IN SEARCH OF EFFECTIVE DIVERSITY: A SIX-LANGUAGE STUDY OF FAULT-TOLERANT FLIGHT CONTROL SOFTWARE

Algirdas Avizienis
Michael R. Lyu
Werner Schutz

November 1987
CSD-870060

FOREWORD

This report presents the first-year summary of an investigation of fault-tolerant N-version software for automatic flight control that is in progress at the UCLA Dependable Computing and Fault-Tolerant Systems Laboratory. The goals of this research are:

(1) to refine the methods for generating demonstrably effective N-version software in an industrial environment;

(2) to study the extent of effective diversity (dissimilarity) that can be attained by the use of a rigorous methodology (a *design paradigm*) and the choice of different high-level programming languages;

(3) to investigate the extent of correlation of software faults that are uncovered during the development of N-version software, as well as those that are discovered after acceptance;

(4) to estimate the effectiveness and to evaluate the relative safety of N-version software versus single-version software.

This research is directed by Professor Algirdas Avižienis as Principal Investigator. The research team consists of the following graduate students in the UCLA Computer Science Department: Michael R. Lyu, coordinator, Werner Schütz, Johnny J. Chen, and Chi S. Wu.

Support for the first year of this research has been provided by the Sperry Commercial Flight Systems Division of Honeywell, Inc., Phoenix, AZ, and by the Microelectronics Innovation and Computer Research Opportunities (MICRO) program of the State of California.

Mr. John F. Williams from Honeywell, Inc. has served as the Honeywell technical liaison person and a consultant on flight control computing to this research group. Problem specifications, software design and test procedures, an aircraft model, sets of test cases, and expert advice has been contributed by Mr. Williams and other members of the Honeywell technical staff.

The UCLA research group that is conducting this investigation is fully responsible for the conclusions of this report and for any inaccuracies that may exist.

Algirdas Avižienis
Michael R. Lyu
Werner Schütz

November 15, 1987

TABLE OF CONTENTS

In Search of Effective Diversity:

A Six-Language Study of Fault-Tolerant Flight Control Software

Algirdas Avižienis
Michael R. Lyu
Werner Schütz

UCLA Dependable Computing and Fault-Tolerant Systems Laboratory
University of California, Los Angeles, CA 90024, U.S.A.

1. Introduction: Origin and Scope of the Project

The investigation being reported here is the consequence of a coincidence of 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).

Four of the long-range goals of UCLA research, which was initiated in 1975 [Aviz85a], are:

(1) The development of rigorous design guidelines (a *paradigm*) that will eliminate all identifiable causes of *related* design faults in two or more independently generated versions of a program or design.

(2) Testing for and detailed study of all potentially related design faults that actually produce similar errors in two or more versions independently generated from a given specification.

(3) Development of qualitative criteria that allow the assessment of the *potential for diversity* through the study of a specification from which the versions are to be generated.

(4) Development of methods for the study of a set of multiple versions to determine to what extent diversity is actually present in the set, and search for the means to quantize the relative diversity of versions that originate from a given specification. The relative benefits of random vs. "enforced" diversity are also of great interest.

Honeywell/Sperry CFSD 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].

