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

Michael R. Lyu, Member IEEE
Bell Communications Research, Morristown
Yu-Tao He, Student Member IEEE
The University of Iowa, Iowa City

Key Words — **N-Version programming, Design paradigm, Experiment, Process evolution, Mutation testing, Reliability analysis, Coverage analysis**

Reader Aids —
Purpose: Present a practical software-design paradigm
Special math needed for explanations: Probability
Special math needed to use results: None
Results useful to: Fault-tolerant software designers/experimenters and reliability analysts

Summary & Conclusions — **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 led to the leak of a pair of coincident faults. This paper reports on a similar project that used a revised NVP design paradigm. This project reused the revised specification of a real, automatic airplane-landing problem, and involved 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 paper identifies & presents: the impact of the paradigm on the software development process; the improvement of the resulting NVS product; the insight, experience, and learning in conducting this project; various testing procedures applied to the program versions; several quantitative measures of the resulting NVS product; and some comparisons with previous projects. The effectiveness of our revised NVP design paradigm in improving software reliability by the provision of fault tolerance is demonstrated.**

We found that no single software engineering experiment or product can make revolutionary changes to software development practices overnight. Instead, modern software engineering techniques evolve through the refinement of software development processes. This is true for 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 projects, this project (based on this evolving paradigm) confirmed that NVS product improvement could come largely from the existence and improvement of the NVS development process. Only through continuous improvement in this process evolution could we build enough confidence in applying and engineering the N-Version Programming techniques successfully for more practical applications.

1. INTRODUCTION

Acronyms

FD	fault density
KLOC	thousand (kilo) lines of code
L-DP	Lyu design paradigm [19][1]
NVP	N-version programming
NVS	N-version software
NVX	N-version executive
RI	Rockwell International, Collins Avionics Div
UI	University of Iowa
UI&RI	UI and RI
UCLA/H6LP	UCLA/Honeywell Six-Language Project.

NVP approach achieves fault-tolerant NVS systems, through the development and use of design diversity [1]. Such systems are usually operated in an NVX environment [2]. The idea of NVP was first proposed in [3]. Since then, there have been numerous papers on modeling NVS systems [4-10] and on empirical studies of NVS systems [11-18]. The effectiveness of NVP, however, has remained highly controversial and inconclusive. Most researchers and practitioners agree that a high degree of version independence and a low probability of failure correlation are important in the success of an NVS deployment.

To maximize the effectiveness of NVP, the probability of similar errors that coincide at the NVS decision points must be reduced to the lowest possible value. To achieve this for NVS, a rigorous NVP development paradigm (initial L-DP) was proposed [19]. It provided disciplined practices in modern software engineering techniques, and incorporated recent knowledge and experience obtained from fault-tolerant system design principles. The main purpose of L-DP was to encourage the investigation and implementation of practical NVS techniques.

The first application of L-DP to a real project was reported in [18] for an extensive evaluation. Some limitations were presented [20], leading to a refined L-DP. To observe the impact of the refined L-DP, a similar project was conducted at UI&RI; this paper is a comprehensive report on this project. Section 2 examines the L-DP. Section 3 describes the UI&RI project. Sections 4-6 thoroughly evaluate the NVS product, including program metrics and statistics (section 4), operational testing and NVS reliability evaluation (section 5), and fault-injection testing for a coverage analysis of the NVS systems (section 6). Section 7 compares this project with three previous projects [16-18].

[1] *Editors' note*: To facilitate reference to this design paradigm, we have assigned the acronym L-DP to it.

0018-9529/93/$3.00 ©1993 IEEE

Standard notation & nomenclature are given in "Information for Readers & Authors" at the rear of each issue.

2. L-DP: ITS ORIGIN & REFINEMENT

NVP is [3]: "the independent generation of $N \geq 2$ functionally equivalent programs from the same initial specification." *Independent generation* means 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 [3]: "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."

The key NVP research effort has been addressed by the formation of a set of guidelines for systematic design to implement NVS systems, in order to achieve efficient tolerance of design faults in computer systems. The evolution of these rigorous guidelines & techniques was formulated [19] as an NVS *design paradigm*[2] by integrating the knowledge and experience obtained from both software engineering techniques and fault tolerance investigations.

