Software Reliability Measurements in N-Version Software Execution Environment

Michael R. Lyu
Bellcore
Information Sciences and Technologies Research Lab.
Morristown, NJ 07962
(319)335-5957

Abstract

N-Version Programming has been proposed as a means to increasing software reliability by the attributes provided in fault tolerance. In this paper we will quantitatively examine the effectiveness of this approach. We look into the details of an academia/industry joint project employing six programming languages, and study the properties of the resulting program versions. We explain how exhaustive testing was applied to the project, and measure the error probability in different *N*-Version Software execution configurations. To explore the manifestations of errors resulting from each programming fault, and to study their impact to *N*-Version Software systems, we apply *mutation testing* techniques to gain more insights. We further define several reliability-related quantities, including error frequency function, error severity function, error similarity function, and safety coverage factor. With the multiple program versions obtained in the project, we create a total of 93 mutants, and measure the reliability quantities for each of them. Based on these quantities, we estimate the safety coverage factor for the N-Version Software system, which reveals the potential of this technique in improving software reliability. A side observation of this research is that the per fault error rate *does not* remain constant in this computation-intensive project. The error rates associated with each program fault differ from each other dramatically. However, they tend to decrease as testing progresses. This information might be helpful for practitioners to decide which models to apply for software reliability measurement in a similar project.

Keywords: Software Failure Data, N-Version Programming, Mutation Testing, Software Reliability, Coverage Analysis.

Software Reliability Measurements In N-Version Software Environment

1. Introduction

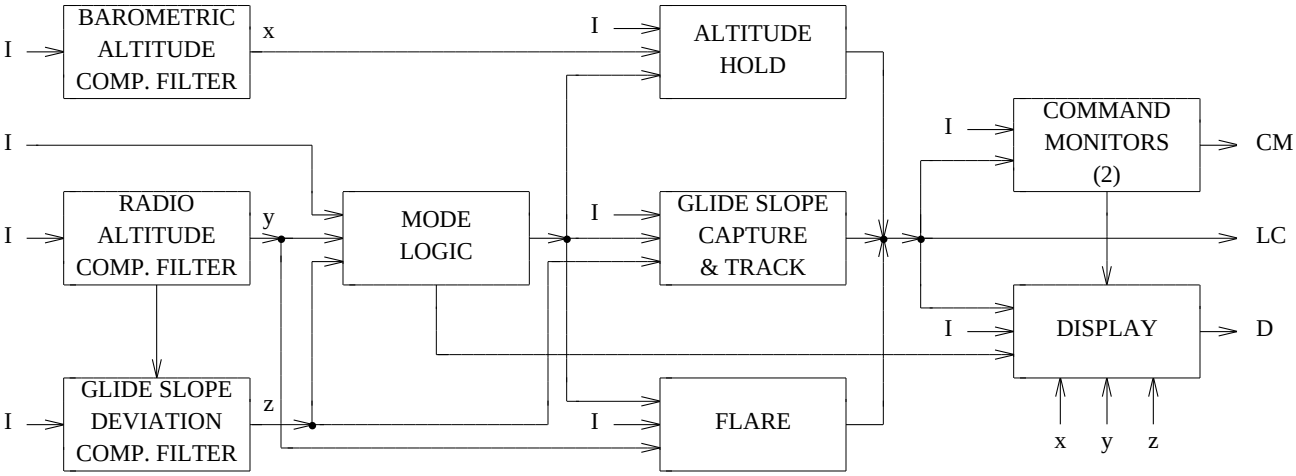
The *N*-Version Programming (NVP) approach to fault-tolerant software systems involves the generation of functionally equivalent, yet independently developed software components, called *N*-version software (NVS)[1]. These components are executed concurrently under a supervisory system that uses a decision algorithm based on consensus to determine final output values[2]. Whenever probability of similar errors is minimized, distinct, erroneous results tend to be masked by a majority vote during NVS execution[3].

NVS systems are gaining acceptance in critical application areas such as the aerospace industry[4] [5] [6] [7], nuclear power industry[8] [9] [10], and ground transportation industry[11] [12]. However, the evaluations of such systems, especially in the context of reliability-related measures, are still left as a controversial issue. There are many conflicting observations about the effectiveness of this fault-tolerance technique in increasing software reliability[13] [14]. In this paper, we will revisit the program versions obtained in a six-language project, using Ada, C, Modula-2, Pascal, Prolog, and T (a lisp dialect)[15], and evaluate the reliability aspects of the resulting NVS executions from various angles.