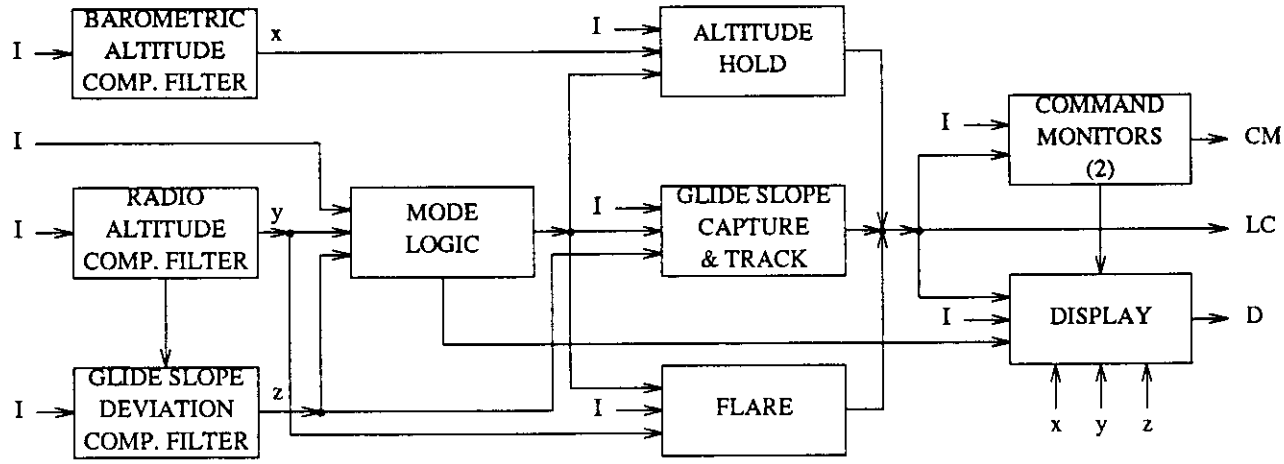
The main research interest of H/S is the generation of demonstrably effective N-version software in an industrial environment, such as exists now and is being further developed by H/S. This objective includes all four above stated topics of UCLA research, referenced to the industrial environment, as well as the estimation of the effectiveness of N-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 the State of California "MICRO" program. 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 November, 1987.

2. The Problem and the Constraints

2.1 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-H/S 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 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 1.



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

Figure 1: Pitch Control System Functions and Data Flow Diagram

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, Glide Slope Capture, Glide Slope Track, Flare, and Touchdown.

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 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.

H/S extracted the information needed for the experiment from their original Demonstrator specification and provided it in a "System Description Document". This document also specified the "test points", i.e., selected intermediate values of each major system function which had to be provided as outputs for additional error checking.

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. The diagrams describing the major system functions were taken directly from the "System Description Document", while the explanatory text was shortened and made more concise. Some general explanations about flight control and the specification of the Display functions were added. A further enhancement to the specification was the introduction of *cross-check points* [Aviz85b] and a *recovery point* [Tso87]. Seven cross-check points are used to cross-check the results of the major system functions (e.g. Complementary Filters, Mode Logic, Outer Loop, Inner Loop, etc.) with the results of the other versions before they are used in any further computation. They have to be executed in a certain predetermined order, but again great care was taken not to overly restrict the possible choices of computation sequence. One recovery point is used to recover a failed version by supplying it with a set of new internal state variables that are obtained from the other versions by the Community Error Recovery technique [Tso87].

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 two members of the coordinating team, plus consultation by H/S experts.

2.2 The Choice of Diversity Dimensions

Design diversity is a potentially effective method to avoid similar errors that are caused by design faults in N-version software systems. The choice of diversity dimensions in this experiment was based on the experience gained from (1) previous experiments at UCLA [Chen78, Kell83, Aviz84, Kell86], (2) recommendations from H/S, and (3) published work from other sites [Gmei79, 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 "forced" diversity [Aviz85a]. 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 had been initiated in this direction [Gmei79, Bish86], significant comparisons of different high order programming languages for the same critical application have not yet been reported.

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.

2.3 Requirements for Software Testing

H/S, along with other avionics suppliers, must adhere to the requirements of the document DO-178A [RTCA85], the industrial software design and test standard approved by the FAA. 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 which is not exposed by system testing based on the system specification.

2.4 The Rationale for Applying Design Diversity in Testing

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.

The software portion of the critical equipment must, then, have a probability of failure less than 10^{-9} depending on the failure rates in remaining portions of the system. To protect against failures in single-version software that cause total loss of a critical function, a *structural-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 which 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 forseeable conditions and critical modes of operation.

3. Guidelines and the Process of Multi-Version Programming

3.1 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. (Their names are given in the Acknowledgement section.) 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.

3.2 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 which was attended by UCLA and H/S principal investigators, the UCLA coordinating team, and an H/S software expert.

Team member	Degree held			CS standing in summer '87		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

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 with the input case on which it failed, for debugging and resubmission. By the end of this two week phase, five programs had passed this 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.

3.3 The Programming Process

The software engineering process involved in this project included formal reviews, well-planned record keeping, isolation rules, a formal communication protocol, and carefully executed testing phases. 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 separate module (before and after unit tests), four snapshots of the complete program (those before and after integration tests, and those before and after acceptance tests), two design documents (preliminary and final versions), program metrics, design walkthrough reports, and code update reports.

