An Experimental Evaluation on Reliability Features of N-Version Programming

Xia Cai and Michael R. Lyu
Department of Computer Science and Engineering
The Chinese University of Hong Kong, Hong Kong
{xcai, lyu}@cse.cuhk.edu.hk

Mladen A. Vouk
Department of Computer Science
North Carolina State University, Raleigh
vouk@ncsu.edu

Abstract

Although N-version programming has been employed in some mission-critical applications, the reliability and fault correlation issues remain a debatable topic in the research community. In this paper, we perform a comprehensive evaluation on our recent project data on N-version programming and present statistical investigations on coincident failures and correlated faults. Furthermore, we compare our project with NASA 4-University project to identify the "variants" and "invariants" with respect to failure rate, fault density, coincident failures, related faults, and reliability improvement for N-version programming. Our experimental results support fault tolerance as an effective software reliability engineering technique.

Keywords: *N-version programming, fault correlation, reliability, software fault tolerance, empirical study*

1 Introduction

As one of the main techniques for software fault tolerance, N-version programming (NVP) aims at tolerating residual software design faults by independently developing multiple program versions from a common specification. Although it has been adopted in some mission-critical applications, the effectiveness of this approach is still an open question. The main issue is how to predict the final reliability as well as how to estimate the fault correlation between multiple versions. Theoretical as well as empirical investigations have been conducted based on experimentation [4, 7, 8, 15, 16], modeling [3, 5, 6, 11, 18], and evaluation [1, 12, 13, 17] for reliability and fault-correlation features of N-version programming.

In our previous work, we conducted a real-world project and engaged multiple programming teams to independently develop program versions based on an industry-scale avionics application. In this paper, we perform comprehensive operational testing on our original versions and collect statistical data on coincident failures and faults. We also conduct comparison with NASA 4-University project [4] and search for the "variant" as well as "invariant" features in N-version programming between our project and NASA 4-University project, two projects with the same application but separated by 17 years. Both qualitative and quantitative comparisons are engaged in development process, fault analysis, average failure rate and reliability improvement of N-version programming.

The terminology used in this paper for experimental descriptions and comparisons is defined as the following. *Coincident failures* refer to the failures where two or more different software versions respond incorrectly (with respect to the specification) on the same test case, no matter whether they produce similar or different results. *Related faults* are defined as the faults which affect two or more software versions, causing them to produce coincident failures on the same test cases, although these related faults may not be identical.

The remaining of this paper is organized as follows: Section 2 describes the experimental background of previous studies on N-version programming, including NASA 4-University project and our project. Qualitative comparisons and quantitative comparisons between the two projects are listed in Section 3 and Section 4, respectively. Section 5 discusses our main findings on the comparisons, and Section 6 concludes the paper.

2 Experimental Background

To investigate and evaluate the reliability and fault correlation features of N-version programming, statistical failure data are highly demanded from experiments or real-world projects. To simulate real environments, the experimental application should be as complicated as real-world projects to represent actual software in practice. In such experiments, the population of program versions should be large enough to provide valid statistical analysis. Furthermore,

Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)
1071-9458/05 $20.00 © 2005 **IEEE**

the development process should be well-controlled, so that the bug history can be recorded and real faults can be studied.

Up to now, a number of projects have been conducted to investigate and evaluate the effectiveness of N-version programming, including UCLA Six-Language project [9, 14], NASA 4-University project [4, 17, 19], Knight and Leveson's experiment [10], and Lyu-He study [3, 15]. Considering the population of programming versions and the complexity of the application, NASA 4-University project was a representative and well-controlled experiment for the evaluation of N-version programming.

Recently, we conducted a large-scale N-version programming experiment which engaged the same application as that in NASA 4-University project and followed well-defined development and verification procedures [16]. Both NASA 4-University project and our project chose the same application of a critical avionics instrument, Redundant Strapped-Down Inertial Measurement Unit (*RSDIMU*), to conduct a multi-version software experiment. RSDIMU is part of the navigation system in an aircraft or spacecraft, where developers are required to estimate the vehicle acceleration using eight redundant accelerometers (also called "sensors") mounted on the four triangular faces of a semi-octahedron in the flight vehicle. As the system itself is fault tolerant, it allows a calculation of the acceleration even when some of the accelerometers fail. Figure 1 shows the system data flow diagram of RSDIMU application.

