Diversity in the Software Development Process

Victoria Hilford, Michael R. Lyu*, Bojan Cukic, Anouar Jamoussi, Farokh B. Bastani
Department of Computer Science
University of Houston
Houston, TX 77204 - 3475
Email: FBastani@uh.edu

Abstract

Various methods have been proposed for building fault-tolerant software in an effort to provide substantial improvements in software reliability for critical applications, such as flight control, air-traffic control, patient monitoring, or power plant monitoring. At execution time, the fault-tolerant structure attempts to cope with the effect of those faults that have survived the development process. The two best-known methods of building fault-tolerant software are n-version programming and recovery blocks. To tolerate faults, both of these techniques rely on design diversity, i.e., the availability of multiple implementations of a specification. Software engineers assume that the different implementations use different designs and, thereby, it is hoped, contain different faults.

Our study uses a novel method of incorporating diversity in the development of one version of the software. We term this approach the pipeline method of software development. Its purpose is to eliminate as many software faults as possible before the testing phase. The method was applied to the specification of a real, automatic airplane-landing problem. The results of the pipeline development method are presented.

1. Introduction

The use of redundant copies of hardware, data, and programs has proven to be quite effective in the detection of physical faults and in subsequent system recovery. However, design faults - which are introduced by human mistakes or defective design tools - are reproduced when redundant copies are made; such replication of faulty hardware or software elements fails to enhance the fault tolerance of the system with respect to design faults.

Design diversity is the approach in which the hardware and software elements that are to be used for multiple computations are not copied, but are independently designed to meet a system's requirements. Different designers and design tools are employed in each effort, and commonalities are systematically avoided. The obvious advantage of design diversity is that reliable computing does not require the complete absence of design faults, but only that those faults not produce similar outcomes in a majority of the designs.

Software faults can be introduced in all phases of the system life cycle: specification, design, implementation, testing, and maintenance. They can arise from defects and omissions in initial requirements or specifications, faulty design methodology, misinterpretations of the specifications, and defective maintenance. Faults in the initial requirements and specification are of extreme concern since they are a major source of similar faults that can cause fault-tolerant architectures to fail. Ambiguities in the specification can often result in failures due to incorrect assumptions or misinterpretation on the part of the implementor. Initial-requirement and initial-specification faults detected in the latter phases of the life cycle can also be very expensive to repair because major system design modifications may be required. Design diversity's primary goal is to minimize the probability that independently designed versions will contain similar faults that may cause all or most of the versions to fail simultaneously. Diversity can be introduced in one or more of the specification, design, implementation, testing, and operational phases in the form of different development teams, specifications, programming languages, algorithms, tools, and testing techniques.

Researchers have investigated two primary design-diversity techniques, both of which tolerate the faults remaining in highly reliable systems: recovery-block software and n-version programming.

In the recovery-block approach [18], alternate software versions are organized similarly to dynamic redundancy (standby sparing) in hardware. The approach's objective is to detect design faults at runtime by a test (called an ac-

*Bell Labs., Lucent Technologies, Room 2A-413 600 Mountain Avenue, Murray Hill, NJ 07974, Email: lyu@research.bell-labs.com

ceptance test) performed on the results of execution of one version and to implement the recovery by rollback (restoring the previous, correct state) followed by execution of an alternate version [18].

The other major approach is n-version programming [1]. Its goal is to adapt the hardware technique of n-fold modular redundancy with majority voting to the tolerance of design faults in software. N-version programming provides run-time fault tolerance by comparing the outputs produced by several diverse versions and tries to mask version failures by propagating only consensus results. This consensus is much more than the result of simple majority voting.

In the initial n-version research, the primary dimension of diversity was the use of independent programmers. In further work, the generation of diverse program versions has relied on diversity in the specification, design, implementation, and testing phases. Therefore, "n-version programming" refers to the process by which these diverse program versions, called n-version software, are generated [3].

The diverse versions developed for either the recovery-block or the n-version programming approach provide additional benefits beyond their use in tolerating design faults. While the rigorous application of testing and other fault-prevention techniques is essential to the development of highly reliable systems, most testing methods simply assume that failures will be observed - that an oracle exists to determine the correct response to a test case. In fact, determining the correct output is often a stumbling block to extensive and more exhaustive testing.

Critics have pointed out that this technique could fail to detect failures if all versions produce similar incorrect results. However, it has also been concluded that this technique is a good approximation to a perfect oracle (an oracle that always makes the correct choice) in detecting failures caused by all faults, except for those rare instances where the programs produce identical incorrect results for all values of the input domain on which the fault causes the failure to occur [4]. To detect such a fault, it may be necessary to execute many more test cases to expose the fault. But only one discrepancy is needed for the fault to reveal itself, and the very many test cases that can easily be executed greatly increases the probability that the fault will be exposed.