2. The Automatic Landing Problem

An industrial investigation concerning the effectiveness of the NVP process was conducted in an academia/industry joint project utilizing an NVP design paradigm[16]. The application was an automatic flight control function for the landing of commercial airliners that had been implemented by the Honeywell Incorporated 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 could be used to develop the software of a flight control computer (FCC, or *autopilot*) for a real aircraft, given that it was adjusted to the performance parameters of a specific aircraft. All algorithms and control laws in the application were specified by diagrams

which had been certified by the Federal Aviation Administration (FAA). The *pitch control* part of the automatic landing problem, i.e., the control of the vertical motion of the aircraft, was selected for the project. 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

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

The *Complementary Filters* preprocess the raw data from the aircraft's sensors. The *Barometric Altitude* and *Radio Altitude Complementary Filters* provide estimates of true altitude from various altitude-related signals, where the former provides the altitude reference for the

Altitude Hold mode, and the latter provides the altitude reference for the Flare mode. The *Glide Slope Deviation Complementary Filter* provides estimates for beam error and radio altitude in the Glide Slope Capture 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 an automatic *elevator* command 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 of each lane, or *land command*) against the results of the other programs. Any disagreement is indicated by the Command Monitor output, so that a supervisor program can take an appropriate action. The Display continuously shows information about the autopilot 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. In summary, this application could be classified as a computation-intensive, real-time system.

3. The NVS Error Detection and Recovery Algorithms

The original software specifications specified *test points*, i.e., selected intermediate values of each major system function that had to be provided as outputs for additional error checking. Error detection was easily identified from the test points with some modifications. A further enhancement to the application was the introduction of seven *cross-check points* (*ccp*'s)[1] [17], and a *recovery point*[18]. Error recovery could happen at each cross-check point, and at the recovery point before the result was fed back as input for the next computation. Cross-check points and recovery points were easily specified with their proper format and sequence. The usage of the ccp's and recovery point in this application program is shown in Figure 2.

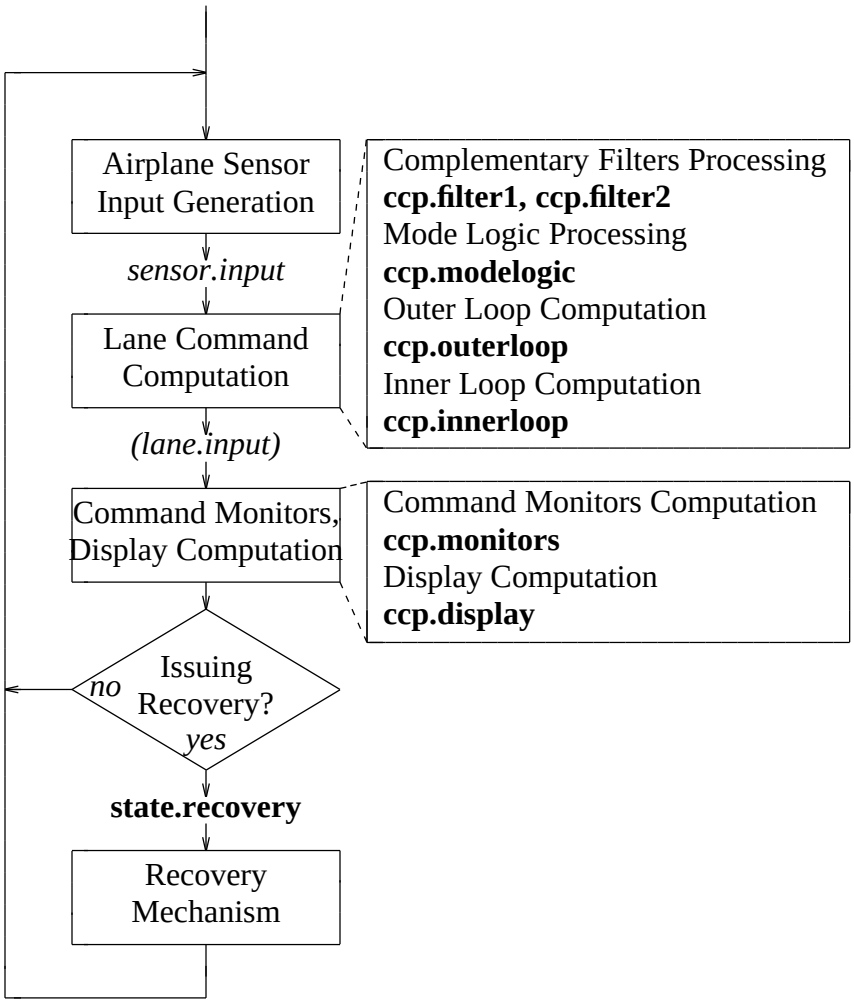


Figure 2: Usage of Cc-points and Recovery-point in the Application

In Figure 2, execution scenario of the application was designed to iterate through the following computations: 1) airplane sensor input generation; 2) lane command computation; 3) command monitors and display computation; and 4) recovery mechanism when necessary. The airplane simulator was separated designed by a coordinating team. Lane command, command monitors and display module were implemented by the programming teams. The recovery mechanism was provided in a supervisory environment[19] [20].