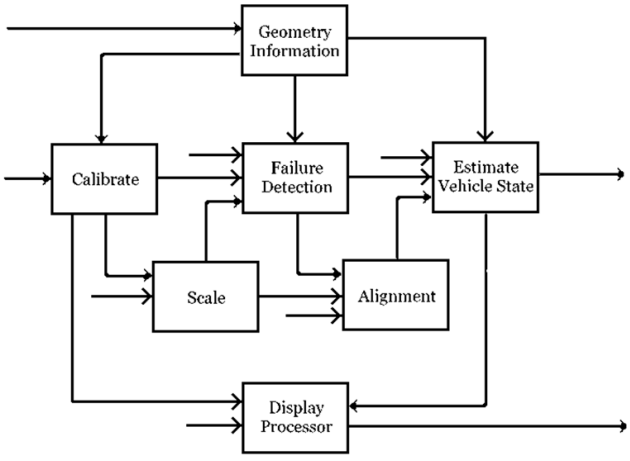

Figure 1. RSDIMU System Data Flow Diagram

The experimental procedures of both NASA 4-University project and our project are described in the following subsections.

2.1 NASA 4-University Project

In 1985, NASA Langley Research Center sponsored an experiment to develop a set of high reliability aerospace application programs to study multi-version software approaches, involving graduate student programmers from four universities in 20 program teams [19]. Details of the experimental procedure and development process can be found in [4, 9, 19].

In this experiment, the development process was controlled to maintain as much independence as possible between different programming teams. In addition, the final twenty programs went through a separate three-phase testing process, namely, a set of 75 test cases for acceptance test, 1196 designed and random test cases for certification test, and over 900,000 test cases for operational test.

The testing data collected in NASA 4-University project have been widely investigated to explore the reliability features of N-version programming [4, 11, 17].

2.2 Our Project Descriptions

In the Spring of 2002 we formed 34 independent programming teams at the Chinese University of Hong Kong to design, code, test, evaluate, and document the RSDIMU application. Each team was composed of four senior-level undergraduate Computer Science students for a twelve-week project in a software engineering course. The project details and the software development procedure are portrayed in [16]. The acceptance test set contains 1196 test cases, including 800 functional test cases and 400 randomly-generated test cases. Another random 100,000 test cases have been conducted recently in an operational test for a final evaluation of all the 34 program versions in our project.

3 Qualitative Comparison with NASA 4-University Project

In NASA 4-University project [9, 19], the final 20 versions were written in Pascal, while developed and tested in a UNIX environment on VAX hardware. As this project has some similarities and differences with our experiment, interesting observations can be made by comparing these two projects, which are widely separated both temporally and geographically, to identify possible "variants" as well as "invariants" of design diversity.

The commonalities and differences of the two experiments are shown in Table 1. The two experiments engaged the same RSDIMU specification, with the difference that NASA 4-University project employed the initial version of the specification which inherited specification incorrectness and ambiguity, while we employed the latest version of the specification and little specification faults were

Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)
1071-9458/05 $20.00 © 2005 **IEEE**

Table 1. Comparisons between the two projects

Features	NASA 4-University project	Our experiment
Commonality		
1.same specification	initial version (faults involved)	mature version
2.similar development duration	10 weeks	12 weeks
3.similar development process	training, design, coding, testing, preliminary acceptance test	initial design, final design, initial code, unit test, integration test, acceptance test
4.same testing process	acceptance test, certification test, operational test	unit test, integration test, acceptance test, operational test
5.same operational test environment (i.e., determined by the same generator)	1196 test cases for certification test	1200 test cases for acceptance test
Difference		
1.Time (17 year apart)	1985	2002
2.Programming Team	2-person	4-person
3.Programmer experience	graduate students	undergraduate students
4.Programmer background	U.S.	Hong Kong
5.Language	Pascal	C

detected during the development. The development duration were similar: 10 weeks vs. 12 weeks. The development process and the testing process are also similar. We engaged 1200 test cases before accepting the program versions, which were then subjected to 100,000 cases for operational test. NASA 4-University project involved the same 1196 test cases in its certification test, and other 900,000 test cases as operational test. Note that all test cases were generated by the same test case generator. The two experiments, on the other hand, differed in time (which is 17 years apart), programming team (2-person vs. 4-person), programmer experience and background, as well as the programming language employed (Pascal vs. C).

