Improving the N-Version Programming Process

Through the Evolution of a Design Paradigm

Michael R. Lyu
Member IEEE
Bell Communication Research
Morristown

Yu-Tao He
Student Member IEEE
The University of Iowa
Iowa City

Keywords – N-Version Programming, Design Paradigm, Experiment, Process Evolution, Mutation Testing, Reliability and Coverage Analysis.

Reader Aids –

Purpose:
Presentation of an N-version software experiment emphasizing a revised design paradigm to show process and product improvements

Special math needed for explanations: Probability theory

Special math needed to use results: None

Results useful to: Fault-tolerant software designers/experimenters and reliability analysts

Abstract – To encourage a practical application of the N-Version Programming (NVP) technique, a design paradigm was proposed and applied in a Six-Language Project. The design paradigm improved the development effort of the N-Version Software (NVS), however, there were some deficiencies of the design paradigm which lead to the leak of a pair of coincident faults. In this paper, we report a similar experiment conducted by using a revised NVP design paradigm, identify its impact to the software development process, and demonstrate the improvement of the resulting NVS product. This project reused the revised specification of an automatic airplane landing problem, and was participated by 40 students at the University of Iowa and the Rockwell International. Guided by the refined NVS development paradigm, the students formed 15 independent programming teams to design, program, test, and evaluate the application. The insights, experiences, and learnings in conducting this project are presented. Several quantitative measures of the resulting NVS product are provided, and some comparisons with other previously-conducted experiments addressed.

Improving the N-Version Programming Process Through the Evolution of a Design Paradigm

1. Introduction

The N-Version Programming (NVP) approach achieves fault-tolerant software systems, called *N-Version Software* (NVS) systems, through the development and use of design diversity [1]. Such systems are normally operated on an *N-Version Executive* (NVX) environment [2]. Since the idea of NVP was first proposed in [3], numerous papers were published in either modeling NVS systems [4] [5] [6] [7] [8] [9] [10], or providing empirical studies of NVS systems [11] [12] [13] [14] [15] [16] [17] [18]. The effectiveness of the NVP approach, however, has remained highly controversial and inconclusive. Nevertheless, most researchers and practitioners agree that a high degree of version independency and a low probability of failure correlation would play the key role in determining the success of an NVS deployment.

To maximize the effectiveness of the NVP approach, the probability of similar errors that coincide at the NVS decision points must be reduced to the lowest possible value. To achieve this key element for NVS, a rigorous NVP development paradigm was proposed in [19]. This effort included the foundation of disciplined practices in modern software engineering techniques, and the incorporation of recent knowledge and experience obtained from fault-tolerant system design principles. The main purpose of this design paradigm was to encourage the investigation and implementation of NVS techniques for practical applications.

The first application of this design paradigm to a real-world project was reported in [18] for an extensive evaluation. Some limitations were also be presented in [20], leading to a couple of refinements to the design paradigm. To observe the impact of the revised NVP development paradigm, a similar project was conducted at the University of Iowa and the Rockwell International. This paper describes a comprehensive report on the application of the design paradigm to this project and the results. In Section 2, we examine the NVP design paradigm and its

refinement. Section 3 describes the University of Iowa/Rockwell NVS project. The three following Sections give a thorough evaluation of the resulting NVS product, including program metrics and statistics (Section 4), operational testing and NVS reliability evaluation (Section 5), and mutation testing for a coverage analysis of the NVS systems (Section 6). Section 7 compares this experiment with three previous experiments in [16] [17] [18]. Conclusions of this paper are provided in Section 8.

2. An NVP Design Paradigm and Refinement

NVP has been defined as "the independent generation of $N \geq 2$ functionally equivalent programs from the same initial specification [3]." "Independent generation" meant that the programming efforts were to be carried out by individuals or groups that did not interact with respect to the programming process. The NVP approach was motivated by the "fundamental conjecture that the independence of programming efforts will greatly reduce the probability of identical software faults occurring in two or more versions of the program [3]."

The key NVP research effort has been addressed to the formation of a set of guidelines for systematic design approach to implement NVS systems, in order to achieve efficient tolerance of design faults in computer systems. The gradual evolution of these rigorous guidelines and techniques was formulated in [19] as an NVS *design paradigm* by integrating the knowledge and experience obtained from both software engineering techniques and fault tolerance investigations. The word "paradigm" means "pattern, example, model," which refers to a set of guidelines and rules with illustrations.

The objectives of the design paradigm are:

(1) to reduce the possibility of oversights, mistakes, and inconsistencies in the process of software development and testing;

(2) to eliminate most perceivable causes of related design faults in the independently generated versions of a program, and to identify causes of those which slip through the

design process;

(3) to minimize the probability that two or more versions will produce similar erroneous results that coincide in time for a decision (consensus) action of an NVX environment.

The application of a proven software development method, or of diverse methods for individual versions, remains the core of the NVP process. This process is supplemented by procedures that aim: (1) to attain suitable isolation and independence (with respect to software faults) of the N concurrent version development efforts, (2) to encourage potential diversity among the multiple versions of an NVS system, and (3) to elaborate efficient error detection and recovery mechanism. The first two procedures serve to reduce the chances of related software faults being introduced into two or more versions via potential "fault leak" links, such as casual conversations or mail exchanges, common flaws in training or in manuals, use of the same faulty compiler, etc. The last procedure serves to increase the possibilities of discovering manifested errors before they are converted to coincident failures. Figure 1 describes the current NVP paradigm for the development of NVS.