While recovery-block and n-version programming both rely on the concept of design diversity to provide diverse versions, the methods they use to tolerate faults are different. Recovery-block uses a *backward* error-recovery approach that tries to return the system to a previous, error-free state from which execution may be retried. N-version programming uses a *forward* error-recovery method to construct a valid, error-free new state from existing redundant information.

Diversity seems to be a valuable approach with beneficial results in creating ultra-reliable software. Besides the recovery-block and n-version techniques, we can think of other ways of using diversity. One approach is possibly using several teams during all the phases of the development process. Then the results are compared at the end of each team's completion process. We call this approach the comparison approach. Another approach is a pipeline approach in which different teams work on different phases of the development process in creating a final program. This is the approach we wanted to study in the context of an automatic airplane-landing application. We call this approach the pipeline development paradigm. Another possible approach is using diversity only during some phases of the development process, such as only during the design phase. For example if 4 different designs are produced, they can be compared two at a time to create 2 intermediate versions which can then be compared to obtain the final version. We call this approach the consensus design approach.

Section 2 of this paper is dedicated to diversity in software development such as n-version programming, recovery-blocks, comparison, pipeline, and consensus. In Section 3, we discuss an experiment carried out in order to evaluate the feasibility of the pipeline approach. Section 4 presents the results of our experiment. Then, in Section 5, we look back at the lessons learned and Section 6 concludes the paper.

2. Diversity in software development

Traditionally, software creation has been perceived as a form of art. Design and programming styles, structure and clarity have not yet emerged into engineering standards. According to Littlewood [11], "this lack of scientific support for the efficacy of our practices is one of the main reasons why software engineering remains an aspiration rather than an actual description of how we engineer software systems." Software development for safety-critical applications is definitely not a form of artistic expression. One of the ways to improve the trust in software is to improve the trust in (and thorough understanding of) its design and construction.

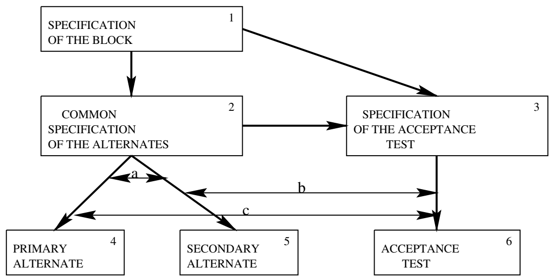
The first step in this process is the identification of the possible types of faults through an analysis of different software development processes. As mentioned earlier, we study the following controlled approaches to software diversity:

- Recovery blocks (RB),
- N-version programming (NVP),
- Comparison approach,
- Consensus design approach, and
- Pipeline development paradigm.

Two most general types of faults resulting from any development process are *independent faults* and *related faults*. Related faults result either from a fault in the common specification, or from dependencies in the separate designs and implementations. Related faults may be further subdivided according to their origin, i.e., 1) among several variants (alternates for RB or versions for NVP) and 2) among one or several variants and the decider (the acceptance test of the RB or voting algorithm of NVP). Related faults manifest themselves under the form of *similar errors*, whereas we shall assume that independent faults cause *distinct errors*.

2.1. Recovery blocks

The development process of the recovery block approach consists of a primary alternate P, a secondary alternate S, and an acceptance test AT, is shown in Figure 1(a). During the diversified design and implementation of P, S, and AT, *independent faults* may be created. However, due to dependencies, some *related faults* between P and S or between P, S, and AT may also be introduced. Faults committed due to the common specification (paths 1→2, 1→3, 1→2→3) are likely to be related faults and, as such, the cause of *similar errors*. Faults created during the implementation can also lead to related faults between P, S, and AT (channels a, b, c); all these faults are summarized in Figure 1(b).



(a) Fault sources in the development process

Path where fault(s) is (are) created or dependency channel(s)	Fault type(s)
1 -> 2 or (a)	Related fault in P and S
1 -> 3, (c) or 1 -> 2 ->3	Related fault in P and AT (or P, S, and AT)
(b)	Related fault in S and AT
2 -> 4 or 2 -> 5	Independent fault in P or S
3 -> 6	Independent fault in AT