Under this scenario, the application software was instrumented by some fault tolerance mechanisms. Two input points (sensor.input, lane.input) were specified to receive external sensor input from an airplane, and to receive flight commends from the other channels (lanes). Moreover, seven cross-check points (ccp.filter1, ccp.filter2, ccp.modelogic, ccp.outerloop, ccp.innerloop, ccp.monitors, ccp.display) were used to cross-check the results of the major system functions (Complementary Filters, Mode Logic, Outer Loop, Inner Loop, Command Monitors, Display) with the results of the other versions before they were used in any further computation. Finally, One recovery point (state.recovery) was used to recover a failed version by supplying it with a set of new internal state variables that were obtained from the other versions by the *Community Error Recovery* technique[18].

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

4. Software Testing Conducted

To emphasize the importance of testing, three phases of testing: unit tests, integration tests, and acceptance tests, were conducted. Different strategies for program testing were provided in order to clean up programs. Table 1 lists the differences among these phases.

category	unit test	integration test	acceptance test
test criteria	structure-based	requirements-based	requirements-based
test case generator	open loop by PC Basic	closed loop by PC Basic	closed loop by multiple languages
test data access	file i/o by each version	interfacing C routines	interfacing C routines
tested by	individual teams	individual teams	coordinating team
tolerance level	0.01 for degrees	0.01 for degrees	0.005 for degrees (0.05 for Prolog)
# of test cases	133	4	9
total executions	1330	960	18,440

Table 1: Different Schemes Used in the Testing Phases

The unit test was considered as structure-based test since its test data was provided in such a way to facilitate programmers in hand-tracing the execution of their program units. The integration test and acceptance test, on the other hand, utilized purely requirements-based test data. A reference model of the control laws was implemented and provided by Honeywell Inc. This version was implemented in Basic on an IBM PC to serve as the test case generator for unit tests and integration tests. Criteria of "open loop testing" and "closed loop testing" were used, respectively. Due to the wide numerical discrepancies between this version and the other six program versions under development, a larger tolerance level was chosen.

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

the lack of accuracy in its internal representation of real numbers.

In the unit test phase, each module of the program received a variant number of test cases. A total of 133 test cases were executed, and since each test case contained 10 program executions, there were 1330 executions in this phase.

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

The acceptance test was a stringent testing phase. Nine testing profiles were designed for this test phase. Data sets 1, 3, and 5 executed for 100 seconds, driving the airplane from Altitude Mode to Glide Slope Capture and Track Modes with different magnitude of wind turbulences. Data sets 2, 4, and 6 were designed similarly, except they exercised all five flying modes for 180 seconds. Data sets 7 and 8 were designed to carry out the recovery command, and Data set 9 the Display module. The total executions required in this phase were 18440 program iterations.

5. Software Metrics and Fault Distributions of the Programs

Table 2 gives several comparisons of the six versions with respect to some common software metrics. The objective of software metrics is to evaluate the quality of the product in a quality assurance environment. However, our focus here is the comparison among the program versions, since design diversity is our major concern.

The following metrics are considered in Table 2: (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); (6) the mean number of statements in between cross-check points (STM/CCP); (7) the number of calls to

programming modules (CALLS); (8) the number of global variables (GBVAR); (9) the number of local variables (LCVAR); and (10) the number of binary decisions (BINDE).

Metrics	ADA	C	MOD-2	PASCAL	PROLOG	T	Range
LINES	2253	1378	1521	2234	1733	1575	1.64:1
LN-CM	1517	861	953	1288	1374	1263	1.76:1
STMTS	1031	746	546	491	1257	1089	2.56:1
MODS	36	26	37	48	77	44	2.96:1
STM/M	29	25	15	10	16	25	2.90:1
STM/CCP	147	107	78	70	179	156	2.56:1
CALLS	97	68	65	93	81	87	1.49:1
GBVAR	139	141	91	81	90	97	1.74:1
LCVAR	117	197	132	127	209	251	2.15:1
BINDE	74	114	78	118	74	86	1.59:1

Table 2: Software Metrics for the Six Programs

A total of 93 faults was found and reported during the whole life cycle of the project. Table 3 shows the test phases during which the faults were detected, and the fault densities (as per thousands of executable statements) of the original version and the accepted version. It also shows test efficiency for fault removal during software development, defined as the number of faults found prior to operational testing divided by the number of total faults.

