant interconnection structure should be developed so that reconfiguration can be possible when physical faults occur on the links or sites.

For applications with a large system state, such as a database, the overhead on the comparison and inter-site communication will be overwhelming for r-point recovery. Research on the practicality of designing exception handlers so that the comparison and communication of version states are done progressively in two or more recovery points should be investigated.

In order that a version with control flow faults can be restarted at any r-point, the versions are constrained with the modular structure described in Chapter 3. Although the method is general and applicable to different programming languages, it would be much better if special linguistic support is developed for the CER algorithm.

7.4.3 Future Experiments

Based on the experience gained in this experiment, the following areas of interest for future experiments are discussed.

- 1. Formal specification**

Although nine months were spent on writing the original RSDIMU English specification by a team of avionics and software specialists, it was found to contain errors and ambiguities. The original specification was

improved based on the feedback of the programmers and preliminary testing results obtained in the validation phase. However, the revised specification was found to have caused the most serious related fault in the certified versions. The conclusion is that an English specification is inherently ambiguous and error-prone. Future experiments should use formal specification languages and appraise their applicability to practical use by application programmers for the specification of fault-tolerant multi-version software.

2. Exploration of other dimensions of diversity

In the NASA-Four University Multi-Version Software Experiment only diverse programming teams and geographical distribution have been employed to generate the independent versions. Many other potential sources of diversity, such as specification languages, programming languages, algorithms, development environment, and testing methods, can be found in the software development process. Each of these dimensions of diversity should be systematically investigated to examine their influence on multi-version software in future experimentation.

3. Better experiment protocol

In order to avoid interaction and be able to share knowledge of the specification among the independent programming teams, it was decided that questions concerning the specification should be submitted to the central project coordinator and then the question-answer pairs be broadcast to every team. The protocol caused a number of problems: 1) The number of questions (over 250) posed by the 40 programmers was overwhelming. The additional

question-answer pairs tripled the bulk of the specification. 2) The questions and answers had caused numerous ambiguities and contradictions. The cause for this confusion appears to be attributable to the requirement that the central coordinator answer every question personally, which put a great deal of pressure on the central coordinator to answer quickly, rather than well. 3) Many implementation details of individual teams were leaked to other teams through the questions, which posed a potential threat of related faults. In future experiments, the coordinator should study a question and the related part of the specification carefully before answering it. Only if it is really a specification error or ambiguity, a well prepared specification revision should be sent to every team. In no case should the question be broadcast.

4. **Avoid Boolean value comparison**

As the results of the extensive testing have shown, comparing Boolean values is harmful to MVS in two respects:

- a. Diverse versions are vulnerable to producing similar errors. Since a Boolean result has two values, incorrect Boolean results of the versions are deemed to be similar.
- b. Good versions may produce different but correct results. If the values which determine the Boolean results differ by an acceptable numerical difference, they may decide on different Boolean results after comparison to a threshold.

We should avoid comparing Boolean results at cc-points. Instead, the values that lead to those Boolean results should be used for comparison in future experiments, as well as in practical use of multi-version software.

5. High quality acceptance test

The preliminary acceptance test for this experiment was grossly inadequate. The resulting initial versions were flawed by many common programming bugs. If these versions had been used in the analysis, the superfluous bugs would have masked the effect of faults that would be found in high quality software, making the result of the experiment unrealistic. We were fortunate to be able to re-hire the original programmers to improve their own programs and apply a much more stringent acceptance test in the certification phase. The acceptance test will have a direct influence on the quality of the resulting versions, and should be well planned and implemented before the software development phase.

6. Careful placement of cc-points

The original specification of cc-points for the RSDIMU module had a state variable which caused a cyclic data dependency problem. The variable was taken out from the cc-point in the initial software development. It was later specified in a separate, additional cc-point in the revised specification and implemented in the certified versions. Without the certification phase, cc-point recovery would not be completed. Another inadequacy of the cc-point specification is that the Fault Detection and Isolation computation of the RSDIMU module is the largest and most complex computation. Most coincident errors observed in the extensive testing occurred in this computation. Placing an additional cc-point after the fault detection of failed sensors in that computation should help in the detection of software errors and their subsequent recovery. The inadequacy of the cc-point specification was due to different people specifying the application and the cc-points, where the

person specifying the cc-points did not have an adequate understanding of the application. In future experiments cc-points should be specified by the same people who write the specification of the application, or by people who have a thorough understanding of the application.

