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

Algirdas Avižienis, Michael R. Lyu, and Werner Schütz

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

ABSTRACT

Multi-version software systems achieve fault tolerance through software redundancy and diversity. In order to investigate this approach, this joint UCLA/Honeywell research project investigated multi-version software systems, employing six different programming languages to create six versions of software for an automatic landing program. The rationale, preparation, execution, and evaluation of this investigation are reported.

Keywords. Software fault tolerance, multi-version software, N-version diverse programming, dependability, software development, flight control systems.

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 [Aviz75], are:

(1) The development of rigorous design guidelines (a *paradigm* [Lyu88]) intended to eliminate all identifiable causes of *related* design faults in two or more independently generated versions of a program or design.

(2) A search for and detailed study of all potentially related design faults that actually produce similar and time-coincident errors in two or more versions that were 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. required (or "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].

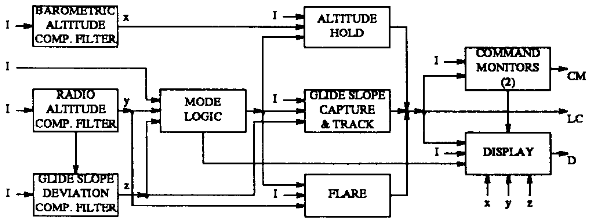
The main research interest of H/S is the generation of demonstrably effective N-version software in an industrial environment, such as exists now and is being further refined by H/S. This objective includes all four above stated topics of UCLA research, referenced to the industrial environment, as well 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 March, 1988.

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-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 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 Autoland 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.

† This research is supported by Honeywell, Inc. and the State of California MICRO (Microelectronics Innovation and Computer Research Opportunities) program.


BAROMETRIC ALTITUDE COMP. FILTER
RADIO ALTITUDE COMP. FILTER
GLIDE SLOPE DEVIATION COMP. FILTER
MODE LOGIC
ALTITUDE HOLD
GLIDE SLOPE CAPTURE & TRACK
FLARE
COMMAND MONITORS (2)
DISPLAY
I
x
y
z
CM
LC
D

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.

H/S extracted the information needed for the experiment from their original Demonstrator specification and provided it in a "System Description Document" (SDD). 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.

A three-member coordinating team began work in October, 1986. They wrote the specification and developed guidelines and procedures, with support from H/S personnel. While writing the specification, the 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 SDD, 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*[Aviz85a] and a *recovery point*[Tso87]. Seven cross-check points are used to cross-check the results of the major system functions 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, and 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 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.

3. Diversity Requirements

The choice of diversity dimensions in this experiment was based on the experience gained from (1) previous experiments at UCLA [Chen78, Kell83, Aviz85b, 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, need to be investigated, and possibly used to assure a certain level of "required" diversity [Aviz85b]. It was decided that different algorithms were not suitable due to potential timing problems and difficulties in proving 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 depend on complete 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 lead 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 Autoland programs.

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. Based on this document, 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.

4. Schedule and Personnel

The recruitment and interviewing of programmers started about 3 months before the 12-week version generation phase in June, 1987. The final choice of 12 programmers and their assignment to six teams were made one month before starting the software generation. The programmers had between 2 and 6 years of programming experience. All held the B.S. degree and had completed at least one year of CS graduate study; six held the M.S. degree and were pursuing doctoral studies in computer science. The effort was directed by the Principal Investigator, and coordinated by the coordinating team. A senior staff expert in flight control computing from H/S maintained continuous contact and regularly made visits to UCLA.

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. The experiment's goals, requirements and the multiple version software techniques were explained. The need for inter-team isolation was thoroughly discussed and acknowledged by all programmers.

(2) *Design phase (4 weeks)*: At the end of this four-week phase, each team delivered and discussed with the UCLA coordinating team, and an H/S software software expert, a design document that followed the guidelines and formats provided at a first-day kick-off meeting.

(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.

(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. 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.

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

5. The Six-Version Programming Process

The software engineering process involved in this project included formal reviews, well-planned record keeping, isolation rules, a formal communication protocol, and phased testing.

The three formal reviews were: the design review, the coding/testing review, and the final review and workshop. These reviews were designed to follow industrial standards as much as possible and involved participation of H/S experts. The first two reviews also 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. They 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 very 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, regardless whether they were due to detected faults, efficiency improvement, specification updates, etc. For each change a standard "Code Update Report" had to be turned in. If a code change was made because of a design change, a "Design Walkthrough Report" had to be submitted as well. In the subsequent discussion, we consider only those changes that were done to correct faults in the programs.