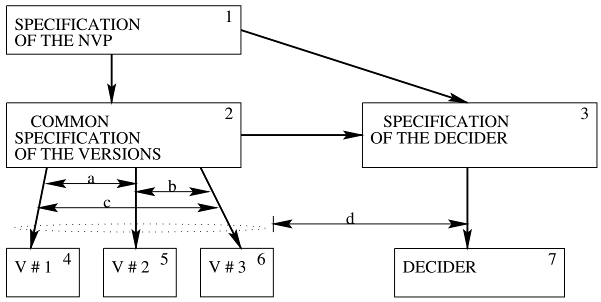
(b) Fault types

Figure 1. Recovery Block (RB) fault analysis

There are at least two different ways to combine hardware redundancy with recovery blocks. The RB/1/1 architecture duplicates the recovery block on two hardware components. In this architecture, both hardware components execute the same variant, and faults are detected by a comparison of the acceptance test and computation results. The distributed recovery block (DRB) [9] executes different alternatives on the different hardware components in order to improve performance in case an error is detected. In the DRB system, one processor executes the primary alternate P while the other executes the secondary alternate S. If an error is detected in the output of the primary, the results from the secondary are immediately available.

2.2. N-version programming

As mentioned earlier, n-version programming (NVP) describes the practice of independently developing versions of software that feed their outputs to a voting unit. Assuming that the goal is fault tolerance, the voting unit takes the majority result. The potential sources of faults in the development process of n-version programming with three versions and one adjudicator are shown in Figure 2(a).



(a) Fault sources in the development process

Path where fault(s) is (are) created or dependency channel(s)	Fault type(s)
1 -> 2	Related fault in the 3 versions
(a), (b), or (c)	Related fault in 2 versions
1 -> 2 -> 3, 1 -> 3 or (d)	Related fault in versions and decider
2 -> 4, 2 -> 5, or 2 -> 6	Independent fault in a version
3 -> 7	Independent fault in the decider

(b) Fault types

Figure 2. NVP fault analysis

The source of the related faults in the NVP approach is the use of a common specification. This has been pointed out in the much publicized study which challenges the independence assumption of NVP [10], and is reflected in Figure 2(b). However, n-version programming can be used to introduce independence in multiple fault classes including [6]:

- Design faults, using different designs for each version,

- Compiler faults, using different compilers for each version,
- Language complexity faults, using different language for each version,
- Implementation faults, using different programmers.

2.3. Comparison approach

Figure 3 depicts the development process of the comparison approach. 3 teams of one or more members each will develop the software product in parallel. Up to this point, the comparison approach resembles the n-version programming approach. Each team goes through all the phases (design, coding, testing, etc.). At the end of this process the results are compared.

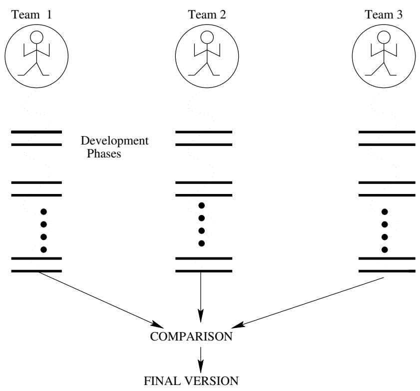


Figure 3. Comparison approach in software development

Comparison results guide the production of only one final version of the software product. Having the insight over all kinds of independent faults discovered by comparing different versions should be very valuable for producing a highly reliable final version. However, it is doubtful whether the advantages of this approach justifies its high cost. In addition to the development of n product versions, putting together the final version may introduce prohibitively large costs and schedule overruns.

2.4. Consensus approach

The consensus approach tries to tackle the problems of the comparison approach. It reduces the number of concurrent software development processes by frequent comparisons between them. Figure 4 depicts the consensus approach that is applied only in the design phase of the development process. Versions V1 and V2 are compared resulting in version V12, versions V3 and V4 are compared resulting in the version V34, then versions V12 and V34 are compared resulting in the final version V1234.

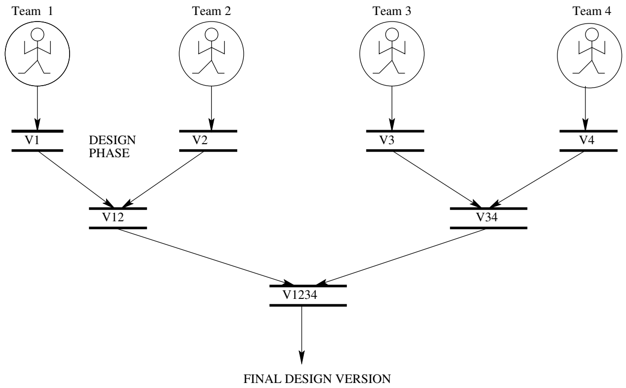


Figure 4. Consensus Design approach in software development

In addition to improved cost effectiveness, the consensus approach may be easier to implement. Software versions developed through the comparison approach may differ so significantly in terms of design and implementation, that compiling “the best" between all of them might be an elusive goal. Frequent comparisons in the consensus approach would smooth these differences throughout the stages of the pre-release life-cycle, while at the same time diversity is preserved wherever desired.