7. Larger Programs

RSDIMU was a well chosen application for this experiment. It is a realistic highly critical software module, with an appropriate size for the 10-week software generation phase for two programmers. However, for better evaluation of the CER, a larger application should be used. In the next experiment the chosen application should consist of a few program modules with r-points and internal states defined, and each of them should consist of a few cc-points similar to those of the RSDIMU module. Such application should allow us to better assess the effort of specifying and implementing CER, its effectiveness in MVS and its limitation of design diversity.

REFERENCES

- [Adri82] W.R. Adrion, M.A. Branstad, and J.C. Cherniavsky, "Validation, Verification, and Testing of Computer Software," *ACM Computing Surveys*, Vol. 14, No. 2, June 1982, pp. 159-192.
- [Ande81] T. Anderson and P.A. Lee, *Fault Tolerance: Principles and Practice*, London, England: Prentice Hall International, 1981.
- [Ande84] T. Anderson, "Can Design Faults be Tolerated?," in *Proceedings 2nd GIIGMR Conference on Fault Tolerant Computing Systems*, Bonn, Germany: 1984, pp. 426-433.
- [Aviz71] A. Avižienis, G.C. Gilley, F.P. Mathur, D.A. Rennels, J.A. Rohr, and D.K. Rubin, "The STAR (Self-Testing And Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," *IEEE Transactions on Computers*, Vol. C-20, No. 11, November 1971, pp. 1312-1320.
- [Aviz75] A. Avižienis, "Fault-Tolerance and Fault-Intolerance: Complementary Approaches to Reliable Computing," in *Proceedings 1975 International Conference on Reliable Software*, Los Angeles, California: April 21-23, 1975, pp. 450-464.
- [Aviz78] A. Avižienis, "Fault-Tolerance: The Survival Attribute of Digital Systems," *Proceedings of the IEEE*, Vol. 66, No. 10, October 1978, pp. 1109-1125.
- [Aviz82] A. Avižienis, "Design Diversity - The Challenge for the Eighties," in *Digest of 12th Annual International Symposium on Fault-Tolerant Computing*, Santa Monica, California: June 1982, pp. 44-45.
- [Aviz84] A. Avižienis and J.P.J. Kelly, "Fault-Tolerance by Design Diversity: Concepts and Experiments," *Computer*, Vol. 17, No. 8, August 1984, pp. 67-80.

- [Aviz85a] A. Avižienis, P. Gunningberg, J.P.J. Kelly, L. Strigini, P.J. Traverse, K.S. Tso, and U. Voges, "The UCLA DEDIX System: A Distributed Testbed for Multiple-Version Software," in *Digest of 15th Annual International Symposium on Fault-Tolerant Computing*, Ann Arbor, Michigan: June 1985, pp. 126-134.
- [Aviz85b] A. Avižienis, P. Gunningberg, J.P.J. Kelly, R.T. Lyu, L. Strigini, P.J. Traverse, K.S. Tso, and U. Voges, "Software Fault-Tolerance by Design Diversity; DEDIX: A Tool for Experiments," in *Proceedings IFAC Workshop SAFECOMP'85*, Como, Italy: October 1985, pp. 173-178.
- [Aviz85c] A. Avižienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, December 1985, pp. 1491-1501.
- [Aviz86] A. Avižienis and J.-C. Laprie, "Dependable Computing: From Concepts to Design Diversity," *Proceedings of the IEEE*, Vol. 74, No. 5, May 1986, pp. 629-638.
- [Bari83] G. Barigazzi and L. Strigini, "Application-Transparent Setting of Recovery Points," in *Digest of 13th Annual International Symposium on Fault-Tolerant Computing*, Milano, Italy: June 1983, pp. 48-55.
- [Bish86] P.G. Bishop, D.G. Esp, M. Barnes, P. Humphreys, G. Dahll, and J. Lahti, "PODS - A Project of Diverse Software," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 9, September 1986, pp. 929-940.
- [Bjor72] L.A. Bjork and C.T. Davies, "The Semantics of the Presentation and Recovery of Integrity in a Data Base System," IBM, San Jose, California, Tech. Rep. TR 02.540, December 1972.
- [Bjor75] L.A. Bjork, "Generalized Audit Trail Requirements and Concepts for Data Base Applications," *IBM System Journal*, Vol. 14, No. 3, 1975, pp. 229-245.
- [Broe75] R.B. Broen, "New Voters for Redundant Systems," *Journal of Dynamic Systems, Measurement and Control*, Vol. 97, No. 1, March 1975, pp. 41-45.
- [Cain75] S.H. Caine and E.K. Gordon, "PDL - A Tool for Software Design," in *Proceedings National Computer Conference*, 1975.