In Figure 1, the NVP paradigm is composed of two categories of activities. The first category, represented by boxes and single-line arrows at the left-hand side, contains typical software development procedures. The second category, represented by ovals and double-line arrows at the right-hand side, describes the concurrent enforcement of various fault-tolerant techniques under the N-version programming environment. Table 1 summarizes the activities and guidelines incorporated in each phase of software development life cycle[19]. This design paradigm has been revised according to the experience of its execution in [18], where the corresponding amendments are shown in italics in Table 1.

These modifications do not show dramatic changes from the original paradigm. Nevertheless, they are important in avoiding two types of specification-related faults: one is the absence of appropriate responses to specification updates, and the other is the incorrect handling of matching features required by an NVX environment (e.g., the placement of voting routines).

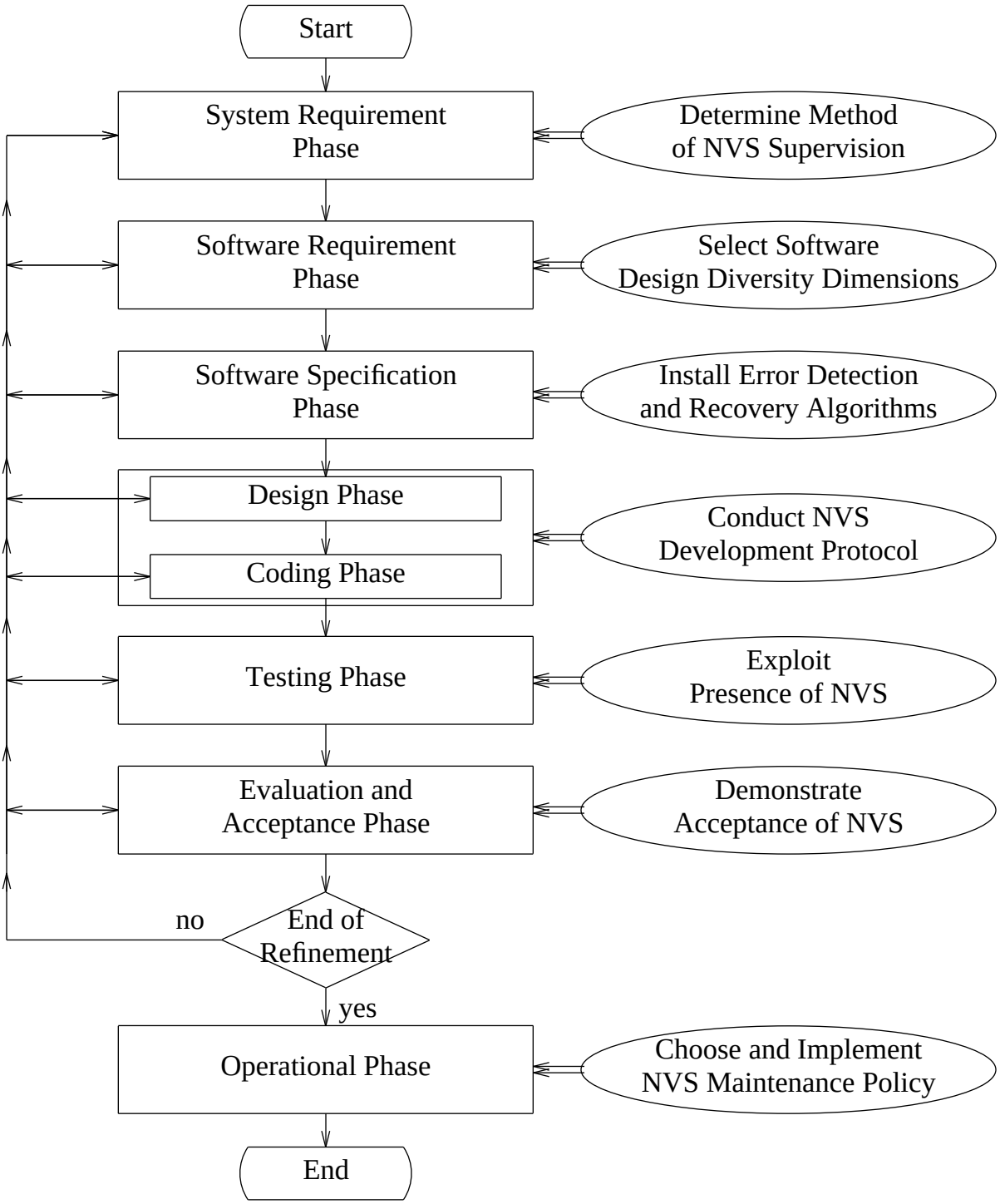

Figure 1: A Design Paradigm for N-Version Programming

Software Life Cycle	Enforcement on Fault Tolerance	Design Guidelines and Rules
System Requirement Phase	Determine Method of NVS Supervision	1. decide NVS execution methods and required resources 2. develop support mechanisms and tools 3. comply with hardware architecture
Software Requirement Phase	Select Software Design Diversity Dimensions	1. compare random diversity vs. enforced diversity 2. derive qualitative design diversity metrics 3. evaluate cost-effectiveness along each dimension 4. obtain the final choice under particular constraints
Software Specification Phase	Install Error Detection and Recovery Algorithms	1. prescribe the matching features needed by NVX 2. avoid diversity-limiting factors 3. require the enforced diversity 4. protect the specification against errors
Design and Coding Phase	Conduct NVS Development Protocol	1. derive a set of mandatory rules of isolation 2. define a rigorous communication and documentation (C&D) protocol 3. form a coordinating team *4. verify every specification update message*
Testing Phase	Exploit Presence *of NVS*	1. explore comprehensive verification procedures 2. enforce extensive validation efforts 3. provide opportunities for "back-to-back" testing *4. perform a preliminary NVS testing under the designated NVX*
Evaluation and Acceptance Phase	Demonstrate Acceptance of NVS	1. define NVS acceptance criteria 2. provide evidence of diversity 3. demonstrate effectiveness of diversity 4. make NVS dependability prediction
Operational Phase	Choose and Implement NVS Maintenance Policy	1. assure and monitor NVX basic functionality 2. keep the achieved diversity work in maintenance 3. follow the same paradigm for modification and maintenance