2.5. Pipeline approach

The pipeline approach makes the further trade-off between the high cost and the level of diversity. The complexity of software development suggests that all members of the team should concentrate on producing a single reliable version of the software.

Figure 5 depicts the development process of the pipeline approach. It is the actual configuration that was used in evaluating this approach using the automatic airplane-landing specification. During the design and coding phase, one team of 3 programmers was used. Each of the programmers was assigned different parts of specification with clearly defined interfaces. During the code review phase, 3 review teams, each consisting of 2 persons, were used. The final program was submitted for acceptance testing and the debugging was done by one person (who was also a member of one of the review teams).

It is obvious that the pipeline approach focuses on efficient in-process reviews. In our experiment, reviews were used at the level of code inspections, but it is easy to envi-

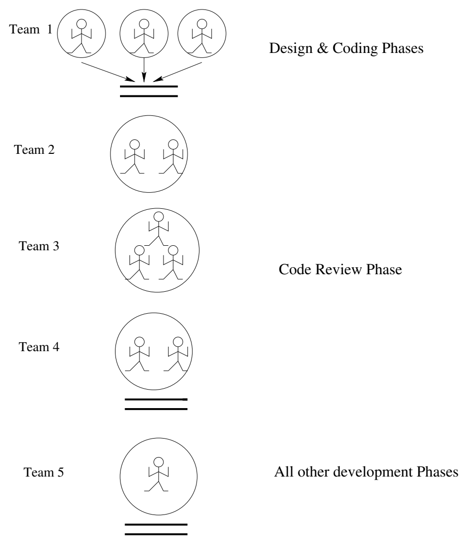


Figure 5. Pipeline approach in software development

sion the pipelined approach including software plans review, software requirements review, software design review, critical design review, code inspection and software test plan review.

3. The experiment set-up

In order to perform the steps of our pipeline paradigm, we selected the revised specification of a real, automatic airplane-landing problem. The development of a suitable specification for the “autopilot" began in 1987 and was used in the Six-Language Project UCLA/H6LP[2]. The automatic airplane-landing specification was finalized in 1991 [12].

3.1. Objective

Our objective was to find out what happens when there are diverse teams working on different phases of the development process according to the described pipeline paradigm. Airplane-landing specification was chosen because it enabled comparing the pipeline development method and its suitability for safety-critical applications with previously published experiences obtained through n-version programming. Based on the gained experience and comparison with the related project, we would like to become aware of drawbacks of the pipeline approach and make recommendations for conducting future experiments.

3.2. Application description

We describe briefly the automatic airplane-landing problem. Simulated flights begin with the initialization of the system in the Altitude Hold mode, at a point approximately 10 miles from the airport. Initial altitude is about 1500 feet, initial speed 120 knots (200 feet/second). The Complementary Filters preprocess the raw data from the aircraft sensors. Pitch-mode entry and exit is determined by the Mode Logic equations, which use the filtered airplane-sensor data to switch the controlling equations at the correct point in the trajectory.

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 Control Law is responsible for maintaining the reference altitude. As soon as the edge of a glide slope beam is reached, the airplane enters the Glide-Slope Capture and Track mode and begins a pitching motion (i.e., the aircraft's vertical motion) to acquire and hold the beam center. 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 2 feet/second at touchdown.

Besides computing the flight control command according to the above sequence, each program checks its final result (the pitch control command) against the results of other programs. Any disagreement is indicated by the Command Monitor output, so that a supervisory program can take an appropriate action. In Figure 6, the data flow of the “autopilot" is summarized.

The system consists of Complementary Filters, the Mode Logic, the actual Laws for each mode entered, the Command Monitors and the Display. There are 3 filters: the Barometric Altitude Complementary Filter, the Radio Altitude Complementary Filter (both provide an estimate of true altitude from the various altitude related signals), the Glide Slope Complementary Filter (provides estimates for beam error and radio altitude in the Glide Slope Capture and Track mode). The Mode Logic consists of logic statements that control the transition from one mode to the next, as a function of the model status and the magnitude of certain system variables. Mode logic provides the mode status for the flight control system. Using data from the Complementary Filters, the points in the landing trajectory at which mode transfer takes place are defined in terms of five states: Altitude Hold, Glide Slope Capture, Glide Slope Track, Flare, and Touchdown. There are 3 control laws. The Altitude Hold Mode Control Law that generates the pitch commands required to hold the aircraft at a reference altitude. The mode is entered when the