The 3 objectives of the design paradigm are to:

- Reduce the possibility of oversights, mistakes, and inconsistencies in the process of software development and testing
- Eliminate most perceivable causes of related design faults in the independently generated versions of a program, and identify causes of those which slip through the design process
- 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 to:

- Attain suitable isolation and independence (with respect to software faults) of the N concurrent version development efforts
- Encourage potential diversity among the multiple versions of an NVS system
- Elaborate efficient error detection & recovery mechanisms. ◄

The first two reduce the chances of related software faults being introduced into two or more versions via potential fault links, such as:

- casual conversations or mail exchanges,
- common flaws in training or in manuals,
- using the same faulty compiler.

[2]A pattern, example, or model that refers to a set of guidelines and rules, with illustrations.

The last procedure increases the probability of discovering manifested errors before they are converted to coincident failures. Figure 1 describes L-DP.

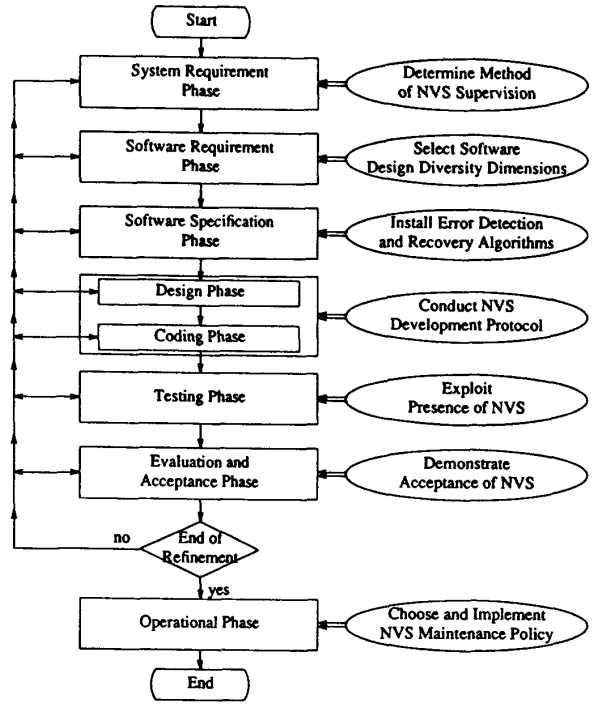

Figure 1. L-DP for N-Version Programming

L-DP has 2 categories of activities:

- Category 1 (boxes and single-line arrows on the l.h.s.) contains typical software development procedures.
- Category 2 (ovals and double-line arrows on the r.h.s.) describes the concurrent enforcement of fault-tolerant techniques under the N-version programming environment.

Table 1 summarizes the activities and guidelines incorporated in each life-cycle phase of software development [19]. The initial L-DP was refined from experience [18]. These refinements are not dramatically different from the initial L-DP. Nevertheless, they are important in avoiding two types of specification-related faults:

- Absence of appropriate responses to specification updates
- Incorrect handling of matching features required by an NVX environment (*eg*, the placement of voting routines).

3. The UI&RI Project

The main purposes in formulating a design paradigm are to:

- eliminate all identifiable causes of related design faults in the independently generated versions of a program,

TABLE 1
Detailed Layout of L-DP
[*Implies a refinement in the refined L-DP]

Software Life Cycle Phase	Enforcement of Fault Tolerance	Design Guidelines & Rules
System Requirement	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	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	Install Error Detection & 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 & Coding*	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*	Exploit Presence of NVS*	1. explore comprehensive verification procedures 2. enforce extensive validation efforts 3. provide opportunities for "back-to-back" testing 4. perform preliminary NVS test under the designated NVX*
Evaluation & Acceptance	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	Choose & 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 & maintenance

- prevent all potential effects of coincident run-time errors while executing these program versions. ◄

An investigation is necessary in which the complexity of the application software reflects a realistic size in highly critical applications. Such investigation must execute & evaluate the design paradigm, and be complete, in the sense that it thoroughly explores all aspects of NVS systems as software fault-tolerant systems. This effort was first conducted in the UCLA/H6LP [18, 20]. There, a real automatic (computerized) airplane landing system (autopilot) was developed and programmed by six programming teams, in which the L-DP was executed, validated, and refined.

