

**Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596**

A DESIGN PARADIGM FOR MULTI-VERSION SOFTWARE

Michael Rung-Tsong Lyu

**October 1988
CSD-880076**

UNIVERSITY OF CALIFORNIA

Los Angeles

A Design Paradigm for Multi-Version Software

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

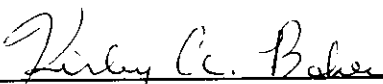
by

Michael Rung-Tsong Lyu


1988

© Copyright by
Michael Rung-Tsong Lyu
1988

The dissertation of Michael Rung-Tsong Lyu is approved.




Kirby A. Baker



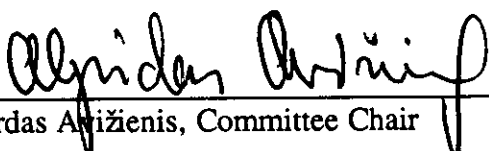
Harold Borko



Mario Gerla



David A. Rennels



Algirdas Avizienis, Committee Chair

University of California, Los Angeles

1988

To my parents, Bor-Hua Lyu and Yu-Ying Chang.

TABLE OF CONTENTS

page

1	INTRODUCTION	1
1.1	The Need for and Approaches to Dependable Computing	1
1.2	Fault Tolerance and Software Fault Tolerance	2
1.3	Multi-Version Software for Fault Tolerance	3
1.4	Exploring Design Diversity in Multi-Version Software	3
1.5	A Design Paradigm	5
1.6	Evaluation of the Design Paradigm by Experimentation	6
1.7	Objectives in this Research	6
2	PREVIOUS NVP EXPERIMENTS AND MVS DESIGNS	7
2.1	Experimental Approach to Design Diversity	7
2.2	Experiments Performed at UCLA	8
2.2.1	Step One: Developing a Methodology	8
2.2.2	Step Two: Impact of Specification	9
2.2.3	Step Three: the NASA/Four-University MVS Experiment	10
2.3	Related Experiments at Other Sites	11
2.3.1	Gmeiner and Voges' Experiment	11
2.3.2	PODS Experiment	12
2.3.3	Knight and Leveson's Experiment	12
2.4	Practical Design of Software Diversity Applications	13
2.4.1	ERICSSON Safety Systems for Railway Control	14
2.4.2	Airbus A310	16
2.4.3	A320 Pitch Control	19
2.4.4	Honeywell/Sperry System SP-300 and Its Successors	20
2.5	Diversity Concerns of Previous Experiments and Designs	22
3	A DESIGN PARADIGM	25
3.1	The Need for a Formal Theoretical Basis	25
3.2	An Overview of the Design Paradigm	25
3.3	State Precise Requirements with Dependability Goals	26
3.4	Define Method of MVS Supervision and Execution Environment	28
3.5	Choose Design Diversity Dimensions	29
3.6	Write High Quality Specifications	30
3.7	Install Error Detection and Recovery Algorithms	31
3.7.1	The Proposed Algorithm	32
3.7.2	Cross-Check Points	32
3.7.3	Recovery Points	33
3.7.4	Introducing Cc-Points and Recovery Points in the Specification	34
3.7.5	Placements of Cc-Points and Recovery-Points	35
3.8	Avoid Uncontrolled Commonalities	36
3.9	Build the Software Versions	37
3.10	Conduct High Quality Testing	39
3.11	Execute Multi-Version Software Systems	39
3.12	Evaluate Effectiveness of MVS	40
3.13	Refine the Design by Iteration	41
3.14	Choose and Implement Modification (Maintenance) Policy	41

4 AN INDUSTRIAL INVESTIGATION	43
4.1 A Practical and Complete Experiment Using the Design Paradigm	43
4.2 The Automatic Landing Problem	45
4.3 Applying the MVS Design Paradigm	48
4.3.1 State Precise Requirements with Dependability Goals	48
4.3.1.1 Requirements for Software Testing	48
4.3.1.2 Apply Design Diversity to Achieve a Dependability Goal	50
4.3.2 Define Method of MVS Supervision and Execution Environment	51
4.3.3 Choose Design Diversity Dimensions	55
4.3.4 Write High Quality Specifications	56
4.3.5 Install Error Detection and Recovery Algorithms	58
4.3.6 Avoid Uncontrolled Commonalities	61
4.3.6.1 Recruit Qualified Personnel	62
4.3.6.2 Define A Formal Communication Protocol	62
4.3.6.3 Experience with the Communication Protocol	64
4.3.7 Build the Software	66
4.3.7.1 Schedule of the Experiment	66
4.3.7.2 The Programming Process	68
4.3.8 Conduct High Quality Testing	69
4.3.9 Execute Multi-Version Systems	74
4.3.10 Evaluate Effectiveness of MVS and Refine the Process	77
5 EVALUATION AND REFINEMENT	78
5.1 Standard Software Metrics of the Programs	78
5.2 Distribution of Faults Detected during Program Development	80
5.3 Evaluation of the Six Programs during Operational Testing	84
5.3.1 Disagreements Detected by Flight Simulations	84
5.3.2 Faults Found During Inspection of Code	85
5.4 Assessment of Structural Diversity	88
5.5 Observations from the Diversity Assessment	94
5.6 Fault Diagnosis and Failure Analysis by Mutation Testing	96
5.7 Coverage Measurement of Multi-Version Software	103
5.8 Evaluation of Fault Tolerance Provisions in the Applications	105
5.9 A Comparison of Three Recent Experiments	108
5.10 Refinements of the Design Paradigm	112
6 CONCLUSIONS AND FUTURE WORK	115
6.1 Original Contributions of This Research	115
6.2 Practicality of the Proposed Design Paradigm	116
6.3 Effectiveness of the Design Paradigm	116
6.4 Future Research Issues in MVS	117
6.4.1 Identify and Avoid Commonalities	117
6.4.2 Measure and Promote Design Diversity	117
6.4.3 Developing Support Tools and Techniques	118
6.4.4 Exploiting the Presence of Multiple Versions for V&V	121
6.4.5 Cost-Effectiveness Measurement and Assessment	122
6.4.6 Incorporating Security Concerns into the Design Paradigm	123
REFERENCES	124

LIST OF FIGURES

	page
Figure 1-1: Multi-Version Software Systems	4
Figure 2-1: Safety Layout of Two Computer Based Interlocking Systems	15
Figure 2-2: ERICSSON Work Organization	17
Figure 2-3: A310 System Configuration (Flap only)	18
Figure 2-4: Slat/Flap Control System Principle	19
Figure 2-5: A320 Pitch Control	21
Figure 2-6: Honeywell/Sperry's Dual Architecture	23
Figure 2-7: Protection from Generic Software and Processor Faults	24
Figure 3-1: A Design Paradigm for Multi-Version Software	27
Figure 4-1: Pitch Control System Functions and Data Flow Diagram	46
Figure 4-2: The Usage of the Cross-check Points and Recovery Point	60
Figure 4-3: Communication Diagram of the UCLA/Sperry Experiment	65
Figure 4-4: 3-Channel Flight Simulation Configuration	75
Figure 5-1: System Data Flow Diagram of RSDIMU	106
Figure 6-1: Classification of Design Diversity	119

LIST OF TABLES

	Page
Table 1: Summary of the UCLA Programmer and Coordinator Background	63
Table 2: Different Schemes Used in the Testing Phases	70
Table 3: Test Data in Unit Test Phase	71
Table 4: Test Data in Integration Test Phase	72
Table 5: Test Data in Acceptance Test Phase	73
Table 6: Software Metrics for the Six Programs	79
Table 7: Fault Distribution by Subfunctions	81
Table 8: Fault Classification by Fault Types	82
Table 9: Fault Classification by Phases	82
Table 10: Fault Classification: Requirements Faults vs. Structural Faults	83
Table 11: Potential for and Observed Diversity	89
Table 12: Error Frequency Function of Each Mutant	101
Table 13: Error Severity Function of Each Mutant	102
Table 14: Reduced Error Similarity Function Matrix in Two-Mutant Sets	103
Table 15: A Comparison of Three Experiments – The Scale	109
Table 16: A Comparison – MVS Software Development Procedure	110
Table 17: A Comparison – Testing and Processing of MVS Systems	111

ACKNOWLEDGEMENTS

I would like to thank my committee members, Professors Algirdas Avižienis, David Rennels, Mario Gerla, Harold Borko, and Kirby Baker for serving on my dissertation committee with great 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. Also thanks to Professor John P. J. Kelly for his early advice and support of this research.

Special thanks to my pal, Mark K. Joseph, who helped and pushed me so hard to complete this dissertation in time. Also to members of the UCLA Dependable Computing and Fault-Tolerant Systems Research Group, Kam Sing Tso, Werner Schütz, Ann Tai, Johnny J. Chen, Chi-Sharn Wu, Bradford T. Ulery and Barbara Swain, for invaluable discussions and support that have made this research possible. It is also my pleasure to acknowledge Mr. John F. Williams for his help and encouragement to my work.

Of course I cannot emphasize too much the importance of the dedication of my lovely wife, Chih-Fen, together with our coming baby.

This work was supported by the Sperry Flight Control Systems Division of Honeywell, Inc. and the State of California "MICRO" program, grant #86-091, under the direction of Professor A. Avižienis, Principal Investigator.

VITA

- March 19, 1959 Born, Taipei, Taiwan, Republic of China.
- 1981 B.S. in Electrical Engineering,
National Taiwan University
- 1984 M.S. in Electrical and Computer Engineering,
University of California, Santa Barbara
- 1983-1984 Teaching Assistant,
UCSB Computer Science Department
- 1985-1988 Research Assistant, Post Graduate Research Engineer,
Dependable Computing and Fault-Tolerant Systems Laboratory,
UCLA Computer Science Department

PUBLICATIONS

- 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.
- A. Avižienis, M.R. Lyu, W. Schütz, K.S. Tso, and U. Voges, "DEDIX 87 - A Supervisory System for Design Diversity Experiments at UCLA," in *Software Diversity in Computerized Control Systems*, Springer-Verlag Wien New York, 1988, pp. 129-168; also *UCLA technical report No. CSD-870029*, July 1987.
- A. Avižienis, M.R. Lyu, and W. Schütz, "In Search of Effective Diversity: A Six-Language Study of Fault-Tolerant Flight Control Software," in *UCLA technical report No. CSD-870060*, November 1987.
- A. Avižienis and M.R. Lyu, "Design and Evaluation of Fault-Tolerant Multi-Version Software," *Annual National Joint Conference on Software Quality and Reliability*, Arlington, Virginia, March, 1988.

A. Avižienis, M.R. Lyu, and W. Schütz, "Multi-version Software Development: A UCLA/Honeywell Joint Project for Fault-Tolerant Flight Control Software," *UCLA technical-report No. CSD-880034*, May 1988.

A. Avižienis, M.R. Lyu, and W. Schütz, "In Search of Effective Diversity: A Six-Language Study of Fault-Tolerant Flight Control Software," in *Proceedings of the Eighteenth Annual International Symposium on Fault-Tolerant Computing*, Tokyo, Japan, June, 1988.

ABSTRACT OF THE DISSERTATION

A Design Paradigm For Multi-Version Software

by

Michael Rung-Tsong Lyu

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1988

Professor Algirdas Avižienis, Chair

In the multi-version software (MVS) implementation of fault-tolerant software, design faults are detected and masked through the consensus of results from two or more independently designed and functionally equivalent versions. To maximize the effectiveness of the MVS approach, the probability of similar errors that coincide at multi-channel decision (consensus) points should be reduced to the lowest possible value. *Design diversity* is potentially an effective method to get this result. It has been the major concern and effort of the author to formulate a set of rigorous guidelines, or a *design paradigm*, for the application and implementation of design diversity in building MVS systems for practical applications.

This design paradigm was developed and formulated based on the results of fault-tolerant software research conducted at UCLA since 1975. The most recent of these studies was a NASA-sponsored four-university MVS experiment, in which the author participated as a site coordinator. Combining software engineering disciplines and fault tolerance techniques, this design paradigm offers rigorous guidelines for the design and implementation of MVS systems.

Using this design paradigm, the author took a major part in the design, coordination and evaluation of a UCLA/Honeywell joint research project on a large-scale MVS system, employing six different programming languages to create six versions of software for pitch control in an automatic aircraft landing program. The rationale, preparation, execution, and evaluation of this project are reported in detail. Moreover, the assessment and refinement of the proposed design paradigm are also presented as results from this project.

CHAPTER 1

INTRODUCTION

1.1 The Need for and Approaches to Dependable Computing

Since the first computer was invented some forty years ago, men have been depending more and more on computers in their daily lives. When the requirements for computers increase, the crises of computer failures also increase. The impact of hardware and software failures range from inconvenience (e.g., malfunctions of home appliances), economic loss (e.g., interceptions of banking systems) to life-threatening (e.g., failures of flight systems). To obtain highly dependable computer systems in such critical applications as nuclear power plant control and aircraft flight control, is a major concern for today's technically advanced societies.

Generally speaking, there are four ways to effect dependable computing [Aviz86] :

- **fault avoidance:** to prevent, *by construction*, fault occurrence;
- **fault tolerance:** to provide, *by redundancy*, service complying with the specification in spite of faults having occurred or occurring;
- **fault removal:** to minimize, *by verification*, the presence of faults;
- **fault forecasting:** to estimate, *by evaluation*, the presence, the occurrence, and the consequences of faults.

Due to their intrinsic complexity, most computing systems currently apply a combination of the above methods for the delivery of dependable service.

1.2 Fault Tolerance and Software Fault Tolerance

Since human activities are imperfect, there is always a need to incorporate redundancy techniques into human designs. Fault tolerance is one such survival attribute of computer systems, which allows delivery of expected service after faults have manifested themselves within a system [Aviz78].

Since the first generation, computer systems have been designed to tolerate faults in hardware components [Renn84, Siew84]. As computer systems get more complicated, the incidence of design faults has grown. In traditional single-version software environment, the provision of fault tolerance mechanisms was enhanced by introducing special fault detection and recovery features, such as program modularity, system closure, atomicity of actions, decision verification, and exception handling [Aviz85b]. While these mechanisms have been explored to their limitations, the achievement of highly reliable software are still much needed.

Since the early 1970s, researchers have been exploring the possibilities of employing similar redundancy methods to create systems tolerant to design faults in software components in a more efficient way. This gave birth to the idea of multi-version software (MVS) † systems [Aviz75].

† The process of generating MVS is also known as NVP, N-Version Programming, initially called "redundant programming."

1.3 Multi-Version Software for Fault Tolerance

The NVP approach to fault-tolerant software systems involves the generation of functionally equivalent, yet independently developed and maintained software components, to be called multi-version software (MVS) [Aviz77]. These components are executed concurrently under a supervisory system that uses a decision algorithm based on consensus to determine final output values [Aviz85c]. Whenever probability of similar errors is minimized, distinct, erroneous results tend to be masked by a majority vote during MVS execution. This is shown in Figure 1-1.

MVS systems are gaining acceptance in critical application areas such as the aerospace industry [Mart82, Youn84, Hill85, Rouq86], nuclear power industry [Rama81, Bish85, Voge85], and ground transportation industry [Tayl81, Hage88]. The construction of such systems is still, however, done mostly in an ad hoc manner. There does not exist a definite procedure which would guarantee that MVS systems could be engineered and evaluated to be software fault-tolerant systems.

1.4 Exploring Design Diversity in Multi-Version Software

Design diversity [Aviz82] is a potentially effective method to avoid identical errors that are caused by design faults in multiple computation systems [Aviz85b]. To build such systems, independent programming efforts are carried out by individuals or groups that do not interact with respect to the programming process. Particularly, different algorithms, programming languages, environments, implementation techniques and tools are used in each effort wherever possible. The goal is to minimize the probability of identical or similar errors at a decision point in the execution of an MVS system. Exploring different dimensions of diversity, or a

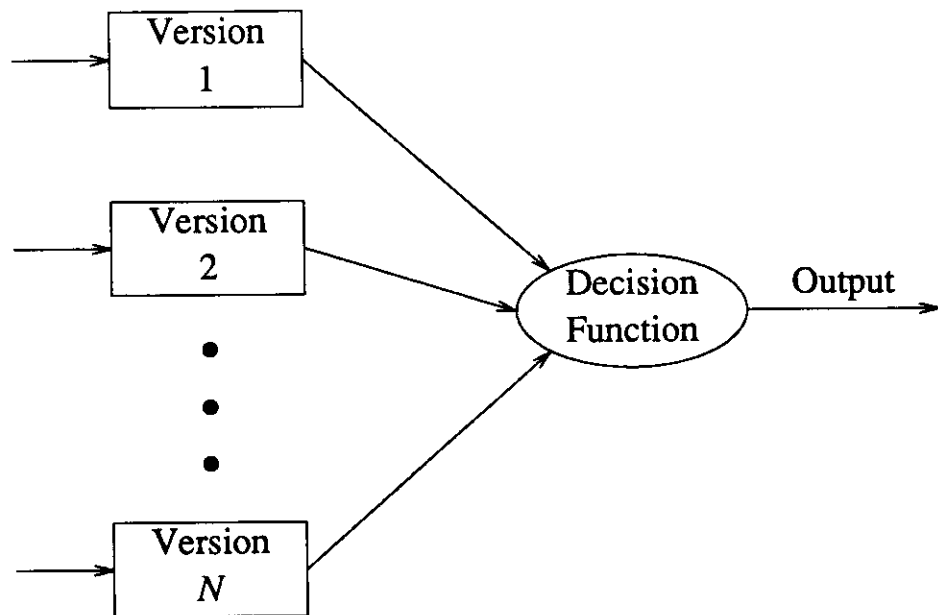


Figure 1-1: Multi-Version Software System

combination of them, may provide a high degree of diversity to the software versions. This could lead to major improvements in system dependability when the multi-version software thus developed is put to use.

1.5 A Design Paradigm

The word "paradigm," used in the dictionary sense, means "pattern, example, model", which refers to a set of guidelines with illustrations. The unifying theme of fault tolerance concept since its introduction [Aviz67] has been that the design of fault-tolerant systems is most likely to succeed if a methodical approach is employed. This suggests that a *design paradigm* is essential to guide the designer in considering fault tolerance as a fundamental issue throughout the design process [Aviz72]. The most recent version of such a design paradigm was presented in [Aviz87a].

Moreover, research results and design experiences accumulated over the past few years point to the need for another design paradigm for the implementation of MVS systems, in order to achieve efficient tolerance of *design faults* in computer systems. The formation of this design paradigm has been revealed in several previous research activities. It is the major goal of this research to abstract and further refine the methodology as developed earlier [Aviz77, Chen78a, Aviz84, Kell86] and then state it as a the design paradigm by combining the knowledge and experience obtained from both software engineering techniques and fault tolerance experiments. Such a design paradigm should be a general scheme for building MVS systems in order to gain acceptance in industry.

1.6 Evaluation of the Design Paradigm by Experimentation

Once the design paradigm is formulated, it is important to evaluate and refine it through realistic applications. Evaluation of the paradigm covers the whole life cycle of the design process itself. Due to cost reasons, an effective approach is to apply it through experimentation, by either pre-constructing or prototyping a system. However, these experimental investigations should reflect a real-world situation, and should be very carefully conducted in a well-planned and controlled process. We shall review the previous experiments and designs that have been conducted to develop MVS, and present a UCLA/Honeywell joint project which thoroughly applied the proposed design paradigm for the purposes of evaluation and refinement.

1.7 Objectives in this Research

In conclusion, the major activities of this research are:

- (1) To state a design paradigm for MVS systems;
- (2) To conduct MVS development, following the paradigm and industrial software engineering standards; and
- (3) To evaluate and refine the paradigm as it was used in (2) above.

Other major concerns include exploring different dimensions of diversity that researchers can investigate. The impact of different programming languages is of special interest.

CHAPTER 2

PREVIOUS NVP EXPERIMENTS AND MVS DESIGNS

This chapter will survey previous experiments † in generating multi-version software, including those conducted at UCLA, and those conducted at other sites. Some practical applications of design diversity will be reviewed as well.

2.1 Experimental Approach to Design Diversity

The scarcity of previous results and an absence of unified design guidelines on MVS systems led to the choice of an experimental approach to implementation of design diversity: to choose some conveniently accessible programming tasks, and then proceed to generate a set of programs. Once generated, the programs were executed as MVS units in a simulated system. The resulting observations were applied to refine the developmental procedures and to enhance the concepts of MVS systems.

There are two advantages in this approach: First, it is easier to control and observe the life-cycle of the whole procedure in a shorter period of time. Second, it is more cost-effective and less risky in gaining the insights of this design technique, especially at its very early stage.

† Throughout the text, we use the word "experiment" as a general term to represent "experimental investigation", instead of its original meaning of "experiment using formal validation procedures".

2.2 Experiments Performed at UCLA

A number of experiments to investigate the role of MVS in the production of fault-tolerant software have been performed at UCLA since 1975 [Aviz77].

2.2.1 Step One: Developing a Methodology

The objectives of this initial MVS research effort at UCLA (1975-1978) were: 1) to study the feasibility and effectiveness of this strategy, and 2) to identify problems or difficulties in using MVS as a means to tolerate software design faults. Two experiments were carried out to study the ease of implementing an MVS system, to gain data on its effectiveness, and to identify problems or difficulties in this approach. The first involved 27 distinct programs of a text editor, and the second used three different algorithms to enforce another aspect of diversity among the resulting 16 programs of a temperature estimation application. Both applications were specified in English and were programmed in PL/1 by students in satisfaction of their class projects.

It was found that the implementation of a MVS system was relatively simple and could be generalized to other applications. Some 3-version systems were effective in preventing failure due to defects in only one of the versions. A number of difficulties were identified, some of which were simply unanticipated implementation constraints. Specification ambiguities and missing logic were discovered to be causes of some similar errors. A hypothesis was raised that MVS could be more effective if applied to the specification or testing phases of the software development rather than only to run-time fault masking. Further research in this approach was deemed necessary. This exploratory research demonstrated the practicality of the MVS

approach and the need for high quality software specifications [Chen78b, Chen78a].

2.2.2 Step Two: Impact of Specification

A common thread running throughout these experiments was the role of specification ambiguities or errors in causing program errors. The objectives of the next phase of UCLA research (1979-1982) concentrated on the investigation of the relative applicability of various software specification techniques. An experiment was conducted at UCLA in 1981 to study the role of specifications in MVS systems [Kell82, Kell83]. Three specification languages were used to compare the effects of using different specification techniques in MVS. They included a formal specification written in OBJ [Gogu79], a non-formal specification written in PDL [Cain75], and a "control" specification written in English. Eighteen programs designed to perform airport scheduling transactions were developed, with an average length of about 500 lines of PL/1 code.

This research revealed again that the MVS approach was 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. Such errors were the most difficult to detect and occurred most often in versions written from the English specification. Also, a few similar errors were discovered to have been caused by *distinct* faults in the different programs.

In addition, it was observed that cosmetic errors (e.g., output misalignment or spelling errors) were produced by several programs in disregard of the specification [Aviz84]. Such errors could potentially cause the decision function to consider

functionally equivalent results as distinct (since a character by character comparison will determine that the results are different). This points to the need for stronger emphasis on *exact* compliance with the specification, and for more sophisticated certification tests under more powerful decision algorithms when non-numerical output is produced.

2.2.3 Step Three: the NASA/Four-University MVS Experiment

Recognizing the importance of software reliability in modern computer technologies, the NASA Langley Research Center sponsored the second generation experiment in fault-tolerant software [Kell86, Kell88]. Four universities (North Carolina State University, UCLA, University of Illinois at Urbana-Champaign, University of Virginia) took part in this experiment. Two other independent institutes (Research Triangular Institute, Charles River Analytics) wrote the specifications and provided tools for testing the modules. Each of the four universities supplied five two-programmer teams to work on the same specification of an aircraft control unit during the summer of 1985. Each of these 20 software versions was required to pass an initial acceptance test that included 75 test cases. This large-scale experiment was designed to evaluate the performance of the MVS approach to fault-tolerant software in a realistic application developed under a controlled software development process that is indicative of industry practice.

A software development process, controlled as uniformly as possible across all four universities, was designed to reflect standard industry practice. In order to maintain a consistent and controlled environment for the 40 programmers and yet ensure independence of the development efforts, the following formal environment was established. The programmers were allowed for ten weeks from the time they

received the specification in which to produce a software version that could pass the acceptance test. These ten weeks were nominally divided into four phases: 1) *design*; 2) *coding*; 3) *testing phase*; and 4) *preliminary acceptance test*. At the end of this period, all 20 programs passed the preliminary acceptance test which was later proved deficient, and a subsequent certification phase was ruled to be necessary. The average length of these programs was 2500 lines of code, ranging from 1600 to 4800.

2.3 Related Experiments at Other Sites

2.3.1 Gmeiner and Voges' Experiment

Gmeiner and Voges reported an experiment in which software diversity was applied to a prototype implementation of a reactor protection system [Gmei79, Voge88]. Three programs were generated, one in IFTRAN, one in Pascal, and one in PHI2 (a structured macro assembly language), in hopes of incorporating diversity into the programs both through unique programmers and languages. Each program was tested first by its programmer, then by another person, and then in the 3-version system (by automatic result comparison). It was discovered that some errors were detected only through the automatic result checking of the 3-version system, and that most of these errors were caused by ambiguities in the specification. Unfortunately, no mention was made of any attempt to locate coincident errors in the 3-version system.

2.3.2 PODS Experiment

The Project on Diverse Software, as known as PODS [Bish86], was conducted in Europe to investigate the impact of a number of software development techniques on software reliability. The application was a nuclear reactor safety system. Three programs were written in two languages (FORTRAN and assembly language), and two different specifications were used. Each program was developed under current software development methods, which included a top-down design approach, documentation, and inspection at each stage of development. After a thorough testing of the individual versions, the three versions were run together to reveal discrepancies between the programs. Some faults were discovered by this "back-to-back" testing which had not been uncovered by the standard development techniques. Seven faults were discovered which were related to the specifications; two of them were due to an error in one specification, and thus caused errors in the two programs which had been developed based on that specification. The other five faults were related to ambiguities in one of the specifications, and each occurred in only one of the versions. Thus, the errors resulting from such faults were masked by the correct results produced by the other two versions. Two additional faults uncovered by the back-to-back testing had been introduced while making corrections to the versions. These faults also occurred in only one of the versions and thus resulted in errors masked by the MVS system.

2.3.3 Knight and Leveson's Experiment

Knight and Leveson conducted an experiment based on their criticism of the feasibility of MVS systems [Knig86]. The experiment was conducted as a class project at the University of Virginia and the University of California, Irvine.