Test Phase	ADA	C	MOD-2	PASCAL	PROLOG	T	Total
Coding/Unit Testing	2	4	4	10	15	7	42
Integration Testing	2	5	0	2	7	4	20
Acceptance Testing	2	4	0	0	4	10	20
Operational Testing	0	5	1	0	3	2	11
Total	6	18	5	12	29	23	93
Original Fault Density	5.8	24.2	9.2	24.4	23.1	21.1	18.0
Test Efficiency	100%	72%	80%	100%	90%	91%	88%
Operational Fault Density	0.0	6.7	1.8	0.0	2.4	1.8	2.1

Table 3: Fault Classification by Phases and Other Attributes

It is interesting to note that there was *only one* incidence of an identical fault, committed by two teams, Ada and Modula-2, during program development phase. This fault was due to a comma being misread as a period, which was classified as a specification misinterpretation. During operational testing, two disagreements were traced to an identical fault occurred in the Prolog and T versions, which was due to the programmers' ignorance to properly incorporate a late specification update. Both pairs of identical faults were related to specification. However, identical faults involving more than two versions have never been observed.

6. *N*-Version Software System Executions and Results

The operational testing of the project was a dynamic, requirements-based testing stressing how the airplane/autopilot interacted and operated in an actual environment. Three channels of diverse software each computed 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 were superimposed in order to represent difficult flight conditions. The individual commands were recorded and compared for discrepancies that could indicate the presence of faults.

The configuration of the flight simulation system (shown in Figure 3) was consist of three lanes of control law computation, three command monitors, a servo control, an Airplane model, and a turbulence generator.