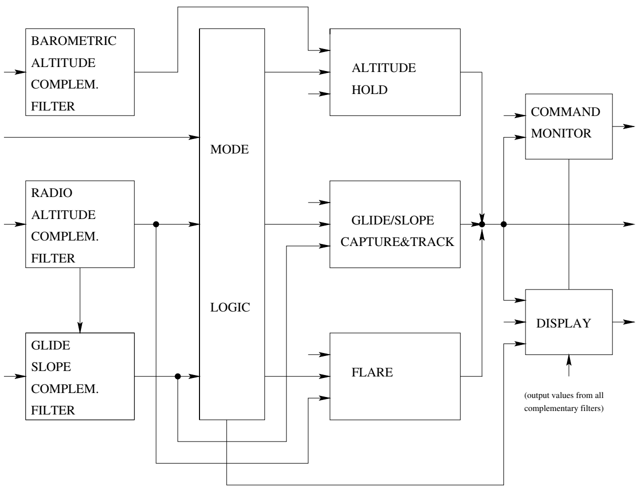


Figure 6. Submodules of the Lane Command Computation and Data Flow

aircraft is in level flight at an altitude of about 1,500 feet. It is the initial mode. The Glide Slope Capture and Track Control Law generates the pitch guidance commands required to maneuver the aircraft to capture and track the glide slope beam upon receipt of the capture discrete from the capture mode logic. The control law is designed to acquire the glide slope beam and hold the airplane on the beam in preparation for flare and touchdown. The Flare Control Law generates the pitch guidance commands required to flare the aircraft, upon generation of the flare discrete by the flare mode logic. The Flare Control Law is designed to bring the airplane to a suitable vertical speed at touchdown (landing). The Command Monitors, by comparing each locally computed lane command with the two others, provide the basic fault detection in critical flight control systems. The Display consists of 3 display panels which provide continuous information to the crew on autopilot modes, fault status, and signal levels for each lane.

4. Experimental application of pipeline development methodology

Before presenting the results of our experiment, we would like to introduce a similar project that used the design diversity paradigm. This is done for two reasons. The first one is that our program was subjected to the same test data as the other project. The second reason is that we can compare the number of faults found in our program during the development phases with the number of faults found in the other project.

4.1. Results from the related NVP experiment

In 1993, another project using the NVP approach [15] also applied the automatic airplane-landing specification [12]. It involved one faculty member and 40 students from the University of Iowa as well as researchers from Rockwell International. Guided by the refined NVS (N-*Version Software*) paradigm, the students were grouped into 15 independent programming teams to design, program, test, and evaluate the *pitch control* of the autopilot problem. The experiment was done in order to see what happens when many teams work independently to build NVS using the same specification.

The software development cycle was conducted in several software engineering phases, including the Initial Design, Detailed Design, Coding, Code Review, Unit Testing, Integration Testing, Acceptance Testing, and Operational Phase. Software testing was a major activity. In the Unit Testing (UT) Phase, each team received sample test data sets for each module (see Figure 6) to check its basic functionality. A total of 133 data files (roughly equivalent to one execution of the completely integrated program) was provided in this phase. In the Integration Testing (IT) Phase, four sets of partial flight-simulation test data, representing 960 complete program executions, were provided to each programming team. This phase of testing was intended to guarantee the software's suitability for a flight simulation environment in an integrated system.

Finally, in the Acceptance Testing (AT) Phase, programmers formally submitted their programs for an acceptance test. In the acceptance test, each program was run in a test harness of flight simulation profiles for both nominal and difficult flight conditions. When a program failed a test, it was returned to the programmers for debugging and resubmission, along with the input case on which it failed.

More than 21,000 different program executions were imposed on these programs before final acceptance. Twelve of the 15 programs passed the acceptance test and went to the Operational Testing(OT) Phase for further evaluations. Program size ranged from 900 to 4,000 uncommented lines of code, with an average of 1,550 lines.

4.2. Pipeline development process

The system described in Subsection 3.2 was divided between 3 programmers. The Complementary Filters and the Mode Logic were assigned to one programmer, the Control Laws were assigned to the second programmer, and the Command Monitor and Display to a third programmer. Each programmer was in charge of the design and implementation of these modules.

After each programmer finished coding, 3 independent teams performed the code review in a pipeline fashion, meaning that each team went through several iterations before passing the modified code to the next team.