In the fall of 1991, a similar project was conducted at UI&RI. Guided by the refined L-DP and a requirement specification with the known defects removed, 40 students (33 from ECE & CS departments at UI, 7 from RI) formed 15 programming teams (12 from UI, 3 from RI) to independently design, code, and test the computerized airplane landing system as the major requirement for a graduate-level software engineering course.

Sections 3.1 - 3.5 describe: a) how the refined L-DP was applied in conducting this project, and b) 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 important command errors could be detected, random wind turbulence of various levels were superimposed to represent difficult flight conditions. The individual commands were recorded and compared for discrepancies that could indicate the presence of faults.

The 3-channel flight simulation system (shown in figure 2) consisted of 3 lanes of control-law computation, three command monitors, a servo control, an airplane model, and a turbulence generator.

The lane-computations and command-monitors were 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

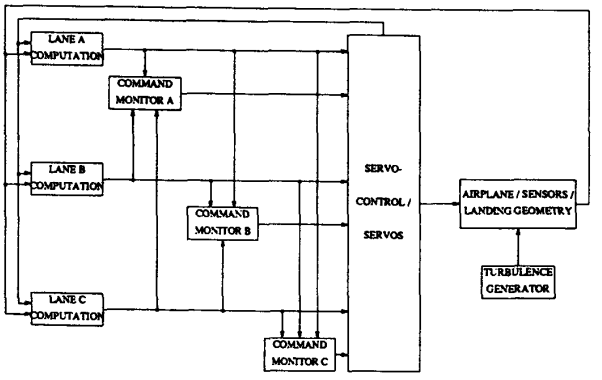

Figure 2. 3-Channel Flight-Simulation Configuration

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 median of the three inputs became the final elevator command. The landing-geometry and turbulence-generator were models associated with the airplane simulator.

Each run of flight simulation was characterized by 5 initial values regarding the landing position of an airplane:

- initial altitude (about 1500 feet)
- initial distance (about 52800 feet)
- initial nose up relative to velocity (range from 0 to 10 degrees)
- initial pitch attitude (range from −15 to 15 degrees)
- vertical velocity for the wind turbulence (0 to 10 ft/sec).

Each simulation consisted of about 5280 iterations of lane command computations (50 msec each) for a total landing time of approximately 264 seconds.

3.2 *Design-Diversity Investigations*

Independent programming teams were the baseline design-diversity. The programming teams represented a wide range of experience, from very experienced programmers (and/or avionics engineers) to novices. Three programming environments were provided to the programmers:

- Iowa Computer-Aided Engineering Network (ICAEN) Center
- UI Dept. Electrical & Computer Engineering computing facility
- RI computing facilities.

Every programming team was required to use the C programming language.

It was hypothesized that various programming environments provided different hardware platforms, working atmosphere, and computing tools and facilities which might add further development diversity.

3.3 *High Quality Specification with Error Detection & Recovery*

The development of a suitable specification began early in 1987; it was initially used in the UCLA/H6LP. During software generation in UCLA/H6LP, several errors and ambiguities in the specification were revealed & corrected — including the two specification defects resulting in two pairs of identical faults. A comprehensive error detection and recovery algorithm was imposed on the specification to include:

- 2 input routines,
- 7 vote routines (cross-check points) to cross-check data,
- 1 recovery routine.

These enhancements were carefully specified and inserted in the initial specification.

The final specification was converted to a single document [21] which was given to the 15 programming teams to develop their program versions independently. This specification has benefited from the scrutiny of more than 16 motivated programmers and researchers. This version of specification followed the principle of supplying only minimal (absolutely necessary) information to the programmers, so as not to unwittingly bias the programmers' design decisions nor 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 prohibited the programmers from discussing any aspect of their work with members of other teams. Work-related communication between programmers and a project coordinator were conducted only via electronic mail, which also served as a documentation protocol. 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, usually in less than 24 hours. The purpose of imposing this isolation rule on programming teams was to assure the independent generation of programs, *ie*, programming efforts are carried out by individuals or groups that do not interact with respect to the programming process.

In the communication protocol, each answer was sent only to the team that submitted the question. The answer was broadcast to all teams if and 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. Thus the individual teams received an average of only 20 messages during the program development phase, in contrast to an average of 58 messages in the UCLA/H6LP, and to an average of over 150 messages in an NASA Experiment [17]. A lesson learned from the UCLA/H6LP, the revised L-DP enforced that each program version was verified to comply

with all the broadcast specification updates before its final acceptance.

