Software Reliability Measurements in N-Version Software Execution Environment

Michael R. Lyu
Bellcore
Information Sciences and Technologies Research Lab.
Morristown, NJ 07962
<lyu@bellcore.com>

Abstract

In this paper we quantitatively examine the effectiveness of the N-Version Programming 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. We also apply mutation testing techniques to measure the safety coverage factor for the N-Version Software system. Results from this investigation reveals the potential of N-Version Software in improving software reliability. Another 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, and they tend to decrease as testing progresses.

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). These components are executed concurrently under a supervisory system that uses a decision algorithm based on consensus to determine final output values. Whenever probability of similar errors is minimized, distinct, erroneous results tend to be masked by a majority vote during NVS execution[1].

NVS systems are gaining acceptance in critical application areas such as the aerospace industry, nuclear power industry, and ground transportation industry. 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[2] [3] [4] [5]. 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)[6], and evaluate the reliability aspects of the resulting NVS executions from various angles.

2. The Application and Its Instrumentation

An industrial investigation concerning the effectiveness of the NVP process was conducted in an industry/academia joint project[6]. The application was an automatic flight control function ("*autopilot*") that had been implemented by the Honeywell Incorporated for the landing of commercial airliners. All algorithms and control commands in the application were specified by diagrams which had been certified by the Federal Aviation Administration (FAA). Details of this project could be found in the technical report[7].

For the purpose of providing reliable operation during software execution, it was critical that the application be instrumented by error detection and recovery mechanisms. This is shown in Figure 1.

In Figure 1, 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 separately 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.[8].

0-8186-2975-4/92 $03.00 © 1992 IEEE

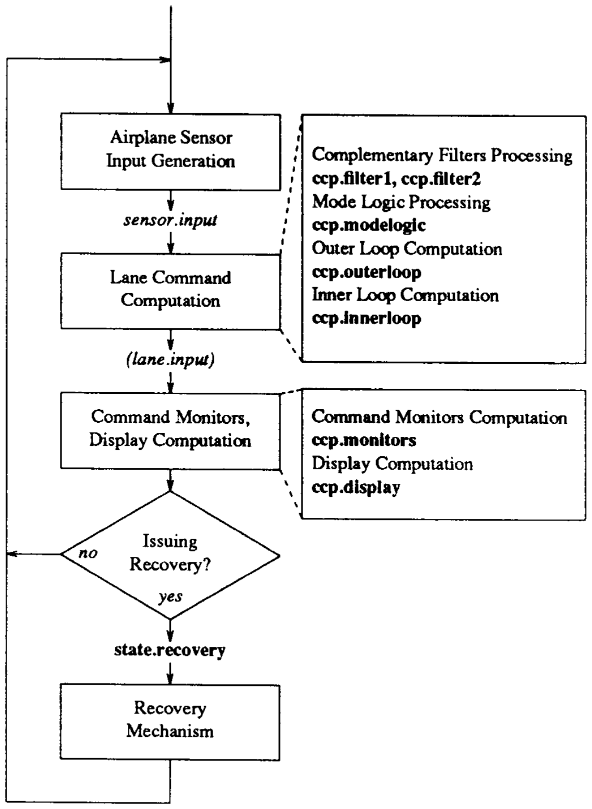

Figure 1: Usage of Cc-points and Recovery-point

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[9].

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 the application. State variables were identified as those variables whose values in the current iteration would affect their new values in the next iteration.

3. 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	acceptance
test criteria	structure-based	requirements-based	requirements-based
test case generator	open loop, PC Basic	closed loop, PC Basic	closed loop, multiple languages
test data access	individual file i/o	interfacing C routines	interfacing C routines
testers	programmers	programmers	coordinators
tolerance level	0.01	0.01	0.005
test cases	133	4	9
total runs	1330	960	18,440

Table 1: Different Schemes in 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 autopilot 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.

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
Inherent 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 2: Fault Classification by Phases and Other Attributes

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. In the integration test, four testing profiles were provided. Each test profile contained 12 seconds of flight simulation, a total of 960 executions. The acceptance test was a stringent testing phase. Nine testing profiles, representing flight situation with different modes and different magnitude of wind turbulences, were designed for this test phase. The total executions required in this phase were 18440 program iterations.

4. Fault Distributions among the Programs

A total of 93 faults was found and reported during the whole life cycle of the project. Table 2 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.