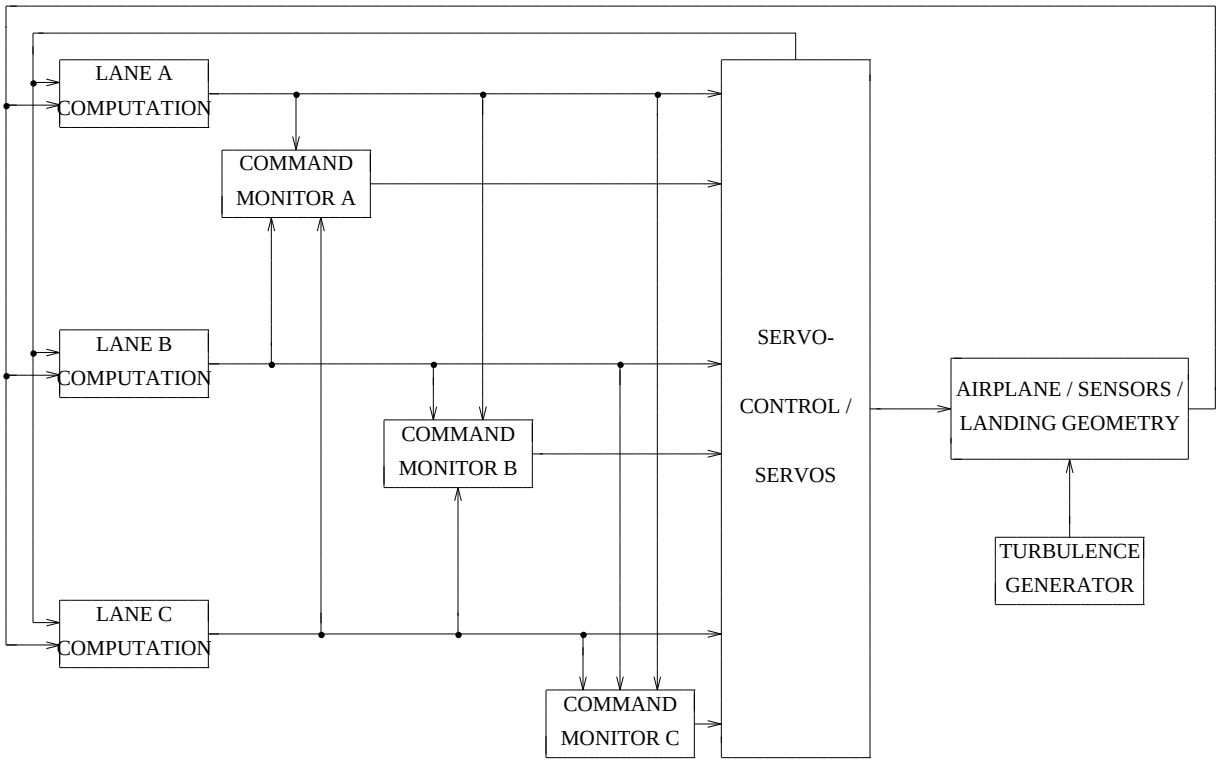


Figure 3: 3-Channel Flight Simulation Configuration

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

The aircraft mathematical model provided the dynamic response of current medium size, commercial transports in the approach/landing flight phase. In this model, the airplane was trimmed, by means of flaps and engine thrust setting, to a landing speed of about 120 knots, and a level of low altitude flight path was chosen. It was assumed that the pilot has engaged the autothrottle while in the low-speed, approach condition, so that forward speed variations were

negligible. In addition, the pilot was assumed to have maneuvered the aircraft to an altitude of 1500 feet in preparation for engagement of the Altitude Hold mode.

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

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

Landing Geometry and *Turbulence Generator* were models associated with the Airplane simulator. The Landing Geometry mathematical model described the deviation from glideslope beam center as a function of aircraft position relative to the end of the runway. Nominal glideslope beam characteristics were defined by a beam angle of 2.5 degrees referenced to the touchdown point, and a width in the vertical plane of ± 0.5 degrees around the beam center. Interception of the lower edge of the beam by the aircraft resulted in a pitch-over command and acquisition of the beam center. The Turbulence Generator model was used to introduce vertical wind gusts. Vertical turbulence was assumed to be frozen with respect to time. This was based on the observation that for reasonable flight speeds, changes in vertical wind velocity were smaller with respect to time than with respect to position.

In summary, one run of flight simulation was 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 consisted of about 5000 iterations of lane command computations (50 milliseconds each) for a total landing time of approximately 250 seconds. In this manner, over 1000 flight simulations (over 5,000,000 program executions) were exercised on the six software versions generated from this

project.

Table 4 shows the errors encountered in each single version, while Table 5 shows different error categories under all combinations of 3-version and 5-version configurations. Note that the discrepancies encountered in the operational testing were called "errors" rather than "failures" due to their non-criticality in the landing procedure, i.e., a proper touchdown was still achieved.

version	size (l.o.c.)	total executions	number of errors	error probability
Ada	2256	5127400	0	.0000000
C	1531	5127400	568	.0001108
Modula-2	1562	5127400	0	.0000000
Pascal	2331	5127400	0	.0000000
Prolog	2228	5127400	680	.0001326
T	1568	5127400	680	.0001326
Average	1913	5127400	321	.00006267

Table 4: Errors in Individual Versions

cate-	3-version configuration		5-version configuration	
gory	# of cases	probability	# of cases	probability
1.	102531685	.9998409	30757655	.9997807
2.	13385	.0001305	5890	.0001915
3.	210	.000002048	70	.000002275
4.	2720	.00002652	680	.00002210
5.	-	-	105	.000003413
Total	102548000	1.0000000	30764400	1.0000000

classifications of the category:
1 - no errors
2 - single errors in one version
3 - two distince errors in multiple versions
4 - two coincident errors in multiple versions
5 - three errors in multiple versions

Table 5: Errors in 3-Version and 5-Version Execution Configurations

From Table 4 we can see that the average error probability for single version is .00006267.

Table 5 shows that for all the 3-version combinations, the error probability concerning reliability is .00002857 (categories 3 and 4), and that for safety is .00002652 (category 4). This is a reduction of roughly 2.3. In all the combinations of 5-version configuration, the error probability for reliability is .000003413 (category 5; Two of the three errors are coincident, resulting in no-decision), a reduction by a factor of 18. This probability becomes zero in the safety measurement.

From these numbers it might be interpreted that the expected dependability improvement of NVS over single-version software is quite moderate. However, it is noted that the coincident errors produced by the Prolog and T programs were all caused by *one identical fault* in both versions, which was due to the ignorance of a slight specification update that was made very late in the programming process. This fault manifested itself right after these program versions were put together for the flight simulation. Had this fault been eliminated in the operational testing, categories 3, 4 and 5 for both 3-version and 5-version configurations in Table 5 would have been all zero, resulting in perfect dependability figures.