3.5 *NVS Software Development Schedule*

The software development was scheduled and conducted in 6 phases:

- Initial Design (4 weeks)

This phase allowed the programmers to get familiar with the problem. At the end of this phase, each team delivered a preliminary design document which followed specific guidelines and formats for documentation.

- Detailed Design (2 weeks)

This phase let each team obtain some feedback from the coordinator to adjust, consolidate, and complete its final design. The feedback related to the feasibility & style of each design rather than to any technical corrections, except when the design violated the specification updates. Each team was requested to conduct at least one design walk-through. At the end of this phase, each team delivered a detailed design document and a design walk-through report.

- Coding (3 weeks)

By the end of this phase, programmers had: a) finished coding, b) conducted a code walk-through by themselves, and c) delivered the initial compilable code. From this moment on, each team was required to use the revision control tool RCS (or to include CVS for concurrent versions) for configuration management of their program modules. Code-update report forms were used for every change made after the code was generated.

- Unit Testing (1 week)

Each team was supplied with sample test data-sets for each module to check the basic functionality of that module. They were required to build their own test harness for this testing purpose. 133 data sets (the same as in UCLA/H6LP) were provided to the programmers.

- Integration Testing (2 weeks)

Four sets of partial flight-simulation test data (the same as those in the UCLA/H6LP), together with an automatic testing routine, were provided to each programming team for integration testing. This test phase was intended to guarantee that the software was suitable for a flight-simulation environment in an integrated system.

- Acceptance Testing (2 weeks)

Programmers formally submitted their programs for a 2-step acceptance test. In step 1 (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 UCLA/H6LP. In step 2 (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 all, there were 23930 executions imposed on these programs before they were accepted and subjected to the final evaluation. By the end of this phase, 12 programs passed the acceptance test and were used in the final evaluation.

4. PROGRAM METRICS / 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 of program versions, since design diversity is our major concern.

Table 2 considers the following metrics.

LINES: number of lines of code, including comments and blank lines

TABLE 2
Software Metrics for the 12 Accepted Programs

Metrics	β	γ	ϵ	ζ	η	θ	κ	λ	μ	ν	ξ	o	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

LN-CM: number of lines, excluding comments and blank lines
STMTS: number of executable statements, such as assignment, control, I/O, or arithmetic
MODS: number of programming modules (*eg*, subroutines, functions, procedures)
STM/M: mean number of statements per module
STM/CCP: mean number of statements in between cross-check points [19]
CALLS: number of calls to programming modules
GBVAR: number of global variables
LCVAR: number of local variables
BINDE: number of binary decisions.

96 faults were found and reported during the life cycle of the project. Table 3 classifies the faults according to [19]:

Implementation Related

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); it includes miscomputation, logic fault, initialization fault, and boundary fault.

Specification Related

4. specification misinterpretation (a misinterpretation of the specification)
5. specification ambiguity (an unclear or inadequate specification which led to a deficient implementation). ◄

Fault type 3 is the most frequent. This result is similar to that in the UCLA/H6LP.

Table 4 shows the test phases during which the faults were detected, and the faults/KLOC (uncommented), of the original and accepted versions. Since the coverage of AT1 data was similar to the Acceptance Test data in the UCLA/H6LP, this snapshot of program versions was of particular interest to be compared with those final versions accepted in the UCLA/H6LP.

There were only 2 incidents of identical faults committed by two programs during the whole life cycle. The first fault, committed by θ & μ versions, was due to an incorrect initialization of a variable. Unit test-data detected this fault when both programs were initially tested. The second fault, committed by γ & λ versions, was an incorrect condition for a switch variable (a Boolean variable) for a late flight mode. This fault did not show up until AT1 where a complete flight simulation was first exercised.

The metrics of this project are compared with those of UCLA/H6LP [18], NASA Experiment [17, 22], and Knight-Leveson Experiment [16, 23].