It is noted 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.

The diversity effect of programming language could also be seen from Table 2. The programming languages of "object-oriented" flavor (Ada, Modula-2) seem to inherit less faults than other programming languages. However, it is unclear whether this new programming style encourages or discourages error detection. It is noted that, since there is only one version per programming language, these observations are not meant to be conclusive.

5. N-Version Software System Executions and Results

The operational testing of this project was a dynamic, requirements-based testing stressing how the airplane/autopilot interacted and operated in an actual environment. Three or five 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.

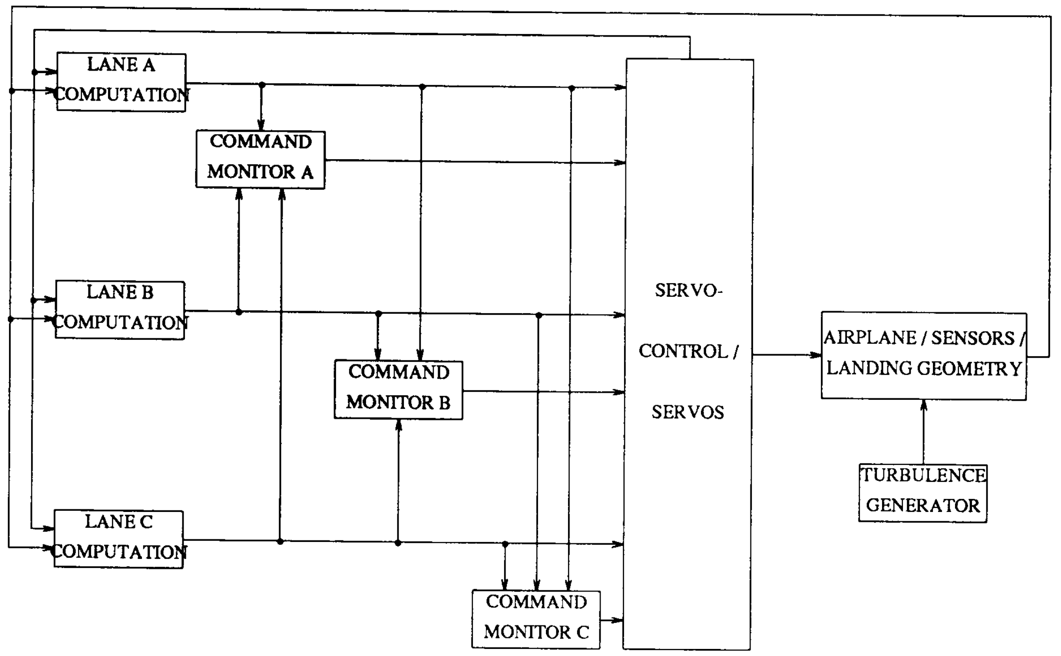

Figure 2: 3-Channel Flight Simulation Configuration

The configuration of the flight simulation system, shown in Figure 2, was consist of three lanes of control law computation, three command monitors, a servo control, an Airplane model, and a turbulence generator.

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

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. The Turbulence Generator model was used to introduce vertical wind gusts.

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 3 shows the errors encountered in each single version, while Table 4 shows different error categories under all combinations of 3-version and 5-version configurations[10]. 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	total executions	# of errors	error prob.
Ada	5127400	0	.0000000
C	5127400	568	.0001108
Mod-2	5127400	0	.0000000
Pascal	5127400	0	.0000000
Prolog	5127400	680	.0001326
T	5127400	680	.0001326
avg.	5127400	321	.00006267

Table 3: 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 4: Errors in 3-Version and 5-Version Execution Configurations

From Table 3 we can see that the average error probability for single version is .00006267. Table 4 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 4 would have been all zero, resulting in perfect dependability figures.

6. 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[11] [12], 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

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

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[13]. 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

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

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 5 and Table 6, respectively.