Since error 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 the subsequent analysis, we consider only those changes that were done to correct faults in the programs.

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. 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 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 phone calls and personal 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 2.

To emphasize the importance of testing, three phases of testing, unit tests, integration tests, and acceptance tests, were introduced for error detection and debugging. 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 the unit tests and the integration tests. 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. 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

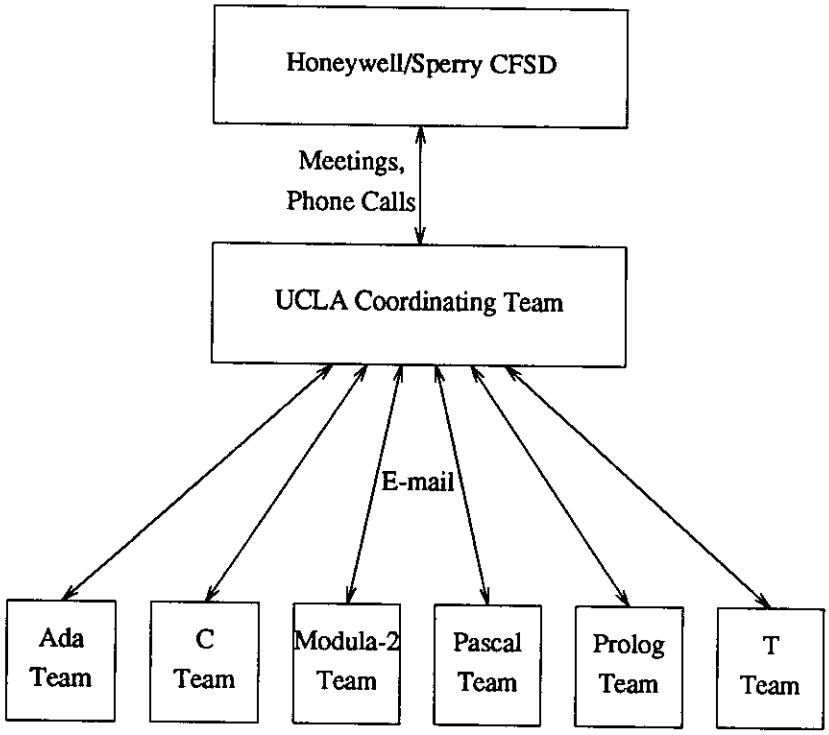


Figure 2: Communication Diagram of the Experiment

points to generate test data during the acceptance tests.

3.4 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, 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 although occasionally it was felt that it was more difficult to write the answer to a certain question, that oral communication would have been easier and more efficient in some cases. The communication with H/S was very efficient; thus it was possible to answer all questions within

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. 10 broadcast messages were not triggered by a question; 5 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 5 of them 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; that 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. Properties of the Versions

As soon as the versions passed the acceptance test, a number of results became available. They are some software metrics, collected by each programming team from its own program, and the record of faults found during program development. All these faults were removed from the versions before any further evaluation began.

4.1 Software Metrics

Table 2 gives a comparison of the six versions with respect to some common software metrics [Li87]. The following metrics are considered: (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 there local or global variables have to be used as "constants"; and (13) the number of binary decisions (BINDE).

4.2 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.

Table 3 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.

Classification of faults according to fault types is shown in Table 4. This category considers the following type of faults: (1) typographical; (2) error of omission (missing code); (3) unnecessary implementation (which was deleted); (4) incorrect algorithm; (5) specification misinterpretation; and (6) specification ambiguity. "Incorrect algorithm" is the most frequent fault type, which includes miscomputation, logic fault, initialization fault, and boundary fault.