Table 1: Detailed Layout of the NVP Design Paradigm

3. The U. of Iowa/Rockwell NVS Project

The main purpose in formulating a design paradigm is to eliminate all identifiable causes of related design faults in the independently generated versions of a program, and to prevent all potential effects of coincident run-time errors while executing these program versions. An investigation to execute and evaluate the design paradigm is necessary, in which the complexity of the application software should reflect a realistic size in highly critical applications. Moreover, this investigation should be complete, in the sense that it should thoroughly explore all aspects of NVS systems as software fault-tolerant systems. This effort was first conducted in a UCLA/Honeywell Six-Language Project (called "Six-Language Project" hereafter) and reported in [18] [20]. In the Six-Language Project, a real-world automatic (i.e., computerized) airplane landing system, or so-called *autopilot*, was developed and programmed by six programming teams, in which the proposed NVS design paradigm was executed, validated, and refined.

In Fall 1991, a similar project was conducted at the University of Iowa and the Rockwell/Collins Avionics Division. Guided by the revised NVP design paradigm and a requirement specification with the known defects removed, 40 students (33 from ECE and CS departments at the University of Iowa, 7 from the Rockwell International) formed 15 programming teams (12 from University, 3 from Rockwell) to independently design, code, and test the computerized airplane landing system for the major requirement of a graduate-level software engineering course.

The following subsections describe how the revised NVP design paradigm was applied in conducting this project and the resulting project characteristics.

3.1 NVS Supervision Environment

The operational environment for the application was conceived as airplane/autopilot interacting in a simulated environment. Three or five channels of diverse software independently 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 a 3-channel flight simulation system (shown in Figure 2) consisted of three lanes of control law computation, three command monitors, a servo control, an Airplane model, and a turbulence generator.

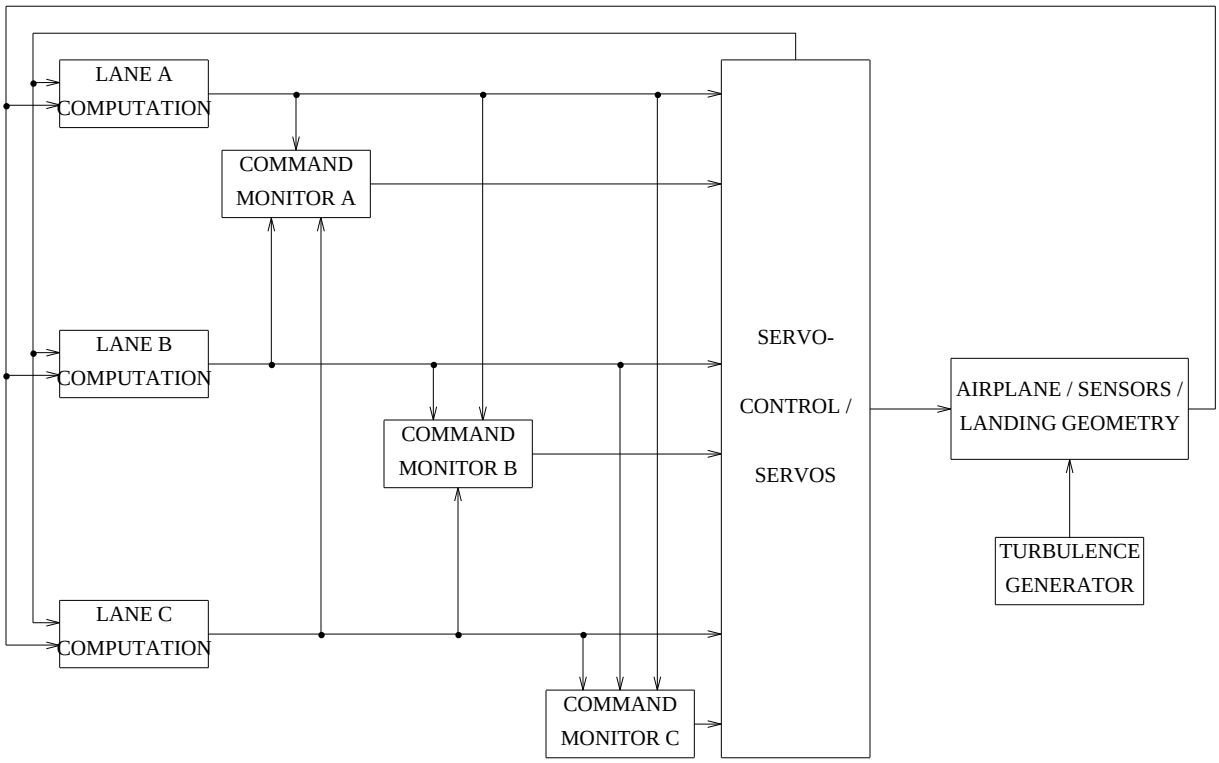

Figure 2: 3-Channel Flight Simulation Configuration

The *lane computations* and the *command monitors* would be the accepted software versions generated by the 15 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. 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 elevator servos. The servos were force-summed at their outputs, so that the mid-value of the three inputs became the final elevator command. The *Landing Geometry* and *Turbulence Generator* were models associated with the Airplane simulator.