7. Fault Diagnosis and Error Analysis by Mutation Testing

To uncover the impact of faults that would have remained in the software version, and to evaluate the effectiveness of NVS mechanisms, a special type of regression testing, similar to *mutation testing* which is well known in the software testing literature[21] [22], was investigated in the six versions. The original purpose of the mutation testing is to ensure the quality of the test data used to verify a program, while our concern here was to examine the relationship of faults and error frequencies in each program and to evaluate the similarity of program errors among different versions. The testing procedure is described in the following steps: 1) The fault removal history of each program was examined and each program fault was analyzed and recreated. 2) Mutants were generated by injecting faults one by one into the final version from where they were removed (i.e., a fault from the C program will be injected to the C program only). Each mutant contains *exactly one* known software fault. 3) Each mutant was executed by the same set

of input data in the Airplane simulation environment to observe errors. 4) Analyze the error characteristics to collect error statistics and correlations.

Using the fault removal history of each version, we created 6 mutants for Ada (a1 - a6), 18 mutants for C (c1 - c18), 5 mutants for Modula-2 (m1 - m5), 12 mutants for Pascal (p1 - p12), 29 mutants for Prolog (pg1 - pg29), and 23 mutants for T (t1 - t23). A higher index number in each mutant represents the injection of a later fault to that version. In order to present the execution results of the above procedure, let us define the following two functions for each mutant:

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

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

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

Since each mutant contains only one known fault, it is hypothesized that errors produced by that fault are always the same for the same test inputs[23]. This hypothesis is considered valid for all the mutants discussed here. Therefore, we can define an Error Severity Function of version x mutant i for test set τ, $\mu(x_i,\tau)$, to be

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

where ε is a specified allowed deviation. If x_i produces run-time exceptions or no results, then

$\mu(x_i, \tau)$ is defined to be 1.

The Error Frequency Function and Error Severity Function applied to each mutant (for a test set of about 15000 executions) are shown in Table 6 and Table 7, respectively.

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

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

Table 6: Error Frequency Function of Each Mutant

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

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

Table 7: Error Severity Function of Each Mutant

From Table 6 we can observe that the error frequency associated with each fault is *not* a constant. The existence of a constant error rate per fault has been a major assumption of many

software reliability models[24] [25] [26] [27]. Table 6, on the other hand, suggests that the per fault error rate is not constant. In fact, this rate tends to decrease as the testing proceeds, suggesting that frequently manifested errors will be detected earlier. This phenomenon suits the assumptions in some other reliability models[28] [29]. However, the exact relationship of the fault sequence and the associated error rates, which has been specified by these models, could not be confirmed in Table 6. Table 7 shows an even more random behavior of the error severity versus testing time, suggesting hard-to-detect errors might, or might not, imply high severity.

As to the nature of errors in two or more versions, three types of relationships are identified: *distinct errors*, *similar errors*, and *identical errors*[30]. Distinct errors are produced by faults whose erroneous results could be distinguished from one another. Similar errors are defined to be two or more results that are within a small range of variation, and the results are erroneous. If the results of the similar errors are identical, they are called identical errors.

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

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

Based on these definitions, we have obtained the Error Similarity Functions for populations of two versions. Table 8 shows the Error Similarity Function matrix for two-mutant sets. The complete layout of this matrix is 93 by 93, but since it is a sparse matrix (most entries are zero), we can reduce it by removing many of the zero entries. In fact, the two incidences of indistinguishable errors shown in this table result from the two pairs of identical faults discussed before.

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

Table 8: Reduced Error Similarity Function Matrix in Two-Mutant Sets

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

Based on the above measures, we can obtain another reliability-related quantity, *safety coverage*, which is important for assessing the effectiveness of fault-tolerant systems. Safety coverage factor is defined as the conditional probability of successful error detection or recovery, given that a fault has manifested itself in the system[31]. In NVS systems, the safety coverage factor depends on the similarity of errors, the severity of errors, and the efficiency of the recovery mechanisms to cope with such errors. Thus, we need to derive a quantitative definition of it for measurement.

Since our main interest here is the analysis of the program versions themselves, without loosing generality, let us assume the NVS supervisory system does not introduce extra errors which corrupt program executions. As a result, the main contribution of the "leak" of the NVS schemes would be the error similarity defined previously. Now we may use the mutants we generated early as the sampling space to represent the condition that a fault has been created. Since this fault is assumed to be manifested into errors during program executions, the probability that it would be covered (or tolerated) is related to its error severity and its similarity to errors of other program versions. Consequently, the safety coverage factor of an n-mutant system (out of a total

population m) with respect to the test set τ, denoted $C_n(\tau)$, could be defined as:

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

For example, $C_1(\tau)$ represents the safety coverage factor of single-version software:

$$\begin{aligned} C_1(\tau) &= 1 - \frac{1}{C(93,1)} \sum_{x_1=1}^{x_1=93} \sigma(x_1;\tau) * \mu(x_1,\tau) \\ &= 1 - \frac{1}{93} \sum_{x_1=1}^{x_1=93} \mu(x_1,\tau) \\ &= 0.504 \end{aligned}$$