Metric	ADA	C	MODULA-2	PASCAL	PROLOG	T
LINES	2253	1378	1521	2234	1475	1575
STMTS	1031	746	546	491	1089	1089
LN-CM	1517	861	953	1288	*	1263
OBJS	85.6k	83.7k	51.9k	37.5k	N.A.	N.A.
MODS	36	26	37	48	73	44
STM/M	29	25	15	10	15	25
CALLS	97	68	65	93	*	87
LIBS	2	2	2	10	7	N.A.
LCALL	3	9	6	12	54	N.A.
GBVAR	139	141	91	81	90	97
LCVAR	117	197	132	127	209	251
CONST	68	21	18	16	N.A.	N.A.
BINDE	74	114	78	118	*	86

* = Metric not provided by the team
N.A. = not applicable

Table 2: Software Metrics for the Six Programs

	ADA	C	MODULA-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
Alt. Hold Outer Loop	0	0	0	1*	0	0	1
Glide Slope 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 3: Fault Distribution by Subfunctions

	ADA	C	MODULA-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	2	0	2	5
Incorrect Algorithm	3	5	2	6	9	13	38
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 4: Fault Classification by Fault Types

Table 5 shows during which phases of testing the faults were detected.

	ADA	C	MODULA-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 5: Fault Classification by Phases

Finally, Table 6 shows the classification of faults according to the categories of "requirements fault" and "structural fault" (see section 2.3).

	ADA	C	MODULA-2	PASCAL	PROLOG	T	Total
Requirements	4	12	3	10	20	18	67
Structural	2	1	1	2	6	3	15
Total	6	13	4	12	26	21	82

Table 6: Fault Classification: Requirements Faults vs. Structural Faults

4.3 Additional Observations

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 4. 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.

It is interesting to note that there was *only one* incidence of an identical fault, committed by two teams, ADA and MODULA-2. In both cases the fault was discovered during unit testing. The fault was the following: the output of an integrator in the Barometric Altitude Complementary Filter must be limited by 65,536. Both teams mistook the comma after the 1000's place for a decimal point and used the constant 65.536. We are not sure whether to classify that fault as a typo or a specification misinterpretation. Although we think that this particular number is easily readable, the example still shows that it is dangerous to provide hand-written numbers in a specification.

In conclusion, we believe that the number of faults found indicates that all six programs were quite thoroughly tested before they were accepted.

5. Testing and Evaluation After Acceptance of the Versions

Requirements-based stress testing and structural analysis are the two employed approaches for the evaluation of the six programs. The efforts of finding more faults (requirements-based or structure-based) and the search for evidence of structural diversity among these programs have been the major concerns.

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

Based on these tools, the UCLA coordinating team has been conducting H/S approved "Level 2" stress testing for months since the software generation phase was completed in early September 1987. 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 3 channels of diverse software each compute 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. The individual commands are recorded and compared for discrepancies which could indicate the presence of faults.

The configuration of the flight simulation system (shown in Figure 3) 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 is based on 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.