Twenty-seven program versions written in Pascal were developed according to a common specification of a launch interceptor function. The length of the resulting programs ranged from 327 to 1004 lines of code. Each program was required to pass 200 pieces of test data for acceptance. There were 241 Boolean results to be computed. A failure for a particular version on a particular test case was recorded when there was any discrepancy between the 241 results produced by that version and those produced by a "gold" program.

From the results of this experiment the investigators observed that the number of coincident failures in this set of independently developed programs was too high. They concluded that the reliability of an MVS system may not be as high as predicted by the assumption of independent failures, an assumption which they felt most people would take for granted while applying MVS techniques.

2.4 Practical Design of Software Diversity Applications

Some architectural designs with MVS approaches have recently been proposed and developed for practical applications. An overview of some applications is given here. The applicability of this technique has been demonstrated by the wide variety of usage of diversity in different applications, especially in safety-critical environments. In general, there is a strong acceptance in industry and government of the assumption that this technique is a cost-effective and necessary one to increase the dependability of computer systems.

2.4.1 ERICSSON Safety Systems for Railway Control

The application to software diversity in industrial applications (beyond an experimental stage) was done first in ERICSSON Safety System for railway control [Ande81, Hage88]. The ERICSSON safety system applies two different packages of software. They are both executed in the same process and the results are compared, which must be equal. This principle is now applied in computerized interlockings and automatic train control systems used in several European countries.

The basic design principle in ERICSSON safety systems is *diversity*. In software this means that *vital* functions (which should work in a fail-safe manner) are designed independently by two teams. Both systems are executed in the processors. The results of the calculations have to match before they are considered correct.

Figure 2-1 shows how diversity is used in the interlocking system. The two interlocking systems (A and B) are running in the interlocking computer. The comparators and detectors are built together into units called object controllers. Each signal and point machine in the station has an individual object controller.

Diversity is achieved by independent teams and design rules. Diversity is enforced by external coding of data and different data organization. External coding is achieved by coding the status messages from the object controllers. The Hamming distance between A-message and B-message is in most cases equal to 4. The two design teams are manned with different people. People are not allowed to move between the teams. In some extreme cases they are even located in different cities. Moreover, each team produces its own set of documentation.

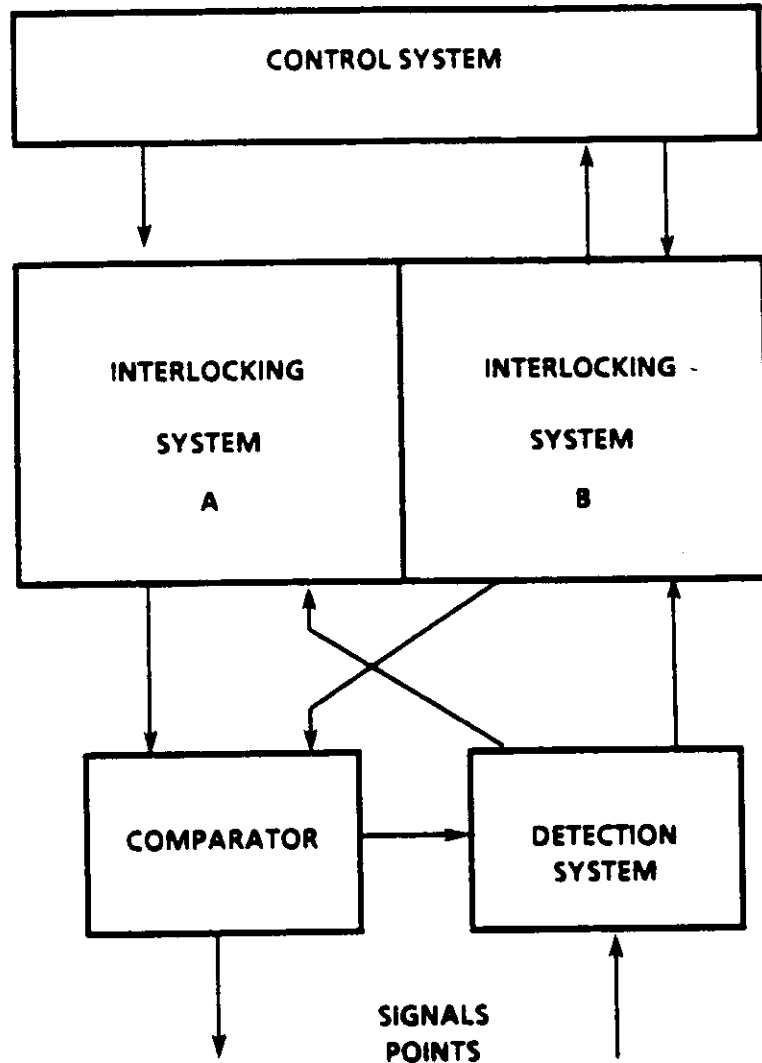


Figure 2-1: Safety Layout of Two Computer Based Interlocking Systems

Safety validations are performed during all phases in development work (see Figure 2-2). Validations are based on hearings. Fault Tree Analysis (which is based on the questions "what can be dangerous and what can cause danger") and Fault Effect Analysis (which is based on the questions "what happens if ...") are used. One observation points out that it is very important to have a stringent development method to utilize diversity to its greatest extent within software design.

The ERICSSON experience indicates that the diversity approach is cost-effective: although cost of some software development steps is doubled (e.g., program specification, coding, and program test), cost of other steps is not (e.g., requirement specification, system specification, test specification, and system test). Most importantly, they conclude that diversity *does* pay. Up till now they have detected one error that could have been dangerous if the software were implemented in the traditional single-version, non-diversity approach.

2.4.2 Airbus A310

The use of software diversity in the Airbus A310 is shown in [Wrig86, Trav88]. The slat/flap control system of the A310 consists of two functionally identical computers with diverse hardware (see Figure 2-3). Within each computer, two diverse programs are executed with results compared via AND-logic (see Figure 2-4), which has a time-window mechanism to provide the synchronization of the two versions.

The diversified architecture is chosen because 1) high integrity requirements exist; 2) the availability is a less stringent requirement; 3) the extent of the task and the ability to use avionic grade microprocessors; and 4) the certification risk. Design

Activity	Documents	Safety Reviewing	Safety Documents
Requirements Specification	Requirements Specification	Review	Inspection Record
	System Test Specification	Review	Inspection Record
System Design	System Description Addition	Review	Inspection Record
Program Specification	Block Specs A Block Specs A Addition	Review	Check Report, Descr. of Principles for Progr.
Programming	Code Lists Code Lists	Procedure	Descr. of Work Routines
Module Tests Block Tests	Test Report Test Report		
System Test	Test Report	Test	Test Report (Checked)

Figure 2-2: ERICSSON Work Organization

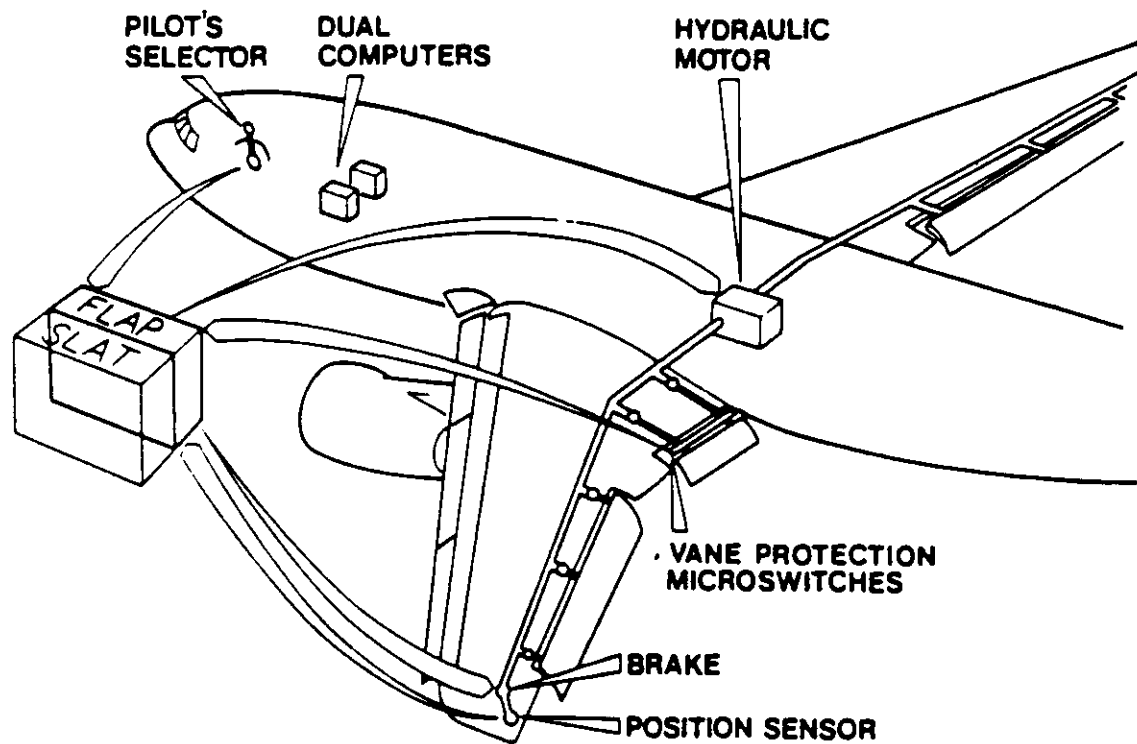


Figure 2-3: A310 System Configuration (Flap only)

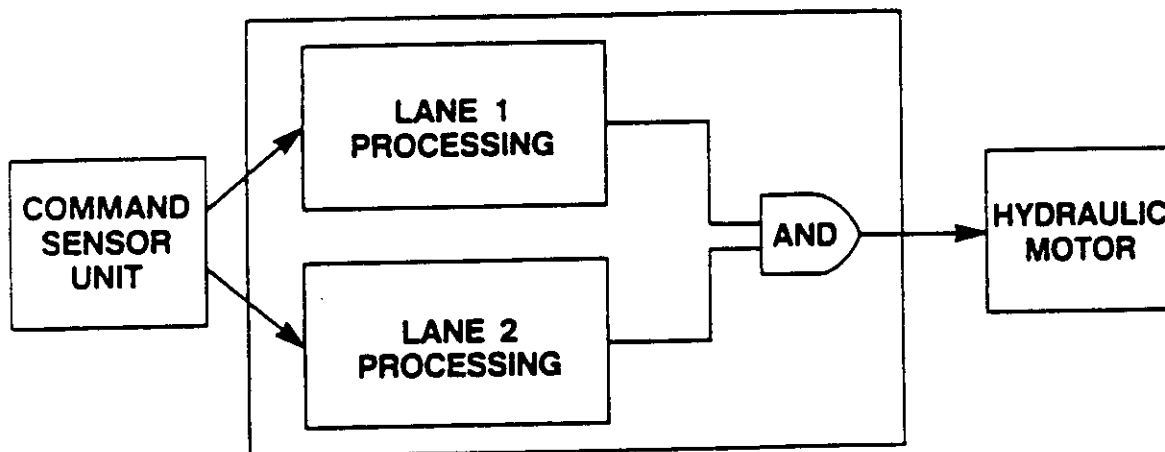


Figure 2-4: Slat/Flap Control System Principle

diversity is ensured by independent design teams, use of diverse hardware and separate host facilities for the software design environment. Besides applying very rigorous testing before the release of the system, there is a continuous real-time test during the use of the system.

Software diversity is applied to high level languages. Since two different languages are used, validation of compiler is considered unnecessary. The use of high level languages also increases the software productivity. As a further fault tolerance means, the outputs of the two programs are compared not only with each other, but also with an estimate generated from the previous cycle. Only the consensus of these data is accepted by the actuators.

The system was certified in March 1983. Since its introduction to airline service, only one revision to the software was necessary, mainly due to performance improvements. No erroneous deployment of the surfaces has been reported, and the reliability of the computers (in the sense of MTBF) exceeded the expectations. Several aspects of this approach are also described in [Duga81, Mart82, Hill83, Hill85, Rouq86].

2.4.3 A320 Pitch Control

The A320 Fly-by-Wire system [Zieg84, Corp85, Rouq86, Trav88] is intended to improve the safety of the aircraft. It will provide a protection against windshear, a protection of the flight envelope, and an alleviation of the burden on the pilots. Two types of computers, each using processors manufactured by different suppliers, are used in the design. Each computer is built with two computation channels and two different application programs. Therefore, four different programs are used.

Four computers are used to control the aircraft on the pitch axis: ELAC1, ELAC2, SEC1, SEC2 (see Figure 2-5). At any time, only one computer is needed to have full authority. The computer in charge of the pitch axis periodically sends "I am alive" like messages to the other computers. If this computer fails, it shuts itself down, and this will be detected by the other computers. According to a predefined priority, one of them will take over the control of the pitch axis.

The whole system is said to be able to tolerate a special type of design fault, namely, one type of computers shutting itself down abruptly [Trav88]. In this case, one computer of the other type can successfully take over control, since it is very unlikely that this computer will commit the same fault. As it turns out, the software quality of this configuration is quite good: either no software error has been reported, or the ones that have been detected are benign. As for related software error, none have been reported. Report mechanisms are the pilots and on-line error logging devices.

2.4.4 Honeywell/Sperry System SP-300 and Its Successors

The autopilot flight director system SP-300 for the Boeing 737-300 aircraft was developed by Sperry Flight Control Division of Honeywell [Will83]. The SP-300 system consists of two redundant computers. Each computer again contains two diverse processors from different manufacturers using different design techniques (micro-chip versus bit-slice). One processor is mainly controlling the pitch axis, and the other one the roll axis. The software for the two processors was developed by independent teams. Part of the functionality is identical, so that a comparison check can be made. The SP-300 system has been a very successful product since its deployment. No software defect caused by the dissimilar processing mechanism has

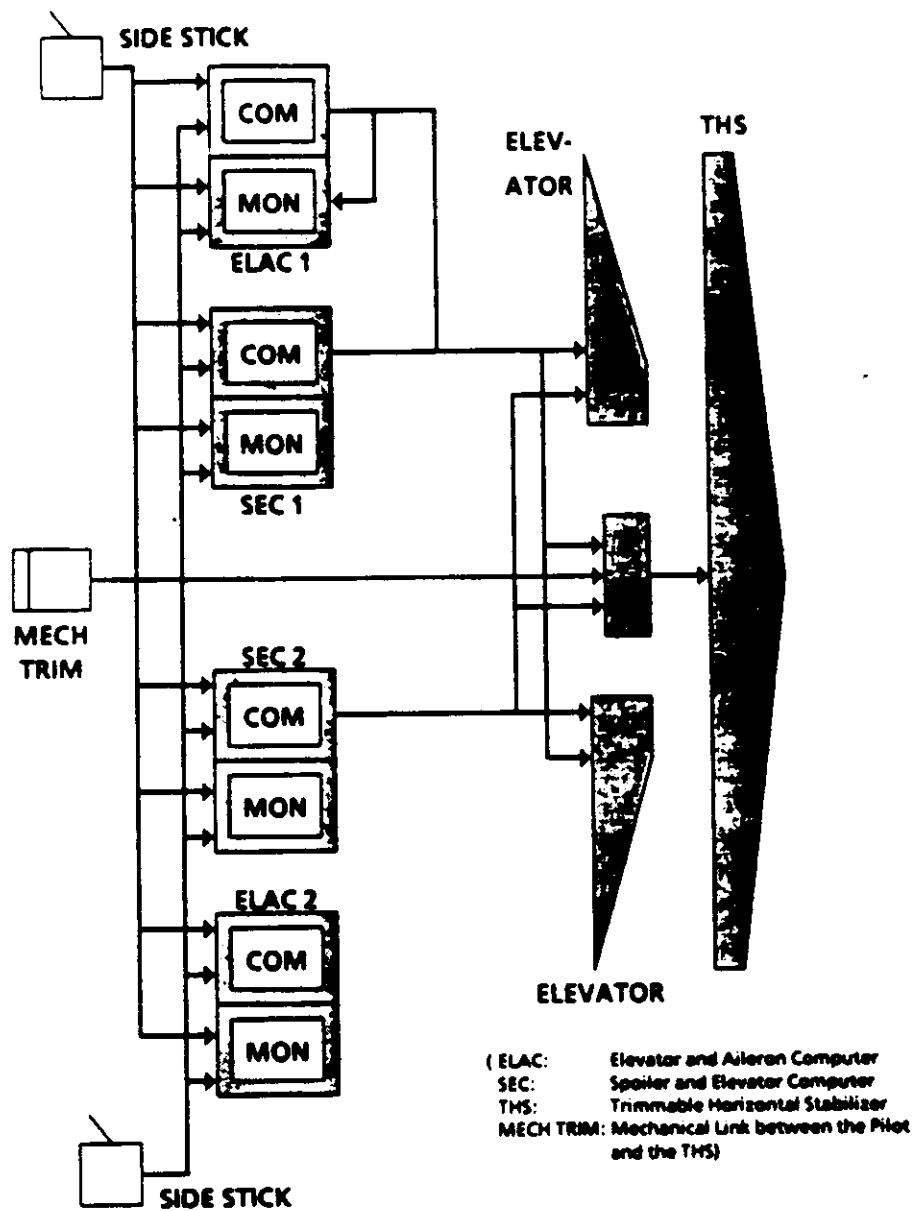


Figure 2-5: A320 Pitch Control

been recorded.

Furthermore, Honeywell/Sperry developed a dual redundant system with diverse hardware and diverse software [Youn85a]. The system consists of two flight control computers (FCC), each containing three diverse redundant CPUs that again are programmed in a diverse way (see Figure 2-6). This system can tolerate single systematic faults within one design unit (that is, CPU and its software). The development process by which Honeywell/Sperry tries to reduce the amount of generic faults is shown in Figure 2-7 [Youn86]. The main advantage of diversity is seen to be the gain in overall system reliability. Moreover, the Boeing 757/767 is equipped with a yaw damper making use of two version programming [Youn85b].

2.5 Diversity Concerns of Previous Experiments and Designs

Design diversity is the baseline concept for the application of MVS to achieve software fault tolerance. The impact of different dimensions of diversity plays an important role in MVS systems, since the correlation of faults tends to decrease as the degree of design diversity increases. Here, "degree" could refer to either "the number of different dimensions where diversity is applied," or "the degree within each dimension." Design diversity concerns shown in previous experiments and designs include specifications, programming languages, algorithms, data structures, programming teams, tools (e.g., compiler), localities (programmer's location), testing methods, hardware environments, and software environments. However, the effect of diversification in each of these dimensions and their potential usefulness still need to be investigated and evaluated.

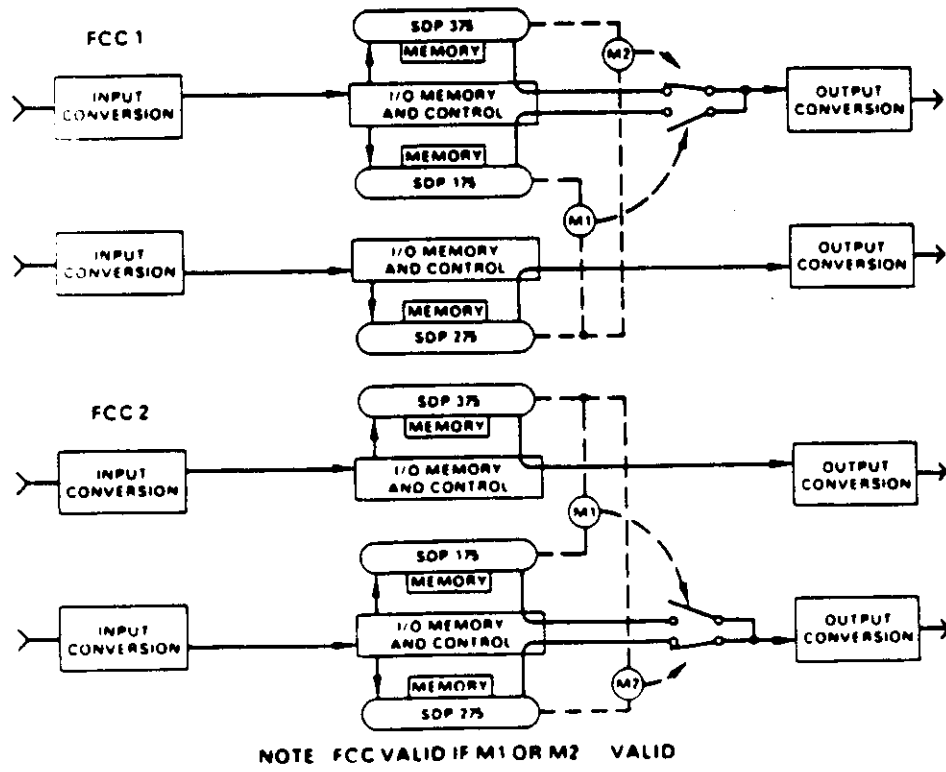


Figure 2-6: Honeywell/Sperry's Dual Architecture

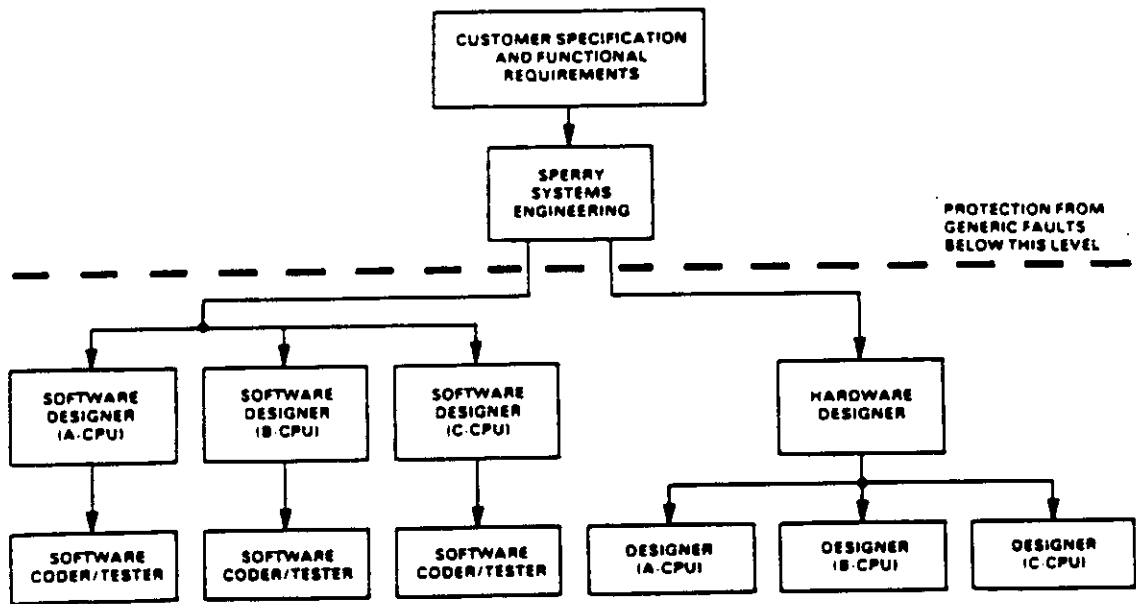


Figure 2-7: Protection from Generic Software and Processor Faults

CHAPTER 3

A DESIGN PARADIGM

3.1 The Need for a Formal Theoretical Basis

Each NVP investigation at UCLA and some elsewhere, discussed in the previous chapter, contributed to an evolving methodology. This is an appropriate time to merge the NVP methodology with the general fault tolerance design paradigm [Aviz87a] and to formulate a set of rigorous guidelines to conduct the generation of MVS. This set of rigorous guidelines for general consideration and treatment of MVS systems will play a crucial role in the success of building such systems. The objective of the design paradigm is to minimize the probability of oversights, mistakes, and inconsistencies in the process of meeting the specified goals of dependable software when an MVS system is under construction.

3.2 An Overview of the Design Paradigm

Based on the experience gained from previous research activities, especially those conducted at UCLA, we have developed a rigorous design paradigm for MVS systems. The overview of the paradigm can be described as follows. At the outset, a software system is devised and the approach to its implementation is selected according to the given dependability specifications, ensuring high quality conditions by use of good software engineering disciplines. This defines the "baseline" design for

the use of *fault avoidance* as a means of assuring dependable performance. Next, *fault-tolerance* can be introduced into the baseline design through a systematic sequence of design activities to guarantee error detection and recovery. At any point where the design is incorrect or inappropriate, a *fault-removal* technique is applied and a system refinement is required. After several iterations for improvements, *fault-forecasting* can ensure that the system has achieved its dependability goal for operation. The maintenance requirement can also be defined at this point.

This entire procedure is outlined below, and summarized in Figure 3-1.

3.3 State Precise Requirements with Dependability Goals

To begin, precise user requirements of the system and their dependability goals are specified. "Dependability" is used here as a generic term meaning "dependable performance", and not as a specific quantitative measure [Aviz86].