- [Camp86] R.H. Campbell and B. Randell, "Error Recovery in Asynchronous Systems," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 8, August 1986, pp. 811-826.
- [Chen78a] L. Chen, "Improving Software Reliability by N-Version Programming," Ph.D. Dissertation, UCLA Computer Science Department, Los Angeles, California, Tech. Rep. ENG-7843, August 1978.
- [Chen78b] L. Chen and A. Avižienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," in *Digest of 8th Annual International Symposium on Fault-Tolerant Computing*, Toulouse, France: June 1978, pp. 3-9.
- [CRA85] CRA and RTI, "Redundancy Management Software Requirements Specification for a Redundant Strapped Down Inertia Measurement Unit.," Version 2.0, Charles River Analytics and Research Triangle Institute, May 30, 1985.
- [Cris82] F. Cristian, "Exception Handling and Software Fault Tolerance," *IEEE Transactions on Computers*, Vol. C-31, No. 6, June 1982, pp. 531-540.
- [Dora86] K.H. Dorato, "Coincident Errors in N-Version Programming," Master Thesis, UCLA Computer Science Department, Los Angeles, California, June 1986.
- [Eckh85] D.E. Eckhardt and L.D. Lee, "A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors," *IEEE Transaction on Software Engineering*, Vol. SE-11, No. 12, December, 1985, pp. 1511-1517.
- [Ehri78] H. Ehrig, H. Kreowski, and H. Weber, "Algebraic Specification Schemes for Data Base Systems," in *Proceedings 10th International Conference on Very Large Data Bases*, West-Berlin, Germany: September 1978, pp. 427-440.
- [Glas80] R.L. Glass, "A Benefit Analysis of Some Software Reliability Methodologies," *ACM SIGSOFT Software Engineering Notes*, Vol. 5, 1980, pp. 26-33.
- [Gmei79] L. Gmeiner and U. Voges, "Software Diversity in Reactor Protection Systems: An Experiment," *Proceedings IFAC Workshop SAFECOMP'79*, May 1979, pp. 75-79.

- [Gogu79] J.A. Goguen and J.J. Tardo, "An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications," in *Proceedings Specifications Reliable Software Technology*, Cambridge, Mass.: 1979, pp. 170-189.
- [Gold80] J. Goldberg, "SIFT: A Provable Fault-Tolerant Computer for Aircraft Flight Control," in *Proceedings IFIP Congress 80*, Tokyo, Japan: October 1980, pp. 151-156.
- [Gray86] J. Gray, "Why Do Computers Stop and What Can Be Done About It?," in *Proceedings Fifth Symposium on Reliability in Distributed Software and Database Systems*, Los Angeles, California: January 1986, pp. 3-12.
- [Gray79] J.N. Gray, "Notes on Data Base Operating Systems," in *Operating Systems, An Advanced Course*, R. Bayer, R.M. Graham, and G. Seegmuller, Ed. Berlin, Germany: Springer-Verlag, 1979, pp. 393-481.
- [Grna80] A. Grnarov, J. Arlat, and A. Avižienis, "On the Performance of Software Fault-Tolerance Strategies," in *Digest of 10th Annual International Symposium on Fault-Tolerant Computing*, Kyoto, Japan: 1980, pp. 251-253.
- [Gunn85] P. Gunningberg and B. Pehrson, "Specification and Verification of a Synchronization Protocol for Comparison of Results.," in *Digest of 15th Annual International Symposium on Fault-Tolerant Computing*, Ann Arbor, Michigan: June 1985, pp. 172-177.
- [Gutt85] J.V. Guttag, J.J. Horning, and J.M. Wing, "Larch in Five Easy Pieces," Master Thesis, Digital Equipment Corporation Systems Research Center, Palo Alto, California, Tech. Rep. Report No. 5, July 24, 1985.
- [Hill83] A.D. Hills, "A310 Slat and Flap Control System Management & Experience," in *Proceedings Fifth Digital Avionics Systems Conference*, Seattle, WA: November 1983, pp. 6.7.1-6.7.7.
- [Hopk78] A.L. Hopkins, T.B. Smith, and J.H. Lala, "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft," *Proceedings of the IEEE*, Vol. 66, No. 10, October 1978, pp. 1221-1239.
- [Horn74] J. Horning, H.C. Lauer, P.M. Melliar-Smith, and B. Randell, "A Program Structure for Error Detection and Recovery," in *Lecture Notes In Computer Science, Vol. 16*, New York: Springer-Verlag, 1974, pp. 171-187.