The isolation rules were desired to assure that programming efforts were carried out by teams 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. Inter-team communications were not allowed. 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 responded within 24 hours. Whenever necessary, the help of the H/S flight control experts was provided to the coordinators by phone calls and personal meetings to resolve questions.

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. 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.

Altogether, 120 questions were received from the six programming teams. The answers to 30 of them were broadcast. The total number of broadcast messages was 40. since 10 other 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. Three broadcasts required an additional follow-up message, to provide further clarification or to correct errors in the original message. The individual teams received between 53 and 64 messages; that is a reduction by a factor of 2 of the number of messages that would have been received if the communication protocol of the NASA experiment [Kell86] (in which the answers to *all* questions were broadcast) had been used.

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 points to generate test data during the acceptance tests.

6. 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 further evaluation or testing.

Table 1 gives a comparison of the six versions with respect to some common software metrics. The following metrics are considered: (1) the number of lines of code, including comments and blank lines (LINES); (2) the number of lines excluding comments and blank lines (LN-CM); (3) the number of executable statements, such as assignment, control, I/O, or arithmetic statements (STMTS); (4) the number of programming modules (subroutines, functions, procedures, etc.) used (MODS); (5) the mean number of statements per module (STM/M).

Metrics	ADA	C	MODULA-2	PASCAL	PROLOG	T
LINES	2253	1378	1521	2234	1733	1575
LN-CM	1517	861	953	1288	1374	1263
STMTS	1031	746	546	491	1257	1089
MODS	36	26	37	48	77	44
STM/M	29	25	15	10	16	25

Table 1: Software Metrics for the Six Programs

A total of 82 faults was found and reported during program development. The next two tables present the distribution of these faults in the six versions under different categories.

Table 2 shows the number of faults found during each one of three phases of testing. Table 3 lists the faults according to the following fault types: (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. The "other" fault was introduced by the Modula-2 compiler.

Test Phase	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 2: Fault Classification by Phases

Fault Class	ADA	C	MODULA-2	PASCAL	PROLOG	T	Total
Typographical	0	1	0	0	9	0	10
Omission	1	3	0	0	8	5	17
Unnecessary Code	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 3: Fault Classification by Fault Types

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 certain whether to classify that fault as a typo or a specification misinterpretation, but it is traceable to the fact that the diagram that was misread was reduced in size to fit a standard page, and the reduction made the handwritten comma small enough to be seen as a point.

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 3. The T interpreter had a problem with its garbage collection which resulted in long test runs not completing. This problem delayed the T program's passing of the acceptance test for over a month.

7. Testing and Evaluation After Acceptance of the Versions

Requirements-based stress testing and structural analysis are the two methods employed to evaluate the six programs. The efforts to find more faults (requirements-based or structure-based) and the search for evidence of structural diversity among these programs have been the major goals.

To attain 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 an aircraft model 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 verified by H/S personnel. Input data was generated and interpretation of the results was suggested by H/S experts.

Based on these tools, the UCLA coordinating team has been conducting H/S approved "Level 2" stress testing since the software generation phase was completed in early September 1987. The major strategy in this requirements-based testing is "dynamic closed-loop" testing, which is intended to verify performance, to detect any tendency towards dynamic mistracking between the different program versions, and to expose requirements faults not caught in static testing. In practice, the three channels of diverse software each compute a surface command to guide a simulated aircraft along its flight path. Random wind turbulences of different levels are superimposed in order to represent difficult flight conditions. The individual commands are recorded and compared for disagreements which could indicate the presence of faults.

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, the following testing procedure was used: 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 disagreements 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 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 paradigm. 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.

8. Disagreements Detected by Flight Simulations

So far, four disagreements have been detected during the flight simulations, all at the Inner Loop cross-check point. 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 produced 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 make use of a broadcast clarification that addressed exactly that problem. Although the errors were similar, the two versions disagreed with the other versions in slightly different ways.

It is noteworthy that all observed disagreements were relatively very small, and further evaluations showed that the versions with these discrepancies are always able to achieve proper aircraft 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.