In summary, one run of flight simulation was characterized by the following five initial values regarding the landing position of an airplane: (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.

3.2 Design Diversity Investigations

Independent programming teams were the baseline design diversity investigated for this project. It was noticed, however, the programming teams represented a wide range of experiences, from very experienced programmers (and/or avionics engineers) to novice ones. Moreover, three programming environments were provided to the programmers: one was located at the Iowa Computer-Aided Engineering Network (ICAEN) Center, another was located at the Department of Electrical and Computer Engineering computing facility, and the other was provided by the Rockwell computing facilities. Every programming team was required to use the C programming language for the coding of the application.

It was hypothesized that different programming environments provided different hardware platforms, working atmosphere, and computing tools and facilities which could add further diversity for the development of the software project.

3.3 High Quality Specification with Error Detection and Recovery

The efforts to develop a specification that would be suitable for the programming teams started early in 1987, which was initially used in the Six-Language Project. During software generation

in the Six-Language Project, many errors and ambiguities in the specification were revealed. The two specification defects resulting in two pairs of identical faults were carefully corrected. Moreover, A comprehensive error detection and recovery algorithm was imposed on the specification to specify two input routines, seven *vote routines* to cross check data, and one *recovery routine* to provide recovery when necessary. These enhancements were carefully specified and inserted in the original specification.

The final specification was restored to a single document [21] which was given to the 15 programming teams to independently develop their program versions in this project. This specification document has benefited from the scrutiny of more than 16 motivated programmers and researchers. This version of specification followed the principle of supplying only minimal (i.e., only absolutely necessary) information to the programmers, so as not to unwillingly bias the programmers' design decisions and overly restrict the potential design diversity. Throughout the program development phase, the specification was maintained as consistent and precise as possible.

3.4 NVP Communication Protocol

This project strictly required the programmers not to discuss any aspect of their work with members of other teams. Work-related communications between programmers and a project coordinator were conducted *only* via electronic mail. The programmers directed their questions to the project coordinator who was very familiar with the NVP process and the specification details. The project coordinator responded to each message very quickly, usually with a turnaround time of less than 24 hours. The purpose of imposing this isolation rule on programming teams was to assure the *independent generation* of programs, which meant that programming efforts were carried out by individuals or groups that did not interact with respect to the programming process.

In the communication protocol, each answer was *only* sent to the team that submitted the corresponding question. The answer was broadcast to all teams only if the answer led to an

update or clarification of the specification. During the software development process, a total of 145 questions were raised by and replied to individual programming teams, among which 11 specification-related message were broadcast for specification changes. In summary, the individual teams received an average of only 20 messages during the program development phase, contrasting to an average of 58 messages in the Six-Language Project and an average of over 150 messages in an NASA Experiment [17]. Also as a lesson learned from the Six-Language Project, each program version was carefully verified to comply with all the broadcast specification updates before its final acceptance.

3.5 NVS Software Development Schedule

The software development for this project was scheduled and conducted in six phases:

(1) *Initial design phase (4 weeks)*:

The purpose of this phase was to allow the programmers to get familiar with the specified problem, so as to design a solution to the problem. At the end of this four-week phase, each team delivered a preliminary design document, which followed specific guidelines and formats for documentation.

(2) *Detailed design phase (2 weeks)*:

The purpose of this phase was to let each team obtain some feedbacks from the coordinator to adjust, consolidate, and complete their final design. The feedbacks were regarding to the feasibility and style of each design rather than any technical corrections, except under the situation where the design did not comply with the specification updates. Each team was also requested to conduct one or several design walkthroughs. At the end of this two-week phase, each team delivered a detailed design document and a design walkthrough report.

(3) *Coding phase (3 weeks)*:

By the end of this 3-week phase, programmers had finished coding, conducted a code walkthrough by themselves, and delivered the initial code which was compilable. From this

moment on, each team was required to use the revision control tool RCS (or to include CVS for concurrent versions) to do configuration management of their program modules. Code update report forms were also distributed to record every change that was made after the code was generated.

(4) *Unit testing phase (1 week)*:

Each team was supplied with sample test data sets for each module to check the basic functionality of that module. They were also required to build their own test harness for this testing purpose. A total of 133 data files was provided to the programmers. These data sets were the same as those in the Six-Language Project.

(5) *Integration testing phase (2 weeks)*:

Four sets of partial flight simulation test data, together with an automatic testing routine, were provided to each programming team for integration testing. This phase of testing was intended to guarantee that the software was suitable for a flight simulation environment in an integrated system. These data sets were the same as those in the Six-Language Project.

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

Programmers formally submitted their programs for a two-step acceptance test. In the first step (AT1), each program was run in a test harness of four nominal flight simulation profiles. These data sets provided the same coverage as those in the Six-Language Project. For the second step (AT2), one extra simulation profile, representing an extremely difficult flight situation, was imposed. When a program failed a test, it was returned to the programmers for debugging and resubmission, along with the input case on which it failed. In summary, there were *23930* executions imposed on these programs before they were accepted and subjected to the final evaluation in the following stage. By the end of this two week phase, 12 programs passed the acceptance test successfully and were used in the following evaluations.