3.1 Fault analysis in development phase

As the two N-version programming experiments exhibit native commonalities and differences, it will be interesting to see what remains unchanged in these two experiments concerning software reliability and fault correlation. As stated above, the original version of the RSDIMU specification involved some faults which were fixed during NASA 4-University project development. We first consider the design and implementation faults detected during the NASA certification test and our acceptance test. These two testing phases employed the same set of test cases, and a comparison between them should be reasonable. In our investigation, we focus our discussion on the related faults, i.e., the faults which occurred in more than one version, and triggered off coincident failures of different versions at the same test case.

The classification of the related faults in our experiment is listed in Table 2. There are totally 15 categories of related faults. The distribution of the related faults is listed in Table 3. All the faults are design and implementation faults. Only class F1 is at low severity level; others are at critical severity level.

Comparing Table 2 with the related faults detected in certification testing of the NASA project [19], we can see that some common faults were generated during the development of these two projects. They are F1(D4 in [19]), F2.3(D1.3), F3.1(D3.7), F3.4(D3.1) and F5(D8). F1 faults are related to the display module of the application. Display algorithm was clearly stated in the RSDIMU specification, but it encountered related faults most frequently in both projects. This may be due to the fact that the module was comparatively simple and less critical in the whole application, and programmers inclined to overlook it in both development and testing phases. Fault F2.3 is related to a calculation in the wrong frame of reference, which involved only one version in both experiments. Fault F3.1 is a fault involved in many program versions, causing fatal failures due to initialization problems. Fault F3.4 is similar to fault D3.1, and both were due to misunderstanding of the content of the specification. Finally, the missing process of masking input from sensors to 12 bits (e.g., mod 4096) results in fault F5, which involved four teams in our project and 11 teams in the NASA project.

For the related faults occurring in both projects, some of them (F1, F2.3, F5) were due to misunderstanding of

Table 2. Related faults detected in our experiment

Faults	Brief Description	severity level
F1	Display error	low
F1.1	improper initialization and assignment	
F1.2	incorrect rounding at Display values	
F1.3	DisplayMode assignment error	
F2	Misalignment problem	critical
F2.1	the misalignment should be in radians	
F2.2	the misalignment should be in milliradians	
F2.3	wrong frame of reference	
F3	Sensor Failure Detection and Isolation problems	critical
F3.1	fatal failures due to initialization problem	
F3.2	wrongly set of the status of system	
F3.3	update problem after failure detection	
F3.4	test threshold for Edge Vector Test is miscalculated or misjudged	
F3.5	wrong OFFRAW array order	
F3.6	arithmetic and calculation error	
F4	Acceleration estimation problem	critical
F4.1	specific force recalculated problem after failure detection	
F4.2	wrongly calculation with G (should multiply 0.3048)	
F5	Input from sensor not properly masked to 12 bits (e.g., “mod 4096” missing)	critical

Table 3. Distribution of related faults detected

Faults	Versions	Fault Span
F1		
F1.1	P2,P4,P9	3
F1.2	P4,P8,P15,P17,P18	5
F1.3	P1,P2,P4,P5,P7,P8,P9,P15,P17,P18,P24,P27,P33	13
F2		
F2.1	P3	1
F2.2	P1,P8,P15	3
F2.3	P15	1
F3		
F3.1	P3,P4,P5,P7,P8,P22,P27	7
F3.2	P1,P8	2
F3.3	P1,P2	2
F3.4	P31,P32	2
F3.5	P12,P31	2
F3.6	P1,P5	2
F4		
F4.1	P2,P18	2
F4.2	P4,P12,P15,P17	4
F5	P12,P17,P15,P33	4

the specification or inadequate efforts spent on the difficult part of the problem, and others (F3.1 and F3.4) were caused by lack of knowledge in the application area or in the programming language area, e.g., omission or wrong variable initialization.