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

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

This indicates an enormous improvement of 2-version software over single-version software. Other higher-order C_n's could be computed similarly, and their effectiveness could also be expected. Such coverage factors can refer to a representative measure of the situations to which the system is submitted during its validation with respect to the actual situations it will be confronted with during its operational life. It is important to note that the coverage defined and measured here is limited to the particular mutant population and the specific test data set. We might argue that most of these mutants would have been killed by a normal testing procedure. However, there is always a non-zero probability for each of these mutants to slip through all practically applied testing schemes in another environment. As a result, sampling from this population is still valid, and they are useful to provide an evidence for the effectiveness of NVS methodology to the assigned application.

8. Conclusions

We have measured several software reliability quantities in an *N*-Version Software environment. The selected application was a real-world avionics control system which performs tedious computation-intensive tasks under severe timing constraints. We described the testing configurations, program properties, and operational testing results in detail. We also applied the idea of mutation testing to measure several other reliability quantities. The difference between the mutation technique applied here and its original usage is two-fold: First, we used *real* mutants, that is, mutants injected with actual faults committed by programmers, instead of mutants with *hypothesized* faults. Secondly, our purpose was to measure the coverage of *N*-Version Software in masking mutants during operation, not merely the coverage of the test data in detecting mutants during testing. In fact, when there are multiple realizations of the same application, test data is no longer the only means for fault treatment and coverage analysis. Study of the error correlations among multiple program versions offers another dimension of investigation in mutation testing. In this mutation investigation, several reliability-related quantities were defined and measured. From these measurements we demonstrated the potential capability of N-Version Software system in improving software reliability.

The observation that the per fault error rate does not remain constant in this project might be due to the nature of the project: a computation-oriented application, a large volume of critical values involved in the system, and the imposed intensive-testing process. Obtaining this kind of information, nevertheless, is useful for software reliability practitioners to determine, in advance, which models are mostly applicable to their projects.

References

1. A. Avižienis and L. Chen, ‘‘On the Implementation of N-Version Programming for Software Fault-Tolerance during Program Execution,’’ in *Proceedings COMPSAC 77*, pp. 149-155, 1977.

2. A. Avižienis, P. Gunningberg, J.P.J. Kelly, M.R. Lyu, L. Strigini, P.J. Traverse, K.S. Tso, and U. Voges, ‘‘Software Fault-Tolerance by Design Diversity; DEDIX: A Tool for Experiments,’’ *Proceedings IFAC Workshop SAFECOMP’85*, pp. 173-178, Como, Italy, October 1985.

3. A. Avižienis, ‘‘The N-Version Approach to Fault-Tolerant Software,’’ *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, pp. 1491-1501, December 1985.

4. D.J. Martin, ‘‘Dissimilar Software in High Integrity Applications in Flight Controls,’’ in *Proceedings AGARD-CPP-330*, pp. 36.1-36.13, September 1982.

5. L.J. Yount, ‘‘Architectural Solutions to Safety Problems of Digital Flight-Critical Systems for Commercial Transports,’’ *Proceedings AIAA/IEEE Digital Avionics Systems Conference and Technical Display*, pp. 1-8, Baltimore, Maryland, December 1984.

6. A.D. Hills, ‘‘Digital Fly-By-Wire Experience,’’ *Proceedings AGARD Lecture Series*, no. 143, October 1985.

7. J.C. Rouquet and P.J. Traverse, ‘‘Safe and Reliable Computing on Board the Airbus and ATR Aircraft,’’ *Proceedings SAFECOMP ’86*, pp. 93-97, Sarlat, France, October 1986.

8. C.V. Ramamoorthy, Y. Mok, F. Bastani, G. Chin, and K. Suzuki, ‘‘Application of a Methodology for the Development and Validation of Reliable Process Control Software,’’ *IEEE Transactions on Software Engineering*, vol. SE-7, no. 6, pp. 537-555, November 1981.

9. P. Bishop, D. Esp, M. Barnes, P. Humphreys, G. Dahl, J. Lahti, and S. Yoshimura, ‘‘Project on Diverse Software - An Experiment in Software Reliability,’’ *Proceedings IFAC Workshop SAFECOMP’85*, Como, Italy, October 1985.

10. U. Voges, ‘‘Application of a Fault-Tolerant Microprocessor-Based Core-Surveillance System in a German Fast Breeder Reactor,’’ *EPRI-Conference*, Phoenix, Arizona, April 9-12 1985.