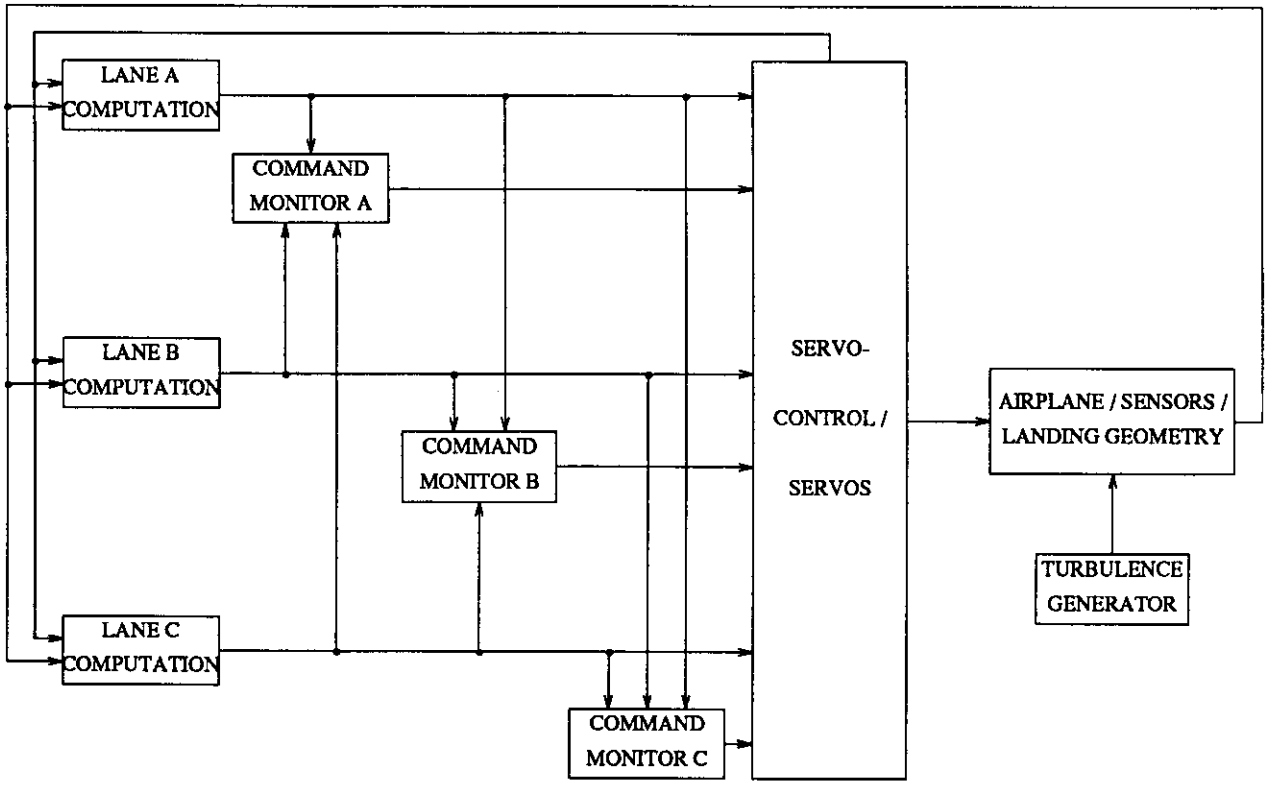


Figure 3: 3-Channel Flight Simulation Configuration

The airplane is a mathematical model that computes the response of the airplane to an elevator command in terms of attitude, attitude rate, flight path, altitude, altitude rate, and vertical acceleration. In a real aircraft these values would be directly measured by sensors. The landing geometry model describes the deviation from the glide slope beam center as a function of aircraft position relative to the end of the runway. Moreover, in order to provide a set of inputs to the airplane model which 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 several 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 3 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 600 flight simulations (over 3,000,000 time frames) have been exercised on the six software versions generated from this project.

In addition to the flight simulations, a structural analysis also was carried out. The six versions were compared to find the differences in structure and implementation that resulted from the application of the N-version programming methodology. 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.

6. Results of Testing and Evaluation

6.1 Disagreements Detected by Flight Simulations

So far, four disagreements at the Inner Loop cross-check point have been detected during the flight simulations. Due to the additional information provided by the test points, it was relatively easy to determine the faulty part of the code in each case. The C version experienced two disagreements. The first one resulted in the detection of two faults, namely initialization with a wrong value (an intermediate value of the present time frame computation was used instead of a result of the previous time frame computation), and the introduction and use of an unnecessary state variable. This latter fault is related to the "underground variables" discussed in the next section; the only difference is that in this case the fault caused a 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.

6.2 Faults Found During Inspection of Code

The following faults were detected during the code inspection performed as part of the structural analysis (see section 5):

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 which 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, an 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.

6.3 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 [Chen78], 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.

The "PFD" column of Table 7 gives 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 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 7 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.

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 7: Potential for and Observed Diversity

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 7 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. These and more details 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.

6.4 Observations from the Diversity Assessment

In general, it can be said that more diversity was observed in the aspects whose method of implementing 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 the upper levels do not have to be concerned with 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; especially this is 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. That 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.

7. Conclusions

The major conclusions of this study are:

(1) The UCLA paradigm for systematic generation of multiple-version software is sufficiently complete and stable for application in industrial environments.

(2) The use of different programming languages has supported very effective inter-team isolation, since different support environments were used. It also has promoted the appearance of diversity in versions that began with a common specification.

(3) Identical faults in two versions were very rare. 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, five faults were uncovered by testing. One pair again was identical, and this fault was due to failure to properly incorporate a clarification to a specification ambiguity. Six more faults were discovered by code inspection, all unrelated and different.

(4) The "Type 3" structural faults (section 6.2) 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. Very strict verification that design rules were followed must be a part of the acceptance test.

(5) The order of computations that is implied by the specification has a strong influence on the programmers' choice, even if other alternatives exist. 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.