4. Program Metrics and Statistics

Table 2 gives several comparisons of the 12 versions (identified by a Greek letter) 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	β	γ	ε	ζ	η	θ	κ	λ	μ	ν	ξ	ο	Range
LINES	8769	2129	1176	1197	1777	1500	1360	5139	1778	1612	2443	1815	7.46
LN-CM	4006	1229	895	932	1477	1182	1251	2520	1168	1070	1683	1353	4.30
STMTS	2663	708	706	720	1208	753	640	1366	759	810	932	858	4.16
MODS	53	11	6	15	6	47	17	17	21	24	17	11	8.83
STM/M	179	64	101	439	201	406	38	80	36	35	67	78	12.5
STM/CCP	380	101	100	103	173	108	91	195	108	115	133	123	7.45
CALLS	84	123	16	23	37	76	31	626	100	106	30	66	39.1
GBVAR	0	55	101	180	86	406	7	0	354	423	421	26	-
LCVAR	1326	179	86	309	553	532	376	402	294	258	328	329	15.4
BINDE	233	68	152	103	224	120	115	118	88	112	73	131	3.29

Table 2: Software Metrics for the 12 Accepted Programs

A total of 96 faults was found and reported during the whole life cycle of the project. Classification of faults according to fault types is shown in Table 3. This category considers the

following type of faults [19]: (1) typographical (a cosmetic mistake made in typing the program); (2) error of omission (a piece of required code was missing); (3) incorrect algorithm (a deficient implementation of an algorithm); (4) specification misinterpretation (a misinterpretation of the specification); and (5) specification ambiguity (an unclear or inadequate specification which led to a deficient implementation). In this category, items (1) through (3) are implementation-related faults, while items (4) and (5) are specification-related faults. It is also noted that "incorrect algorithm" of item (3) is the most frequent fault type, which includes miscomputation, logic fault, initialization fault, and boundary fault. This result was similar to that observed in the Six-Language Project.

Fault Class	β	γ	ε	ζ	η	θ	κ	λ	μ	ν	ξ	ο	Total
(1) Typo	0	0	0	1	2	2	0	0	0	0	0	0	5
(2) Omission	0	0	4	0	1	0	3	1	0	0	1	0	10
(3) Incorrect Algorithm	7	1	3	6	2	1	3	3	4	3	6	2	41
(4) Spec. Misinterpretation	2	2	0	1	1	4	3	3	4	2	2	4	28
(5) Spec. Ambiguity	0	4	3	0	0	0	0	1	0	0	1	0	9
(6) Other	0	0	0	1	1	0	0	0	1	0	0	0	3
Total	9	7	10	9	7	7	9	8	9	5	10	6	96

Table 3: Fault Classification by Fault Types

Table 4 shows the test phases during which the faults were detected, and the fault density (as per thousand lines of uncommented code, abbreviated as F.D.) of the original version and the accepted version. "F.D. after AT1" represents the fault density of the program versions after passing the Acceptance Test Step 1. Since the coverage of AT1 data was similar to the Acceptance Test data in the Six-Language Project, this snapshot of program versions was of particular interest to be compared with those final versions accepted in the Six-Language Project.

Test Phase	β	γ	ε	ζ	η	θ	κ	λ	μ	ν	ξ	ο	Total
Coding/Unit Test	2	2	3	1	3	3	5	3	2	1	2	2	29
Integration Test	4	3	4	4	1	0	3	2	2	2	3	1	29
Acceptance Test 1	1	2	3	4	1	2	1	2	3	2	5	3	29
Acceptance Test 2	1	0	0	0	2	2	0	1	2	0	0	0	8
Operational Test	1	0	0	0	0	0	0	0	0	0	0	0	1
Total	9	7	10	9	7	7	9	8	9	5	10	6	96
Original F.D.	2.2	5.7	11.2	9.7	4.7	5.9	7.2	3.2	7.7	4.7	5.9	4.4	5.1
F.D. after AT1	0.5	0	0	0	1.4	1.7	0	0.4	1.7	0	0	0	0.48
F.D. after AT2	0.2	0	0	0	0	0	0	0	0	0	0	0	0.05

Table 4: Fault Classification by Phases and Other Attributes

It is interesting to note that there was *only two* incidences of identical faults committed by two programs during the whole life cycle. The first fault, committed by θ version and μ version, was due to an incorrect initialization of a variable. Unit test data detected this fault very early when both programs were initially tested. The second fault, committed by γ and λ version, was an incorrect condition for a switch variable (a Boolean variable) for a late flight mode. This fault did not manifested itself until the Acceptance Test Step 1 where a complete flight simulation was first exercised.

It is interesting to compare the metrics of this project with those of the Six-Language Project [18], the NASA Experiment [17] [22], and the Knight-Leveson Experiment [16] [23]:

(1) Due to the revised NVS design paradigm which included a validated design process and a cleaner specification with better specification update policy, the program quality obtained from this project was higher. For the 12 finally accepted programs, the average F.D. was 0.05 faults per KLOC (thousand lines of code). This number is close to the field data from current industrial software engineering practice. As comparisons with similar experiments, the average F.D. of the finally accepted programs was 2.1 faults per KLOC in the Six-Language Project, 1.0 faults per KLOC in the NASA Experiment, and at least 2.8 faults per KLOC in the Knight-Leveson Experiment.