- [Joy86] W.N. Joy, S.L. Graham, C.B. Haley, M.K. McKusick, and P.B. Kessler, "Berkeley Pascal User's Manual," Version 3.1, Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California, April 1986.
- [Kell82] J.P.J. Kelly, "Specification of Fault-Tolerant Multi-Version Software: Experimental Studies of a Design Diversity Approach," Ph.D. Dissertation, UCLA Computer Science Department, Los Angeles, California, Tech. Rep. CSD-820927, September 1982.
- [Kell83] J.P.J. Kelly and A. Avižienis, "A Specification Oriented Multi-Version Software Experiment," in *Digest of 13th Annual International Symposium on Fault-Tolerant Computing*, Milan, Italy: June 1983, pp. 121-126.
- [Kell86] J.P.J. Kelly, A. Avižienis, B.T. Utery, B.J. Swain, R.T. Lyu, A.T. Tai, and K.S. Tso, "Multi-Version Software Development," in *Proceedings IFAC Workshop SAFECOMP'86*, Sarlat, France: October 1986, pp. 43-49.
- [Kim84] K.H. Kim, "Distributed Execution of Recovery Blocks: An Approach to Uniform Treatment of Hardware and Software Faults," in *Proceedings IEEE 4th International Conference on Distributed Computing Systems*, San Francisco, California: May 1984, pp. 526-532.
- [Koo87] R. Koo and S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 1, January 1987, pp. 23-31.
- [Lapr84] J.-C. Laprie, "Dependability Evaluation of Software Systems in Operation," *IEEE Transaction on Software Engineering*, Vol. SE-10, No. 6, November 1984, pp. 701-714.
- [Leff86] S.J. Leffler, R.S. Fabry, and W.N. Joy, "A 4.2BSD Interprocess Communication Primer," Draft, Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California, February 1986.
- [Lisk79] B.H. Liskov and A. Snyder, "Exception Handling in CLU," *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 6, November 1979, pp. 546-558.

- [Luck80] D.C. Luckham and W. Polak, "ADA Exception Handling: An Axiomatic Approach," *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 2, April 1980, pp. 225-233.
- [Maka82] S.V. Makam, A. Avižienis, and G. Grusas, "UCLA ARIES 82 User's Guide," UCLA Computer Science Department, Los Angeles, California, USA, Tech. Rep. CSD-820830, August 1982.
- [Mart82] D.J. Martin, "Dissimilar Software in High Integrity Applications in Flight Controls," in *Proceedings AGARD-CPP-330*, September 1982, pp. 36.1-36.13.
- [Merl78] P.M. Merlin and B. Randell, "Consistant State Restoration in Distributed Systems," in *Digest of 8th Annual International Symposium on Fault-Tolerant Computing*, Toulouse, France: June 1978, pp. 129-134.
- [Mili85] A. Mili, "Towards a Theory of Forward Error Recovery," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 8, August 1985, pp. 735-748.
- [Parn72a] D.L. Parnas, "A Technique for Software Module Specification with Examples," *Communications of the ACM*, Vol. 15, No. 5, May 1972, pp. 330-336.
- [Parn72b] D.L. Parnas, "On the Criteria to be used in Decomposing Systems into Modules," *Communications of the ACM*, Vol. 15, No. 12, December 1972, pp. 1053-1058.
- [Pope81] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel, "LOCUS: A Network Transparent, High Reliability Distributed System," in *Proceedings 8th Symposium on Operating Systems Principles*, Pacific Grove, California: December 1981, pp. 169-177.
- [Rand75] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, June 1975, pp. 220-232.
- [Rand78] B. Randell, P.A. Lee, and P.C. Treleaven, "Reliability Issues in Computing System Design," *Computing Surveys*, Vol. 10, No. 2, June 1978, pp. 123-165.

- [Russ80] D.L. Russell, "State Restoration in Systems of Communicating Processes," *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 2, March 1980, pp. 183-194.
- [Siew84] D.P. Siewiorek, "Architecture of Fault-Tolerant Computers," *Computer*, Vol. 17, No. 8, August 1984, pp. 9-18.
- [Swai86] B.J. Swain, "Group Branch Coverage Testing of Multi-Version Software," Master Thesis, UCLA Computer Science Department, Los Angeles, California, Tech. Rep. CSD-860013, December 1986.
- [Tai86] A.T. Tai, "A Study of the Application of Formal Specification for Fault-Tolerant Software," Master Thesis, UCLA Computer Science Department, Los Angeles, California, June 1986.
- [Tayl81] R. Taylor, "Redundant Programming in Europe," *ACM SIGSOFT Software Engineering Notes*, Vol. 6, No. 1, January 1981.
- [Toy78] W.N. Toy, "Fault-Tolerant Design of Local ESS Processors," *Proceedings of the IEEE*, Vol. 66, No. 10, October 1978, pp. 1126-1145.
- [Wens78] J.H. Wensley, L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shostak, and C.B. Weinstock, "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proceedings of the IEEE*, Vol. 66, No. 10, October 1978, pp. 1240-1255.
- [Youn84] L.J. Yount, "Architectural Solutions to Safety Problems of Digital Flight-Critical Systems for Commercial Transports," *Proceedings AIAA/IEEE Digital Avionics Systems Conference and Technical Display*, December 1984, pp. 1-8.