Some fault types occurred in the NASA project but did not occur in our experiment, e.g., D5 (division by zero on all failed input sensors), D6 (incorrect conversion factor) and D7 (Votelinout procedure call placement fault and/or error in using the returned values). Some of the reasons are: 1) As mentioned above, the initial specification contained incorrectness and inconsistency. Some design and implementation faults (e.g., D7) may be caused by the ambiguity or wrong statements in the specification. 2) Certain exception faults, such as division by zero, did not occur in our experiment. This is an interesting phenomenon, and the possible reason is that nowadays students learned the principle of avoiding exception faults as a common practice in the programming language courses.

From the comparison of the two experiments, we can see that both cause and effect of some related faults remain the same. As the application can be decomposed into different subdomains, related faults often occurred in most difficult parts. Furthermore, no matter which programming language was used, the common problems with programming remained the same, e.g., the initialization problem. Finally, the most fault-prone part being the easiest part of the application (i.e., Display module) confirms that a comprehensive testing and certification procedure towards the easiest module is important.

3.2 Fault analysis in operational test

As all the 34 versions have already passed the acceptance test, their failures revealed in the operational test deserve to be scrutinized. To investigate the features of these failures, especially those coincident failures occurring in more than two versions, we identify all the operational faults in each version and list them in Table 4. There are totally six faults detected during operational test. We denote each individual fault by the version number in which the fault occurs, followed by a sequence number. For example, version 22 contains one single fault 22.1, and version 34 is associated with three different faults: 34.1, 34.2, and 34.3. Table 4 also shows the input condition where a corresponding fault is manifested, together with a brief description of the fault.

Since these four versions have already passed the acceptance test, their faults were detected by some extreme situations. It is noticed that these faults are all sensor Failure Detection and Isolation problems, i.e., F3 category in Table 2, including wrong setting of system status (F3.2) and arithmetic and calculation error (F3.6). These faults were triggered only under special combinations of individual sensor failures due to input, noise and failure detection process.

In Table 4, versions 22, 29 and 32 exhibit single faults, while version 34 contains multiple faults. There is no related fault or coincident failure among the former three versions. For version 34, one of its faults (34.2) is related to the fault in version 22 (22.1), resulting in coincident failures on 25 test cases, as shown in Table 6. The other fault 34.3 is related to the one in version 29 (29.1), leading to 32 coincident failures. Although causing coincident failures, these two fault pairs are quite different by nature.

Compared with other program versions, version 34 shows the lowest quality in terms of its program logic and design organization. Particularly, hard code is found in the source code, i.e., some intermediate result was manually assigned according to specific input data of a particular test case, but not through the required computational functions. Because of its lack of detailed report and poor quality, we omitted this version (as well as a number of other versions) when applying mutation testing in our previous study [16]. Since related faults and coincident failures exist in this version, we took a pessimistic approach to include it (and all other program versions) in the following analysis. It is noted that the overall performance of N-version programming derived from our data would be much better, if the failures in version 34 are ignored (as no related faults or coincident failures would then be observed in other versions).

4 Quantitative Comparison with NASA 4-University Project

In this comprehensive testing study, we generate 100,000 test cases randomly to simulate the operational environment for RSDIMU application. The failures occurring in all these 34 versions are recorded according to their input and output domains. Here we adopt the similar partition method described in [4], i.e., all the normal operations are classified as one of the following six system states exclusively:

$$S_{i,j} = \{ i \; sensors \; previously \; failed \; and \; j \; of \; the \; remaining \; sensors \; fail \mid i = 0, 1, 2; \; j = 0, 1 \}.$$

In addition, we introduce a new category S_{others} to denote all the exceptional operations which do not belong to any of the above six system states. In the following analysis, we partition the whole input and output domain into seven categories for an investigation in software reliability measurement.