1. First, the classes of design faults that are to be tolerated are explicitly identified. For MVS systems, most independent faults can be identified and tolerated easily by error masking, and some related faults can be tolerated with proper error detection and recovery mechanisms (with system degradation when necessary). Other types of faults, e.g., those which cause the aborting of individual versions, should also be considered [Trav88].
2. Second, quantitative dependability goals are specified. This often is done for all fault classes at once, but preferably should be detailed for individual classes which have been previously identified. It is important that the numbers specified here should be sensible; they cannot be unrealistic conjectures which

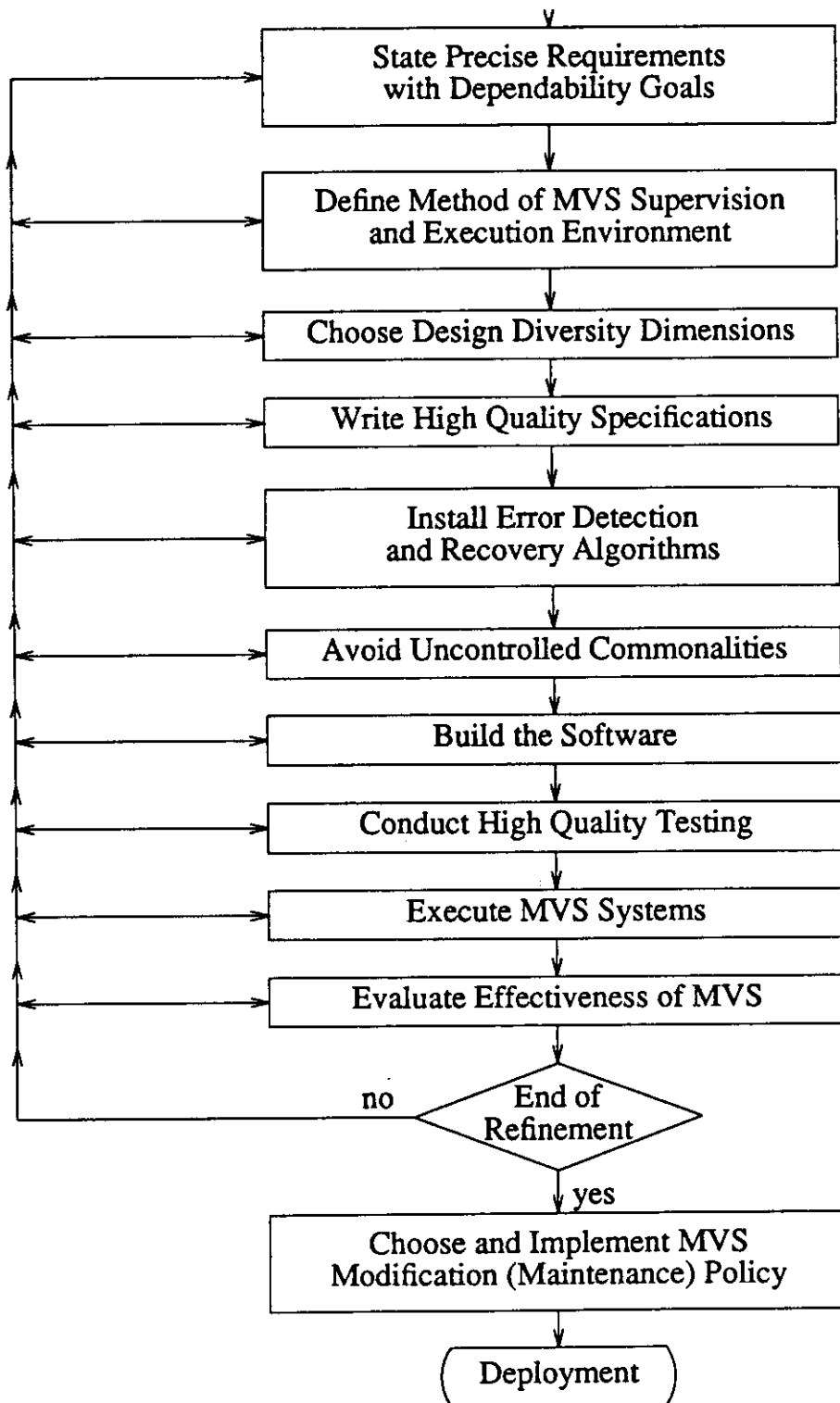


Figure 3-1: A Design Paradigm for Multi-Version Software

lack integrity.

3. Third, the methods for the evaluation of the dependability actually attained by the MVS system are postulated in detail. If adequate analytical modeling is not likely, other methods, such as experimentation or simulation techniques, should be chosen.

3.4 Define Method of MVS Supervision and Execution Environment

The number of versions and their configurations have to be decided before the execution of MVS systems. The executing method and required resource should also be defined and investigated. Generally speaking, a generic class of support mechanisms forming the MVS execution support environment is necessary. A prototype system called DEsign DIversity eXperiment testbed (DEDIX) [Aviz85a, Aviz85c] has been built to serve this purpose, which supervise the MVS execution on a general operating system (UNIX). Moreover, it is possible to build a customer-oriented, special-purpose supporting environment for the execution of MVS systems. In this case, special tools or hardware processors would have to be implemented or obtained in advance for the execution of MVS systems, especially when the MVS supporting environments need to operate under certain stringent requirements (e.g., accurate supervision, efficient CPUs).

3.5 Choose Design Diversity Dimensions

The removal of the cause of common mode error is a traditional concern in hardware, especially in VLSI designs. Most pioneering efforts introduce diversity through multi-channel designs in a single chip [Sedm78, Tami84], or offer a more general method of diversification by independently designed, redundant circuits [Arla79]. The same idea should be employed in software systems as well. The major reason for choosing design diversity is to eliminate the commonalities between the separate efforts, as it has the potential to cause related faults among the multiple versions. Examples of a commonality are: an ambiguous specification, a conversation between designers from two separate efforts, use of the same faulty compiler, use of the same erroneous programmer's manual, or a misleading source of information (e.g., reference books).

Fundamental development of MVS systems relies on "random" diversity due to the programmers' independent efforts. Choosing different dimensions of diversity can lead to another level of controlled dissimilarities among program versions due to specified diversity. This manner of enforcing design diversity for the best configuration of the MVS system could have an important effect upon software fault tolerance.

It was mentioned early in Chapter 2 that different dimensions of diversity can be applied to the building of MVS systems. However, since adding more diversity implies adding more resources, it is important to derive design diversity metrics for the evaluation of the effect of these dimensions. These metrics will enable trade-off studies between cost and efficiency. Such metrics might be application dependent, and thus may need to be elaborated, possibly by several iterations of investigations.

Another interesting concept in examining diversity is the diversified development methodologies in producing program versions. Different software development methodologies, e.g., prototyping with incremental refinements, step-by-step software engineering, or even the "clean room" approach which requires program inspection and certification before applying any test data [Dyer88], could be investigated, compared and evaluated.

Finally, in the idea of "international mail-order" experiments, the members of fault tolerance research groups from several countries would perform design and implementation from a common specification [Aviz85b]. This may include both hardware and software. It is expected that the several versions produced at widely separated locations, by different designers with different training and experience, applying different software development methodologies, will contain substantial design diversity.

3.6 Write High Quality Specifications

A completely documented specification for the required functions, interfaces, and performance for the software product should be provided. Such specifications should be as accurate and correct as possible. This is the "hard core" of design diversity [Aviz86]. Latent defects in the specification such as incorrectness, inconsistencies, ambiguities, and omissions are likely to bias otherwise independent efforts toward related design faults, which could lead to unrecoverable system failure. According to previous experiments, many related faults have been ascribed to this cause [Bish86, Gmei79].

Thus, a condition which is critical in avoiding related design faults is the existence of a complete and accurate specification of the service to be delivered by the diverse designs. In order to build high quality into the specification, two techniques can be applied.

- Formal specification for fault-avoidance.

The use of formal, very high-level specification languages is the most promising approach [Gogu79, Gutt83, Kemm85, Zave86]. When such specifications are executable, they can be automatically tested for latent defects [Berl87].

- Independently generated multiple versions for fault-tolerance.

The independent writing and subsequent comparison of two or more specifications [Rama81], presumably by using multiple formal languages, is the next step to be taken. This is expected to increase the dependability of specifications beyond the present limits.

3.7 Install Error Detection and Recovery Algorithms

Next, error detection algorithms are selected and incorporated into the original design. Their purpose is to generate the initial error signal. After error detection, recovery algorithms are devised which are invoked by the error signals. Their goal is to return the system to some level of proper operation or to shut it down safely. Recovery consists of all actions that take place after the error signal has been received.

3.7.1 The Proposed Algorithm

Error detection and recovery techniques in MVS systems have the following objectives and constraints:

- 1) Recovery from design faults.
- 2) Minimal limitation on design diversity.
- 3) Matching the type and severity of the errors.
- 4) Minimal impact on system performance.
- 5) Simplicity in implementation.

This problem has been substantially studied and solved by the community error recovery scheme in [Tso87a, Tso87b]. Its 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.

3.7.2 Cross-Check Points

A cross-check point (cc-point) is a decision point at which the redundant versions produce their output results after a computation for comparison [Aviz77]. Each cc-point has the following attributes: 1) a cc-point id (*ccp-id*), which uniquely identifies the cc-point; 2) a *format* string, which indicates the types and number of state variables to be compared; and 3) a set of pointers to the state variables (*cc-vector*), which contains the values to be checked. The MVS supervisor 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 produce the

decision output so that errors due to a minority of versions will be masked. Hence the cc-points serve the following functions in MVS:

- *error masking* - a decision result is obtained through comparison of the cc-vectors, thus masking errors in the versions.
- *error detection* - faults in failed versions will manifest themselves as either erroneous or missing cc-vectors to produce detectable errors. This information is accumulated for the second level of recovery.
- *error recovery* - erroneous versions will obtain the decision result for its future computation, in order to attempt a recovery from the effects of errors.

Error recovery at the cc-point level will not be effective if some *state variables* (variables whose value affect further computation in the program) that are not in the cc-point have been corrupted. Another level of recovery is necessary to deal with this situation.

3.7.3 Recovery Points

Complete error recovery of failed versions is performed at recovery points [Tso87b]. Each r-point specifies the following items: 1) a unique recovery point id (*rp-id*), which uniquely identifies the r-point, and 2) two exception handlers, the *state-input exception handler* and the *state-output exception handler*, which are required for input and output, respectively, the internal state of the version (*version state*) in a specified format. At a recovery point, the rp-ids of the versions are submitted to the MVS supervisor and compared. 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 produce its version state as output. The version states are compared by the decision function of the MVS supervisor to produce a *decision state* which will be used as input by the state-input exception handler to every failed version. Missing or incorrect rp-ids indicate some versions have control flow faults. These versions are restarted at the current recovery point by means of the decision rp-id. In this way, failed versions will proceed to the next program module with a correct internal state after the recovery point.

3.7.4 Introducing Cc-Points and Recovery Points in the Specification

When design diversity is applied to implement MVS, the common specification must incorporate the specification of the cc-points and recovery points described above. The two-way communication link between these two phases in the design paradigm allows this activity to happen interactively. Very careful attention must be paid to this endeavor.

Concerning fault tolerance techniques, an initial specification should define: 1) the cross-check (and recovery) points at which the decision algorithm will be applied to the results of each version; 2) the content and format of the cross-check (and recovery) vectors to be generated at each cc-point (and recovery point); 3) the decision algorithm to be used at each cc-point, as well as the exception handler to be used at each recovery point; and 4) the responses to the possible outcomes of decisions. The decision algorithm explicitly states the allowable range of variation in numerical results, if such a range exists, as well as any other acceptable differences in the version results, such as extra spaces in text output or other "cosmetic" variations.

In review, the specification procedure concerning the design of error detection and recovery algorithms of a MVS system takes the following steps:

1. Draw the control and data flow diagram of the application,
2. Recognize the modularity of the application,
3. Identify cc-points across those modules,
4. Identify the sequence of the cc-points,
5. Specify contents and formats of cc-points in the specification according to their sequence,
6. Identify recovery points, and
7. Specify contents and formats of recovery points, together with their recovery procedures.

3.7.5 Placements of Cc-Points and Recovery-Points

It was observed that the placements of the cc-points and recovery points were very important in the MVS development, and that sometimes it tended to be treated inadequately [Tso87a]. In the NASA Experiment [Kell86], the original specification of cc-points for the application 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 in that experiment was that there was a large module in the application which required the most complex computation. Most coincident errors

observed in the extensive testing occurred in this computation. Placing an additional cc-point in the middle of that module 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. The lesson from this experiment is that cc-points and recovery 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.

3.8 Avoid Uncontrolled Commonalities

The main purpose in identifying and avoiding uncontrolled commonalities is to prevent them from contributing to potential contamination of the whole design. Any kind of contamination might lead to deterioration of the quality of the software to be produced when related faults are introduced inadvertently. Examples of uncontrolled commonalities include: other (unaddressed) dimensions of diversity, defects in specifications, unqualified personnel in the design procedure, programmers' common training and experience, technical information exchange during software generation, supervisory system faults which cause identical errors in all the application software, and other unknown factors.

A procedure to handle the uncontrolled commonalities can be stated as follows: 1) list all the hypothesized uncontrolled commonalities; 2) decide which ones to address before building the software; and 3) leave others as input to later evaluations. There is a need for research to measure and elucidate what are the possible uncontrolled commonalities that cause software deterioration. The purpose

of such research is to provide guidance on how to perform the design rigorously, without disturbing the integrity of the system.

Another effective method to avoid commonalities lies in the control process in the multi-version software development, which is conducted in the next phase. Namely, communication protocols used during software development should be rigidly prescribed. An example guideline for such protocol looks like this: programmers are strictly admonished not to discuss any aspect of their technical work with members of other teams directly. All work-related communications are allowed only between programmers and a coordinator, and this is conducted via a formal protocol (e.g., electronic mail) to avoid ambiguity and preserve retrospection.

3.9 Build the Software Versions

The methodology for building the software basically follows the well known software life cycle practice. Each program should be developed in the environment as if there is only one version in construction. There are two directions for the discussion here:

1. *Applying software engineering technologies*

Well defined software development techniques should be applied [Boeh81]. In short, the following design phases should be scheduled and conducted to ensure high quality software development.

- **Overall design:** A complete, verified design of the overall software architecture, control structure, and data structure for the system, along with such other necessary components as draft user's manuals and test

plans.

- **Detailed design:** A complete, verified design of the control structure, data structure, interface relations, sizing, key algorithms, and assumptions of each program component.
- **Coding and unit testing:** A complete, verified set of program components.
- **Integration:** A properly functioning software product composed of the software components.
- **Validation testing:** A validation process to ensure quality of the functioning software system after integration.

2. *Conducting MVS software development protocol*

In addition to specifying the above phases, software development communication protocols prescribed in the previous stage should be carefully conducted. Careful coordination among multiple software programming teams plays a crucial role in the success of such practices. Certain formal protocols, especially those related to communication procedures defined previously, should be carefully conducted and evaluated.

3.10 Conduct High Quality Testing

It should be emphasized that MVS systems require high quality programs. In order to guarantee the quality of programs, we should apply high quality testing or even extensive verification and validation procedures before we accept the usage of any given version.

An integral part of the achievement of suitable program versions is to show by the verification and validation (V&V) procedures that the intermediate software products do indeed satisfy their objectives. Verification is to establish the truth of correspondence between a software product and its specification. Validation is to establish the fitness or worthiness of a software product for its operational mission. Informally, verification concerns "building the software right" while validation concerns "building the right software".

A more speculative, and also more general, application of MVS is its reinforcement for current software V&V procedures. Instead of extensive preoperational V&V of a single program, two or three independent versions can be executed "back-to-back" in an operational environment, completing V&V concurrently with productive operation [Aviz85b].

3.11 Execute Multi-Version Software Systems

The generic MVS support mechanisms provided by DEDIX includes the following attributes: 1) a decision algorithm; 2) assurance of input consistency; 3) interversion communication; 4) version synchronization and enforcement of timing constraints; 5) local supervision for each version; 6) the global executive and decision

function for the treatment of faulty versions; and 7) the user interface for observation, debugging, injection of stimuli, and data collection during multi-version execution of application programs.

The approach of a customer-oriented, special-purpose MVS supporting environment has the advantages of simplicity, efficiency, and maintainability. It is, however, restricted to only several types of faults. Such supporting mechanisms, in turn, need to be specified, implemented, and protected against failures due to physical or design faults. This idea of multilayer design diversity is the pinnacle of software fault tolerance.

3.12 Evaluate Effectiveness of MVS

After the implementation of the MVS and its fault detection and error recovery algorithms, an evaluation of the software fault-tolerance of a design is performed by means of analytic modeling, simulation, experiments, or combinations of these techniques. Usually, the dependability prediction of the MVS system is compared to that of the single version baseline system. The choice of suitable software models and the definition of quantitative measures that characterize the level of fault-tolerance present in a MVS system are two essential aspects of evaluation.

The usefulness of the MVS system depends on the validity of the conjecture that residual software faults in separate versions will cause very few, if any, similar errors at the same cc-points. Credible evidence should be statistically gathered and analyzed for evaluation.

3.13 Refine the Design by Iteration

Refinement of the design is carried out as the final step. When a minimally acceptable dependability value is specified, the software versions will reach that value at different "mission times." Additional maintenance activities (including debugging and upgrading) of the individual software versions or other choices of error detection and recovery techniques can be made to raise the predicted dependability of the inferior software. It is also possible that the software may prove to be overprotected, and some fault-tolerance mechanisms may be removed. The goal of design refinement is to balance the protection provided to MVS systems in such a manner that the dependability goal is attained without a single dominating contributor of undependability, and at the lowest cost of additional resources. Interactive modeling and evaluation tools, such as SURF [Cost81], CARE III [Stif79], ARIES 81 [Maka82], SAVE [Goya86], HARP [Duga86], and SHARPE [Sahn86], are essential for cost-effective evaluation and refinement of the design.

3.14 Choose and Implement Modification (Maintenance) Policy

As for the modification and maintenance of the MVS system, the same design paradigm should be followed, i.e., a common "specification" of the maintenance action should be "implemented" by independent maintenance teams. The modification of MVS due to a fix or improvement in functionality or performance should take advantage of sufficient modularity of the specification so that a given modification will affect only a few modules [Aviz85b]. The extent to which each module is affected can then be used to determine whether the existing software should be modified according to specification changes, or the existing versions should be

discarded and new versions generated from the appropriately modified specification. It is even argued that patching the software, as has been widely used in industry, might more easily reveal the existence of faults by exhibiting dissimilarities among the independently generated software versions. This would be a valuable feature of MVS systems since such a patching technique could create the potential of high risks in an originally well (and possibly elegantly) designed single version software.

Notice that some of the above stages occur at progressively later times, but backtracking from a given stage to its previous one may occur at any time. Alteration of requirements arising from use, revision of specification, change in environment, and erroneous implementation may interrupt the flow of the normal design paradigm or spawn sub-processes having their own life cycles.

CHAPTER 4

AN INDUSTRIAL INVESTIGATION

For the purpose of self-contained completeness, Chapters 4 and 5 of this dissertation includes portions of an early Technical Report [Aviz88a].

4.1 A Practical and Complete Experiment Using the Design Paradigm

The main purpose in formulating a design paradigm is to eliminate all identifiable causes of related design faults in the independently generated versions of a program, and to prevent all potential effects of coincident run-time errors while executing these program versions. An investigation to execute and evaluate the design paradigm is necessary, in which the complexity of the application software should reflect a realistic size in highly critical applications. Moreover, this investigation should be complete, in the sense that it should thoroughly explore all aspects of MVS systems as software fault-tolerant systems.

An industrial investigation concerning this subject might be the most suitable experiment in which to obtain an appropriate conclusion for applying the design paradigm to implement MVS systems. This would be a very attractive approach, since it reflects real-world situations under severe quality requirements. Based on this reasoning, a joint project was initiated as a consequence of mutual research interests in design diversity [Aviz82] at the UCLA Dependable Computing & Fault-Tolerant Systems (DC & FTS) Laboratory and at the Sperry Commercial Flight Systems

Division of Honeywell, Inc., in Phoenix, Arizona (abbreviated as "H/S" in the following discussion), which has been a very successful builder of aircraft flight control systems for over 30 years. A recent major product of H/S is the flight control system for the Boeing 737/300 airliner, in which a two-channel diverse design is employed [Will83], as was previously mentioned in Chapter 2.

The main research interest of H/S is the generation of demonstrably effective multi-version software in an industrial environment, such as exists now and is being further developed by H/S [Youn85a]. Its objective includes most recent UCLA research activities, referenced to the industrial environment, as well as the estimation of the effectiveness of multi-version software and of its relative safety as compared to a single-version approach.

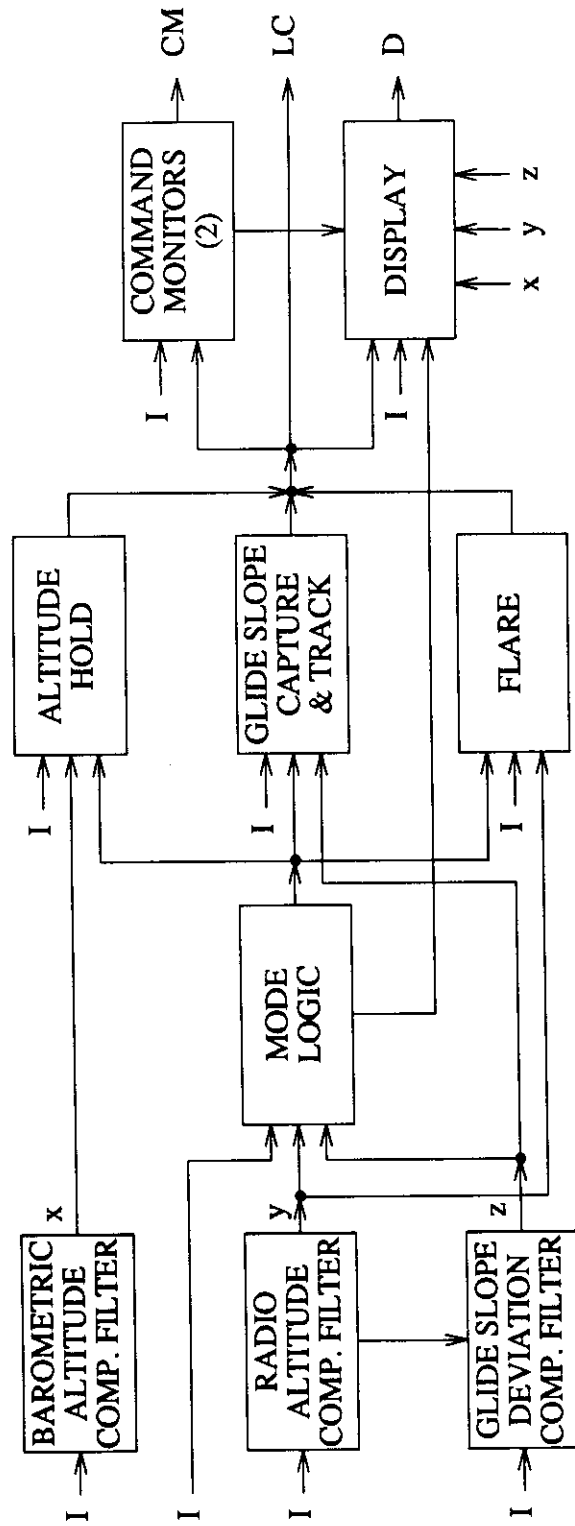
It was mutually agreed that an experimental investigation was necessary, in which H/S would supply an automatic flight control problem specification, specify H/S software design and test procedures, deliver an aircraft model and sets of realistic test cases, and also provide prompt expert consultation. The research was initiated in October 1986 and carried out at the UCLA DC & FTS Laboratory, funded jointly by H/S and Microelectronics Innovation and Computer Research Opportunities (MICRO) program of the State of California. A six-version programming effort in which six programming languages were used and 12 programmers were employed took place during 12 weeks of the summer of 1987. An intensive evaluation followed, and is continuing as of April 1988 [Aviz87b, Aviz88b].

4.2 The Automatic Landing Problem

Automatic (computer-controlled) landing of commercial airliners is a flight control function that has been implemented by H/S and other companies. The specification used in the UCLA/Honeywell experiment is part of a specification used by H/S to build a 3-version Demonstrator System (hardware and software), employed to show the feasibility of N-version programming for this type of application. The specification can be used to develop the software of a flight control computer (FCC) for a real aircraft, given that it is adjusted to the performance parameters of a specific aircraft. All algorithms and control laws are specified by diagrams which have been certified by the Federal Aviation Administration (FAA). The *pitch control* part of the auto-land problem, i.e., the control of the vertical motion of the aircraft, has been selected for the experiment in order to fit the given budget and time constraints. The major system functions of the pitch control and its data flow are shown in Figure 4-1.

Simulated flights begin with the initialization of the system in the Altitude Hold mode, at a point approximately ten miles from the airport. Initial altitude is about 1500 feet, initial speed 120 knots (200 feet per second). Pitch modes entered by the autopilot/airplane combination, during the landing process, are: Altitude Hold (AHD), Glide Slope Capture (GSCD), Glide Slope Track (GSTD), Flare (FD), and Touchdown (TD).

The *Complementary Filters* preprocess the raw data from the aircraft's sensors. The *Barometric Altitude* and *Radio Altitude Complementary Filters* provide estimates of true altitude from various altitude-related signals, where the former provides the altitude reference for the Altitude Hold mode, and the latter provides the altitude reference for the Flare mode. The *Glide Slope Deviation Complementary Filter* provides estimates for beam error and radio altitude in the Glide Slope Capture



Legend: I = Airplane Sensor Inputs
 LC = Lane Command
 CM = Command Monitor Outputs
 D = Display Outputs

Figure 4-1: Pitch Control System Functions and Data Flow Diagram

and Track modes.

Pitch mode entry and exit is determined by the Mode Logic equations, which use filtered airplane sensor data to switch the controlling equations at the correct point in the trajectory.

Each Control Law consists of two parts, the Outer Loop and the Inner Loop, where the Inner Loop is very similar for all three Control Laws. The Altitude Hold Control Law is responsible for maintaining the reference altitude, by responding to turbulence-induced errors in attitude and altitude with automatic *elevator* control motion. (The elevator is the surface of an airplane that controls the vertical motion.) As soon as the edge of the glide slope beam is reached, the airplane enters the Glide Slope Capture and Track mode and begins a pitching motion to acquire and hold the beam center. A short time after capture, the track mode is engaged to reduce any static displacement towards zero. Controlled by the Glide Slope Capture and Track Control Law, the airplane maintains a constant speed along the glide slope beam. Flare logic equations determine the precise altitude (about 50 feet) at which the Flare mode is entered. In response to the Flare control law, the vehicle is forced along a path which targets a vertical speed of two feet per second at touchdown.

Each program checks its final result (elevator command) against the results of the other programs. Any disagreement is indicated by the Command Monitor output, so that the supervisor program can take appropriate action.

The Display continuously shows information about the FCC on various panels. The current pitch mode is displayed for the information of the pilots (Mode Display), while the results of the Command Monitors (Fault Display) and any one of sixteen possible signals (Signal Display) are displayed for use by the flight engineer.

Upon entering the Touchdown mode, the automatic portion of the landing is complete and the system is automatically disengaged. This completes the automatic landing flight phase.

4.3 Applying the MVS Design Paradigm

For the purpose of investigating and determining how much credit could be given to the design diversity approach, the rigorous design paradigm described in the previous chapter was applied to carry out the implementation of six versions of diverse processing software. The general rules fit into the experiment domain are described in the following sections.

4.3.1 State Precise Requirements with Dependability Goals

The techniques and methods used to develop and manage software for airborne, digital computer-based equipment and systems are usually carefully formulated under stringent regulations. The major tasks involved in a typical software development program and the guidance and recommendations for performing these tasks are provided in the document DO-178A [RTCA85], which is the FAA-approved software design and test standard for the aeronautic industry.

4.3.1.1 Requirements for Software Testing

The required control laws to be implemented in this experiment are part of the H/S SDD that defines the system requirements for the Demonstrator Cat IIIb Digital Flight Control System (DFCS). H/S, along with other avionics suppliers, must adhere

to the requirements of DO-178A. The following definitions apply to software testing, as specified in [RTCA85].