(2) Both this project and the Six-Language Project were guided by an NVS design paradigm, and the resulting identical faults were relative very low. Only two pairs of identical faults were found in the life cycle of either experiment (out of a total of 93 and 96 faults, respectively). In contrast, the NASA Experiment reported as many as 7 identical or similar faults after acceptance testing (out of a total of 26 faults), and the Knight-Leveson Experiment experienced 8 identical or similar faults after acceptance testing (out of a total of 45 faults.)

(3) Even the occurrences of identical or similar faults of this project remained as low as the Six-Language Project, the causes for these faults were quite different. In the Six-Language Project, the two identical faults were related to specification, while in this project, both identical faults were due to incorrect program initialization. Other experiments encountered identical or similar faults of various kinds, including specification deficiencies, voting routine mismatches, program initialization errors, code omissions, roundoff problems, and boundary case errors.

(4) In this project and the Six-Language Project, identical faults involving three or more versions have never been observed. This was not only true for program versions within each experiment, but also valid for cross-comparing all the accepted program versions (a total 18 of versions). On the other hand, the NASA Experiment observed identical or similar faults involving up to 5 versions, and the Knight-Leveson Experiment reported identical or similar faults spanning as many as 4 versions.

More comparisons among these experiments will be addressed in Section 7.

5. Operational Testing and NVS Reliability Evaluation

During the operational testing phase, 1000 flight simulations, or over five million program executions, were conducted. Only one fault (in the β version) was found during this operational testing phase.

To measure the reliability of the NVS system, we took the program versions which passed the Acceptance Test Step 1 for study. The reason behind this was that had the Acceptance Test not included an extreme situation of Step 2, more faults would have remained in the program versions. It would be interesting to see how the remaining faults would be manifested in this application, and how they would or would not be tolerated during the NVS operation.

Table 5 shows the errors encountered in each single version, while Tables 6 and 7 show different error categories under all combinations of 3-version and 5-version configurations. We examine two levels of granularity in defining execution errors and correlated errors: "by case" or "by time." The first level was defined based on test cases (1000 in total). If a version failed at *any* time in a test case, it was considered failed for the whole case. If two or more versions failed in the same test case (no matter at the same time or not), they were said to have coincident errors for that test case. The second level of granularity was defined based on execution time frames (5,280,920 in total). Errors were counted only at the time frame upon which they manifested themselves, and coincident errors were defined to be the multiple program versions failing at the same time in the same test case (with or without the same variables and values).

version	number of errors	error probability	
		by case	by time
β	510	.51	.000096574
γ	0	.00	.000000000
ε	0	.00	.000000000
ζ	0	.00	.000000000
η	1	.001	.000000189
θ	360	.36	.000068169
κ	0	.00	.000000000
λ	730	.73	.000138233
μ	140	.14	.000026510
ν	0	.00	.000000000
ξ	0	.00	.000000000
o	0	.00	.000000000
Average	145.1	.1451	.000027472

Table 5: Errors in Individual Versions in 1000 Flight Simulations

category	3-version configuration		5-version configuration	
	# of cases	probability	# of cases	probability
1 – no errors	163370	0.742590909	470970	0.594659095
2 – single error	51930	0.236045454	275740	0.348156565
3 – two coincident errors	4440	0.020181818	36980	0.046691919
4 – three coincident errors	260	0.001181818	8220	0.010378787
5 – four coincident errors	-	-	89	0.000112372
6 – five coincident errors	-	-	1	0.000001262
Total	220000	1.000000000	792000	1.000000000

Table 6: Errors by Case in 3-Version and 5-Version Execution Configurations

category	3-version configuration		5-version configuration	
	# of cases	probability	# of cases	probability
1 – no errors	1159930420	0.998388728	6789295000	0.998933423
2 – single error	1869280	0.001608948	7161240	0.001053659
3 – two coincident errors	2700	0.000002324	87800	0.000012918
4 – three coincident errors	0	0.000000000	0	0.000000000
Total	1161802400	1.000000000	6796544040	1.000000000

Table 7: Errors by Time in 3-Version and 5-Version Execution Configurations

From Table 5 we can see that the average error probability for single version is 0.1451 measured by case, or .00002747 measured by time. Table 6 shows that when measured by case, for all the 3-version combinations the error probability is .02136 (categories 3 and 4), an improvement over the single version by 7. In all the combinations of 5-version configuration, the error probability is .01049 (categories 4, 5, and 6), an improvement of 14.

From Table 7 we see that when measured by time, for all the 3-version combinations, the error probability is .000002324 (categories 3 and 4). This is a reduction of roughly 12 comparing with the single version execution. In all the combinations of 5-version configuration, the error probability is further reduced to 0 since there was no incidence of more than two coincident errors at the same time in the same case.

If we carry the error comparison granularity further to define coincident errors as the program versions failing with the same variables and values at the same time in the same test case, then no such coincident errors exist among the 12 program versions, resulting in perfect reliability improvement in both NVS configurations.

6. Mutation Testing and Coverage Analysis

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 [24] [25], was investigated in the 12 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 flight simulation environment to observe errors. (4) The error characteristics were analyzed to collect error statistics and correlations.

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 detecting errors 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 dimen-

sion of investigation in mutation testing.

Using the fault removal history of each version, we created a total of 96 mutants from the 12 program versions. 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.
- *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 [26]. 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.