In our experiment, each of the three programmers performed the Initial Design, Detailed Design, and Coding on the parts assigned to them. See Figure 6 for the data flow of the “autopilot". No Unit Testing or Integration Testing phases were carried out. Instead, the 3 review teams, in a pipeline fashion, did a code walkthrough in several iterations. Review team one went over the program twice, before passing it to review team two. Review team two performed 3 iterations before passing the code to review team three. Review team three went over the program once. After each iteration, the code was changed and the changes were verified for correctness.

Unfortunately, correctness verification steps performed by the review teams appear to have been carried out without proper attention. Somehow, after the review by team one, the verification of the changes was not made, and many of the same faults were found during the first iteration of team two. Also, it turns out that some of the changes done during the last iteration of team two introduced an additional fault that was caught during the review by team three. Also, the changes during the second iteration of team two were not checked, so a fault was introduced that was caught only later during the Integration Testing phase.

4.3. Fault classification

The following is a classification of the faults found during the development phases. These can be broadly described as being either *implementation related* or *specification related*. Implementation related faults are:

1. Typographical: a mistake made in typing the program, without violating language syntax.
2. Error of omission: a piece of required code was missing.
3. Incorrect algorithm: a deficient implementation of an algorithm; it includes miscomputation, logic fault, initialization fault, and boundary fault.

Specification related faults are:

1. Specification misinterpretation.
2. Specification ambiguity: an unclear or inadequate specification which led to a deficient implementation.

4.4. Faults detected in code inspections

As far as the faults found during each iteration of the review teams, none was of the specification related types and quite a few were of incorrect algorithm type. The remaining faults were mainly comment and type inconsistency errors. Listed below are the faults, their type, and the total number of faults found during the code review performed by the three teams. One comment about this table is the further classification of the Incorrect Algorithm faults. When the declared type of a variable was “double" and the variable was initialized to “0", that was counted as one fault. Also, in the same category are incorrect type declarations, such as “double" instead of “int".

Fault Type	Number of Faults
1. Typo	7
2. Omission	1
3a. Incorrect Algorithm	6
3b. Incorrect Algorithm(type)	6
4. Spec. Misinterpretation	0
5. Spec. Ambiguity	0
Total	20

Table 1. Number of Faults in Each Type

4.5. Faults detected in acceptance phase

After the code review by the three teams was completed, the program was submitted for Acceptance Testing (AT) Phase. Let us refer to this original submission as version 1 of the program. The program contained 2140 lines of code. During the Acceptance Testing phase, the program was subjected to 8 test data sets. The first 4 data sets, data1 through data4, were each 12 seconds long. The last 4 data sets were complete flight simulations of the following length: DATA1 was 264.10 seconds, DATA2 was 264.35 seconds, DATA3 was 264.35 seconds, and DATA4 was 264.35 seconds. When the program failed a test data, it was returned for debugging and resubmission, along with the input case on which it failed. Only one programmer worked on debugging and resubmission of the program. This programmer was not one of the three programmers that wrote the original program.

The time at which the program failed is in increments of the frame length, which in this case was set to 0.05 seconds.

- The first time the program failed was on test data data1 at time 0.05. It was determined after debugging that the module in which it failed was the Glide Slope Complementary Filter (refer to Figure 6). And the cause of failure was the way an integrator (I8 referring to the specification [12]) was initialized. This would qualify the fault as being of type 3, *incorrect algorithm*, (initialization fault, see the classification above). Rereading the specification, we thought that we fixed the initialization, so with that change, the program was resubmitted as version 2.

- The second time the program failed was again on test data data1 at time 0.05. It was determined after debugging that the program failed in the same manner as the previous submission. So, we determined that perhaps we had not understood the specification. After consultations, we decided that the specification was ambiguous as far as it concerned the initialization of Integrator I8. We then reclassified the fault as being of type 5, *specification ambiguity*, fixed the code and resubmitted the program as version 3.

- The third time the program failed was on test data data1 at time 0.10. It was determined after debugging that a variable that needed to keep an initialized value was erroneously declared local to the Altitude Hold Control Law (refer to Figure 6). This fault was classified as of type 3, *incorrect algorithm*. After fixing the program, it was submitted as version 4.

- The fourth time the program failed was on test data data2 (it passed on test data data1) at time 4.50. After debugging, it was determined that the cause of the fault was reversal of the order of the arguments in the mode logic (refer to Figure 6). This fault was classified as type 3, *incorrect algorithm* and, analyzing corrections made during code inspections, it was determined that it was introduced during the code review of the second team. It is not that the team review suggested this incorrect change, but it was the way the programmer carried out the suggested change. Another fault was also discovered in the Glide Slope Capture and Track Control Law. The variable that was used in determining when the switch SW3 was to be closed was implemented as type real instead of type integer (to count to the 11th frame instead of 0.5 seconds of 0.05 seconds per frame).Due to the wrong variable type, the test was missed. This fault was classified as type 4, *specification misinterpretation*. After the code was modified, the program was submitted as version 5.