(a) *Requirements-Based Tests (black box testing)*. Test cases are derived from the software requirements independent of the software structure. Primarily, these are the requirements specified in the Software Requirements Document (Software Specification), but further requirements may emerge during the design process (e.g., scheduler requirements). These tests demonstrate that the software performs its *intended functions*. Each software requirement should be traceable to an associated verification test or tests.

(b) *Software Structure-Based Tests (white box testing)*. Test cases are derived from the software design itself. As such, they can address features of the implementation which may or may not be apparent from a requirements perspective. Typically, requirements-based tests are analyzed for structural coverage and augmented as necessary. In this sense, the structure-based tests complement the requirements-based tests to provide sufficient test coverage. Such structure-based tests are necessary to provide some measure of protection from *unintended functions* in software that may pass all of its requirements-based tests. All of the software must be exercised to a degree commensurate with its software certification level.

Therefore, software errors are postulated to be caused by two types of human-made faults: *requirement* faults and *structural* faults. A requirement fault exists when a specified requirement is not or not completely complied with. A structural fault is the complement of the requirement fault, i.e., it is any fault that is not exposed by system testing based on the system specification.

4.3.1.2 Apply Design Diversity to Achieve a Dependability Goal

Three categories of aircraft systems are distinguished by the FAA, namely *flight critical*, *flight essential*, and *non-essential*, with different testing efforts required for each. In general, avionics equipment is designated as *critical* when loss of the function provided by the equipment can cause a catastrophic aircraft failure. The probability of such an occurrence must be demonstrated by test or analysis to be 10^{-9} or less over the duration of the flight. Avionics equipment is designated as *essential* when loss of its function can significantly impact safety. For essential equipment the probability of loss of function must be demonstrated to be 10^{-5} or less over the duration of the flight.

Thus, the software portion of the critical equipment must have a probability of failure less than 10^{-9} depending on the failure rates in the remaining portions of the system. To protect against failures in single-version software that cause total loss of a critical function, a *structure-based testing* methodology is required in addition to *requirements-based testing*. Any fault will manifest itself identically in all redundant computation channels that use identical software; but this exhaustive testing procedure (Level 1) is assumed to assure the desired reliability. For software that can fail and cause loss of an essential function only, requirements-based testing alone is required (Level 2).

While requirements-based testing may be extensive, the number of test cases is bounded by the system requirements. Structure-based testing, on the other hand, is likely to be very extensive, possibly involving permutations of all inputs together with a rather subjective evaluation of each result. If more than a few inputs are involved, the time required to prepare and run the test, and to analyze the results becomes prohibitive and may present a serious scheduling and cost problem. Structural testing

appears to be analogous to the hardware "failure modes and effects analysis" procedure with LSI circuits, which is acknowledged to be extremely difficult to implement fully [Trea82]. Therefore, the FAA encourages manufacturers, where practical, to reduce the level of testing by architectural means.

The architectural techniques to reduce test levels that the FAA has accepted, or is likely to accept, employ design diversity as their central attribute. The application of threefold diversity in critical software is based on the conjecture that the likelihood of two identical, critical structural faults in 3-version software is, in the verified and validated release, substantially reduced from the likelihood of a critical structural fault in a single version; thus only Level 2 testing may be required in 3-version architectures. The FAA has recognized, however, that the conflicting requirements for design independence and of having the diverse elements perform the same function impose an important design constraint. Therefore, these systems must be shown to monitor each other under all foreseeable conditions and critical modes of operation.

4.3.2 Define Method of MVS Supervision and Execution Environment

For the purpose of industrial-standard validation and verification, a Model Definition Document (MDD) was supplied by H/S. This document provides mathematical models for functions within the landing/approach control loop but external to the control laws defined in the System Description Document.

Consequently, a closed loop testing of multiple executions with random inputs was conducted. Millions of test runs have been executed in a DEDIX-like environment for suitable aircraft control and flight simulation. Moreover, another

open loop testing strategy, called "Square Wave testing," was also provided for extra operational testing. Statistical data related to execution of MVS systems were gathered.

The aircraft mathematical model provided in the MDD is representative of the dynamic response of current medium size, commercial transports in the approach/landing flight phase. In this model, the airplane is trimmed, by means of flaps and engine thrust setting, to a landing speed of about 120 knots, and a level of low altitude flight path is chosen. It is assumed that the pilot has engaged the autothrottle while in the low-speed, approach condition, so that forward speed variations are negligible. In addition, the pilot is assumed to have maneuvered the aircraft to an altitude of 1500 feet in preparation for engagement of the Altitude Hold mode.

The three control signals from the autopilot computation lanes are inputs to three elevator servos. The servos are force-summed at their outputs, so that the mid-value of the three inputs becomes the elevator command.

Sensed airplane attitude, attitude rate, altitude, flight path, and vertical acceleration motions are directly measured by various sensors mounted in the fuselage, which are modeled in MDD. It is assumed that these sensors are at the center-of-gravity of the aircraft, and have unity gain characteristics. The reaction of the airplane (e.g., the pitch attitude, PA) should be able to stabilize the path of the flight in responding to fluctuating inputs.

Several other models are associated with the Airplane simulator. They include:

- Landing Geometry

The landing geometry mathematical model describes the deviation from glideslope beam center as a function of aircraft position relative to the end of the runway. Nominal glideslope beam characteristics are defined by a beam angle of 2.5 degrees referenced to the touchdown point, and a width in the vertical plane of ± 0.5 degrees about the beam center. Interception of the lower edge of the beam by the aircraft results in a pitch-over command and acquisition of the beam center.

- Turbulence Generator

Turbulence generator is used to introduce vertical wind gusts. Vertical turbulence is assumed to be frozen with respect to time. This is based on the observation that for reasonable flight speeds, changes in vertical wind velocity are smaller with respect to time than with respect to position. *Dryden spectra* are readily simulated and show reasonable agreement with measured spectra. Three more functions are required to compute the vertical turbulence: Random Number Generator, Gaussian Noise Generator, and Dryden Vertical Filter.

- Gaussian Noise Generator

The derivation for the Gaussian Noise Generator utilized here is provided in [Box50]. The requirement is to convert a uniform probability distribution between 0 and +1 to a specified Gaussian distribution with a high degree of statistical independence.

- Dryden Vertical Filter

The Dryden Vertical Filter is simplified to cover only low altitude conditions.

Filter constants are a function of several variables: gust intensity, altitude, longitudinal speed, sample interval, and gust scale. Physically, gust scale is the longest distance between two points in a turbulent field before correlation becomes zero.

Another model for the purpose of verifying the Flight Control Computer software is the Square Wave model. Unlike the Airplane model, this model applies open loop testing strategy with various stringent conditions to saturate the execution of the control laws. Specifically, the control laws receive constant values as all the sensor input data for the computation of each time frame, and a clock is set to record the Square Wave computation time. The input values are not changed until the clock reaches half of a selected time period for the Square Wave. At this point the input data are complemented (reverse the sign but remain the same magnitude) and the clock is reset for the second half period of the computation, repeating for many executions of the same period.

These models were programmed by the UCLA coordinating team to provide a suitable control problem for the experiment. Two program versions of the aircraft models, one in C and the other in Pascal, were independently generated. They were rather short programs of about 100 lines of code. Nevertheless, "back-to-back" testing between these two versions effectively revealed a bug in one of them. These versions were later certified by H/S personnel. Generation of input data and interpretation of the results were also performed and suggested by H/S experts.

4.3.3 Choose Design Diversity Dimensions

Design diversity is a potentially effective method to avoid similar errors that are caused by design faults in multi-version software systems. The choice of diversity dimensions in this experiment was based on the experience gained from: (1) previous experiments at UCLA [Chen78a, Kell83, Aviz84, Kell86], (2) recommendations from H/S, and (3) published work from other sites [Gmei79, Ande85, Bish86, Knig86].

Independent programming teams are the baseline dimension for design diversity. This allows the diversity to be generated with an uncontrolled factor of *randomness*. However, different dimensions of design diversity, including different algorithms, programming languages, environments, implementation techniques and tools, should be investigated and explored, and possibly used to assure a certain level of *enforced* diversity [Aviz85b]. It was decided that different algorithms were not suitable for the scope of FCCs due to potential timing problems and difficulties in proving their correctness (guaranteed matching among them). The investigation of different programming languages was attractive since it provides protection from subtle compiler errors and avoids the need to certify compiler correctness. Moreover, although research was initiated in this direction [Gmei79, Bish86], significant comparisons of different high order programming languages for the same critical application have not yet been reported.

Under the budget constraint and availability of the compilers, six programming languages were determined for the design diversity investigation for this project. The six programming languages chosen consist of two widely used conventional procedural languages (C and Pascal), two modern object-oriented programming languages (Ada and Modula-2), a logic programming language (Prolog), and a functional programming language (T, a variant of Lisp). It was hypothesized that

different programming languages will force people to think differently about the application problem and the program design, which could lead to significant diversity of programming efforts. Choices of the Prolog and T versions presented challenges to this project, since it was thought that they might not be suitable for this computation-intensive application. Nevertheless, it was still considered to be worthwhile to investigate this unexplored area, especially to assess the impact of Prolog and T on the structure of the auto-land programs.

4.3.4 Write High Quality Specifications

A high quality specification of the software requirement document has to be provided. This was carried out under the cooperation of a three-member UCLA coordinating team and H/S engineers. Previous experience from the NASA project in writing and maintaining specifications proved to be helpful in the fulfillment of this task [Tai86].

The efforts to develop a specification that is suitable to be used by the programming teams started early in 1987. Since it was clear from the beginning that programming the complete autopilot system would be too complex and too large a task for a twelve week programming experiment, the first major task was to find a subset of the autopilot that, when programmed, would result in a program of *reasonable* size. "Reasonable" was informally defined as "as large as possible while still being manageable within twelve weeks." It was decided that the *pitch* function of the autopilot with some added display functions should be programmed.

In the next step, representatives from H/S extracted the information needed for the experiment from their original Demonstrator specification and provided it in a

System Description Document (SDD), which was subsequently reviewed by the members of the coordinating team. The goal was to understand the problem as thoroughly as possible, in order to avoid as many ambiguities as possible and to provide a clear specification. Many meetings of the coordinating team with representatives from H/S were devoted to clarify various aspects, partly on a very detailed level.

To write the specification that was given to the programmers, the UCLA coordinating team followed the principle of supplying only minimal (i.e., only absolutely necessary) information to the programmers, so as not to unwillingly bias the programmers' design decisions and overly restrict the potential design diversity. The diagrams describing the major system functions were taken directly from the original SDD, while the explanatory text was shortened and made more concise.

In an appendix of the specifications document, the symbols used in the graphical representations of the system functions were explained, and it was explained how to deal with feed-back loops that appeared in the charts. In addition, the coordinating team imposed the requirement that two input routines, seven so-called *vote routines*, and one *recovery routine* be inserted at well defined points of the computation sequence. The purpose of the input routines was to facilitate the reading of sensor data for each channel. The purpose of the vote routines was to allow cross-checking and comparison of different versions' outputs of major system functions, as well as of some selected intermediate results (test points). The purpose of the recovery routine was to provide error recovery mechanism for the internal states of each version. Detailed discussions in using these routines is presented in the next section. A second appendix defined the syntax of all these vote routines.

The original specification given to the programmers was a 64 page document (including tables and figures) written in English. Its development required about 10 weeks of effort by the coordinating team, plus consultation by H/S experts.

We have noted that a small number of errors in the original specification could lead to numerous ambiguous and contradictory addenda in the form of question and answer pairs [Kell86]. To prevent the confusion, the coordinating team at UCLA was very careful about message replying in order to minimize the broadcast information. Since there was always a quick response from H/S flight control engineers when the question needed to be forwarded to them, the turn-around time for the programmers to receive their answers was very short.

During software generation, many errors and ambiguities in the specification were revealed. All questions by the programming teams were handled according to a communication protocol described later. Throughout the program development phase, the specification has been maintained as clear and precise as possible. The specification has now been restored to a single document [Aviz88a], a document that has benefited from the scrutiny of more than 16 motivated programmers and researchers.

4.3.5 Install Error Detection and Recovery Algorithms

The original software specifications specified *test points*, i.e., selected intermediate values of each major system function that had to be provided as outputs for additional error checking. Error detection was easily identified from the test points with some modifications.

A further enhancement to the specification was the introduction of *cross-check points* [Aviz77, Aviz85a] and a *recovery point* [Tso87b]. Error recovery could happen at each cross-check point, and at the recovery point at the end of each computation, before the result is fed back as input for the next computation in a close loop testing environment. Cross-check points and recovery points were easily specified with their proper format and sequence. These have been elaborated as part of the specifications. The usage of the cross-check points and recovery point in this application program is shown in Figure 4-2.

In Figure 4-2, execution scenario of the application was designed to iterate through the following computations: 1) airplane sensor input generation; 2) lane command computation; 3) command monitors and display computation; and 4) recovery mechanism when necessary. The airplane simulator was designed by the coordinating team. Lane command, command monitors and display module were implemented by the programming teams. The recovery mechanism was provided in a DEDIX-like environment [Aviz87c, Aviz88c].

Under this scenario, the application software was instrumented by some fault tolerance mechanisms. Two input points (sensor.input, lane.input) were specified to receive external sensor input from an airplane, and to receive flight commends from the other channels. Moreover, seven cross-check points (ccp.filter1, ccp.filter2, ccp.modelogic, ccp.outerloop, ccp.innerloop, ccp.monitors, ccp.display) were used to cross-check the results of the major system functions (Complementary Filters, Mode Logic, Outer Loop, Inner Loop, Command Monitors, Display) with the results of the other versions before they were used in any further computation. They were executed in a certain predetermined order, but again great care was taken not to overly restrict the possible choices of computation sequence. Finally, One recovery point

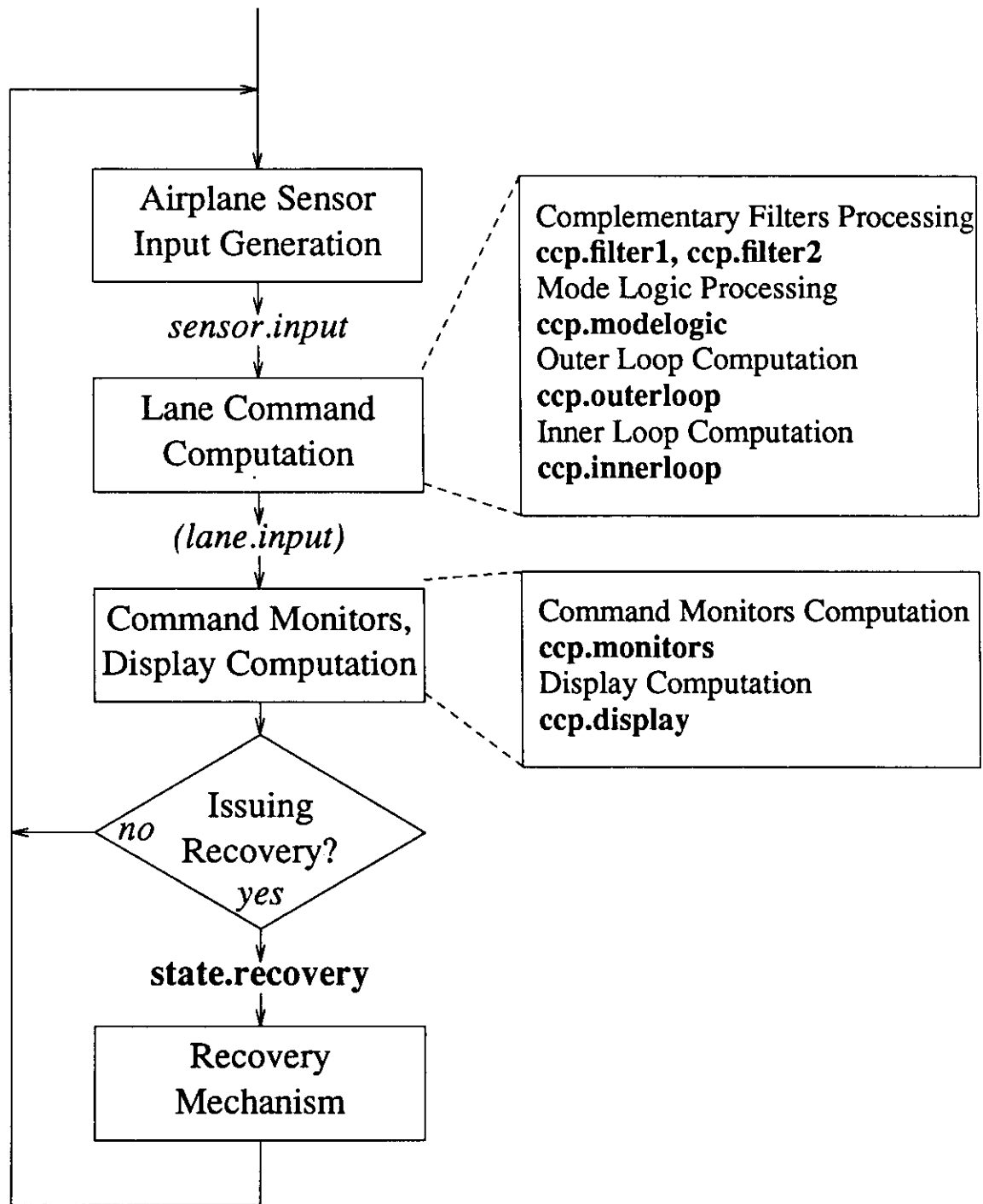


Figure 4-2: Usage of Cc-points and Recovery-point in the Application

(state.recovery) was used to recover a failed version by supplying it with a set of new internal state variables that were obtained from the other versions by the Community Error Recovery technique [Tso87b].

In summary, these fault tolerance mechanisms introduce 14 external variables (for input functions), 68 intermediate and final variables (for cross-check functions), and 42 state variables (for recovery function) in this application. State variables were identified as those variables whose value in its current iteration would affect that of its next iteration. *Integrators, filters, and rate-limiters* in the system denoted these variables.

4.3.6 Avoid Uncontrolled Commonalities

Uncontrolled commonalities include inexperienced programmers, technical information exchange among programmers, and design defects in the simulation environment. For the purpose of avoiding them, special effort was devoted in recruiting the programmers, and in assigning tasks to each programming team. Furthermore, a formal communication protocol was applied to monitor the information flow in the experiment and to impose isolation requirements among programming teams. Experience gained from the previous NASA experiment facilitated the necessary coordination [Kell86]. And finally, the UCLA research team was very careful in implementing the airplane simulation model and the associated recovery mechanism in order to prevent any possible contamination due to a supervisory fault.

4.3.6.1 Recruit Qualified Personnel

The recruitment and interviewing of programmers started about 3 months before the 12-week version generation phase in June, 1987. The summer is an especially favorable time to recruit highly qualified personnel from the about 260 CS graduate students at UCLA, since about 20 Teaching Assistants (many of them from programming classes) and several fellowship holders are able to accept summer employment. About 20 candidates, most of them graduate students at UCLA, submitted applications. The final choice of 12 programmers and their assignment to six teams were made one month before starting the software generation. Table 1 shows the specialties, graduate standing, and qualifications of the programmers identified by their assigned languages. The data indicate a mature, experienced, and well qualified group of research programmers. The effort was directed by the Principal Investigator, and coordinated by a three-member coordinating team, who started the work of writing the specification and developing guidelines and procedures, with support of H/S personnel, in November, 1986. A senior staff expert in flight control computing from H/S maintained continuous contact and regularly made visits to UCLA.

4.3.6.2 Define A Formal Communication Protocol

The purpose of imposing isolation rules on the teams was to assure the *independent generation* of programs, which meant that programming efforts were carried out by individuals or groups that did not interact with respect to the programming process [Aviz85b]. In order to keep this constraint, the programming teams were assigned physically separated offices for their work. Additionally, programmers were strictly admonished not to discuss any aspect of their work with

Team member	Degree held			CS standing		Programming experience
	Field	Degree	Year	Program	Year	
Ada-1	CS	B.S.	1984	M.S.	2nd	3 years
Ada-2	ECE ECE	B.S. M.S.	1982 1984	Ph.D.	2nd	3 years
C-1	IE CS	B.S. M.S.	1981 1983	Ph.D.	3rd	2 years
C-2	CS CS	B.S. M.S.	1982 1984	Ph.D.	2nd	5 years
Modula2-1	ECE ECE	B.S. M.S.	1982 1984	Ph.D.	2nd	3 years
Modula2-2	ECE	B.S.	1984	M.S.	2nd	2 years
Pascal-1	EE	B.S.	1984	M.S.	4th	6 years
Pascal-2	EECS ECE	B.S. M.S.	1984 1986	Ph.D.	2nd	2 years
Prolog-1	EE CS	B.S. M.S.	1984 1986	Ph.D.	2nd	3 years
Prolog-2	CS	B.S.	1986	M.S.	2nd	3 years
T-1	EECS	B.S.	1983	M.S.	3rd	2 years
T-2	CS	B.S.	1986	M.S.	2nd	3 years
Coord-1	ECE CS	B.S. M.S.	1981 1984	Ph.D.	4th	6 years
Coord-2	CS CS	B.S. M.S.	1984 1986	M.S.	2nd	3 years
Coord-3	ECE	B.S.	1986	M.S.	2nd	2 years

Table 1: Summary of the UCLA Programmer and Coordinator Background

members of other teams. The coordinating team monitored the progress of each team. Work-related communications between programmers and the coordinating team were conducted *only* via a formal tool (electronic mail). The programmers directed their questions to the coordinating team, who then tried to respond as quickly as possible. Whenever necessary, the help of the H/S flight control experts was provided by telephone and meetings to resolve questions.

Generally, each answer was *only* sent to the team that submitted the corresponding question. The answer was broadcast to all teams only if the answer led to an update or clarification of the specification, if there was an indication of a misunderstanding common to some teams, or if the answer was considered to be important or relevant for other teams for some other reason. In the first case, a broadcast constituted an official amendment to the original specification. This contrasts with the communication protocol used in the NASA experiment [Kell86], where the answers to *all* questions were broadcast, regardless of which team submitted the question. The resulting flood of messages proved to be a bothersome overload, that was avoided this time. The communication diagram among H/S experts, the UCLA coordinating team and the programming teams is presented in Figure 4-3.

4.3.6.3 Experience with the Communication Protocol

The communication protocol was designed in order to: (1) prevent the ambiguity of oral communications; (2) give the coordinating team time to think and discuss before answering a question, and to summon the help of H/S flight control experts, if necessary; (3) provide a record of the communication for possible analysis; (4) reduce the number of messages sent to each individual team; and (5) adhere to the principle of supplying only absolutely necessary information to the programming teams, thus aiming to avoid any bias on a team's design decisions by supplying unnecessary and/or unrequested information.

With respect to the first three goals the protocol was very successful. However, it frequently proved to be more difficult to write the answer to a certain question, whereas oral communication would have been easier and more efficient.

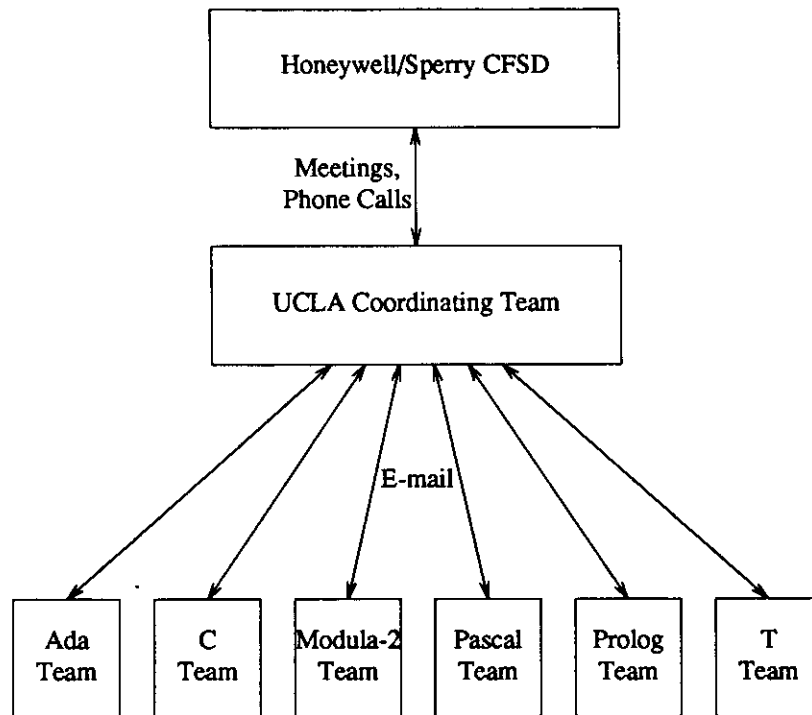


Figure 4-3: Communication Diagram of the Experiment

The communication with H/S was very efficient; thus it was possible to answer all questions within a short time – usually less than one day.

Altogether, about 120 questions were sent by the programming teams. The answers to only 30 of them were broadcast. The total number of broadcast messages was 40, three of which required an additional follow-up message, to provide further clarification or to correct errors in the original message. Ten broadcast messages were not triggered by questions: five of them were sent because either the coordinating team, or H/S detected an error in the specification or for some other reason decided to update it, and the other five were a result of the Design Review at which some common misinterpretations of the specification were observed. The individual teams received between 53 and 64 messages. This constitutes a reduction by a factor of 2 in

comparison with the number of messages that would have been received if the communication protocol of the NASA experiment [Kell86] had been used.

4.3.7 Build the Software

4.3.7.1 Schedule of the Experiment

The software version generation for this experiment was conducted in six phases:

1) *Training meetings (five in total, 2-4 hours each):*

One project-introduction meeting was offered to all the applicants, and all other four meetings were held after the selection of personnel. H/S presented a discussion of flight control systems as background information. Introductory presentations were made summarizing the experiment's goals, requirements and the multiple version software techniques. Issues of different programming languages were also discussed. A kick-off meeting was held on the first day of the software development phase. At that meeting, the programmers were given the written specifications and documentation on system tools to start their 12-week effort. Rules and guidelines about schedules, deliverables, and communication protocols were also clearly defined. The programmers were strongly motivated and showed serious concerns about the project in these meetings. The need for inter-team isolation was thoroughly discussed and clearly acknowledged by all programmers.

2) *Design phase (4 weeks):*

At the end of this four-week phase, each team delivered a design document

following the guidelines and formats provided at the kick-off meeting. Each team delivered a design walkthrough report after conducting a walkthrough that was attended by UCLA, H/S principal investigators, the UCLA coordinating team, and an H/S software expert.