(6) The original specification, as received from H/S, contained too much information on implementation issues, which would tend to limit diversity. Our concentrated effort to reduce the specification as much as possible to the "what", removing the "how", paid off by encouraging diversity.

We also note that we found only two identical faults that cause similar errors, described in (3) above. 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. For this reason, our conjecture is that a rigorous application of the paradigm described in this paper would have led to the elimination of most faults described in [Knig86] before acceptance of the programs.

Acknowledgements

This research has been supported jointly by the Sperry Commercial Flight Systems Division of Honeywell, Inc., in Phoenix, Arizona, and the State of California "MICRO" program. It is with great pleasure that we acknowledge the efforts of the representative from Honeywell/Sperry, Mr. John F. Williams, who offered tremendous help and encouragement to this project. The collaboration of Johnny J. Chen as a member of the coordinating team is highly appreciated. The programming efforts were contributed by the following individuals: Hsin-Chou Chi, Andy Y. Hwang, Ting Y. Leung, Paul C. Lin, Eugene Paik, Chien-Chung Shen, Michael D. Stiber, Tsung-Yuan Tai, Charles Tong, Tella Vijayakumar, Ping-Hann Wang, and Chi S. Wu. We would also like to thank Rick Tyo from Honeywell/Sperry and Mark Joseph from UCLA for their valuable suggestions, and Nick Lai for his technical assistance.

References

[Aviz82] A. Avižienis, "Design Diversity - The Challenge for the Eighties," in *Digest of 12th Annual International Symposium on Fault-Tolerant Computing,* Santa Monica, California: June 1982, pp. 44-45.

[Aviz84] A. Avižienis and J.P.J. Kelly, "Fault-Tolerance by Design Diversity: Concepts and Experiments," *Computer,* Vol. 17, No. 8, August 1984, pp. 67-80.

[Aviz85a] A. Avižienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Transactions on Software Engineering,* Vol. SE-11, No. 12, December 1985, pp. 1491-1501.

[Aviz85b] A. Avižienis, P. Gunningberg, J.P.J. Kelly, L. Strigini, P.J. Traverse, K.S. Tso, and U. Voges, "The UCLA DEDIX System: A Distributed Testbed for Multiple-Version Software," in *Digest of 15th Annual International Symposium on Fault-Tolerant Computing,* Ann Arbor, Michigan: June 1985, pp. 126-134.

[Bish86] P.G. Bishop, D.G. Esp, M. Barnes, P. Humphreys, G. Dahll, and J. Lahti, "PODS - A Project of Diverse Software," *IEEE Transactions on Software Engineering,* Vol. SE-12, No. 9, September 1986, pp. 929-940.

[Chen78] L. Chen and A. Avižienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," in *Digest of 8th Annual International Symposium on Fault-Tolerant Computing,* Toulouse, France: June 1978, pp. 3-9.

[Gmei79] L. Gmeiner and U. Voges, "Software Diversity in Reactor Protection Systems: An Experiment," *Proceedings IFAC Workshop SAFECOMP'79,* May 1979, pp. 75-79.

[Kell83] J.P.J. Kelly and A. Avižienis, "A Specification Oriented Multi-Version Software Experiment," in *Digest of 13th Annual International Symposium on Fault-Tolerant Computing,* Milan, Italy: June 1983, pp. 121-126.

[Kell86] J.P.J. Kelly, A. Avižienis, B.T. 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.

[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.

[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.

[RTCA85] RTCA, Radio Technical Commission for Aeronautics, "Software Considerations in Airborne Systems and Equipment Certification," Technical Report DO-178A, Washington, D.C., March 1985. Order from: RTCA Secretariat, One McPherson Square, 1425 K Street, N.W., Suite 500, Washington, DC 20005.

[Schu87] W. Schuetz, "Diversity in N-Version Software: An Analysis of Six Programs," Master Thesis, UCLA Computer Science Department, Los Angeles, CA, November 1987.

[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.

[Tso87] K.S. Tso and A. Avižienis, "Community Error Recovery in N-Version Software: A Design Study with Experimentation," in *Digest of 17th Annual International Symposium on Fault-Tolerant Computing,* Pittsburgh, Pennsylvania: July 1987, pp. 127-133.

[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.