4.1 Failure probability and fault density

Version failures which are detected in operational test are listed in Table 5. Most of the 34 versions passed all

Table 4. Fault description during operational test

Version	Fault	Input condition	Fault description
22	22.1	At least one sensor fail during the test	Incorrect calculation in sensor failure detection process
29	29.1	No sensor fail before the test, and more than two bad faces due to noise and failure detection, but at least one sensor pass the failure detection algorithm	Omission of setting all sensor failure output to TRUE when the system status is set to FALSE
32	32.1	Three or four sensors fail in more than two faces, due to input, noise, or failure detection algorithm	Incorrectly setting system status to FALSE when more than five sensors fail
34	34.1	Sensor failure in the input	Wrongly setting the second sensor as failure when the first sensor in the same face fails
	34.2	No sensor fail due to input and noise	Incorrect calculation in Edge Vector Test is wrong
	34.3	At most two faces fail due to input and noise, no. of bad faces is greater than 2, and at least one more sensor on the third face fail in failure detection algorithm	Only counting the number of bad face prior to the Edge Vector Test

Table 5. Failures collected in our project

Version ID	$S_{0,0}$	$S_{0,1}$	$S_{1,0}$	$S_{1,1}$	$S_{2,0}$	$S_{2,1}$	S_{others}	Total	Probability
22	0	210	0	246	0	133	29	618	0.00618
29	0	0	0	0	0	0	2760	2760	0.0276
32	0	0	0	0	0	0	2	2	0.00002
34	0	617	0	407	0	74	253	1351	0.0135
Total no. of failures	0	827	0	653	0	207	3044	4731	0.0473
Conditional average failure probability	0	0.0024	0	0.0017	0	0.0011	0.0023	0.0014	
No. of test cases	9928	10026	11558	11624	12660	5538	38666	100000	

these 100,000 test cases, while only four of them exhibited a number of failures, ranging from 2 to 2760. Compared with the failure data of NASA 4-University project with over 900,000 test cases described in [4], our data demonstrate some similar as well as different reliability features with respect to the same RSDIMU application.

In both projects, the programs are generally more reliable for the cases where no sensor failure occurs during operation (i.e., States $S_{0,0}$, $S_{1,0}$, and $S_{2,0}$) than in the situations where one false sensor occurs during operation (i.e., States $S_{0,1}$, $S_{1,1}$, and $S_{2,1}$). This is because there are some complicated computations under the latter situations. Particularly in our project data, no new failure was detected in the former three states, while all the detected failures were revealed under the latter three states and the exceptional states (State S_{others}). Especially for version 29, all the 2760 failures occurred under exceptional system states.

In our operational test, totally 4731 failures were collected for 34 versions, representing an average 139 failures per $100,000$ executions for each version. The failure probability is thus 0.00139 for a single version, with highest probability in state $S_{0,1}$ (with 0.0825). This indicates on average the programs inherit high reliability.

We further investigate the coincident failures between version pairs. Here two or more versions are regarded as correlated if they fail at the same test case, whether their outputs are identical or not. Two correlated version pairs were observed, as listed in Table 6.

It can be seen that for version pair 22 & 34, there were 25 coincident failures, thus the percentage of coincident failures versus total failures is 4% for version 22 and 1.85% for version 34. For the other version pair 29 & 34, 32 coincident failures were observed with the percentage 1.16% for version 29 and 2.37% for version 34. The low probability of coincident failures versus total failures supports the effectiveness of N-version programming, meaning a more reliable system can be expected by this approach from out project data. Moreover, as seen in Table 4, there are totally six faults identified in four out of the 34 versions. As noted in [16], the size of these versions varies from 1455 to 4512 source lines of code. We can calculate the average fault density to be roughly one fault per 10,000 lines. This figure is close to industry-standard for high quality software systems.

Table 6. Coincident failures between versions

Version pairs	$S_{0,0}$	$S_{0,1}$	$S_{1,0}$	$S_{1,1}$	$S_{2,0}$	$S_{2,1}$	S_{others}	Total failures
22 & 34	0	15	0	6	0	4	0	25
29 & 34	0	0	0	0	0	0	32	32
Total no. of failures	0	827	0	653	0	207	3044	4731
Conditional frequency	0	0.0005	0	0.0003	0	0.0006	0.0003	0.0004

4.2 Reliability improvement by N-version programming

To estimate the reliability improvement of N-version programming compared with one single version, we first adopt the simplest statistical method. For 2-version system, there are $C(34, 2) = 561$ combinations out of 34 versions. Considering there are 57 coincident failures observed, the average failure is 57/561=0.102 over $100,000$ executions. In a 3-version system there are $C(34, 3) = 5984$ combinations, so that the average failure probability is $57/5984 = 0.0095$ for 100,000 executions, which implies a 11 times higher reliability than that of a 2-version system. Recall the average failures in single version listed in Table 5 is 139; therefore, we obtain the reliability improvement from 3-version system versus one single version of about 15000 (139/0.0095) times. For more accurate comparisons, failure bound models can be exercised to investigate the reliability features for N-version programming compared with one single version. Consequently, we fit the observed operational test failure data to Popov and Strigini failure bound model [2, 17] and estimate the reliability improvement of N-version programming. The estimated failure bounds for version pair (22,34) and (29,34) are listed in Table 7.