3) *Coding phase (3 weeks):*

By the end of this 3-week phase, programmers had finished coding, conducted a code walkthrough by themselves, and delivered a code development plan and a test plan. Code Update Report forms were distributed for them to record every change that was made after the code was generated.

4) *Unit testing phase (1 week):*

Each team was supplied with sample test data sets (generated by H/S) for each module that were suitable to check the basic functionality of that module. They had to pass all the unit testing data before they could proceed to the next phase. One week was allotted to this phase. At the end of this phase, each team conducted a coding/testing review with UCLA coordinators and H/S representatives to present their progress and testing experience.

5) *Integration testing phase (2 weeks):*

Four sets of partial flight simulation test data were produced by H/S and provided to each programming team for integration testing. This phase of testing was intended to guarantee that the software was suitable for the closed-loop simulation of the integrated system.

6) *Acceptance testing phase (2 weeks):*

Programmers formally submitted their programs. Each program was run in a test harness of nine flight simulation profiles. When a program failed a test, it

was returned to the programmers for debugging and resubmission, with the input case on which it failed. By the end of this two week phase, five programs had passed the acceptance test successfully. The T program encountered difficulties in using the T interpreter, and it was necessary to do additional work over the next month before that version passed the acceptance test.

All the participants of this project presented concluding talks and met each other socially at a final one-day workshop when the software generation phase ended. During that occasion programmers were free to talk with each other and exchange their experiences. A large variety of experiences, viewpoints and difficulties encountered were brought out during this final workshop and following party.

4.3.7.2 The Programming Process

The software engineering process involved in this project for the program generation included periodic formal reviews, step-by-step milestone deliverables, and standard documentation facilities. This controlled process provided continuous interactions between the coordinators from UCLA and H/S, and each individual team.

The design review, the coding/testing review, and the final review and workshop were the three formal reviews within this project, all with the participation of H/S experts. These reviews were designed to follow industrial standards as much as possible. Moreover, they served as checkpoints to observe the progress of each programming team and to adjust the development process according to their feedback.

For the purpose of keeping a complete record, several *deliverables* were required from each team. These deliverables, representing the products of the project, included two *snapshots* of each module (before and after unit tests), four snapshots of the complete program (those before and after integration tests, and before and after acceptance tests), two design documents (preliminary and final versions), program metrics, design walkthrough reports, and code update reports. These deliverables helped to set up the bug removal history of each program, and to facilitate the retrieval of on-line program faults for *mutation testing* reported in the evaluation phase.

Since fault-reporting was considered extremely important for this project, each team was required to report all the changes made to their program, starting from the time when the program first compiled successfully. All changes had to be reported, no matter whether they were due to detected faults, efficiency improvement, specification updates, etc. For each change a Code Update Report, a standardized form designed by the coordinating team, had to be turned in. If a code change was made because of a design change, a Design Walkthrough Report (another standardized form) had to be submitted as well. For subsequent analysis, we considered only those changes that were done to correct faults in the programs.

4.3.8 Conduct High Quality Testing

A high quality design for testing should consider the effect of different high order programming languages used in this project. For efficiency purposes, a general interfacing routine between C (in which the airplane simulator was implemented) and other languages was resolved and provided by the coordinating team (with some special treatments for Prolog and T). With the suggestions from H/S engineers, proper testing granularities were elaborated. They were roughly at the same order of

magnitude as the resolution of the application, which was a very fine calibration. Meaningful input test data (with high coverage) were generated together with their validated output values. Tests were carried out at each defined cross-check point by an error comparator in the interfacing routine.

To emphasize the importance of testing, three phases of testing: unit tests, integration tests, and acceptance tests, were introduced for error detection and fault removal. Different strategies for program testing were provided, during the program generation phase, in order to clean up programs before they were subjected to a final evaluation. Table 2 lists the differences among these phases.

category	unit test	integration test	acceptance test
test case generator	open loop by PC Basic	closed loop by PC Basic	closed loop by multiple languages
test data access	file i/o by each version	interfacing C routines	interfacing C routines
tested by	individual teams	individual teams	coordinating team
tolerance level	0.01 for degrees	0.01 for degrees	0.005 for degrees (0.05 for Prolog)

Table 2: Different Schemes Used in the Testing Phases

At first, a reference model of control laws was implemented and provided by H/S flight control software engineers. This version was implemented in Basic on an IBM PC to serve as the test case generator for unit tests the integration tests. Criteria of "open loop testing" and "closed loop testing" were used, respectively. Detailed descriptions for these two testing strategies are provided later in this Chapter. Due to the wide numerical discrepancies between this version and the other six versions, a larger tolerance level was chosen.

Later in the acceptance test, this reference model proved to be less reliable (several faults were found) and less efficient, since the PC was quite slow in numerical computations and I/O operations. Thus, it was necessary to replace it with a more reliable and efficient testing procedure for a large volume of test data. For this procedure, the outputs of the six versions were voted and the majority results were used as the reference points to generate test data during the acceptance tests. This was also the test phase during which programmers were required to submit their programs to the coordinating team and wait for the test results. A finer tolerance level was used based on the observation that less discrepancies were expected if programs computed the right results. An exception had to be made for the Prolog program due to the lack of accuracy in its internal representation of real numbers.

As to the numbers of test cases performed in each phase, the detailed information is presented in the following three tables.

name of the module	number of test cases
Baro Altitude Filter	7
Radio Altitude Filter	11
Glideslope Filter	7
Mode Logic	11
Altitude Hold Mode Outerloop	10
Glideslope Mode Outerloop	12
Flare Mode Outerloop	15
Innerloop	23
Command Monitor	5
Display Module	32
total	133
total frames	about 1330

Table 3: Test Data in Unit Test Phase

In the Unit Test phase, each module of the program received a variant number of test cases (see Table 3). A Total of 133 test cases were executed, and since each test case contained 10 frames †, there were 1330 frames in this phase.

id	testing time	involved modes	wind turbulence
data.1	12 sec	AHD mode only	no wind turbulence
data.2	12 sec	AHD-GSCD-GSTD	no wind turbulence
data.3	12 sec	AHD mode only	average wind turbulence
data.4	12 sec	AHD-GSCD-GSTD	average wind turbulence
total frames		960 frames*	

* Each second has 20 frames of execution.

Table 4: Test Data in Integration Test Phase

Four testing profiles were provided for the Integration Test as shown in Table 4. Each test profile contained 12 seconds of flight simulation, and in total 960 frames were executed. These four test data sets differed from each other in the flying modes that were involved (either Altitude Hold Mode only, or from Altitude Hold Mode to Glide Slope Capture and Track Modes), and by the level of wind turbulence (either no wind turbulence or an average wind turbulence) being superimposed.

† One frame denotes one execution run for the module or the program.

id	time	involved modes	turbulence	other test
data.1	100 sec	AHD-GSCD-GSTD	no	no
data.2	180 sec	AHD-GSCD-GSTD-FD-TD	no	no
data.3	100 sec	AHD-GSCD-GSTD	average	no
data.4	180 sec	AHD-GSCD-GSTD-FD-TD	average	no
data.5	100 sec	AHD-GSCD-GSTD	large	no
data.6	180 sec	AHD-GSCD-GSTD-FD-TD	large	no
data.7	30 sec	AHD mode only	large	recovery
data.8	22 sec	AHD-GSCD-GSTD-FD-TD	large	recovery
data.9	30 sec	AHD mode only	large	display
total frames	18440 frames*			

* Each second has 20 frames of execution.

Table 5: Test Data in Acceptance Test Phase

The Acceptance Test was a stringent testing phase. As indicated in Table 5, nine testing profiles were designed for this test phase. Data sets 1, 3, and 5 executed for 100 seconds, drove the airplane from Altitude Mode to Glide Slope Capture and Track Modes, with either no, average, or large wind turbulences, respectively. Data sets 2, 4, and 6 were designed similarly, except they exercised all five flying modes for 180 seconds. Data sets 7 and 8 were designed to carry out the recovery command, and Data set 9 tested the Display module. The total executions required in this phase were 18440 frames.

In summary, there were 20730 executions imposed on these programs before they were accepted and subjected to the final evaluation in the following stage.

4.3.9 Execute Multi-Version Systems

Since the software generation phase was completed in early September 1987, the UCLA coordinating team has conducted H/S approved stress testing for nine months. The major strategy in this requirements-based testing is so-called *dynamic closed-loop* tests, which have the purpose of verifying performance, detecting any tendency towards dynamic mistracking between the different program versions, and exposing requirements faults not caught in static testing.

In practice, the three channels of diverse software each computes a surface command to guide a simulated aircraft along its flight path. To ensure that significant command errors could be detected, random wind turbulences of different levels are superimposed in order to represent difficult flight conditions. The individual commands are recorded and compared for discrepancies that could indicate the presence of faults.

The configuration of the flight simulation system (shown in Figure 4-4) consists of three lanes of control law computation, three command monitors, a servo control, an Airplane model, and a turbulence generator.

The *lane computations* and the *command monitors* are the redundant software versions generated by the six UCLA programming teams. Each lane of independent computation monitors the other two lanes. However, no single lane can make the decision as to whether another lane is faulty. A separate servo control logic function is required to make that decision, based on the monitor states provided by all the lanes. This control logic applies a strategy that ignores the elevator command from a lane when that lane is judged failed by both of the other lanes, and these lanes are judged valid.

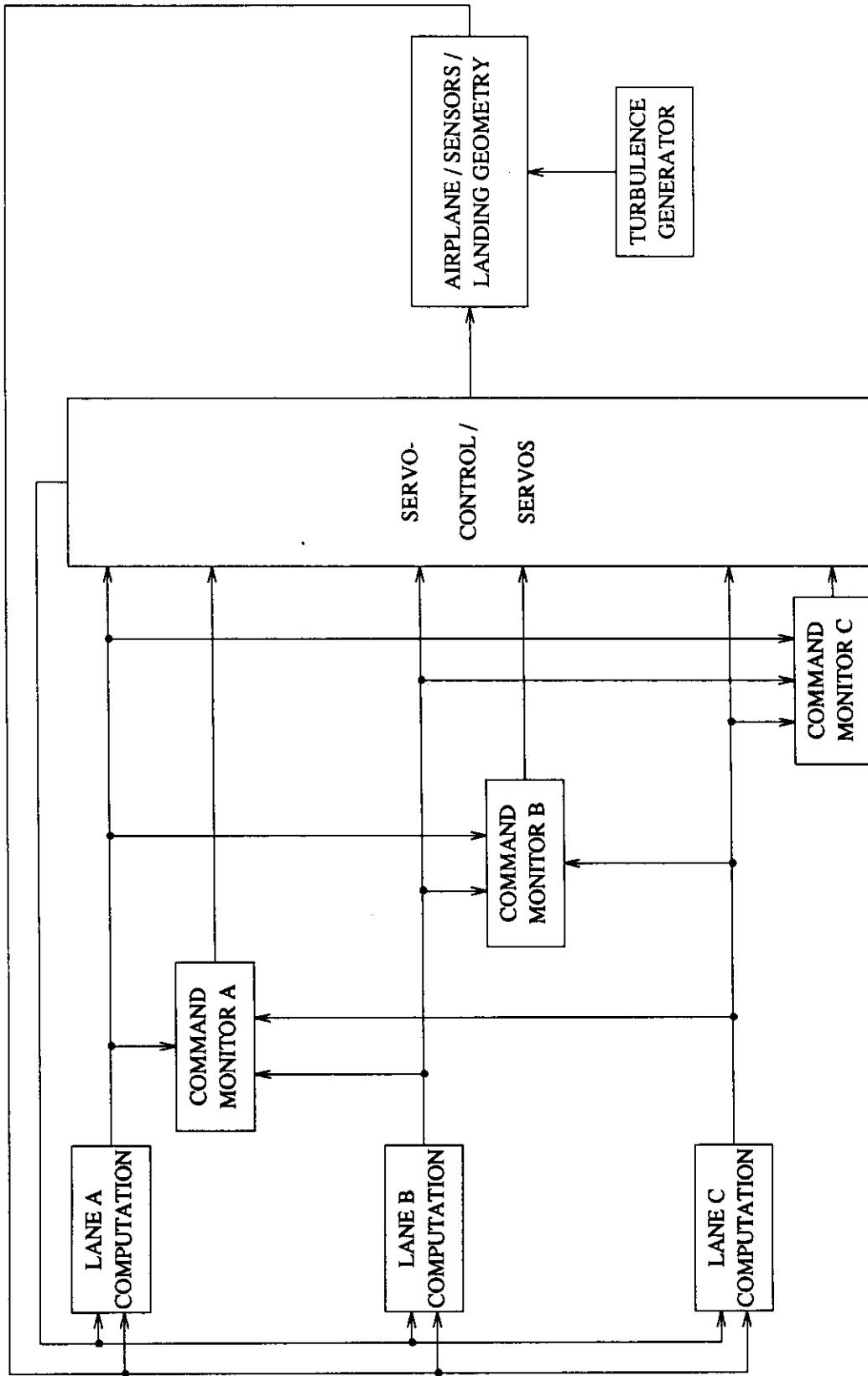


Figure 4-4: 3-Channel Flight Simulation Configuration

The Airplane simulator computes the response of an airplane to each elevator command. In a real aircraft these values would be directly measured by sensors. The landing geometry model describes the deviation relative to the glide slope beam. Moreover, in order to provide a set of inputs to the Airplane model that create large error magnitudes, and thereby force off-nominal software operating conditions, turbulence in the form of vertical wind gusts is introduced.

One run of flight simulation is characterized by the following five initial values: (1) initial altitude (about 1500 feet); (2) initial distance (about 52800 feet); (3) initial nose up relative to velocity (range from 0 to 10 degrees); (4) initial pitch attitude (range from -15 to 15 degrees); and (5) vertical velocity for the wind turbulence (0 to 10 ft/sec). One simulation consists of 5000 time frame computations of 50 msec/frame, for a total landing time of 250 seconds.

For the purpose of efficiency, a testing procedure equivalent to Figure 4-4 was used (approved by H/S): first, each lane by itself guided the airplane for a complete landing; second, the whole history of the flight simulation was recorded; and finally, the flight profiles of all versions were compared and analyzed to observe discrepancies and determine faults. In this manner, over 1000 flight simulations (over 5,000,000 time frames) have been exercised on the six software versions generated from this project.

In addition to the flight simulations, a structural analysis was also carried out. The six versions were compared to find the differences in structure and implementation that resulted from the application of the multi-version software methodology. The efforts of finding more faults (either requirements-based or structure-based) and the search for evidence of structural diversity among these

programs have been the major concerns. An additional benefit of this analysis was that it necessitated a thorough code inspection, during which some additional faults that were not caught by any tests were detected [Schu87].

4.3.10 Evaluate Effectiveness of MVS and Refine the Process

Detailed evaluation of MVS generated in this UCLA/Honeywell joint project has been a major final result. Moreover, there have been several improper treatments of in the software development (e.g., incorrect test data supplied by H/S), and backtracking to a previous stage in the paradigm has proven necessary. Proper coordination in conducting the overall design procedure played an crucial role in the success of the design. Although the paradigm allows the entire design procedure to be repeated if necessary, the experiment did not suffer this overhead since the product was shown to meet the original requirements. The final evaluation of the experiment also contributed to the refinement of the proposed design paradigm. All the final observations are provided in the next chapter.

CHAPTER 5

EVALUATION AND REFINEMENT

For the purpose of assessment and refinement of the design paradigm, we have evaluated many aspects of the UCLA/Honeywell investigation, in order to assess the impact of the proposed design paradigm during the design of a MVS system.

5.1 Standard Software Metrics of the Programs

At first, we should evaluate the quality of the programs produced in the UCLA/Honeywell joint project. The objective of software metrics is to evaluate the quality of the process or product in a quality assurance environment. However, our focus here is the comparison among the program versions, since design diversity is our major concern.

Table 6 gives several comparisons of the six versions with respect to some common software metrics [Li87]. These comparisons were based on the report delivered from the programmers, and certified by the UCLA coordinating team.

Metrics	ADA	C	MOD-2	PASCAL	PROLOG	T	Range
LINES	2253	1378	1521	2234	1733	1575	1.64:1
STMTS	1031	746	546	491	1257	1089	2.56:1
LN-CM	1517	861	953	1288	1374	1263	1.76:1
OBJS	85.6k	83.7k	51.9k	37.5k	N.A.	N.A.	2.28:1
MODS	36	26	37	48	77	44	2.96:1
STM/M	29	25	15	10	16	25	2.90:1
CALLS	97	68	65	93	81	87	1.49:1
LIBS	2	2	2	10	7	N.A.	5.00:1
LCALL	3	9	6	12	61	N.A.	20.33:1
GBVAR	139	141	91	81	90	97	1.74:1
LCVAR	117	197	132	127	209	251	2.15:1
CONST	68	21	18	16	N.A.	N.A.	4.25:1
BINDE	74	114	78	118	74	86	1.59:1
COMP	2'3"	23"	4'22"	33"	N.A.	N.A.	11.39:1
EXEC	5'30"	2'41"	12'18"	2'37"	19'17"	250'	95.54:1

N.A. = not applicable

Table 6: Software Metrics for the Six Programs

The following metrics are considered in Table 6: (1) the number of lines of code, including comments and blank lines (LINES); (2) the number of executable statements, such as assignment, control, I/O, or arithmetic statements (STMTS); (3) the number of lines excluding comments and blank lines (LN-CM); (4) the size of the object code (OBJS), we note that this metric is not applicable to the PROLOG and the T programs; (5) the number of programming modules (subroutines, functions, procedures, etc.) used (MODS); (6) the mean number of statements per module (STM/M); (7) the number of calls to programming modules (CALLS); (8) the number of library functions used (LIBS), we note that this metric is not applicable to the T

program since there is no notion of "library functions;" (9) the number of calls to library functions (LCALL), we note that this metric is also not applicable to the T program; (10) the number of global variables (GBVAR); (11) the number of local variables (LCVAR); (12) the number of constants (CONST), we note that this metric is not applicable to the PROLOG and the T programs, since their local or global variables have to be used as *constants*; (13) the number of binary decisions (BINDE); (14) the compile time to generate the final object code (COMP); and (15) the execution time for one flight simulation profile (EXEC).

Note that the maximum value of each row is typed in bold, while the minimum is typed in italic. The ratio between these two values is shown in the last column to represent the range of each item. Also note that the compile times and the execution times were measured on a lightly loaded Sun-3/280 machine (except that Modula-2 version was measured on Vax-11/750 system, which was estimated to be four to five times slower). Moreover, the T interpreter (Yale University Version 3.0) ran very slowly due to the tremendous overhead spent on its garbage collection facility.

5.2 Distribution of Faults Detected during Program Development

A total of 82 faults was found and reported during program development. The following four tables present the distribution of these faults in the six versions under different categories. Detailed descriptions of these faults can be found in Appendix A.

Table 7 shows the fault distribution in each system function. The total adds up to more than 82 since all the modules affected by one fault are counted. An asterisk indicates such a case.

Subfunction	ADA	C	MOD-2	PASCAL	PROLOG	T	Total
Main Program	1	2	0	0	7	6*	16
BACF	1	0	1	0	2*	3*	7
RACF	0	0	0	0	1*	1*	2
GSCF	1	1	1	4	7	2*	16
Mode Logic	1	4	0	0	1*	2*	8
AH Outer Loop	0	0	0	1*	0	0	1
GS Outer Loop	0	1	0	0	0	0	1
Flare Outer Loop	1	2	0	1	2*	1	7
Inner Loop	1	3	0	4*	4*	2	14
Command Monitor	0	0	0	0	1	2	3
Display	0	0	1	3	1	1	6
General, other	0	0	1	0	5*	4	10
Total	6	13	4	13	31	24	91

*: This fault affected more than one subfunction.

Table 7: Fault Distribution by Subfunctions

Classification of faults according to fault types is shown in Table 8. This category considers the following type of faults [Kell82] (1) typographical (a cosmetic mistake made in typing the program); (2) error of omission (a piece of required code was missing); (3) unnecessary implementation (an unintended extra piece of code which was deleted); (4) incorrect algorithm (a deficient implementation of an algorithm); (5) specification misinterpretation (a misinterpretation of the specification); and (6) specification ambiguity (an unclear or inadequate specification which led to a deficient implementation). In this category, items (1) through (4) are

implementation related faults, while items (5) and (6) are specification related faults. Item (2) and (3) are complementary to each other. It is also noted that "incorrect algorithm" of item (4) is the most frequent fault type, which includes miscomputation, logic fault, initialization fault, and boundary fault. The "other" fault was introduced by the Modula-2 compiler.

Fault Class	ADA	C	MOD-2	PASCAL	PROLOG	T	Total
Typo	0	1	0	0	9	0	10
Omission	1	3	0	0	8	5	17
Unnecessary	1	0	0	1	0	2	4
Incorrect Algorithm	3	5	2	7	9	13	39
Spec. Misinterpretation	1	3	1	4	0	1	10
Spec. Ambiguity	0	1	0	0	0	0	1
Other	0	0	1	0	0	0	1
Total	6	13	4	12	26	21	82

Table 8: Fault Classification by Fault Types

Table 9 shows during which phases of testing the faults were detected.

Test Phase	ADA	C	MOD-2	PASCAL	PROLOG	T	Total
Coding/Unit Testing	2	4	4	10	15	7	42
Integration Testing	2	5	0	2	7	4	20
Acceptance Testing	2	4	0	0	4	10	20
Total	6	13	4	12	26	21	82

Table 9: Fault Classification by Phases

Finally, Table 10 shows the classification of faults according to the categories of "requirements fault" and "structural fault."

	ADA	C	MOD-2	PASCAL	PROLOG	T	Total
Requirements	5	12	3	11	21	19	71
Structural	1	1	1	1	5	2	11
Total	6	13	4	12	26	21	82

Table 10: Fault Classification: Requirements Faults vs. Structural Faults

All cross-check and recovery point routines were written in the C programming language, and therefore five of the six programs had the additional problem of interfacing to another language. The Prolog and the T team had the most severe problems. The Prolog team had to modify the Prolog interpreter; the solution of the T team was to convert all parameters to ASCII strings, pass them to a C routine, convert them back into numbers, do the cross-checking, convert the results into strings, and pass them back to the T functions.

Three compiler or interpreter bugs were found during program development: the Ada compiler did not support nested generic packages (which resulted in a design change to avoid using this feature). With the Modula-2 compiler the expression "i+i" had to be used as an array index instead of "2*i" to achieve the desired result. This fault is classified as the type "other" in Table 8. The T interpreter had a problem with its garbage collection which resulted in uncompleted long test runs. This problem delayed the T program's passing of the acceptance test for over a month.

In addition, we experienced a computing environment change during the experiment. This did cost some time, but finally all teams were moved to the new Sun workstations. Only the Modula-2 team had to continue to use the original VAX computers, due to their compiler not being available on the Sun.

disagreement. This fault is traceable to an ambiguity in the specification: the graphical language used was not powerful enough to express the exact semantics of the required operation. The third fault discovered in the C version is the too frequent initialization of a state variable (it is re-initialized at every pitch mode change, while it should be initialized only once at the entry of Altitude Hold mode). In this case, the team did not follow a specification update that was made very late in the programming process (during integration testing).

Two disagreements were traced to an identical fault; they occurred in the Prolog and T versions. Both teams made the same design decision to update a state variable of the Inner Loop twice during one computation of the Inner Loop. This fault is due to the same specification ambiguity as mentioned above, but in addition these teams did not pay attention to a broadcast clarification that addressed exactly that problem. Although similar in nature, the two versions disagreed in slightly different ways from the other versions.

It is noteworthy that all observed disagreements were very small, and further experiments showed that the versions with these discrepancies are always able to achieve proper Touchdown. Furthermore, all these faults are specification related. It is interesting to note that the Inner Loop was the program part that was most thoroughly tested during all test phases.

5.3.2 Faults Found During Inspection of Code

The following faults were detected during the code inspection performed as part of the structural analysis:

One *requirements fault* was found in the Display, where rounding to 5 significant digits was not done correctly. The error occurs only when rounding overflow (e.g., 6 or more subsequent 9's) changes the decimal point position. This special case was not triggered by any of the acceptance test or flight simulation data. Other teams, however, had discovered the same kind of fault during unit testing. Therefore one explanation might be that this team did not perform the unit test sufficiently carefully.

The other six faults were three types of structural faults, discovered in the C, Modula-2, Pascal, Prolog, and T versions. They and their possible impacts are discussed next.

One fault was Type 1, as described next. Normally, the boundaries within which the output of certain functions (integrator, rate limiter, and magnitude limiter) had to be limited was a finite constant. There were a few cases (in the Inner Loop and the Command Monitor), however, where the bound was either $+\infty$ or $-\infty$. To implement these special cases, the C version used the arbitrarily chosen values +99999.0 or -99999.0 and passed them as parameters to the subprogram that implements the functions mentioned above. This is a structural fault because an unintended (unspecified) function (i.e. the limiting of an output value) is performed if this value exceeds the arbitrarily chosen values. In this application, however, this might not be a problem since the output of the Inner Loop (elevator command) will be further limited to ± 15 degrees. Similarly, the Command Monitor will indicate a disagreement between two versions long before this structural fault has any effect.

Type 2 faults are more serious. They are caused by the introduction of new, unspecified state variables which we call "underground variables," since they are neither checked nor corrected in any cross-check or recovery point. This may lead to

an inconsistent state that is impossible to recover from. An example follows: the C team decided to move the computation of some parameters for the Glide Slope Deviation Complementary Filter outside of this Filter. Unfortunately, this computation depends on some other, state dependent computations in this Filter. These latter computations were re-implemented outside the Glide Slope Deviation Complementary Filter which also led to a duplication of their state variables. Therefore, a new design rule for multi-version software must be stated as "Do not introduce any *underground* variables." Note that this rule is irrelevant if only cross-check points are used, since these do not attempt to recover the internal state of the version. Only one Type 2 fault was uncovered.

Type 3 faults occurred when the C, Modula-2, Prolog, and T teams used the output of the Mode Logic in some further – but different – computations before it was voted upon. This was in violation of a rule stated in the specification, explicitly forbidding that. If the Mode Logic output is corrected by the Decision Function, a fault of this kind could lead to a situation where the Mode Logic output is correct, but the variables dependent on this output are not, since they were computed using the old, uncorrected values of the Mode Logic output. Then an inconsistent state between different variables of the version might exist which could be impossible to recover from. Apparently, more programmer training is necessary to prevent these types of mistake since the reason for this fault is obviously a misunderstanding or unawareness of some of the multi-version software design rules. Although this might seem a dangerous possibility of introducing common faults, faults of this kind are easily checked for. Thus they can be eliminated by the acceptance test. We conclude that the acceptance test should always check for compliance with all the N-version software design rules specified.