9. Faults Found During Inspection of Code

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

One *requirements fault* was found in the Prolog Display, where rounding to 5 significant digits was not done correctly. 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 may remain incompletely recovered at a *recovery point* [Tso87]. 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.

All six discovered structural faults that are described above are distinct and independent, and thus would be tolerated by the multi-version software approach.

10. 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 potential for "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 minimum requirement must be (1) the isolation of programming efforts, and (2) specified 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 specified 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. Due to space constraints, detailed discussions and examples could not be given here. These and more details can be found in [Aviz87, Schu87].

We must 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.

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.

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 expected 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 requirement of "test points" is the second factor that tended 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 have proved to be 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 outside of cross-check points.

11. Conclusions

The major conclusions of this study are:

(1)The design guidelines (i.e., the "UCLA paradigm") for the systematic generation of multiple-version software that were employed in this effort are sufficiently complete and stable for application in industrial environments.

(2)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.

(3)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.

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

(5)The failure to follow clearly stated multi-version software design rules led to potentially identical and therefore dangerous faults that were identified as Types 2 and 3 in section 9. Very strict verification that design rules were followed must be a part of the acceptance test.

(6) Similar and time-coincident errors (due to identical faults in two versions) were rare. Only one identical pair existed in the 82 faults removed from the six versions before acceptance. During post-acceptance testing and inspection, five faults were uncovered by testing. One pair again was identical. Six more faults were discovered by code inspection, all unrelated and different.

Thus far we have found only two pairs of identical faults that cause similar errors, described in (6) above. Both pairs 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, discussed in section 8, 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. Such a requirement has been added to the paradigm. This result is different from previously published results by Knight and Leveson [Knig86]. We have reviewed [Knig86] and find differences that may account for this outcome. First, the scale of the problem is smaller: 6 pages of (typeset) specification vs. our 64; 327-1004 lines of code vs. 2234 in our Pascal version. Second, the testing seems rather limited: 15 input data sets were used before acceptance, and 200 randomly generated test cases as an acceptance test. This seems inadequate for a program that would launch missile interceptors. It is our conjecture 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.

12. 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, Mr. John F. Williams, who offered tremendous help and encouragement to this project. Johnny J. Chen served well as a member of the coordinating team. 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 Prof. John P. J. Kelly of UCSB for help in the planning of this effort, Rick Tyo from Honeywell, Mark Joseph from UCLA and Lorenzo Strigini from IEI-CNR for their valuable suggestions, and Nick Lai of UCLA for his technical assistance.

13. References

Ande85. T. Anderson, P.A. Barrett, D.N. Halliwell, and M.R. Moulding, "Software Fault Tolerance: An Evaluation," *IEEE Transactions on Software Engineering* SE-11(12), pp. 1502-1510 (December 1985).

Aviz75. A. Avižienis, "Fault-Tolerance and Fault-Intolerance: Complementary Approaches to Reliable Computing," *Proceedings 1975 International Conference on Reliable Software*, Los Angeles, California, pp. 450-464 (April 21-23, 1975).

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

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*, Ann Arbor, Michigan, pp. 126-134 (June 1985).

Aviz85b. A. Avižienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Transactions on Software Engineering* SE-11(12), pp. 1491-1501 (December 1985).

Aviz87. A. Avižienis, M. R. Lyu, and W. Schütz, "In Search of Effective Diversity: A Six-Language Study of Fault-Tolerant Flight Control Software," Technical Report CSD-870060, UCLA Computer Science Department, Los Angeles, California (November 1987).

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* SE-12(9), pp. 929-940 (September 1986).

Chen78. 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*, Toulouse, France, pp. 3-9 (June 1978).

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

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*, Milan, Italy, pp. 121-126 (June 1983).

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*, Sarlat, France, pp. 43-49 (October 1986).

Knig86. J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming," *IEEE Transactions on Software Engineering* SE-12(1), pp. 96-109 (January 1986).

Lyu88. M. R. Lyu, *A Design Paradigm for Multi-Version Software*, Ph. D. Dissertation, UCLA Computer Science Department, Los Angeles, California (May 1988).

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. Schutz, *Diversity in N-Version Software: An Analysis of Six Programs*, Master Thesis, UCLA Computer Science Department, Los Angeles, California (November 1987).

Tso87. 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*, Pittsburgh, Pennsylvania, pp. 127-133 (July 1987).

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).