Note that DP1, DP2 and DP3 stand for three different testing profiles with various probabilities on different input domains, i.e., $S_{i,j}$ [2]. For simplicity, we denote an insignificant value of the lower bound for version pair (29,34) under DP1 to be zero, which stands for independence between the versions. Since only two version pairs show correlations, we add these bounds up, and then divided by 34, to get the average lower and upper bounds for all 2-version combinations. The average lower bound and upper bounds for any 2-version system under different profiles are shown in Table 7. Compared with the average failure probability for single version 0.00139, the reliability improvement of 2-version system versus one single version under DP3 (which is the closest to the real distribution) is 90 to 2,000 times. As the reliability of 3-version system is 11 times of that for 2-version system, the reliability improvement by 3-version system is thus about 900 to 20,000 times over single version system under DP3. Similar improvement can be obtained in DP2, and DP1 achieves even higher improvement.

Moreover, we apply the same failure bound model on the NASA data and get the average failure bounds, as shown in Table 7. It is surprise to see that the failure bounds in the NASA project are at least an order of magnitude larger than those in our project. The experimental data in [4] indicate that although there were only seven faults identified in 14 versions, they produced almost 100,000 failures. Particularly, the single fault in version 7 caused more than half of these failures. The fault was caused by incorrect initialization of a variable; however, it was not stated that why the initialization problem was not detected by certification test. We can see that the failure bounds would have been comparable to our project results, had the fault in version 7 been detected and removed in the NASA project.

4.3 Comparison with NASA 4-University Project

In NASA 4-University operational test involving over 900,000 test cases, only seven out of 20 versions passed all the test cases. Moreover, a number of these versions, ranging from 2 to 8, were observed to fail simultaneously on the same test cases. In addition, seven related faults were identified which caused the coincident failures. Two of the NASA faults are also related to sensor failure detection and isolation problem. Other faults, e.g., variable initialized incorrectly and failure isolation algorithm implemented in wrong coordinate system, were not observed in our data. In our future research we will investigate failure coincidences between these two projects.

We compare the failure data collected in operational test in both our project and NASA 4-University project, and list some of the reliability related features in Table 8. In our experiment, only 2-version coincident failures occurred, and no coincident failures among three or more versions were detected. The number of failures and coincident failures in the NASA project is much larger than that in our project, meaning we have achieved a significantly higher reliability figure in our project. Interestingly, the difference on fault number and fault density is not significant for these two projects. Note there is a number of coincident failures in 2- to 8-version combinations in the NASA project, yet the reliability improvement for 3-version system still achieves 80 to 330 times over the single version. Our project obtains similar improvement for 3-version system (i.e., 900 to 20,000 times) over the single version. Considering the average failure rate for a single version is already 50 times better than that in the NASA project, it is understandable that the

Table 7. Failure bounds for 2-version system

Version pair	DP1		DP2		DP3	
	Lower bound	Upper bound	Lower bound	Upper bound	Lower bound	Upper bound
(22,34)	0.000007	0.000130	0.000342	0.006721	0.000353	0.008396
(29,34)	0.000000	0.000001	0.000009	0.000131	0.000047	0.000654
Average in our project	$1.25 \cdot 10^{-8}$	$2.34 \cdot 10^{-7}$	$6.26 \cdot 10^{-7}$	0.000012	$7.13 \cdot 10^{-7}$	0.000016
Average in NASA project	$2.32 \cdot 10^{-7}$	0.000007	0.000023	0.000103	0.000072	0.000276
Average in NASA project after omitting version 7	$6.15 \cdot 10^{-9}$	0.000006	$2.16 \cdot 10^{-7}$	0.000024	$5.97 \cdot 10^{-7}$	0.000028

Table 8. Quantitative comparison in operational test with NASA 4-University project

Item	Our project	NASA 4-University project
no. of test cases	100,000	920,746
failure probability	0.00139	0.06881
number of faults	6	7
fault density	1 per 10,000 lines	1.8 per 10,000 lines
2-version coincident failures	57	21173
3 or more version coincident failures	0	372
3-version improvement	900 to 20,000 times	80 to 330 times

improvement for 3-version system in our experiment is 30 to 60 times of that in the NASA project.