The six discovered structural faults that are described above are uncorrelated, and thus will be tolerated by the multi-version software approach.

5.4 Assessment of Structural Diversity

A fundamental first step in assessing the diversity that is present in a set of versions must be an assessment of the *potential for diversity* (PFD) that is indicated by a given specification. Some reasonable evidence that *meaningful diversity* can occur is needed in order to justify the effort of multi-version programming. Here we exclude the "pseudo-diversity" that can be attained by rearranging code, using simple substitutions of identities, etc. It is introduced too late in the programming process to be effective, and is likely to replicate and camouflage already existing faults.

After the PFD assessment, a decision must be made whether certain diversity shall be "enforced", i.e., specified; examples would be a requirement to use different algorithms [Chen78a], several versions of the specification [Kell83], different compilers, programming languages, etc. The alternative is to depend on the isolation between programmers and on the differences in their backgrounds and approaches to the problem as the means to get diversity. This is the "random" approach to the attainment of diversity.

It is our position that the minimal requirement must be (1) the isolation of programming efforts, and (2) "enforced" diversity that is needed to avoid predictable causes of common faults, such as compiler bugs and other defects that could exist in a shared support environment.

In the present investigation the only additional choice of "enforced" diversity is the use of six different programming languages. One of our goals is to evaluate the effectiveness of this choice in attaining meaningful diversity between the six versions that originated from one specification. A summary of the observations follows [Schu87].

Program Module	PFD	Observed Diversity
Main Program	good	level of detail implemented, information handling, organization of state variable initialization, placement of calls to vote routines
Radio Altitude Complementary Filter	poor	grouping, sequence
Barometric Altitude Complementary Filter	medium	grouping, sequence
Glide Slope Deviation Complementary Filter	medium	grouping, sequence, time-dependent computation
Mode Logic	good	constants, sequence, algorithm
Altitude Hold Control Law, Outer Loop	poor	constants, grouping, sequence
Glide Slope Capture and Track Control Law, Outer Loop	good	constants, grouping, sequence, time-dependent computation
Flare Control Law, Outer Loop	good	constants, grouping, sequence
Inner Loop	poor	constants, grouping, sequence, organization
Command Monitor	poor	grouping, algorithm, organization
Mode Display	poor	algorithm
Fault Display	poor	algorithm
Signal Display	medium	algorithm
Primitive Operations	poor	choice, organization

Table 11: Potential for and Observed Diversity

The "PFD" column of Table 11 presents our assessment of the extent of diversity (structural differences) that may be expected for each program module. A module has *poor* potential for diversity if it is either so small and simple, or else if its computation sequence (in terms of primitive operations) is so well-defined by data dependencies, that there is little room for diversity in implementation and organizational aspects. In the modules with *good* potential for diversity, many (between 5 and 10) independent computation paths exist which could be traversed in any order. In the case of the Main Program the sequence of the major system functions is determined by data dependencies (cf. Figure 4-1); here the PFD lies in the organizational aspects. Modules with "medium" PFD are estimated to lie somewhere between these two limiting cases. It must be noted that the PFD assessment is somewhat subjective; the factors used in the assessment include the specification of each program module as well as the observed structural differences.

The column "Observed Diversity" of Table 11 lists the attributes in which structural diversity actually was observed between two or more of the six versions. Further explanations and comments on this column follow.

The first notable difference between the Main Programs is the level of detail implemented there. The Ada version is one extreme example; it deals with all the organizational details, such as initialization of state variables, or determination of which function to perform at a given instant, in the Main Program. This leads to a calling hierarchy which is exactly one level deep, if some auxiliary subprograms and the calls to primitive operations are ignored. The T version is similar in the sense that all the system functions are called directly by the Main Program. However, most of the organization (especially initialization of state variables) is done locally by these system functions. The other versions (C, Modula-2, Pascal) generally show a two-

level calling hierarchy, i.e., they define relatively general subprograms like "Filter Module," "Mode Logic," or "Altitude Hold Control Law," and deal with the organization of the appropriate system functions locally. Nevertheless, there are some differences between these latter versions too. For instance, the C and Modula-2 versions organize the Control Laws into three different Control Laws (one for each pitch mode), each consisting of an Outer and an Inner Loop. The Pascal version, on the other hand, divides the Control Laws into an Outer and an Inner Loop, where the Outer Loop consists of three different Outer Loop procedures. Finally, the C and Modula-2 versions differ also in the organization of their Filter Module, or their Mode Logic. The Prolog version is a special case. It has a rather large and complex calling hierarchy because the language is such that IF-statements have to be implemented by function calls.

Another important difference that was noted is the strategy chosen to handle information, i.e., state, interface, and output variables. Solutions range from extensive parameter passing (Pascal) to the exclusive use of global variables (C, Prolog). We note that this choice was unavoidable for the Prolog version because of the language properties. The other versions use solutions between these two extremes, by trying to define as many variables as possible locally. The choices are partly programming language dependent, e.g., dependent on the availability of local static variables. A related aspect is the organization of state variable initialization: the two basic solutions are initialization by the Main Program, or initialization within each program module.

The third aspect of diversity in the Main Program concerns the placement of vote routines: either all vote routines are called in the Main Program, or they are called in the system function whose result they check. The recovery point routine,

however, is always called by the Main Program.

The notation "constants" in Table 11 indicates that some teams chose to simplify the computation by manually evaluating some expressions consisting of constants only. "Grouping" refers to the fact that different teams chose different ways of combining primitive operations into statements of their programming language. "Sequence" denotes that some versions use a different computation sequence (in terms of primitive operations) to implement a system function than others. Sometimes the differences are very minor, for instance in the Outer Loop of the Altitude Hold Control Law or in the Inner Loop.

"Time-dependent computation" means that this system function contains an algorithm that is dependent on real time. In both cases, we observe much variety among the strategies chosen (1) to keep track of real time; and (2) to guard against effects of limited precision of real number representation. (Note: Real time was simulated in this application.)

"Algorithm" indicates that different versions use different algorithms to implement a certain system function. These differences are mostly minor ones; only in the Signal Display more interesting differences can be found, both in the structure of the algorithm and in implementation details.

"Organization" refers mainly to the fact that some versions chose to implement a certain subprogram as a procedure (results are returned via parameter passing), while other versions used a function (a RETURN statement or similar construct is used). In the case of the Inner Loop, slightly different requirements existed for different pitch modes. A variety of solutions to cope with these has been found.

Primitive operations are integrators, linear filters, magnitude limiters, and rate limiters. The algorithms for these operations were exactly specified, however, different choices of which primitive operations to implement as subprograms have been made, mainly whether the integrators include limits on the magnitude of the output value (as is required in most cases), or not. Only the Prolog version implemented a "switch" subprogram; this choice has clearly been influenced by programming language properties – all other versions just use IF-statements. The T version defined only a subprogram for magnitude limiting, all other primitive operations are implemented directly in each system function. The Prolog version uses procedures to implement these operations, all other versions use functions. Lastly, the Ada functions also do the state update, in the case of state dependent primitive operations; all other versions have to do this within each system function.

Due to space constraints, no examples could be given here. More detailed discussions and presentations can be found in [Schu87]. In conclusion, we note that both the PFD assessment and the search for meaningful structural differences were based on individual judgements of the investigators and are somewhat subjective. However, it is evident that (1) aspects of meaningful diversity can be identified; and (2) diversity in programming languages definitely motivates structural diversity between the versions. We hope that our modest first steps will stimulate further investigations into the problems of qualitative and quantitative assessment of meaningful diversity in a set of program versions.

5.5 Observations from the Diversity Assessment

In general, it can be said that more diversity was observed in a module with the aspect that its implementation method was not explicitly stated in the specification, such as the Signal Display, the organization of different Inner Loop algorithms (depending on the pitch mode), the organization of state variable initialization, or the implementation of time-dependent computations. Furthermore, not all the design choices outlined above can be made independently. For instance, whether a primitive operation is defined as a function or a procedure determines if it can be combined with other operations in a single statement, or not. Similarly, if the update of state variables is performed as part of the primitive operation, then the upper levels do not have to concern this. As a last example, if the state variables of a system function are defined as local static variables, then they cannot be initialized by the Main Program.

Two factors that limit actual diversity have been observed in the course of this assessment. One of them is that programmers obviously tend to follow a *natural* sequence, even when coding independent computations that could be performed in any order. The observation made was that algorithms specified by figures were generally implemented by following the corresponding figure from top to bottom. In this case the "natural" order was given by the normal way to read a piece of paper, i.e. from left to right and from top to bottom. Only when enforced by data dependencies, a different order was chosen, e.g. from bottom to top. It can be safely assumed that the same phenomenon would occur if the specification was stated in another form than graphical; this is especially true for a textual description. The latter can be exemplified by the Display Module: only one team chose the order of computation Fault Display, Mode Display, Signal Display; all other teams chose the order Mode

Display, Fault Display, Signal Display which was also the order used in the specification. This means that if there is a number of independent computations that could be performed in any order there exist some permutations of these computations that are more likely to be chosen than other permutations, due to human, psychological factors.

The Outer Loops of the Glide Slope Capture and Track and the Flare Control Law, and the Mode Logic were affected the most; their good potential diversity was not exploited as much as expected and possible, due to this phenomenon. In retrospect, a second reason for this lack of diversity is that we have concluded that the logic part of the Mode Logic was overspecified. A description of the conditions that have to be met to enter the next pitch mode would have been more appropriate than the logic diagram which biased the programmers too much towards using identical or very similar algorithms.

One possible solution to the *natural* sequence problem is to provide different specifications to individual teams. They could either be required to follow a specific unique computation sequence, or the order of presenting the independent computations could be different in each specification while still having each team decide which sequence to follow. The problem of this approach is the possibility of introducing additional faults into the specification, i.e., more faults than would have been made in a single specification, unless the process of generating different versions of a specification can be proven to be correct.

The H/S concept of *test points* is the second factor that tends to limit diversity. Their purpose is to output and compare not only the final result of the major subfunctions, but also some intermediate results. However, that restricted the programmers on their choices of which primitive operations to combine – efficiently –

into one programming language statement. In effect, the intermediate values to be computed were chosen for them. These restrictions are rather unnecessary and can easily be removed. An additional benefit is that output and the use of vote routines would become simpler. On the other hand, the test points proved very beneficial in version debugging. A way to preserve this useful feature is to add test points only during the testing and debugging phase, and to remove them afterwards. Each team should be free to choose its own test points; in addition, the program development coordinator can request specific test points if it is intended to compare the results of two or more different versions.

5.6 Fault Diagnosis and Failure Analysis by Mutation Testing

Software failure behavior is affected by two principal factors [Musa87]:

- (1) the execution environment or operational profile of execution, and
- (2) the number of faults in the software being executed.

The Airplane model and the Square Wave model discussed in the previous chapter provide the reference for extensive testing requirements of this project. Moreover, faults were carefully analyzed to determine what faults the six programs contained in order to facilitate their avoidance during development and during testing in revising the design paradigm. Special interest was devoted to the potentiality of similar errors in order to evaluate the effectiveness of the MVS approaches.

Appendix I describes all the faults found in the programs. As has been discussed before, there were two pairs of identical faults in the six versions: one pair was uncovered before or during acceptance test (82 faults found in total), the other

pair was detected after the acceptance test (11 faults found in total). Both pairs of faults were specification related. The first pair was severe, since it reduced a factor by a thousand, but it was easily detected at an early stage. On the contrary, the second pair was not detected until after the acceptance test; however, it was less severe and the airplane *always* obtained proper touchdown in the flight simulation. Moreover, the errors that resulted from each of these faults appeared a little bit different from each other numerically, due to the discrepancies of different programming languages and compilers; i.e., they are not 100% correlated. Here we observe that *error frequency* and *error severity* are two major attributes of an error, and *error similarity* is an important property in describing a pair of errors. These concepts facilitate the following discussion.

To uncover the impact of faults that would have remained in the software version, and to evaluate the effectiveness of MVS mechanisms, a special type of regression testing, similar to *mutation testing* which is well known in the software testing literature [Budd78, Howd82], was investigated in the six versions. Mutation testing, like other testing schemes, applies test data to the program and to a reference of what is expected from the software, and compares the results. However, the testing is done with "mutants" instead of the original software that is being validated. Mutants are programs derived from the original software by the inclusion of some faults. However, there is a major difference in the objective: the original purpose of the mutation testing is to ensure the quality of the test data used to verify a program, while our concern here is to evaluate the similarity of program errors and their treatment by MVS systems.

In order to do this, the mutants created for the testing should be *real* mutants, that is, they contain real programmer errors. Thus we injected all the program faults

(93 in total) back into the versions where they originally occurred. The procedures of this testing is described in the following steps:

Step 1: First, one should obtain the final version of the six programs from the project.

They are executed to provide consensus "reference values" for the testing procedure. Then we can identify and compose the fault removal history of each version according to the audit of faults in Appendix I. These data will be used to derive a representative set of mutants.

Step 2: In this step, one should generate mutants by injecting faults into the final version (from which they were removed) one by one. Thus a representative sample of programs that can be executed for comparisons is created. For simplicity, each mutant is injected with only one fault. The definition of data to be applied to the versions is also done at this step.

Step 3: This step is the main body of the mutation testing. Each mutant is executed by the same set of input data both in the airplane simulation environment and the Square Wave testing environment. Testing results are carefully collected and logged for further analysis.

Step 4: Finally, analysis of the error behavior due to each fault is examined. This includes the definition and the measurement of certain functions (error frequency function, error severity function). Special interest is devoted to the similarity of coincident errors in each program version.

Using the fault removal history of each version, we have created 6 mutants for Ada (a1 - a6), 18 mutants for C (c1 - c18), 5 mutants for Modula-2 (m1 - m5), 12 mutants for Pascal (p1 - p12), 29 mutants for Prolog (pg1 - pg29), and 23 mutants for T (t1 - t23). Each mutant contains *one* fault introduced by its original designers.

In order to present the execution results of the above steps, applied to the Flight Control Computer software in our project, let us define the following two functions for each mutant:

- *Error Frequency Function* (for a given set of test data) – the frequency of the error being triggered by the specified test data set in this mutant.
- *Error Severity Function* (for a given set of test data) – the severity of the error when manifested in the system by the specified test data set.

An Error Frequency Function of version x mutant i for test set τ , denoted as $\lambda(x_i, \tau)$, is computed by

$$\lambda(x_i, \tau) = \frac{\text{total number of errors when executing test set } \tau \text{ on mutant } x_i}{\text{total number of executions}}$$

Since each mutant contains only one known fault, it is hypothesized † that errors produced by that fault are always the same for the same test inputs. Therefore, we can define an Error Severity Function of version x mutant i for test set τ , $\mu(x_i, \tau)$, to be

$$\mu(x_i, \tau) = \begin{cases} 0 & , \text{ if } 0 \leq \left| \frac{\text{reference value} - \text{error of } x_i}{\text{reference value}} \right| \leq \epsilon \\ \left| \frac{\text{reference value} - \text{error of } x_i}{\text{reference value}} \right| & , \text{ if } \epsilon < \left| \frac{\text{reference value} - \text{error of } x_i}{\text{reference value}} \right| < 1 \\ 1 & , \text{ otherwise} \end{cases}$$

where ϵ is a specified allowed deviation.

† This hypothesis is valid for all the mutants in our experiment here.

If x_i produces run-time exceptions or no results, then $\mu(x_i, \tau)$ is defined to be 1.

The Error Frequency Function and Error Severity Function applied to each mutant (for a test set of about 15000 executions) are shown in Table 12 and Table 13, respectively.

As to the nature of errors in two or more channels, three types of relationships are identified: *distinct errors*, *similar errors*, and *identical errors* [Aviz86]. Distinct errors are produced by faults whose erroneous results could be distinguished from one another. Similar errors are defined to be two or more results that are within a small range of variation, and the results are erroneous. If the results of the similar errors are identical, they are called identical errors.

Thus we can define a *Error Similarity Function*, $\sigma(x_1, \dots, x_n)$, for a set of mutants $\{x_1, \dots, x_n\}$ and a test set τ , to be

$$\sigma(x_1, \dots, x_n; \tau) = \begin{cases} 0 & \text{if } (x_1, \dots, x_n) \text{ produce distinct errors in test set } \tau \\ 1 & \text{if } (x_1, \dots, x_n) \text{ produce identical or similar errors in test set } \tau \end{cases}$$

Based on these definitions, we have obtained the Error Similarity Functions for populations of two versions. Table 14 shows the Error Similarity Function matrix for two-mutant sets. The complete layout of this matrix is 93 by 93, but since it is a sparse matrix (most entries are zero), we can reduce it by removing many of the zero

Id	ADA	C	MOD-2	PASCAL	PROLOG	T
1	0.0002	1	0	1	0	1
2	0.001	0.0037	0.001	0	0.0006	1
3	1	0.5	0.005	0	1	1
4	1	~0	0	0.0001	1	1
5	~0	0.0037	0	1	0.0005	1
6	0	0	-	0.002	1	0
7	-	0.001	-	0.001	1	1
8	-	0	-	0.001	0.1	1
9	-	0	-	~0	0.0005	1
10	-	0	-	0	1	0.5
11	-	0.0005	-	0.001	1	1
12	-	0	-	0.0002	1	0
13	-	0	-	-	1	0
14	-	0	-	-	~0	0
15	-	0.03	-	-	1	0.002
16	-	0	-	-	1	0
17	-	0	-	-	1	0.0028
18	-	0	-	-	0	0
19	-	-	-	-	1	0
20	-	-	-	-	1	0
21	-	-	-	-	1	0
22	-	-	-	-	1	0.001
23	-	-	-	-	~0	0
24	-	-	-	-	0	-
25	-	-	-	-	~0	-
26	-	-	-	-	0	-
27	-	-	-	-	0.001	-
28	-	-	-	-	0	-
29	-	-	-	-	0	-

"-" means the mutant for that version does not exist.

Table 12: Error Frequency Function of Each Mutant

Id	ADA	C	MOD-2	PASCAL	PROLOG	T
1	1	0.025	0	1	0	1
2	0.51	1	0.51	0	1	1
3	1	1	1	0	1	1
4	1	1	0	0.03	1	0.017
5	1	0.05	0	1	1	1
6	0	0	-	0.017	1	0
7	-	1	-	1	1	1
8	-	0	-	0.004	0.1	1
9	-	0	-	1	0.038	1
10	-	0	-	0	1	1
11	-	0.001	-	0.23	1	1
12	-	0	-	1	1	0
13	-	0	-	-	1	0
14	-	0	-	-	0.022	0
15	-	0.6	-	-	1	1
16	-	0	-	-	1	0
17	-	0	-	-	1	1
18	-	0	-	-	0	0
19	-	-	-	-	1	0
20	-	-	-	-	1	0
21	-	-	-	-	1	0
22	-	-	-	-	1	0.02
23	-	-	-	-	1	0
24	-	-	-	-	0	-
25	-	-	-	-	1	-
26	-	-	-	-	0	-
27	-	-	-	-	0.02	-
28	-	-	-	-	0	-
29	-	-	-	-	0	-

"-" means the mutant for that version does not exist.

Table 13: Error Severity Function of Each Mutant

entries (see Table 14).

σ	a2	m2	pg27	t22
a2	–	1.0	0.0	0.0
m2	1.0	–	0.0	0.0
pg27	0.0	0.0	–	1.0
t22	0.0	0.0	1.0	–

Table 14: Reduced Error Similarity Function Matrix in Two-Mutant Sets

Analysis of three-mutant sets becomes much more tedious since a three-dimensional matrix will be needed. However, it should be similar to the analysis of two-mutant sets, whose error similarity is shown to be weak. Moreover, since we have not seen any common errors affecting more than two mutants, the results that would be obtained from the analysis of higher-order mutant sets should also be promising in MVS schemes.

5.7 Coverage Measurement of Multi-Version Software

Coverage is an important measurement for the effectiveness of fault-tolerant systems. It is defined as the conditional probability of successful recovery, given that a fault has occurred [Bour69]. In MVS systems, the coverage value depends on the similarity of errors, and the efficiency of the recovery mechanisms to cope with such errors. Obviously, this is an important justification of MVS as a valuable software fault tolerance technique. Thus, we need to derive a definition of coverage and

measure it, using the software produced in this project, and perform a proper analysis.

Since our main interest currently is the analysis of the program versions themselves, without loosing generality, let us assume the supervisory system in supporting MVS executions is outside this analysis. In this case, the main contribution of the "leak" of the MVS schemes, as applied to the FCC software, would be the error similarity defined in the previous section. Furthermore, let us assume a uniform distribution for all the mutants. Consequently, the coverage factor of an n-mutant system (out of a total population m) with respect to the test set τ , denoted $C_n(\tau)$, could be defined as:

$$C_n(\tau) = 1 - \frac{1}{C(m,n)} \sum_{1 \leq x_1 \leq \dots \leq x_n \leq m} \sigma(x_1, \dots, x_n; \tau) * \mu(\text{median_of}(x_1, \dots, x_n), \tau)$$

For example, the coverage factor $C_2(\tau)$ of a two-version system from our sample mutants is:

$$\begin{aligned} C_2(\tau) &= 1 - \frac{1}{C(93,2)} \sum_{x_2=x_1+1}^{x_2=93} \sum_{x_1=1}^{x_1=92} \sigma(x_1, x_2) * \mu(x_1) \dagger \\ &= 1 - 0.000234 (1*0.51 + 1*0.02) \\ &= 0.99988 \end{aligned}$$

Other C_n 's could be computed similarly. Such a coverage factor can serve to validate previous software reliability models [Cost78, Litt80, Rama82, Lapr84]. Coverage can also refer to a representative measure of the situations to which the system is submitted during its validation with respect to the actual situations it will be confronted with during its operational life. It is important to note that coverage defined and measured in our experiment is limited to the particular mutant population and the specific test data set. Nevertheless, it might be useful to provide an evidence of the effectiveness of MVS methodology for the assigned application.

5.8 Evaluation of Fault Tolerance Provisions in the Applications

It is interesting to compare the selected application for the two large scale fault-tolerant software experiments [Kell88, Aviz88b] to see how fault tolerance attributes were introduced in the application, and how these attributes could be explored by our design paradigm. The application of the Flight Control Computer (FCC) software was shown in Figure 4-1. The application of the Nasa/Four-University Experiment, Redundant Strapped Down Inertial Measurement Unit (RSDIMU), is show in Figure 5-1. From these two applications, we can observe the following characteristics:

(1) System Partitioning

Both systems employed modular designs, and each module was self-contained

† $\mu(x_1)$ is picked as the median (for $n = 2$).

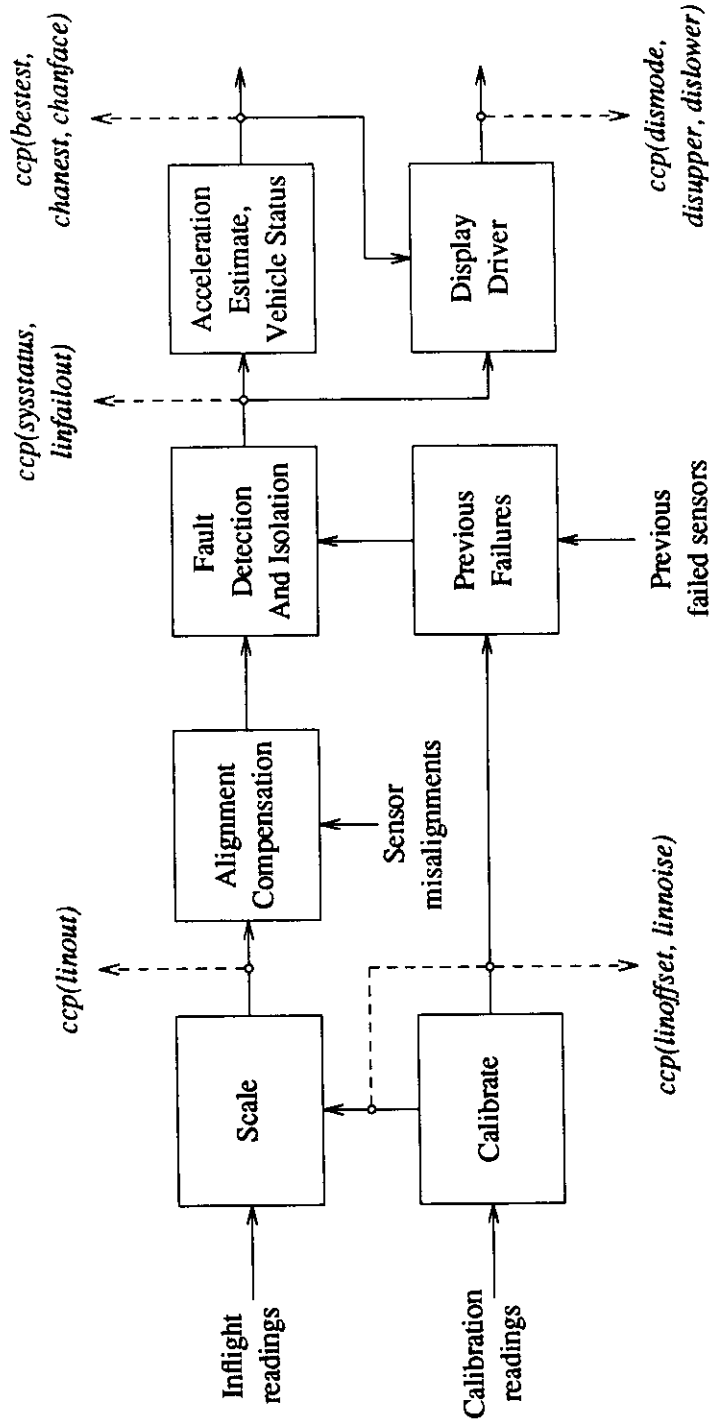


Figure 5-1: System Data Flow Diagram of RSDIMU

and relatively independent. Partition of systems naturally facilitates the placement of error detection and recovery mechanisms. This characteristic helps to establish isolation requirements and to select isolation mechanisms against propagation of errors.

(2) Self Acceptance Test

In the application of the FCC software, magnitude limiters and rate limiters were required to limit the output and output changes of the landing command, in order to prevent unbounded values or any dramatic changes. These limiters were designed not only from the physical requirements (e.g., pitch degrees must be between +/- 15 degrees), but also from certain safety considerations. This introduces the possibility of incorporating the *acceptance test* idea of the Recovery Block approach [Rand75] into MVS systems, and thus each version could disqualify itself in order to ensure early treatment of errors [Kell82].