- The fifth time the program failed on test data data2 at time 6.45. After debugging, two faults were found, both of which were classified as type 3, *incorrect algorithm*. The first fault was discovered in the the Altitude Hold Control Law where a variable was supposed to be declared as global (rather than local) to record past values. The second fault was discovered in the Glide Slope and Track Control Law where the initialization was done twice, once when entering Glide Mode and the second time when entering Track Mode. After fixing the code, the program was submitted as version 6.

- The sixth time the program failed was on test data DATA4 at time 4.20 (it passed data1, data2, data3, data4, DATA1, DATA2, and DATA3 test data files). After debugging, it was determined that in dealing with the Control Laws the specification was ambiguous. This type of fault was classified as type 5, *specification ambiguity*. After clarifying the intended meaning of the specification, the code was modified and submitted as version 7.

The seventh version of the program then passed all the test data files: data1, data2, data3, data4, DATA1, DATA2, DATA3, and DATA4. Then the program was subjected to Operational Testing Phase (OT) where the program was tested for 2500 landing simulations, which represented 5200X2500=13M program executions. No further faults were detected.

Fault Type	Number of Faults
1. Typo	0
2. Omission	0
3. Incorrect Algorithm	4
4. Spec. Misinterpretation	1
5. Spec. Ambiguity	2
Total	7

Table 2. Number of faults in each type during acceptance testing

4.6. Discussion

Listed above are the faults, their type, and the total number of faults detected during acceptance testing.

According to Table 2, faults of type 3, *Incorrect Algorithm*, are the most frequent.

Also, the number of faults found in each of the system functions is shown in Table 3. Please also refer to the automatic airplane-landing specification [12].

According to the data in Table 3, representing fault distributions in system functions, it it interesting to note that most of the faults reside in the Inner Loop function, which can be considered a tough spot for this project. Other tough spots occurred in GSCF, Mode Logic, AH Outer and GS Outer.

A very interesting comparison can be made between the number of faults found using our methodology versus the number of faults found in the 1993 12-version project (see

System Function	Number of Faults
Main controls the sequence of computations	0
BACF (RADIO ALTITUDE COMPLEM. FILTER)	0
RACF (RADIO ALTITUDE COMPLEM. FILTER)	0
GSCF (GLIDE SLOPE COMPLEM. FILTER)	1
MODE LOGIC	1
ALTITUDE HOLD outer loop	1
GLIDE/SLOPE CAPTURE and TRACK outer loop	1
FLARE outer loop	0
All 3 Control Laws inner loop	3
COMMAND MONITOR	0
DISPLAY	0
Total	7

Table 3. Fault distribution per system function

reference [15]). During the life cycle of that project, 96 faults were discovered. One version had a minimum of 5 faults and two versions had a maximum of 10 faults. The average number of faults was 8. Again faults of type 3, *Incorrect Algorithm*, were dominant.

Comparing the *number* of faults remaining in our version of the automatic airplane-landing program with the average of the 12 version experiment, it is not possible to conclude that the pipeline approach results in significantly better software. Nevertheless, it is beneficial to eliminate as many faults as possible through reviews and inspections since debugging during the testing phase is extremely time-consuming and laborious for complex process-control programs (see the following section). Also, all implementation ("incorrect algorithm") faults were detected by the first 4 data sets which together totaled just 48 seconds of execution time. No further implementation faults were detected in spite of 13M program executions. This implies that the size of the faults that escaped the multiple independent review process were relatively large and easily detectable by testing.

5. Lessons learned

We have presented a methodology to model and analyze some important trade-offs in the development of fault-tolerant software. This methodology is applied to one version of an n-version software process. Its main contribution is in the pipeline manner in which the design, coding, and code review phases are carried out. One conclusion we came up to is that a lot of faults were eliminated during code review. This is an important saving since testing takes a major part of time out of the whole project life cycle. Eliminating as many faults as possible will create a more reliable product. Since each member of the 3 code review teams got to follow the code based on the specification, the problem of misinterpreting the specification was diminished; this diversity of opinions on how to interpret the specification helped minimize this type of faults.

Once the program was submitted to Acceptance Testing (AT) Phase, it was submitted to nominal and difficult flight conditions. When the program failed, it was returned to us for debugging and resubmission, along with the input case on which it failed. Also, the failure occurred if any of the intermediate or output variables deviated from those of the "gold version" beyond the threshold. This made the debugging process very tedious. One has the inputs and the expected outputs up to the time frame when the program failed. Debugging has to be done over many frames verifying during each frame that the expected outputs and intermediate variables match. Looking back at the cause of some of the faults, it appears that these faults could have been easily discovered during a code review. We conclude then that it is harder to locate faults (i.e., to debug) during testing than during code review.