Overall, from the above comparison between NASA 4-University project and our project, we can derive some variances as well as invariances on N-version programming. The invariances are:

- Both experiments yielded reliable program versions with low failure probability, i.e., 0.06881 and 0.00139 for single version respectively.
- The number of faults identified in the operational test was of similar size, i.e., 7 versus 6 faults.
- The fault density was of similar size, i.e., 1.8 versus 1 fault detected per $10,000$ lines of code.
- Remarkable reliability improvement was obtained for N-version programming in both experiments, i.e., hundreds to tens of thousands of times enhancement.
- Related faults were observed in both projects, in both difficult and easy parts of the application.

Nevertheless, there are some variances between the two projects:

- Some faults identified in the NASA project did not appear in our project, e.g., divide by zero, wrong coordinate system, and incorrect initialization problems.
- More failures were observed in the NASA project than in our project, causing their average failure probability to be an order of magnitude higher than ours.
- In the NASA project, more coincident failures are observed and the failure correlation between versions was more significant, especially among more than three versions. In our project, the fault correlation was reduced, especially if we omit version 34, which contains hard code and poor logic.
- The overall reliability improvement derived from our data is at least an order of magnitude larger than that from the NASA project. This is true for both single version system and N-version system.

The reasons behind the variance and invariance between the two projects can be concluded from the following aspects: First, as the first RSDIMU experiment, NASA 4-University project had to deal with some specification-related faults. Its N-version programming development process became a bit messy, as faults in the specification would be identified and revised, then distributed to the developers for further evaluation. In our project, as we employed a stable version of the specification, such revisions no longer occurred, and programmers could concentrate on the problem solving process. Secondly, there is apparently a significant progress on programming course and training to computer science students over the past 20 yeas. Modern programmers are well disciplined to avoid common programming fault such as divide by zero and uninitialization problems. Lastly, based on the experience accumulated in all the former projects and experiments on N-version programming, we were able to follow a more well-controlled protocol in our experimental procedure. We believe that the N-version programming design and development process

can keep further improvement as a vital software reliability engineering technique.

5 Discussions

In this experimental evaluation, we perform comprehensive testing and compare the two projects addressing N-version programming. The empirical data show that both similar and dissimilar faults were observed in these two projects. We analyze the reasons behind their similarities and differences. It is evident that the program versions in our project are more reliable than those in the NASA project in the terms of total number of failures and coincident failures revealed in operational test. Our data show the reliability improvement by N-version programming is significantly high, i.e., from 900 to 20,000, over single program versions already with very high reliability. The effectiveness of N-version programming in mission-critical applications is confirmed in our project data.

Moreover, when we examine the faults identified in our project, especially for 22.1, 29.1 and 32.1, we can find that these hard-to-detected faults are only hit by some rare input domains. This means a new strategy should be employed for such faults. As discussed in our previous study [2, 16], code coverage is a good estimator for testing effectiveness. Experimental data show that in our acceptance test, test cases with higher code coverage tend to detect more faults, especially for exceptional test cases. Nevertheless, in this operational test, none of these faults can be detected by the code coverage indicator.

6 Conclusion

In this paper, we perform an empirical investigation on evaluating reliability features by a comprehensive comparison between two projects. We conduct operational testing involving 100,000 test cases on our 34 program versions and analyze their failures and faults. The data collected in this testing process are compared with those in NASA 4-University project.

Similar as well as dissimilar faults are observed and analyzed, indicating common problems related to the same application in these projects. Less failures are detected in our operational test of our project, including a very small number of coincident failures. This result provides a supportive evidence for N-version programming, and the improvement is attributed to cleaner development protocol, stable specification, experience in N-version programming experiment, and better programmer training.