(3) Internal Redundancies

Internal redundancies are very important if only one software version is relied on to produce results. In the application of RSDIMU, eight linear accelerometers were used for the inertial measurement to decide the movement vector of an airplane, while only three accelerometers would suffice to make the required computation. Redundancies are used for fault detection, error isolation, and more precise computation. The MVS approach combined with such features provides *multi-level redundancies* in protecting highly-critical software systems.

(4) **System Monitoring**

System monitoring is usually achieved by using independent subsystems. In single-version software environment, a monitoring subsystem could be very complicated. In MVS environment, however, such a subsystem could easily be designed and provided. In RSDIMU, while used in a single-version environment, the monitoring function (which declared the error of one accelerometer, or the failure of the whole system) was provided by using the complicate redundant information of the accelerometers. In FCC, designed as a 3-channel system, the monitoring function was merely a simple combinatorial logic circuit, and it could detect a single failure twice. These applications indicate that the existence of an MVS facilitates the requirement of system monitoring.

5.9 A Comparison of Three Recent Experiments

Tables 15, 16, and 17 compare among three recent MVS experiments, including the Knight and Leveson study (experiment A) [Knig86], NASA investigation (experiment B) [Kell88], and Honeywell/UCLA project (experiment C) [Aviz88b]. We focus our attention on the methodology of developing MVS each of these experiments has applied, since it displays the rigor of MVS development and execution that each experiment employed.

category	experiment A	experiment B	experiment C
specification size	6 pages	64 pages	64 pages
lines of code	327-1004	1600-4800	1253-2234
# of programs	27	20	6
duration	one Quarter	one Summer (10 weeks)	one Summer (12 weeks)
members per team	1	2	2
programmers' reward	class grade	full RA pay	full RA pay

Table 15: A Comparison of Three Experiments – The Scale

From Table 15 we can see that experiment A was of rather small scale, since the specification was 6 pages long and could be programmed in 327 lines of code. The scale of both experiments B and C, with 64 pages of specification and at least over one thousand lines of code, were significantly larger. experiment A was a class project during one Quarter term. Motivated by wanting to pass the class, each student carried out his program alone. On the contrary, programmers in experiments B and C worked in two-member teams and were paid a full-time research assistant (RA) salary during a class-free period (in summer).

Regarding the software development concerns (see Table 16), there was no industrial involvement (except description of the application) in experiment A. There were infrequent contacts with industrial consultants for the application in experiment B, while there were frequent technical meetings and communications with Honeywell in experiment C. The diversity required in experiment A and B was programmers' locality. Team separation was enforced in experiment C and the extra diversity investigated in experiment C was several high order programming languages. The team coordination, team synchronization, and software engineering practices were not

category	experiment A	experiment B	experiment C
industrial involvement	none	weak	strong
# of involved academia	2	4	1
programming languages	Pascal	Pascal	Ada, C, Modula-2 Pascal, Prolog, T
team coordination	none	weak	strong
team synchronization	none	loosely controlled	tightly controlled
software engineering practices	none	incomplete	complete
team separation effort	no	yes	yes
communication protocol	poorly defined	well defined, poorly executed	well defined, well executed

Table 16: A Comparison – MVS Software Development Procedure

employed in experiment A. They were of concern but caused difficulties in experiment B, and were systematically and successfully treated in experiment C. There was *no* communication protocol for the programmers in experiment A, and programmer separation was not emphasized. The communication protocols for experiment B and C were well defined, but it was inadequately executed in experiment B.

For the testing and processing of the MVS systems (shown in Table 17), the failure detection granularity for experiment A was coarse since it used *one* Boolean variable to represent 241 Boolean conditions for a "missile launching decision." Experiments B and C both incorporated real number comparisons for inexact matching with specified tolerances. Error detection and recovery mechanisms were *missing* in experiment A, while the cross-check points and recovery points for experiments B and C were specified. The recovery point in experiment B was injected after the software generation, however. Both experiment A and B did not apply high

category	experiment A	experiment B	experiment C
testing granularity	coarse	fine	fine
error detection	no	yes	yes
error recovery	none	post-injected	pre-designed
testing schemes	acceptance test only	acceptance test only	phased test: unit, integration, and acceptance test
# of test cases before acceptance test	15 runs	4 runs	2290 runs
# of test cases for acceptance test	200 runs	75 runs	18440 runs
testing discipline	one-at-a-time, random executions	one-at-a-time, random executions	open loop execution + closed loop flight simulation
required deliverables	only one	several, poorly recorded	several, well recorded
project delays	no	yes	no

Table 17: A Comparison – Testing and Processing of MVS Systems

testing before program acceptance, whereas experiment C employed three testing phases by different testing criteria (open loop testing, close loop flight simulation) with a reasonable amount of meaningful test data. There was only one required deliverable in experiment A: the final code. There were several required deliverables in experiments B and C, but they were inadequately recorded in experiment B. Lastly, both experiments A and C were continuously conducted and completed on time; however, the original acceptance test of experiment B was judged inadequate during its software generation, which caused overhead in its software certification and delayed the project.

The comparison of the results in experiment A and experiment C (the final results of experiment B are unavailable due to the delays in certification) also shows major disagreements. Unlike experiment C [Aviz88b], experiment A [Knig86] found a number of faults in the program versions after their acceptance, and according to their definition of "coincident errors," a significant number of such errors were identified. We have reviewed experiment A and find reasons that may account for this outcome: the small scale of that experiment (Table 15), lack of MVS software development disciplines (Table 16), and inappropriate as well as inadequate testing and processing of MVS systems (Table 17). It is our conjecture that a rigorous application of the design paradigm, as described in this thesis, would have led to the elimination of most faults described in experiment A [Knig86] before acceptance of the programs.

5.10 Refinements of the Design Paradigm

In the UCLA/Honeywell project, thus far we have found only two pairs of identical faults that caused similar errors. Both pairs of faults were caused by readily avoidable procedural deficiencies. In the first case, a handwritten comma was misread as a period because of excessive reduction of the size of a diagram. In the other case, two teams failed to respond properly to a broadcast clarification of an ambiguity in the specification. This pair of faults could be avoided by requiring a positive acknowledgment that the clarification had been understood and accounted for. As a consequence, several important points have been observed in refining the design paradigm:

- 1) *Educating and requiring programmers to obey MVS design rules*

As has been pointed out earlier, certain peculiar mistakes have been committed by the programmers due to disobeying the MVS design rules. The *Type 3* structural faults are due to disregard of clearly stated multi-version software design rules. They are potentially identical and therefore dangerous. This is also true of the *Type 2 underground variable* fault (see Section 5.3.2).

2) *Designing special cases for recovery*

To guarantee that recovery routines function properly, special test cases could be designed beforehand to certify their effectiveness. In any case, a strict verification process must be a part of the acceptance test to ensure that MVS design rules are followed.

3) *Acknowledging each broadcast message and verifying each update message*

Ignorance of a specification update message is a potential commonality for related faults, thus should be carefully prevented. Such incidence could be caused by overlooking the message, or even loss of a broadcast message. Under this circumstance, each broadcast message during the software development phase should have acknowledgement from each programming team, and each specification update message should be confirmed by a mandatory code update report for the corresponding changes in the code. Special mechanisms, e.g., program testing or code inspection, could also be applied to verify the completion of the update message.

4) *Prototyping the system prior to the software generation phase*

System prototyping has been accepted as an important approach in evaluating a large software system before its deployment. Many insights can be gained when a compact prototype system is built and exercised. Additionally, the main purposes of a fast prototyping of the system are to clean up the

specifications, and to clarify how the recovery mechanisms are to work. Both specifications and recovery mechanisms have been proved to be key issues for the effectiveness of MVS systems.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Original Contributions of This Research

In summary, the major contributions of this research include the following activities:

1. An MVS design paradigm has been formulated, applied, and evaluated. Its practicality and effectiveness are elaborated in the following sections.
2. The design paradigm has been used during the entire life-cycle of the UCLA/Honeywell experiment. This experiment served to execute and evaluate the proposed paradigm. With some refinements, the design paradigm has been demonstrated to be effective in MVS development.
3. The experiment has been the first to apply six high-level programming languages to a real-world critical software application. The use of different programming languages has supported effective inter-team isolation, since different support environments were used. It has also promoted the appearance of diversity in versions that began with a common specification.

6.2 Practicality of the Proposed Design Paradigm

The proposed design paradigm for MVS design and development, which integrates software engineering discipline and fault tolerance system design paradigm, is simple and efficient. It explores and supports design diversity to its full extent, and prevents commonalities that could produce related software faults. This design paradigm has been successfully applied in conducting the UCLA/Honeywell experiment. Overall, the UCLA paradigm for systematic generation of multi-version software is judged to be sufficiently complete and stable for application to industrial environment.

6.3 Effectiveness of the Design Paradigm

The effectiveness of the design paradigm is shown by the experimental result that identical faults in two versions have occurred very rarely. In the experiment, only one identical pair existed in the 82 faults removed from the six versions before acceptance, and it was due to a comma being misread as a period. During post-acceptance testing and inspection, eleven faults were uncovered. One pair again was identical, and this fault was due to failure to properly incorporate a clarification to a specification ambiguity. *Identical faults involving more than two versions have never been observed.* This is very different from previously published results by Knight and Leveson [Knig86]. Upon reviewing that reference, we conclude that there are several significant differences: the previous problem had limited potential for diversity, the programming process was rather informally formulated, testing was limited, and the acceptance test was totally inadequate according to industrial standards that we have followed. It is clear that the design paradigm would be efficient to eliminate the design defects in that experiment.

6.4 Future Research Issues in MVS

Several research issues have been brought to consideration for the MVS technique in achieving efficient fault-tolerant software. These topics are discussed one by one in the following sub-sections.

6.4.1 Identify and Avoid Commonalities

Related design faults existing in the major population of the software versions are the Achilles heels of the MVS techniques. To rule out these faults, all possible commonalities that have the potential to create them should be identified and avoided. Identification and prevention of the common links of design faults is very important, since programmers should have few chances to make identical design faults if no design commonalities exist. Moreover, observable dissimilarities among program versions could serve as reasonable evidence for the certification of a software product, especially when the justification of its reliability requirement is *impossible* to demonstrate experimentally (e.g., 10^{-9} failure rate).

6.4.2 Measure and Promote Design Diversity

In the UCLA/Honeywell project, we observed that the order of computations that was implied by the specification had a strong influence on the programmers' choice, even if other alternatives existed. This is especially true of graphical specifications used in this effort. *Test points* given in the specification also tend to limit diversity. There is a need to develop effective means to minimize these diversity-limiting factors.

Improving design diversity from other dimensions should also be considered. A tentative classification of possible diversification means is in Figure 6-1. Design diversity could be achieved either by *randomness* or by *enforcement*. The "random" diversity, such as that provided by independent personnel, leaves the dissimilarity to be generated in an uncontrolled manner. The enforced diversity, on the other hand, investigates different aspects in several dimensions, and deliberately requires them to be implemented into different program versions.

In any case, quantitative evaluations of design diversity have proven to be very important. Investigation of branch coverage testing [Adri82, Prat83, Swai86] might be an intriguing approach in demonstrating the assurance of diversity, and giving guidance for the enforcement of this attribute in each program.

6.4.3 Developing Support Tools and Techniques

Different support tools and techniques could be investigated for the MVS development. They include:

- **Specification**

Although writing of the original English specification was carefully conducted by the UCLA coordinating team and H/S avionics and software specialists, it was found to contain errors and ambiguities. The original specification was improved, according to the feedback from the programmers and preliminary testing results obtained in the validation phase. This indicates that English specifications are inherently ambiguous and error-prone.

Hardware

- Different microprocessors
- One more microprocessor in the control channel
- Two different types of computer

Project organization

- Different Software Design Teams
(One in Los Angeles, one in Toulouse, and one in Tokyo)
- Two test sets designed by two different teams
- Different optimization goals: Timing performance vs program size
- Validation oriented vs. testing oriented

Inherent differences

- Hardware differences
- Some functions are required in only one channel
- Different input
- Different necessary precision
 - 12 bits for the control
 - 8 bits for the monitoring
- Function in the control channel, inverse function in the monitoring channel

Forced differences

- Different languages
 - PASCAL vs ASSEMBLER
 - PLM vs ASSEMBLER
 - Division of the instruction into two subsets
- Different automatic programming tools
- Different software specifications
- Different algorithms
- Different flowcharts
- A function can be tabulated or calculated
- Interrupts allowed in only one channel
- Trigonometric functions (polar coordinate vs Cartesian)

Figure 6-1: Classification of Design Diversity

Specifications need to be as complete as possible; however, providing too much information may tend to limit diversity. In order to encourage design diversity, concentrated effort should be devoted to reduce the specifications as much as possible to address "what should be done" and remove "how it is done." To provide accurate problem specifications, formal specification methods [Gogu79, Gutt83, Kemm85, Zave86], especially executable specifications [Berl87], should play an important role in the future development of MVS systems.

- Facilitate software development tools

A full complement of development tools was used (e.g. csh script, vi and emacs editors, makefile facility, rcs revision control, etc.) in the project. The following tools could be further investigated:

Branch coverage testing tool – to provide branch coverage analysis of each program and adjust the quality of the test data [Swai86].

Statistics collector – to gather the pertinent data in order to make adaptations to the testing process.

Software reliability measurer – to monitor the operational performance of software and to control new features added and design changes made to the software.

Diversity measurer – to reveal diversification of different pieces of software during program generation. This is considered a very difficult tool to build.

- Supervisors

DEDIX was designed to supervise and observe the execution of multiple diverse versions functioning as a fault-tolerant MVS system [Aviz88c]. However, the system (implemented in the C programming language) does not support the execution of programming languages other than C and Pascal. Modification of DEDIX was necessary in order to provide the interface for object oriented languages (e.g., Ada, Modula-2). Moreover, some fundamental problems would have to be resolved in order for DEDIX to support the execution of applicative languages (e.g., Prolog, Lisp) running on interpreters.

6.4.4 Exploiting the Presence of Multiple Versions for V&V

In both the NASA/Four-University Experiment and the UCLA/Honeywell Experiment, the majority results of existing multiple versions are conjectured to be more reliable than those of a "gold" model or version (usually provided by an application expert). This is true during most of the later stages in the software life cycle, including the testing phase and the operational phase. Running back-to-back comparison and cross-checking of MVS system is a simple testing configuration; moreover, the criteria for correct results are easily determined by applying a tight skew for allowable numerical discrepancy. Therefore, the V&V process, a process that is usually very complex and costly, would be much easier to achieve by using the existence of multiple versions. This argument has been recently supported in [Knig88].

Another strong reason of using MVS for verification and validation is to provide protective redundancy around specification misinterpretations and ambiguities. Any single version approach gets a single interpretation of the specification, no matter how carefully a development procedure is followed. Especially when the software development group is small, everyone can share a misunderstanding. MVS approaches force the specifications to be assessed from independent observations and viewpoints, and any inconsistency and incorrectness between the specifications and the V&V process should be uncovered more effectively and thoroughly.

6.4.5 Cost-Effectiveness Measurement and Assessment

Another dimension of evaluation of a system is in cost investigations. The generation of multiple versions of a given program instead of a single version results in an immediate increase in the cost of software prior to the verification and validation phase. The question is whether the subsequent cost can be reduced because of the ability to employ two or more versions to attain mutual validation under testing and operational conditions. Cost advantages may accrue because of 1) the faster operational deployment of new software; and 2) replacement of costly verification and validation tools and operations by a generic MVS environment in which the versions validate each other. However, cost of modifications and maintenances of MVS systems is also significant.

In summary, successful completion of the design of an MVS system requires a convincing verification of two properties: the completeness of the design and its potential to meet the stated goals of availability, reliability, maintainability, and safety. The verification therefore must consist of the application of two distinct

evaluations: first *qualitative*, then *quantitative*. The qualitative evaluation requires realistic assumptions to prevent unjustified simplifications, while quantitative evaluation requires system reliability models. Both these two factors need to be formally established and validated.

6.4.6 Incorporating Security Concerns into the Design Paradigm

Several issues involving fault tolerance and computer security have been recently addressed [Turn86]. These include the methods to make the security mechanisms employed provide proper service in the presence of faults, and the impact on security when complex fault tolerance mechanisms are added to a computing system. Moreover, faults deliberately designed and hidden in the complexity of the system's software or hardware, with the intent to cause loss or harm to the system, (e.g., *malicious logic*: Trojan horses, trap doors, and computer viruses) are an important issue. The introduction of MVS techniques as an approach in treating malicious logic has been explored in [Jose88]. We believe that computer security considerations could be incorporated into our design paradigm for a unified and general treatment of MVS systems that provide both fault tolerance and security attributes.

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.
- [Ande85] T. Anderson, P.A. Barrett, D.N. Halliwell, and M.R. Moulding, "Software Fault Tolerance: An Evaluation," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, December 1985, pp. 1502-1510.
- [Ande81] H. Andersson and G. Hagelin, "Computer Controlled Interlocking System," Ericsson Signal Systems, Stockholm, Sweden, Tech. Rep. 2, 1981.
- [Arla79] J. Arlat, *Design of a Microcomputer Tolerating Faults Through Functional Diversity*, Toulouse, France: National Polytechnic Institute, 1979.
- [Aviz67] A. Avižienis, "Design of Fault-Tolerant Computers," *Proceedings AFIPS Conference, 1967 Fall Joint Computer Conference*, 1967, pp. 733-743.
- [Aviz72] A. Avižienis, "The Methodology of Fault-Tolerant Computing," in *Proceedings The 1st USA-Japan Computer Conference*, Tokyo, Japan: October 1972, pp. 405-413.
- [Aviz75] A. Avižienis, "Fault-Tolerance and Fault-Intolerance: Complementary Approaches to Reliable Computing," *Proceedings 1975 International Conference on Reliable Software*, April 21-23, 1975, pp. 450-464.
- [Aviz77] A. Avižienis and L. Chen, "On the Implementation of N-Version Programming for Software Fault-Tolerance during Program Execution," in *Proceedings COMPSAC 77*, 1977, pp. 149-155.
- [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," *Digest of 12th Annual International Symposium on Fault-Tolerant Computing*, 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," *Digest of 15th Annual International Symposium on Fault-Tolerant Computing*, June 1985, pp. 126-134.
- [Aviz85b] 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.
- [Aviz85c] 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," *Proceedings IFAC Workshop SAFECOMP'85*, October 1985, pp. 173-178.
- [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.
- [Aviz87a] A. Avižienis, "A Design Paradigm for Fault-Tolerant Systems," *Proceedings AIAA Computers in Aerospace VI Conference*, October 1987, pp. 52-57.
- [Aviz87b] A. Avižienis, M. R. Lyu, and W. Schütz, "In Search of Effective Diversity: A Six-Language Study of Fault-Tolerant Flight Control Software," UCLA Computer Science Department, Los Angeles, California, Tech. Rep. CSD-870060, November 1987.
- [Aviz87c] A. Avižienis, M. R. Lyu, W. Schütz, K. Tso, and U. Voges, "DEDIX 87 - A Supervisory System for Design Diversity Experiments at UCLA," Computer Science Department, UCLA, Los Angeles, California, Tech. Rep. CSD-870029, July 1987.
- [Aviz88a] A. Avižienis, M. R. Lyu, and W. Schütz, "Multi-Version Software Development: A UCLA/Honeywell Joint Project for Fault-Tolerant Flight Control Software," Los Angeles, California, Tech. Rep. CSD-880034, May 1988.

- [Cain75] S.H. Caine and E.K. Gordon, "PDL - A Tool for Software Design," *Proceedings National Computer Conference*, 1975.
- [Chen78a] L. Chen and A. Avižienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," *Digest of 8th Annual International Symposium on Fault-Tolerant Computing*, June 1978, pp. 3-9.
- [Chen78b] 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.
- [Corp85] S. G. Corps, "A320 Flight Controls," in *Proceedings The 29th Symposium of the Society of Experimental Test Pilots*, September 1985.
- [Cost78] A. Costes, C. Landrault, and J.C. Laprie, "Reliability and Availability Models for Maintained Systems Featuring Hardware Failures and Design Faults," *IEEE Transactions on Computers*, Vol. C-27, No. 6, June 1978, pp. 548-560.
- [Cost81] A. Costes, J.E. Doucet, C. Landrault, and J.C. Laprie, "SURF: A Program for Dependability Evaluation of Complex Fault-Tolerant Computing Systems," in *Digest of 11th Annual International Symposium on Fault-Tolerant Computing*, Portland, Maine: June 1981, pp. 72-78.
- [Duga81] J. B. Dugan and D. J. Martin, "Dissimilar Redundancy for Fly-by-wire Secondary Flight Controls," in *Proceedings Advanced Flight Controls Symposium*, Colorado Springs, Colorado: 1981.
- [Duga86] J. B. Dugan, K. S. Trivedi, M. K. Smotherman, and R. M. Geist, "The Hybrid Automated Reliability Predictor," in *AIAA Journal of Guidance, Control and Dynamics*, May-June 1986, pp. 319-331.
- [Dyer88] M. Dyer, "Certifying the Reliability of Software," in *Proceedings Annual National Joint Conference on Software Quality and Reliability*, Arlington, Virginia: March 1-3 1988.
- [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.
- [Goya86] A. Goyal, W. C. Carter, E. de Souza e Silva, S. S. Lavenberg, and K. S. Trivedi, "The System Availability Estimator," *Proceedings 16th Annual International Symposium on Fault Tolerant Computing*, July 1-3, 1986.
- [Gutt83] J.V. Guttag and J.J. Horning, "An Introduction to the Larch Shared Language," in *Proceedings IFIP Congress 83*, 1983, pp. 809-814.
- [Hage88] G. Hagelin, "ERICSSON Safety System for Railway Control," in *Software Diversity in Computerized Control Systems*, U. Voges, Ed. Austria: Springer-Verlag/Wien, 1988, pp. 11-21.
- [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.
- [Hill85] A.D. Hills, "Digital Fly-By-Wire Experience," *Proceedings AGARD Lecture Series*, No. 143, October 1985.
- [Howd82] W.E. Howden, "Weak Mutation Testing and Completeness of Test Sets," *IEEE Transactions on Software Engineering*, Vol. SE8, No. 4, July 1982, pp. 371-379.
- [Jose88] M. K. Joseph, "Architectural Issues in Fault-Tolerant, Secure Computing Systems," Ph. D. Dissertation, UCLA Computer Science Department, Los Angeles, California, May 1988.
- [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," *Digest of 13th Annual International Symposium on Fault-Tolerant Computing*, June 1983, pp. 121-126.

- [Kell86] 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," *Proceedings IFAC Workshop SAFECOMP'86*, October 1986, pp. 43-49.
- [Kell88] J. P. J. Kelly, D. E. Eckhardt, A. Caglavan, J. C. Knight, D. F. McAllister, and M. A. Vouk, "A Large Scale Second Generation Experiment in Multi-Version Software: Description and Early Results," *Proceedings The Eighteenth International Symposium on Fault-Tolerant Computing*, June 27-30, 1988.
- [Kemmm85] R. A. Kemmerer, "Testing Formal Specifications to Detect Design Errors," *IEEE Transactions on Software Engineering*, Vol. SE-11, January 1985, pp. 32-43.
- [Knig86] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 1, January 1986, pp. 96-109.
- [Knig88] J. C. Knight, "Using Multiple Version for Verification," in *Proceedings Annual National Joint Conference on Software Quality and Reliability*, Arlington, Virginia: March 1988.
- [Lapr84] J.-C. Laprie, "Dependability Evaluation of Software Systems in Operation," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 6, November 1984, pp. 701-714.
- [Li87] H. F. Li and W. K. Cheung, "An Empirical Study of Software Metrics," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 6, June 1987, pp. 697-708.
- [Litt80] B. Littlewood, "Theories of Software Reliability: How Good Are They and How Can They Be Improved?," *IEEE Transactions on Software Engineering*, September 1980, pp. 489-500.
- [Maka82] S.V. Makam and A. Avižienis, "ARIES 81: A Reliability and Life-Cycle Evaluation Tool for Fault-Tolerant Systems," *Digest of 12th Annual International Symposium on Fault-Tolerant Computing*, June 1982, pp. 267-274.
- [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.

- [Musa87] J. D. Musa, A. Iannino, and K. Okumoto, *Software Reliability – Measurement, Prediction, Application*, New York, New York: McGraw-Hill Book Company, 1987.
- [Prat83] R. E. Prather, “Theory of Program Testing – An Overview,” *The Bell System Technical Journal*, Vol. 62, No. 10, Part 2, December 1983, pp. 3073-3105.
- [Rama81] C.V. Ramamoorthy, Y. Mok, F. Bastani, G. Chin, and K. Suzuki, “Application of a Methodology for the Development and Validation of Reliable Process Control Software,” *IEEE Transactions on Software Engineering*, Vol. SE-7, No. 6, November 1981, pp. 537-555.
- [Rama82] C.V. Ramamoorthy and F.B. Bastani, “Software Reliability - Status and Perspective,” *IEEE Transactions on Software Engineering*, Vol. SE8, No. 4, July 1982, pp. 334-370.
- [Rand75] B. Randell, “System Structure for Software Fault Tolerance,” *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, June 1975, pp. 220-232.
- [Renn84] D.A. Rennels, “Fault-Tolerant Computing -- Concepts and Examples,” *IEEE Transactions on Computers*, Vol. C-33, No. 12, December 1984, pp. 1116-1129.
- [Rouq86] J. C. Rouquet and P. J. Traverse, “Safe and Reliable Computing on Board the Airbus and ATR Aircraft,” *Proceedings SAFECOMP '86*, October 1986, pp. 93-97.
- [RTCA85] RTCA, Radio Technical Commission for Aeronautics, “Software Considerations in Airborne Systems and Equipment Certification,” Washington, D.C., Tech. Rep. DO-178A, March 1985. Order from: RTCA Secretariat, One McPherson Square, 1425 K Street, N.W., Suite 500, Washington, DC 20005.
- [Sahn86] R. A. Sahnner and K. S. Trivedi, “A Hierarchical, Combinatorial-Markov Method for Solving Complex Reliability Models,” in *Proceedings ACM/IEEE Fall Joint Computer Conference*, Dallas Texas: November 1986.
- [Schu87] W. Schutz, “Diversity in N-Version Software: An Analysis of Six Programs,” Master Thesis, UCLA Computer Science Department, Los Angeles, California, November 1987.