Applying disciplined practices in modern software engineering techniques during the software development is a must in creating ultra-reliable software. The role of observing that all the steps of the development are followed should be assigned to a person or a group that is not involved in any of the development phases. This authority would enforce that each program version be verified to comply with all the changes suggested during the code review. This is in our mind the most critical mistake we made during this project. We left the correction of detected faults and verification that the correction was performed correctly at the developer's discretion. Some of the suggested changes made during the code review were not done or were done incorrectly. If the verification would have been done, some faults could have been avoided.

6. Summary

We have looked at improving a version of an n-version software environment. The method's intention is to remove as many faults as possible before any kind of testing is done. We are basing our method on the assumption that it is easier and less time consuming to uncover faults during this phase than during the testing phase. Considering the lessons learned during the initial experiment, we believe that pipeline approach to incorporate diversity in software development is a viable complement to existing approaches. Further experiments are needed to obtain further data for quantitative assessment of this methodology.

References

[1] A. Avizienis and L. Chen, "On the implementation of n-version programming for software fault-tolerance during execution," *In Proc. Computer Software and Applications Conf.*, 1977, pp. 149-155.

[2] A. Avizienis, M.R. Lyu, and W. Schutz, "In search of effective diversity: A six-language study of fault-tolerant flight control software," *In Proc. 18th Annual Int. Symp. on Fault-Tolerant Computing*, 1988, pp. 15-22.

[3] A. Avizienis, "The methodology of n-version programming," In *Software Fault Tolerance*, (M.R. Lyu, ed.), John Wiley, 1995, pp. 23-46.

[4] S.S. Brilliant and J. Knight, "On the performance of software testing using multiple versions," *In Digest 20th Ann. Fault-Tolerant Computing* 1990, pp. 408-415.

[5] L. Chen and A. Avizienis, "N-version programming: A fault tolerance approach to reliability of software operation," *In Dig. Papers, FTCS-8*, 1978, pp. 3-9.

[6] E. Collins, L. Dalton, P. Perry, G. Polloc, C. Sicking, "A review of research and methods for producing high consequence software,' *In Proceedings of 1995 IEEE Aerospace Applications Conference*, Vol. 1, Aspen, CO, Feb. 4-11, 1995, pp. 197-245.

[7] H. Hecht and M. Hecht, "Fault-tolerant software," *Fault Tolerant Computing*, 1986, pp. 658-692.

[8] J.R. Horgan, S. London, and M.R. Lyu, "Achieving software quality with testing coverage measures," *In Computer* 1994, pp. 60-69.

[9] K.H. Kim and H.O. Welch, "Distributed execution of recovery block: An approach for uniform treatment of hardware and software faults in real-time applications," *In IEEE Transactions on Computers*, 1989, pp. 626-636.

[10] J. C. Knight, N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *In IEEE Transactions on Software Engineering*, Vol SE-12, No. 1, Jan 1986, pp. 96-109.

[11] B. Littlewood, "Learning to live with uncertainty on our software," *Proceedings of the* 2^{nd} *International Software Metrics Symposium*, London, UK, Oct. 1994.

[12] M.R. Lyu, "Software requirements document for a fault-tolerant flight control computer," *U. of Iowa ECE55:195 Project Specification*, 1991, 64 pages.

[13] M.R. Lyu, "Software reliability measurements in n-version software execution environment," *In Proc. Third Int. Symp. on Software Reliability Engineering*, 1992, pp. 254-263.

[14] M.R. Lyu, J.-H. Chen, and A. Avizienis, "Software diversity metrics and measurements," *In Proc. The Sixteen Annual Int. Computer Software and Applications Conf.* 1992, pp. 69-78.

[15] M.R. Lyu and Y.-T. He, "Improving the n-version programming process through the evolution of a design paradigm," *In IEEE Transactions on Reliability* 1993, pp. 179-189.

[16] M.R. Lyu, J.R. Horgan, and S. London, "A coverage analysis tool for the effectiveness of software testing," *In IEEE Transactions of Reliability*, 1994, pp. 527-535.

[17] B. Randell, "System structure for software fault tolerance," *In IEEE Trans. on Software Engineering*, 1975, pp. 220-232.

[18] B. Randell and J. Xu, "The evolution of the recovery block concept," In *Software Fault Tolerance*, (M.R. Lyu, Ed.), John Wiley, 1995, pp. 1-21.