From Table 5 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[14] [15] [16]. Table 5, 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[17] [18]. However, the exact relationship of the fault sequence and the associated error rates, which has been specified by these models, cannot be confirmed in Table 5.

id	Ada	C	Mod-2	Pascal	Prolog	T
1	.0002	1	0	1	0	1
2	.001	.0037	.001	0	.0006	1
3	1	.5	.005	0	1	1
4	1	.0001	0	.0001	1	1
5	.0001	.0037	0	1	.0005	1
6	0	0	–	.002	1	0
7	–	.001	–	.001	1	1
8	–	0	–	.001	.1	1
9	–	0	–	.0001	.0005	1
10	–	0	–	0	1	.5
11	–	.0005	–	.001	1	1
12	–	0	–	.0002	1	0
13	–	0	–	–	1	0
14	–	0	–	–	.0001	0
15	–	.03	–	–	1	.002
16	–	0	–	–	1	0
17	–	0	–	–	1	.0028
18	–	0	–	–	0	0
19	–	–	–	–	1	0
20	–	–	–	–	1	0
21	–	–	–	–	1	0
22	–	–	–	–	1	.001
23	–	–	–	–	.0001	0
24	–	–	–	–	0	–
25	–	–	–	–	.0001	–
26	–	–	–	–	0	–
27	–	–	–	–	.001	–
28	–	–	–	–	0	–
29	–	–	–	–	0	–

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

Table 5: Error Frequency Function of Each Mutant

Table 6 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*[19]. 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.

id	Ada	C	Mod-2	Pascal	Prolog	T
1	1	.025	0	1	0	1
2	.51	1	.51	0	1	1
3	1	1	1	0	1	1
4	1	1	0	.03	1	.017
5	1	.05	0	1	1	1
6	0	0	–	.017	1	0
7	–	1	–	1	1	1
8	–	0	–	.004	.1	1
9	–	0	–	1	.038	1
10	–	0	–	0	1	1
11	–	.001	–	.23	1	1
12	–	0	–	1	1	0
13	–	0	–	–	1	0
14	–	0	–	–	.022	0
15	–	.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	.02
23	–	–	–	–	1	0
24	–	–	–	–	0	–
25	–	–	–	–	1	–
26	–	–	–	–	0	–
27	–	–	–	–	.02	–
28	–	–	–	–	0	–
29	–	–	–	–	0	–

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

Table 6: Error Severity Function of Each Mutant

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

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

Based on these definitions, we have obtained the Error Similarity Functions for populations of two versions. Table 7 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
m2	1	–	0	0
pg27	0	0	–	1
t22	0	0	1	–

Table 7: 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[20]. 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:

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

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

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

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

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

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.

7. 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 effectiveness of NVS as observed by our mutation testing is quite different from previously published results (e.g.,[4].) The major reasons lie on the investigated NVP design paradigm and design diversity, and the effectiveness of error detection and recovery mechanisms. In this project, the critical, final output of the application does not rely on a single Boolean variable; moreover, there are many variables installed for the purpose of cross-checking intermediate values and recovering internal states. Incorporation of this comprehensive fault-tolerant procedure has made a significant contribution in assessing the effectiveness of NVS systems.

Finally, 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.

Acknowledgement

The author wishes to thank Prof. Algirdas Avižienis for providing the program versions and the DeDiX tool to do the analysis and evaluation required in this research.

References

1. 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.
2. 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.
3. R.K. Scott, J.W. Gault, and D.F. McAllister, "Fault Tolerant Software Reliability Modeling," *IEEE Transactions on Software Engineering*, vol. SE-13, pp. 582-592, May 1987.
4. 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.
5. M.A. Vouk *et al.*, "Identification of Correlated Failures of Fault-tolerant Software Systems," in *Proceedings COMPSAC-85 the IEEE Computer Software and Applications Conference*, 1985.
6. 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.
7. A. Avižienis, M.R. Lyu, and W. Schütz, "Multi-Version Software Development: A UCLA/Honeywell Joint Project for Fault-Tolerant Flight Control Software," CSD-880034, Los Angeles, California, May 1988.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. J. D. Musa, A. Iannino, and K. Okumoto, *Software Reliability – Measurement, Prediction, Application*, McGraw-Hill Book Company, New York, New York, 1987.
14. Z. Jelinski and P.B. Moranda, "Software Reliability Research," in *Statistical Computer Performance Evaluation*, ed. W. Freiberber, pp. 465-484, Academic, New York, 1972.
15. N.F. Schneidewind, "Analysis of Error Processesin Computer Software," in *Proceedings International Conference on Reliable Software*, pp. 337-346, Los Angeles, 1975.
16. 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.
17. 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.
18. 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.

19. 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.
20. 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.