- [Sedm78] R. M. Sedmak and H. L. Liebergot, "Fault-Tolerance of a General Purpose Computer Implemented by Very Large Scale Integration," *Proceedings 8th International Symposium on Fault-Tolerant Computing*, June 1978, pp. 137-143.
- [Siew84] D.P. Siewiorek, "Architecture of Fault-Tolerant Computers," *Computer*, Vol. 17, No. 8, August 1984, pp. 9-18.
- [Stif79] J. J. Stiffler, "CARE III Final Report, Phase I, Vol. I and II," Raytheon Co., California, November 1979.
- [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.
- [Tami84] Y. Tamir and C. H. Sequin, "Reducing Common Mode Failures in Duplicate Modules," in *Proceedings International Conference on Computer Design: VLSI in Computers ICCD'84*, Port Chester, New York: October 1984, pp. 302-307.
- [Tayl81] R. Taylor, "Redundant Programming in Europe," *ACM SIGSOFT Software Engineering Notes*, Vol. 6, No. 1, January 1981.
- [Trav88] P. Traverse, "AIRBUS and ATR System Architecture and Specification," in *Software Diversity in Computerized Control Systems*, U. Voges, Ed. Austria: Springer-Verlag/Wien, 1988, pp. 95-104.
- [Trea82] J. J. Treacy, "Certification of Digital Avionics: A Review of Recent FAA Experience," in *Aerospace Congress and Exposition*, Anaheim, California: October 1982, pp. 3-7.
- [Tso87a] K.S. Tso, "Error Recovery in Multi-Version Software," Ph.D. Dissertation, UCLA Computer Science Department, Los Angeles, California, Tech. Rep. CSD-870013, March, 1987.
- [Tso87b] K.S. Tso and A. Avižienis, "Community Error Recovery in N-Version Software: A Design Study with Experimentation," *Digest of 17th Annual International Symposium on Fault-Tolerant Computing*, July 1987, pp. 127-133.

- [Turn86] R. Turn and J. Habibi, "On the Interactions of Security and Fault Tolerance," *Proceedings 9th National Computer Security Conference*, September 1986, pp. 138-142.
- [Voge85] U. Voges, "Application of a Fault-Tolerant Microprocessor-Based Core-Surveillance System in a German Fast Breeder Reactor," *EPRI-Conference*, April 9-12 1985.
- [Voge88] U. Voges, "Use of Diversity in Experimental Reactor Safety Systems," in *Software Diversity in Computerized Control Systems*, U. Voges, Ed. Austria: Springer-Verlag/Wien, 1988, pp. 29-49.
- [Will83] J. F. Williams, L. J. Yount, and J. B. Flannigan, "Advanced Autopilot Flight Director System Computer Architecture for Boeing 737-300 Aircraft," in *Proceedings Fifth Digital Avionics Systems Conference*, Seattle, WA: November 1983.
- [Wrig86] N. C. J. Wright, "Dissimilar Software," in *Proceedings Design Diversity in Action Workshop*, Baden, Austria: June 1986.
- [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.
- [Youn85a] L. J. Yount, K. A. Liebel, and B. H. Hill, "Fault Effect Protection and Partitioning for Fly-by-Wire and Fly-by-Light Avionics Systems," in *Proceedings AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference*, Long Beach, California: October 1985, pp. 275-284.
- [Youn85b] L. J. Yount, "Generic Fault-Tolerance Techniques for Critical Avionics Systems," in *Proceedings AIAA Guidance and Control Conference*, Snowmass, Colorado: June 1985.
- [Youn86] L. J. Yount, "Use of Diversity in Boeing Airplanes," in *Proceedings Workshop of Design Diversity in Action*, Baden, Austria: June 27-28 1986.
- [Zave86] P. Zave and W. Schell, "Salient Features of an Executable Specification Language and Its Environment," *IEEE Transaction on Software Engineering*, Vol. SE-12, No. 2, February 1986, pp. 312-325.

[Zieg84]

B. Ziegler and M. Durandeu, "Flight Control System on Modern Civil Aircraft," in *Proceedings International Council of the Aeronautical Sciences (ICAS'84)*, Toulouse, France: September 1984.

APPENDIX I

Summary of All Faults Found during Testing and Evaluation

With the help of the change documentation submitted by each team at the end of each test phase (Coding and Unit Test, Integration Test, and Acceptance Test), the following summary of faults was produced. Changes that were not made due to a fault are ignored.

Starting with a particular id, each fault is described, the location and the type of the fault is given (e.g. typo, omission, unnecessary, incorrect algorithm, specification misinterpretation/ambiguity etc.), and the method of detection is mentioned, followed by special remarks, if any. Reference to the corresponding Design Walkthrough Report(s) (DWR) and/or Code Update Report(s) (CUR) is given. Finally, a classification of the fault into requirements or structural fault (unintended function) is attempted.

I.1 ADA Version

I.1.1 Faults detected during Coding and Unit Test

- a1. Some unnecessary statements in the initialization of the Inner Loop (CUR #1)

Type: unnecessary. Detected by: reading the code

Classification: structural fault

- a2. Wrong constant (wrong place of the decimal point) for the upper limit in the declaration of an integrator in the specification part of the Complementary Filter Module (CUR #3)

Type: typo / *spec misinterpretation* / due to poor readability. Detected by: test data data.2, data.13

Classification: requirements fault

The other changes were due to floating point representation problems (CUR #2), compiler peculiarities (CUR #4), and a specification update (CUR #5).

I.1.2 Faults detected during Integration Test

- a3. There were some unnecessary local variables in the main program which interfered with global (imported) variables (CUR #6).

Type: incorrect algorithm. Detected by: test data data.1

Classification: requirements fault

- a4. Two initialization statements were omitted when initializing the Glide Slope Complementary Filter (CUR #8).

Type: incorrect algorithm (due to lack of communication with team mate).

Detected by: test data data.2

Classification: requirements fault

The other changes were due to a specification update (CUR #7) and to adjust to the C interface (CUR #9).

I.1.3 Faults detected during Acceptance Test

a5. In the outer loop of the Flare Control Law, some test points which are not used in Flare Mode were not reset to the specified default value (CUR #10).

Type: omission. Detected by: test data data.2

Classification: requirements fault

a6. A new version of the Mode Logic was written because a numeric error exception resulted from execution (CUR #11).

Type: incorrect algorithm. Detected by: test data data.2

Classification: requirements fault

I.1.4 Faults detected after Acceptance Test

No faults were detected.

I.2 C Version

I.2.1 Faults detected during Coding and Unit Test

c1. A wrong constant was used in the Mode Logic (CUR #4).

Type: specification misinterpretation (due to poor readability of Fig. 4.2).

Detected by: reading the code

Classification: requirements fault

c2. OR-gates in Mode Logic were implemented wrong (CUR #5).

Type: incorrect algorithm. Detected by: team's own test data

Classification: requirements fault

c3. In the Inner Loop, another state variable for the determination of the condition for switch SW2 was added (CUR #9).

Type: incorrect algorithm. Detected by: test data data.18.

Classification: requirements fault

Note: As it was detected during operational testing, this change resulted in an incorrect algorithm.

c4. Part of the Outer Loop of the Flare Control Law was not implemented correctly ($\frac{200}{VA}$ was used instead of $\frac{VA}{200}$) (CUR #10).

Type: incorrect algorithm. Detected by: reading the code and the coordinators' test data.

Classification: requirements fault

The other changes were made in order to conform to the C interface (CUR #1), to obtain a clear structure of the program (CUR #2, 3), or were due to a specification update (CUR #6, 7, 8).

1.2.2 Faults detected during Integration Test

c5. In the Outer Loop of the Glide Slope Control Law, a comparison statement of "time >= 0.5" had to be changed to "time > 0.45", due to the lack of infinite

precision in real number representation (CUR #11).

Type: due to peculiarities of real number representation. Detected by: test data data.1

Classification: structural fault of the C compiler

- c6. In the Inner Loop, initialization of the integrator I1 was not done upon the transition from Altitude Hold to Glide Slope Track Mode (CUR #12).

Type: incorrect algorithm (due to specification ambiguity). Detected by: test data data.2

Classification: requirements fault

- c7. In the Inner Loop, the state of the rate limiters LR1 and LR2 was not saved correctly (CUR #13, 14).

Type: incorrect algorithm (count as only one fault here) Detected by: test data data.2

Classification: requirements fault

- c8. In the main program, the test points of the Outer Loops were not properly reset to the specified default value (CUR #15).

Type: omission. Detected by: test data data.1

Classification: requirements fault

- c9. In the Mode Logic, the values FPEC1 and FPDC1 were not reset to zero when not in Altitude Hold mode (CUR #16).

Type: spec misinterpretation or ambiguity. Detected by: test data data.1

Classification: requirements fault

I.2.3 Faults detected during Acceptance Test

- c10. Some functions used in the Outer Loop of the Flare Control Law were not declared to return double precision values (CUR #17).

Type: omission. Detected by: test data data.2

Classification: requirements fault

- c11. The constant K3 of the Glide Slope Complementary Filter was computed incorrectly while in Altitude Hold mode. It was assumed that the factor $\frac{200}{RAGSF}$ is always limited to $\frac{1}{5.5}$ minimum, but this applies during Glide

Slope Capture and Track modes only (CUR #18).

Type: spec misinterpretation. Detected by: test data data.1

Classification: requirements fault

- c12. In the Mode Logic, a wrong variable was used in an expression (CUR #19).

Type: typo. Detected by: test data data.8

Classification: requirements fault

- c13. In the main program, all the parameters of the routine "VOTESTATES" were forgotten (CUR #20).

Type: omission. Detected by: test data data.2

Classification: requirements fault

I.2.4 Faults detected after Acceptance Test

- c14. In the Inner Loop, initialization of integrator I1 should be in the beginning of the module (CUR #21).
Type: incorrect algorithm. Detected by: flight simulation test data.
Classification: requirements fault
- c15. Introduction of an underground variable "rr_old" which made the internal states inconsistent (CUR #22).
Type: specification ambiguity. Detected by: flight simulation test data.
Classification: structure fault (which happened to be detected by requirements test)
- c16. LR1 and LR2 of the Inner Loop should be initialized only once upon entering AHD mode instead of every mode change (CUR #23, #24 and #25).
Type: incorrect algorithm. Detected by: flight simulation test data.
Classification: requirements fault
- c17. Using +/- 99999.0 instead of +/- ∞ in the Inner Loop and the Command Monitor.
Type: incorrect algorithm. Detected by: code inspection.
Classification: structure fault
- c18. Output of Mode Logic was used in some further computations before it was voted upon.

Type: incorrect algorithm. Detected by: code inspection.

Classification: structure fault

I.3 MODULA-2 Version

I.3.1 Faults detected during Coding and Unit Test

m1. "2*i" as index to an array did not work; "i+i" had to be used instead (CUR #3).

Type: compiler fault Detected by: team's own test data

Classification: structural fault of the compiler

m2. In the Barometric Altitude Complementary Filter, the wrong constant (wrong position of the decimal point) was used for the upper limit of the output of integrator I3 (CUR #4).

Type: typo / *spec misinterpretation* / due to poor readability. Detected by: test data data.2, data.13

Classification: requirements fault

Note: same as ADA team!

m3. A wrong computation sequence was used in the Glide Slope Complementary Filter: The constants K0, K2, and K3 were computed without first computing the value of the variable RAGSF (DWR #18; CUR #5).

Type: incorrect algorithm. Detected by: test data data.1

Classification: requirements fault

m4. The Signal Display Algorithm (procedure todigit) could not handle a boundary

case (DWR #21; CUR #9).

Type: incorrect algorithm, partly due to language peculiarities (floating point operations for LONGREAL variables). Detected by: test data sig_words

Classification: requirements fault

The other changes were due to specification updates (DWR #17; CUR #2, 6) and to compiler peculiarities (too many LONGREAL parameters in a procedure call – DWR #19, 20; CUR #7, 8 – and too complicated an expression in a RETURN statement – CUR #1).

I.3.2 Faults detected during Integration Test

No faults were detected.

Changes were made because of an specification update (DWR #22; CUR #10), and the interface to "VOTESTATES" had to be changed because the compiler could not handle so many parameters (see above) (CUR #11, 12).

I.3.3 Faults detected during Acceptance Test

No faults were detected.

I.3.4 Faults detected after Acceptance Test

m5. Output of Mode Logic was used in some further computations before it was voted upon.

Type: incorrect algorithm. Detected by: code inspection.

Classification: structure fault

I.4 PASCAL Version

I.4.1 Faults detected during Coding and Unit Test

- p1. The Signal Display did not round correctly (DWR #32, CUR #1).
Type: incorrect algorithm. Detected by: reading the code
Classification: requirements fault
- p2. A wrong constant was used in the Fault-word Display (CUR #2).
Type: incorrect algorithm. Detected by: test data fault_words
Classification: requirements fault
- p3. In the Signal Display, zero was displayed as "+.00000" instead of as ".00000" (CUR #3).
Type: spec misinterpretation / incorrect algorithm. Detected by: test data sig_words
Classification: requirements fault
- p4. The constants of the Glide Slope Complementary Filter were not computed correctly in all system modes (CUR #4).
Type: spec misinterpretation. Detected by: test data data.18
Classification: requirements fault
- p5. The output of integrator I8 in the Glide Slope Complementary Filter was not

initialized correctly (CUR #5, 6).

Type: spec misinterpretation. Detected by: test data data.11

Classification: requirements fault

- p6. The initialization of the variable HREF in the Outer Loop of the Altitude Hold Control Law was incorrect (CUR #7, 8, 10).

Type: incorrect algorithm. Detected by: reading the code

Classification: requirements fault

- p7. The initialization of Inner Loop variables was incorrect (CUR #9).

Type: incorrect algorithm. Detected by: reading the code

Classification: requirements fault

- p8. Logic Fault in the evaluation of the condition for switch SW2 in the Inner Loop (DWR #36; CUR #19).

Type: incorrect algorithm. Detected by: test data data.20

Classification: requirements fault

- p9. In the Outer Loop of the Flare Control Law, the function F4 was evaluated incorrectly (wrong setting of parenthesis) (DWR #34; CUR #13).

Type: typo / spec misinterpretation. Detected by: test data data.11

Classification: requirements fault

- p10. Redundant computation of test point 7 in the Inner Loop (DWR #37; CUR #20).

Type: unnecessary. Detected by: test data data.20

Classification: requirements fault

Other changes were made in response to specification updates in the Mode Logic (CUR #11, 21), to aid in debugging (CUR #14 - 16, 18), and to remove a compiler warning message (CUR #12, 17).

I.4.2 Faults detected during Integration Test

p11. The variable RAGSF in the Glide Slope Complementary Filter was not initialized correctly (CUR #26, 27).

Type: incorrect algorithm. Detected by: test data data.1

Classification: requirements fault

p12. The integration of the integrator I8 in the Glide Slope Complementary Filter was not performed during the very first computation frame (CUR #26).

Type: incorrect algorithm. Detected by: test data data.1

Classification: requirements fault

Other changes were made to be compatible with the C interface and to remove debugging facilities introduced before Unit testing.

I.4.3 Faults detected during Acceptance Test

No faults were detected; some changes were made for the sake of compatibility with the C interface.

I.4.4 Faults detected after Acceptance Test

No faults were detected.

I.5 PROLOG Version

I.5.1 Faults detected during Coding and Unit Test

pg1. The names of some Mode Logic variables (Fig. 4.2) conflicted with variable names in the Flare Control Law (DWR #24).

Type: incorrect algorithm. Detected during coding

Classification: requirements fault

pg2. In the Inner Loop, the variable names THETA_C and THCI were confused (DWR #29; CUR #15).

Type: incorrect algorithm. Detected during coding

Classification: requirements fault

pg3. Integrator I8 in the Glide Slope Complementary Filter was not initialized correctly (DWR #30; CUR #7).

Type: incorrect algorithm / spec misinterpretation. Detected by: test data filter.gs/data.9

Classification: requirements fault

pg4. A comma was misplaced in the Barometric Altitude Complementary Filter (CUR #2).

Type: typo. Detected during coding

Classification: requirements fault

pg5. Initialization of the Inner Loop was forgotten in the main program (DWR #32).

Type: omission. Detected during coding

Classification: requirements fault

pg6. The order of parameters in the definition of the integrator function was inconsistent with its use (CUR #3).

Type: typo / incorrect algorithm. Detected by: test data filter.ba/data.1

Classification: requirements fault

pg7. The order of parameters in the definition of the linear-filter function was inconsistent with its use (CUR #5).

Type: typo / incorrect algorithm. Detected by: test data filter.ba/data.1

Classification: requirements fault

pg8. A wrong global variable was used in the Glide Slope Complementary Filter (CUR # 8).

Type: typo. Detected by: test data filter.gs/data.9

Classification: requirements fault

pg9. The constant K3 in the Glide Slope Complementary Filter was computed incorrectly (wrong magnitude limitation when not in Glide Slope Capture or

Track modes) (CUR #9).

Type: incorrect algorithm. Detected by: test data filter.gs/data.18

Classification: requirements fault

pg10. In the main program, the initialization of the variable "first_ahd" was forgotten (CUR #11).

Type: omission. Detected by: team's own test data

Classification: requirements fault

pg11. Missing declaration of THETA_C as a global variable (CUR #14).

Type: omission. Detected by: reading the code

Classification: structural fault

pg12. In the Inner Loop, a wrong algorithm was used (CUR #16).

Type: typo / incorrect algorithm. Detected by: test data inner/data.6

Classification: requirements fault

pg13. In the Inner Loop, SU4 and LM1 are computed twice, but no new variable names were used for the second computation (CUR #17).

Type: omission, due to language peculiarities (one cannot reassign a value to a bound variable). Detected by: test data inner/data.6

Classification: structural fault due to the language

pg14. Wrong constants were used in the Outer Loop of the Flare Control Law (CUR #49).

Type: typo (readability problems). Detected by: coordinators' test data

Classification: requirements fault

pg15. The value of SIGIN was not input to the Display procedure, to determine which signal should be displayed (CUR #54).

Type: omission. Detected by: coordinators' test data

Classification: requirements fault

The other changes were due to specification updates (DWR #23; CUR #53), efforts to increase the efficiency (DWR #25, 27, 28; CUR #1, 4, 6, 51), were made to add debugging facilities (CUR #10, 12, 13, 18), and to add an "absolute value" routine to the Prolog interpreter (CUR #58).

CUR #55 is questionable: it seems that this update was actually not done (comparison with code).

I.5.2 Faults detected during Integration Test

pg16. Wrong initialization of Filter F10 in the Glide Slope Complementary Filter (DWR #34, 35; CUR #19, 20, 33, 34, 35).

Type: typo / incorrect algorithm. Detected by: test data data.1

Classification: requirements fault (count as only one fault)

pg17. A wrong function name was used in the definition of "do_ahd" (main program) (CUR #23).

Type: typo

Classification: requirements fault

pg18. Incorrect organization of the command monitors (global variables) (CUR #24).

Type: incorrect algorithm. Detected by: reading the code

Classification: requirements fault

pg19. Syntax error in a comment in the main program (CUR #28).

Type: typo. Detected by: Prolog interpreter

Classification: requirements fault

pg20. Inconsistencies between the global database and the Barometric Altitude Complementary Filter, the Radio Altitude Complementary Filter, the Inner Loop, and the routine "VOTESTATES". Also, a "retract" statement was forgotten in the initialization of the Barometric Altitude Complementary Filter (CUR #29, 30, 37, 44).

Type: omission. Detected by: reading the code

Classification: structural fault

pg21. A state variable of the Glide Slope Complementary Filter was not entered into the global database (CUR #31, 32).

Type: omission. Detected by: reading the code

Classification: structural fault

pg22. A wrong variable name was used in the "VOTEFILTER2" function in the interface (CUR #39).

Type: typo.

Classification: requirements fault

The other changes were due to adding/deleting debugging facilities (CUR #21, 22, 25, 26, 38, 42), to efforts to increase efficiency and/or clarity (CUR # 27, 43, 56, 58), to numerical difficulties (CUR #36), and to the installation of the Display procedure in the Prolog interpreter (CUR #40, 41). CUR #33, 34 are duplications of CUR #19, 20, respectively.

I.5.3 Faults detected during Acceptance Test

pg23. Fault in "do_gscf" function: VOTEFILTER2 must be called for every frame computation (DWR #33; CUR #47).

Type: incorrect algorithm. Detected by: test data data.4

Classification: requirements fault

pg24. The condition for initializing the Outer Loop of the Flare Control Law was wrong (CUR #45).

Type: incorrect algorithm. Detected by: test data data.2

Classification: requirements fault

pg25. The function name "set_gsp" was misspelled in the main program (CUR #46).

Type: typo. Detected by: reading the code

Classification: requirements fault

pg26. In the definition of "callVotestates" for the C interface, new variable names

had to be introduced for the return values (CUR #60).

Type: omission, due to language peculiarities (one cannot reassign a value to a bound variable). Detected by: test data data.8

Classification: structural fault due to the language

I.5.4 Faults detected after Acceptance Test

pg27. A state variable of the Inner Loop was updated twice during one computation (CUR #61).

Type: specification ambiguity. Detected by: flight simulation test data.

Classification: requirements fault

pg28. Rounding errors in the Display module (CUR #62).

Type: incorrect algorithm. Detected by: code inspection.

Classification: requirements fault

pg29. Output of Mode Logic was used in some further computations before it was voted upon.

Type: incorrect algorithm. Detected by: code inspection.

Classification: structure fault

I.6 T Version

I.6.1 Faults detected during Coding and Unit Test

t1. No magnitude limitation was performed on the output of integrator I2 in the

Barometric Altitude Complementary Filter (CUR #1).

Type: omission. Detected by: test data data.11

Classification: requirements fault

- t2. The output of integrator I2 in the Barometric Altitude Complementary Filter was initialized by zero, instead of by HR (DWR #10; CUR #2).

Type: omission. Detected by: test data data.13

Classification: requirements fault

- t3. The output of integrator I8 in the Glide Slope Deviation Complementary Filter was initialized by zero, but it should be initialized by the current value of the output variable GSEL (DWR #8, 9; CUR #3).

Type: The team claimed the fault was due to a spec ambiguity, but we feel that it could be a spec misinterpretation or a oversight.

Classification: requirements fault

- t4. Several changes in the Mode Logic (CUR #5).

Type: wrong algorithm, typos, omissions. Detected by: test data data.1, data.6

Classification: requirements fault

- t5. In all the Complementary Filters, the variables of the current computation state were initialized, instead of the variables of the previous computation state (CUR #6).

Type: incorrect algorithm. Detected by: test data data.1

Classification: requirements fault

t6. Incorrect computation of the output of the Command Monitor, "smaller" and "smaller or equal" was confused (CUR #7).

Type: incorrect algorithm. Detected by: reading the code

Classification: requirements fault

t7. Incorrect rounding in the Signal Display function (CUR #8).

Type: incorrect algorithm. Detected by: test data

Classification: requirements fault

Other changes were due to specification updates concerning the Mode Logic (DWR #11; CUR #4).

I.6.2 Faults detected during Integration Test

t8. In the Inner Loop, the rate limiters and switch SW2 were implemented incorrectly (CUR #9).

Type: incorrect algorithm. Detected by: test data data.1

Classification: requirements fault

t9. The global variable definitions in the main program were incomplete (CUR #11). Type: omission. Detected by: reading the code

Classification: requirements fault

t10. The overall organization of the initialization (main program, Mode Logic) had to be fixed, so that all functions would get properly initialized. Originally, a flag indicating a mode change was erroneously reset by the Mode Logic (CUR

#13, 14).

Type: incorrect algorithm. Detected by: reading the code

Classification: requirements fault

- t11. The arguments standing for global variables were deleted from the functions VOTEMODE and VOTEFILTER1. Only test point variables are passed as arguments (CUR #15, 16).

Type: unnecessary code. Detected by: reading the code

Classification: structural fault

The other changes were made to add comments, and to correct the interface between C and T (CUR #10). To make the program more clear, arguments were explicitly added to the VOTEFILTER2 function (CUR #12) – this could also indicate a fault, however.

I.6.3 Faults detected during Acceptance Test

- t12. In the VOTESTATES function, some variables were omitted from the format statements, and some variables were not assigned their value as returned by VOTESTATES (CUR #19, 20).

Type: omission. Detected by: reading the code, test data data.7

Classification: requirements fault

- t13. In the VOTESTATES function, some wrong variable names were used when assigning the return value (CUR #21).

Type: typo, incorrect algorithm, due to inconsistent naming conventions among team mates. Detected by: test data data.7

Classification: requirements fault

- t14. The transition from Glide Slope Capture to Glide Slope Track mode was considered a mode change requiring reinitialization of the Inner Loop (CUR #22).

Type: incorrect algorithm. Detected by: test data data.3

Classification: requirements fault

- t15. In the interface to the C routines, the parameters of VOTESTATES were called by value instead of by reference (CUR #23).

Type: omission, incorrect algorithm. Detected by: test data data.7

Classification: requirements fault

- t16. The current value of RAE was used in function F3 in the Flare Outer Loop, instead of the value of RAE at Flare initiate (CUR #24).

Type: incorrect algorithm, possibly caused by typo or omission. Detected by: test data data.2

Classification: requirements fault

- t17. A parameter was omitted in the definition of VOTEOUTER (CUR #25).

Type: omission. Detected by: test data data.2

Classification: requirements fault

- t18. A global variable was initialized twice in the Main Program (CUR #27).
Type: unnecessary statement. Detected by: code reading
Classification: structural fault (but did not cause any error)
- t19. The routine LANEINPUT, the Command Monitors, and the Display are not called when the mode is Touchdown.
Type: incorrect algorithm. Detected by: code reading
Classification: requirements fault
- t20. The Command Monitors are re-initialized at every mode change.
Type: incorrect algorithm. Detected by: code reading
Classification: requirements fault
- t21. The rate limiters LR1 and LR2 in the Inner Loop are re-initialized at every mode change.
Type: incorrect algorithm. Detected by: code reading
Classification: requirements fault

I.6.4 Faults detected after Acceptance Test

- t22. A state variable of the Inner Loop was updated twice during one computation (CUR #28).
Type: specification ambiguity. Detected by: flight simulation test data.
Classification: requirements fault
Note: same as PROLOG team!

t23. Output of Mode Logic was used in some further computations before it was voted upon.

Type: incorrect algorithm. Detected by: code inspection.

Classification: structure fault

The other changes were made in order to increase storage efficiency (CUR #17), to increase the size of the buffer in which arguments are passed from the C routines to the T functions, just to be on the safe side (CUR #18), and to make the structure of the Mode Logic more clear (CUR #26).