11. R. Taylor, ‘‘Redundant Programming in Europe,’’ *ACM SIGSOFT Software Engineering Notes*, vol. 6, no. 1, January 1981.

12. G. Hagelin, ‘‘ERICSSON Safety System for Railway Control,’’ in *Software Diversity in Computerized Control Systems*, ed. U. Voges, pp. 11-21, Springer-Verlag/Wien, Austria, 1988.

13. D.E. Eckhardt, A. Caglavan, J.C. Knight, L.D. Lee, D.F. McAllister, M.A. Vouk, and J.P.J. Kelly, ‘‘An Experimental Evaluation of Software Redundancy as a Strategy For Improving Reliability,’’ *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 692-702, July 1991.

14. 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, pp. 96-109, January 1986.

15. A. Avižienis, M.R. Lyu, and W. Schütz, ‘‘In Search of Effective Diversity: A Six-Language Study of Fault-Tolerant Flight Control Software,’’ *Proceedings 18th Annual International Symposium on Fault Tolerant Computing*, Tokyo, Japan, June 27-30 1988.

16. M.R. Lyu and A. Avižienis, ‘‘Assuring Design Diversity in N-Version Software: A Design Paradigm for N-Version Programming,’’ in *Proceedings 2nd International Working Conference on Dependable Computing for Critical Applications*, pp. 89-98, Tucson, Arizona, February 1991.

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

18. 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*, pp. 127-133, Pittsburgh, Pennsylvania, July 1987.

19. A. Avižienis, M.R. Lyu, W. Schütz, K. Tso, and U. Voges, ‘‘DEDIX 87 - A Supervisory System for Design Diversity Experiments at UCLA,’’ CSD-870029, Computer Science Department, UCLA, Los Angeles, California, July 1987.

20. A. Avižienis, M.R. Lyu, W. Schütz, K. S. Tso, and U. Voges, ‘‘DEDIX 87 - A Supervisory System for Design Diversity Experiments at UCLA,’’ in *Software Diversity in Computerized Control Systems*, ed. U. Voges, pp. 129-168, Springer-Verlag/Wien, Austria, 1988.

21. T.A. Budd, R.J. Lipton, F.G. Sayward, and R.A. DeMillo, ‘‘The Design of a Prototype Mutation System for Program Testing,’’ in *Proceedings NCC*, pp. 623-627, 1978.

22. W.E. Howden, ‘‘Weak Mutation Testing and Completeness of Test Sets,’’ *IEEE Transactions on Software Engineering*, vol. SE8, no. 4, pp. 371-379, July 1982.

23. J. D. Musa, A. Iannino, and K. Okumoto, *Software Reliability – Measurement, Prediction, Application*, McGraw-Hill Book Company, New York, New York, 1987.

24. Z. Jelinski and P.B. Moranda, ‘‘Software Reliability Research,’’ in *Statistical Computer Performance Evaluation*, ed. W. Freiberber, pp. 465-484, Academic, New York, 1972.

25. M. Shooman, ‘‘Operational Testing and Software Reliability During Program Development,’’ in *Proceedings 1973 IEEE Symposium on Computer Software Reliability*, pp. 51-57, New York, April 1973.

26. N.F. Schneidewind, ‘‘Analysis of Error Processesin Computer Software,’’ in *Proceedings International Conference on Reliable Software*, pp. 337-346, Los Angeles, 1975.

27. A.L. Goel and K. Okumoto, ‘‘Time-Dependent Error-Detection Rate Model for Software Reliability and Other Performance Measures,’’ *IEEE Transactions on Reliability*, vol. R-28, pp. 206-211, 1979.

28. J.D. Musa and K. Okumoto, ‘‘A Logarithmic Poisson Execution Time Model for Software Reliability Measurement,’’ in *Proceedings Seventh International Conference on Software Engineering*, pp. 230-238, Orlando, Florida, 1984.

29. B. Littlewood and J.L. Verrall, ‘‘A Bayesian Reliability Growth Model for Computer Software,’’ *Journal Royal Statistics Society C*, vol. 22, pp. 332-346, 1973.

30. A. Avižienis and J.-C. Laprie, ‘‘Dependable Computing: From Concepts to Design Diversity,’’ *Proceedings of the IEEE*, vol. 74, no. 5, pp. 629-638, May 1986.

31. W.G. Bouricius, W.C. Carter, and P.R. Schneider, ‘‘Reliability Modeling Techniques for Self-Repairing Computer Systems,’’ in *Proceedings 24th National Conference of the ACM*, pp. 295-383, 1969.