According to the above definitions, the Error Frequency Function and Error Severity Function for

all the mutants were measured according to 100 flight simulation (500,000 executions) of each mutant program. As to the nature of errors in two or more versions, three types of relationships are identified: *distinct errors*, *similar errors*, and *identical errors* [27]. 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 an *Error Similarity Function*, $\sigma(x_1,..,x_n)$, for a set of mutants $\{x_1,..,x_n\}$ and a test set τ, to be

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

Based on these definitions, we 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 96 by 96, 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	$\theta 4$	$\mu 1$	$\gamma 6$	$\lambda 7$
$\theta 4$	–	1.0	0.0	0.0
$\mu 1$	1.0	–	0.0	0.0
$\gamma 6$	0.0	0.0	–	1.0
$\lambda 7$	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. The error similarity

function of the three-mutant sets is also considered as a sparse matrix, since we have not seen any common errors affecting more than two mutants.

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 [28]. 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 for measurement.

Since our main interest here is the analysis of the program versions themselves, without losing 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 earlier 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, measured by the testing of 100 simulations ($\tau = \{$ *100 data sets* $\}$), $C_1(\tau)$ represents the safety coverage factor of single-version software:

$$\begin{aligned} C_1(\tau) &= 1 - \frac{1}{C(96,1)} \sum_{x_1=1}^{x_1=96} \sigma(x_1;\tau) * \mu(x_1,\tau) \\ &= 1 - \frac{1}{96} \sum_{x_1=1}^{x_1=96} \mu(x_1,\tau) \\ &= 0.437 \end{aligned}$$

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

$$C_2(\tau) = 1 - \frac{1}{C(96,2)} \sum_{x_2=x_1+1}^{x_2=96} \sum_{x_1=1}^{x_1=95} \sigma(x_1,x_2;\tau) * \mu(x_1,\tau)$$
$$= 1 - 0.0002193\ (1*0.02 + 1*0.056)$$
$$= 0.99998$$

Furthermore,

$$C_3(\tau) = 0.99995,$$

$$C_n(\tau) = 1.0 \quad \text{for } n \geq 4.$$

Note that $C_2(\tau)$ should indeed be greater than $C_3(\tau)$, since it is known that 2-version systems are more capable of error detection than 3-version systems.

Evaluation of these safety coverage factors indicates an enormous improvement of N-version software over single-version software, which was similarly observed in the Six-Language Project [29]. It is important to note, however, that the coverage defined and measured here is limited to the particular mutant population and the 100 specific test data sets. We might argue that many of the faults injected in these mutants could be detected by a normal testing procedure. However, there is always a non-zero probability for each of these faults to slip through all practically applied testing schemes in another environment. As a result, sampling from this fault population is still valid, and the resulting measures are useful to provide an evidence for the effectiveness of NVS methodology to the assigned application.

7. Comparisons to Other Experiments

This project, guided by an evolving NVP design paradigm, revealed some major differences in the underlying process and the resulting product comparing with other similar NVS experiments. Table 9 illustrates an overall comparison between this project and three other NVS experiments: The Knight-Leveson Experiment [16], the NASA Experiment [17], and the Six-Language Project [18].

Item	Knight-Leveson	NASA	Six-Language	This Project
number of teams	27	20	6	15
number of accepted versions	27	20	6	12
program acceptance rate	100%	100%	100%	80%
average messages received per team	not available	150	58	20
average size (comments included)	about 600 lines	2175 lines	1782 lines	2558 lines
average size (uncommented)	not available	1246 lines	1209 lines	1564 lines
# inherent faults	not available	162	93	96
# faults during operation	45	26	11	1(9 after AT1)
inherent fault density (per KLOC)	not available	6.5	18	8.1
accepted version fault density (per KLOC)	2.8 or more	1.0	2.1	0.05 (0.48 after AT1)
# identical or similar faults	8	7	2	2
maximum span of identical or similar faults	4	5	2	2
3-version reliability improvement over single version	not available	2 to 5 (estimated)	2.3 (by time)	12 (by time)
5-version reliability improvement over single version	not available	8 to 20 (estimated)	18 (by time)	∞ (by time)

Table 9: Comparisons Among Several NVS Experiments

It is clear that the quality of the programs obtained in this project and the associated NVS reliability improvement were superior to those achieved in the previous experiments. In evaluating the differences among these NVS development projects, we noticed that the U. of Iowa/Rockwell Project took advantages of the previous projects in many aspects:

(1) A more diversified background of students (from academia and industry, in both areas of electrical engineering and computer science) and working environment.

(2) A better specification which was less error-prone and less ambiguous.

(3) A better error detection and recovery mechanism.

(4) A coordinator who was more familiar with the NVS development protocol.

(5) A better validation policy to enforce specification updates.

(6) A smoother team communication procedure.

(7) An initial testing under a designated NVX environment.

(8) A more thorough and more elaborated testing procedure to accept program versions.

In summary, these improvements were all contributed by the refined NVP design paradigm and the experience learned in conducting the process associated with this NVP design paradigm.

8. Conclusions

In this paper we demonstrated the effectiveness of our revised NVP design paradigm in improving software reliability by the provision of fault tolerance. The selected application was a real-world avionics control system which performed tedious computation-intensive tasks under severe testing constraints. We described the design diversity investigations, the program development phases, the testing configurations, the program properties, and the operational testing results in detail. We also applied the idea of mutation testing to measure several other reliability quantities, and demonstrated the software reliability improvement of the NVS configurations over the single-version configurations. We also compared our findings with those obtained from similar experiments conducted previously.