Acknowledgement

The work described in this paper was fully supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK4205/04E).

References

[1] F. Belli and P. Jedrzejowicz. Fault-tolerant programs and their reliability. *IEEE Transactions on Reliability*, 29(2):184–192, 1990.

[2] X. Cai and M. R. Lyu. An empirical study on reliability modeling for diverse software systems. In *Proc. of the 15th International Symposium on Software Reliability Engineering (ISSRE'2004)*, pages 125–136, Saint-Malo, France, Nov. 2004.

[3] J. B. Dugan and M. R. Lyu. Dependability modeling for fault-tolerant software and systems. In M. R. Lyu, editor, *Software Fault Tolerance*. Wiley, New York, 1995.

[4] D. E. Eckhardt, A. K. Caglavan, J. C. Knight, L. D. Lee, D. F. McAllister, M. A. Vouk, and J. P. Kelly. An experimental evaluation of software redundancy as a strategy for improving reliability. *IEEE Transactions on Software Engineering*, 17(7):692–702, July 1991.

[5] D. E. Eckhardt and L. D. Lee. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Transactions on Software Engineering*, 11(12):1511–1517, Dec. 1985.

[6] M. Ege, M. A. Eyler, and M. U. Karakas. Reliability analysis in n-version programming with dependent failures. In *Proc. of the 27th Euromicro Conference*, pages 174 –181, Warsaw, Poland, Sept. 2001.

[7] K. E. Grosspietsch. Optimizing the reliability of the component-based n-version approaches. In *Proc. of International Parallel and Distributed Processing Symposium (IPDPS 2002)*, pages 138–145, Florida, Apr. 2002.

[8] K. E. Grosspietsch and A. Romanovsky. An evolutionary and adaptive approach for n-version programming. In *Proc. of the 27th Euromicro Conference*, pages 182–189, Warsaw, Poland, Sept. 2001.

[9] J. Kelly, D. Eckhardt, M. Vouk, D. McAllister, and A. Caglayan. A large scale generation experiment in multi-version software:description and early results. In *Proc. of the 18th International Symposium on Fault-Tolerant Computing (FTCS-18)*, pages 9–14, Tokyo, Japan, June 1988.

[10] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, 12(1):96–109, Jan. 1986.

[11] B. Littlewood and D. Miller. Conceptual modeling of coincident failures in multiversion software. *IEEE Transactions on Software Engineering*, 15(12):1596–1614, Dec. 1989.

[12] B. Littlewood, P. Popov, and L. Strigini. Design diversity: an update from research on reliability modelling. In *Proc. of Safety-Critical Systems Symposium 21*, Bristol, U.K., 2001.

[13] B. Littlewood, P. Popov, and L. Strigini. Modelling software design diversity - a review. *ACM Computing Surveys*, 33(2):177–208, June 2001.

[14] M. R. Lyu. *Software Fault Tolerance*. Wiley, New York, 1995.

[15] M. R. Lyu and Y. He. Improving the n-version programming process through the evolution of a design paradigm. *IEEE Transactions on Reliability*, 42(2):179–189, June 1993.

[16] M. R. Lyu, Z. Huang, K. S. Sze, and X. Cai. An empirical study on testing and fault tolerance for software reliability engineering. In *Proc. of the 14th IEEE International Symposium on Software Reliability Engineering (ISSRE'2003)*, pages 119–130, Denver, Colorado, Nov. 2003.

[17] P. T. Popov, L. Strigini, J. May, and S. Kuball. Estimating bounds on the reliability of diverse systems. *IEEE Transactions on Software Engineering*, 29(4):345–359, Apr. 2003.

[18] X. Teng and H. Pham. A software-reliability growth model for n-version programming systems. *IEEE Transactions on Reliability*, 51(3):311–321, Sept. 2002.

[19] M. A. Vouk, A. Caglayan, D. E. Eckhardt, J. Kelly, J. Knight, D. McAllister, and L. Walker. Analysis of faults detected in a large-scale multi-version software development experiment. In *Proc. of the 9th Digital Avionics Systems Conference*, pages 378–385, Virginia Beach, Virginia, Oct. 1990.