APPENDIX A
RSDIMU Defined Constants

```
const

    alb = 1;           { bounds for axes of }
    aub = 3;           { frames of reference arrays }

    baddat = 9999.000;

    chlb = 1;
    chub = 4;          { bounds for channel array }

    clb = 1;
    cub = 50;          { bounds for calibration array }

    dlb = 1;
    dub = 3;           { bounds for display arrays }

    elb = 1;           { bounds for misalignment }
    eub = 6;           { angle arrays }

    flb = 1;
    fub = 4;           { bounds for face-oriented arrays }

    g = 32.0;          { gravitational constant in ft/sec2}

    maxint = 65535;

    modemin = 0;
    modemax = 99;      { range values for display modes }

    mlb = 0;           { bounds for unsigned }
    mub = maxint;      { 16 bit machine integer }
```

```
nsigmin = 3;
nsigmax = 7;      { bounds for nsigt }

pi = 3.1415926535;

pairmin = 0;
pairmax = 6;      { bounds for face-pair arrays }

slb = 1;           { bounds for sensor }
sub = 8;           { and related arrays }
```

APPENDIX B
RSDIMU Defined Types

type

```
{ base types for portability }
iint = integer;
ireal = real;

{ index types }
aindex = alb..aub;
chindex = chlb..chub;
cindex = clb..cub;
dindex = dlb..dub;
eindex = elb..eub;
findex = flb..fub;
modet = modemin..modemax;
nsigset = nsigmin..nsigmax;
pairt = pairmin..pairmax;
sindex = slb..sub;

mint = mlb..mub;
    { represents a 16-bit nonnegative machine integer }

system = (normal, analytic, undefined);
    { computational modes }

ararray = array [aindex] of ireal;
    { holds information for 3 axes }

cmarray = array [sindex, cindex] of mint;
    { holds calibration data points for 8 sensors }

cparray = array [chindex] of pairt;
    { holds facepair used to compute channel estimate }
```

```

dmarray = array [dindex] of mint;
    { holds 'words' of display information }

erarray = array [findex, eindex] of ireal;
    { holds misalignment angles of 8 sensors }

frarray = array [findex] of ireal;
    { holds temp or normface of 4 faces }

sbarray = array [sindex] of boolean;
    { holds sensor failure indications }

smarray = array [sindex] of mint;
    { holds count data for 8 sensors}

srarray = array [sindex] of ireal;
    { holds slope coefficients for 8 accel's }

state = record
    status: system;
    acceleration: ararray
end;
    { holds one vehicle state estimation }

vsearray= array [chindex] of state;
    { holds vehicle state estimation of 4 channels }

```


APPENDIX C
RSDIMU Input and Output Variables

```
var

  { input variables }

  obase : ireal;
         { semi-octahedron base }

  offraw : carray;
         { calibration data for 8 sensors }

  linstd : mint;
         { noise standard deviation for sensors }

  linfailin : sbarray;
         { accererometer failure initial conditions }

  rawlin : smarray;
         { raw data for acceleration computation }

  dmode : modet;
         { display mode }

  temp : frarray;
         { current temperature on each face }

  scale0, scale1, scale2 : srarray;
         { linear sensor slope coefficients }

  misalign : erarray;
         { sensor misalignment angles }

  normface : frarray;
         { accelerations normal to i faces }
```

```

nsigt : nsigset;
        { noise tolerance }

phiv, thetav, psiv : ireal;
        { n to v rotations }

phii, thetai, psii : ireal;
        { v to i rotations }

{ output variables }

linoffset : srray;
        { sensor offset values }

linnoise : sbrray;
        { sensor calibration results }

linout : srray;
        { individual sensor outputs }

linfailout : sbrray;
        { failure detection results }

sysstatus : boolean;
        { operational status of system }

dismode : mint;
        { display panel mode }

disupper : dmarray;
        { upper display panel encodings }

dislower : dmarray;
        { lower display panel encodings }

bestest : state;
        { vehicle state estimate using
          all operational sensors}

chanest : vsearray;
        { vehicle state estimates for the 4 channels }

chanface : cparray;
        { maps face pairs to channels }

```