No single software engineering experiment or product could make revolutionary changes to software development practices overnight. Instead, modern software engineering techniques evolve through the refinement of software development processes gradually. We believe this will also be true for the fault-tolerant software techniques. Without a paradigm to guide the development and evaluation of NVS, software projects by nature can get out of control easily. The N-Version Programming design paradigm offers a documented process model which is subject to readjustment, tailoring, refinement, and improvement. Compared to previous NVS experiments, the U. of Iowa/Rockwell Project, which was conducted based on this evolving paradigm, confirmed that the NVS product improvement could come largely from the existence and

improvement of the NVS development process. Only through the experience in this process evolution could we build up enough confidence in applying and managing the N-Version Programming techniques successfully for more practical applications.

Acknowledgement

The authors wish to thank the following individuals who participated in this project: B. Ajango, T. Beckmann, N. Bloom, S. Caswell, B. Cholasamvdram, D. Clark, R. Dietz, S. Evans, D. Goetzinger, V. Goyal, D. Haverkamp, W. Headlee, J. Hsu, J. Kohl, M. Krenz, J. Laird, S. Lee, J. Lumpp, B. Mehta, R. Miller, C. Minter, R. Mostaert, D. Newmister, C. Park, W. Park, C. Rajaraman, R. Reddy, K. Sangareddi, S. Seth, S. Shafer, M. Steffensmeier, Y. Sun, S. Sundaram, P. Sunkerneni, R. Tangirala, J. Trepka, W. Wang, P. Wiley, and M. Wolf.

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. 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.
3. 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.
4. A. Costes, C. Landrault, and J.C. Laprie, ‘‘Reliability and Availability Models for Maintained Systems Featuring Hardware Failures and Design Faults,’’ *IEEE Transactions on Computers*, vol. C-27, no. 6, pp. 548-560, June 1978.
5. A. Grnarov, J. Arlat, and A. Avižienis, ‘‘On the Performance of Software Fault-Tolerance Strategies,’’ in *Digest of 10th Annual International Symposium on Fault-Tolerant Computing*, pp. 251-253, Kyoto, Japan, 1980.
6. D.E. Eckhardt and L.D. Lee, ‘‘A Theoretical Basis for the Analysis of Multiversion Software Sub-

ject to Coincident Errors,'' *IEEE Transaction on Software Engineering*, vol. SE-11, no. 12, pp. 1511-1517, December 1985.

7. 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.

8. B. Littlewood and D. Miller, ''Conceptual Modeling of Coincident Failures in Multiversion Software,'' *IEEE Transactions on Software Engineering*, vol. 15, no. 12, pp. 1596-1614, December 1989.

9. V.F. Nicola and A. Goyal, ''Modeling of Correlated Failures and Community Error Recovery in Multiversion Software,'' *IEEE Transactions on Software Engineering*, vol. SE-16, no. 3, pp. 350-359, March 1990.

10. J.C. Laprie and K. Kanoun, ''X-Ware Reliability and Availability Modeling,'' *IEEE Transactions on Software Engineering*, vol. SE-18, no. 2, pp. 130-147, February 1992.

11. L. Chen, ''Improving Software Reliability by N-Version Programming,'' ENG-7843, Ph.D. Dissertation, UCLA Computer Science Department, Los Angeles, California, August 1978.

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

13. 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.

14. J.P.J. Kelly and A. Avižienis, ''A Specification Oriented Multi-Version Software Experiment,'' *Digest of 13th Annual International Symposium on Fault-Tolerant Computing*, pp. 121-126, Milan, Italy, June 1983.

15. P.G. Bishop, D.G. Esp, M. Barnes, P. Humphreys, G. Dahl, and J. Lahti, ''PODS - A Project of Diverse Software,'' *IEEE Transactions on Software Engineering*, vol. SE-12, no. 9, pp. 929-940, September 1986.

16. 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.

17. 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.

18. 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.

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

20. M.R. Lyu and A. Avižienis, ‘‘Assuring Design Diversity in N-Version Software: A Design Paradigm for N-Version Programming,’’ in *Dependable Computing and Fault-Tolerant Systems*, ed. J.F. Meyer R.D. Schlichting, pp. 197-218, Springer-Verlag/Wien, Austria, 1992.

21. M.R. Lyu, ‘‘Software Requirements Document for a Fault-Tolerant Flight Control Computer,’’ ECE55:195 Project Specification, p. 64 pages, Iowa City, Iowa, September 1991.

22. M.A. Vouk, D.F. McAllister, A.K. Caglayan, J.L. Walker, Jr., D.E. Eckhardt, J.P.J. Kelly, and J. Knight, ‘‘Analysis of Faults Detected in a Large-Scale Multi-Version Software Development Experiments Environment,’’ in *DASC’90 Proceedings*, pp. 378-385.

23. S.S. Brilliant, J.C. Knight, and N.G. Leveson, ‘‘Analysis of Faults in an N-Version Software Experiment,’’ *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 238-247, February 1990.

24. 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.

25. 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.

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

27. 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.

28. 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.

29. M.R. Lyu, ‘‘Software Reliability Measurements in N-Version Software Execution Environment,’’ in *Proceedings of 1992 International Symposium on Software Reliability Engineering*, pp. 254-263, Raleigh, North Carolina, October 1992.