- Due to the refined L-DP 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 FD was 0.05 faults/KLOC. This number is close to the field data from the best current industrial software

TABLE 3
Number of Faults in Each Type

Fault Type	β	γ	ϵ	ζ	η	θ	κ	λ	μ	ν	ξ	o	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 4
Fault Types by Phases and Other Attributes

Test Phase	β	γ	ϵ	ζ	η	θ	κ	λ	μ	ν	ξ	o	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 (AT1)	1	2	3	4	1	2	1	2	3	2	5	3	29
Acceptance Test 2 (AT2)	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 FD	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
FD after AT1	0.5	0	0	0	1.4	1.7	0	0.4	1.7	0	0	0	0.48
FD after AT2	0.2	0	0	0	0	0	0	0	0	0	0	0	0.05

engineering practice. Compared with similar projects, the average FD of the finally accepted programs was:

- 2.1 faults/KLOC in UCLA/H6LP
- 1.0 faults/KLOC in the NASA Experiment
- > 2.8 faults/KLOC in the Knight-Leveson Experiment.

- Both this project and the UCLA/H6LP were guided by an NVS design paradigm, and the resulting identical faults were relative very low. Only two pair of identical faults were found in the life cycle of either project (out of a total of 93 and 96 faults, respectively). In contrast, the NASA Experiment reported as many as 7 types of identical or similar faults after acceptance testing (out of a total of 26 faults), and the Knight-Leveson Experiment experienced 8 types of identical or similar faults after acceptance testing (out of a total of 45 faults).
- Even though the number of identical or similar faults of this project were as low as the UCLA/H6LP, the causes for these faults were quite different. In the UCLA/H6LP the two identical faults were related to specification, while in this project, both identical faults were due to incorrect program initialization. Other projects 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.
- In this project and UCLA/H6LP, identical faults involving three or more versions have never been observed. 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.

5. OPERATIONAL TESTING AND NVS RELIABILITY EVALUATION

During the operational testing phase, 1000 flight simulations (over 5×10^6 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 AT1 for study. The reason was: Had the Acceptance Test not included an extreme situation of AT2, more faults would have remained in the program versions.

Table 5 shows the errors encountered in each single version, while tables 6 & 7 show various 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* — based on 1000 test cases. Each case contained about 5280 execution time frames. 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 (whether at the same time or not), they were said to have coincident errors for that test case.
- *by time* — based on 5280920 execution time frames of the 1000 test cases. Errors were counted only at the time frame in 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).

TABLE 5
Errors in Individual Versions in 1000 Flight Simulations

version	number of errors	error probability	
		by case (%)	by time (10^{-6})
β	510	51	96.6
γ	0	0	.000
ϵ	0	0	.000
ζ	0	0	.000
η	1	.1	.189
θ	360	36	68.2
κ	0	0	.000
λ	730	73	138
μ	140	14	26.5
ν	0	0	.000
ξ	0	0	.000
o	0	0	.000
Average	145	14.5	27.47

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

category	3-version configuration		5-version configuration	
	incidents	prob. (%)	incidents	prob. (%)
1. 0 errors	163370	74.3	470970	59.5
2. 1 error	51930	23.6	275740	34.8
3. 2 coincident errors	4440	2.0	36980	4.67
4. 3 coincident errors	260	.118	8220	1.04
5. 4 coincident errors	—	—	89	.011
6. 5 coincident errors	—	—	1	.00012
Total	220000	1.00	792000	1.00

Table 5 shows that the average error probability for single version is 14.5% measured by case, or 0.0027% measured by time. Table 6 shows that when measured by case, for all the 3-version combinations the error probability is 2.1% (categories 3 & 4), an improvement over the single version by a factor of 7. In all the combinations of 5-version configuration, the error probability is 1.0% (categories 4 - 6), an improvement by a factor of 14.

Table 7 shows that, when measured by time, for all the 3-version combinations, the error probability is 0.0002% (categories 3 & 4). This is a reduction by a factor of 13 compared 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 2 coincident errors at the same time in the same case.

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

	3-version configuration		5-version configuration	
category	incidents (10^3)	prob. (%)	incidents (10^3)	prob. (%)
1. 0 errors	1160744	99.909	6778421	99.733
2. 1 error	1056	.0909	18036	.265
3. 2 coincident errors	2.7	.0002	87.8	.0013
4. 3 coincident errors	0	.0000	0	.0000
Total	1161802	100	6796544	100

If we analyze the error comparison level in more detail 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 in both NVS configurations.

6. FAULT-INJECTION 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 by fault-injection, 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 mutation testing is to ensure the quality of the test data used to verify a program; our concerns here are to examine the relationship of faults and error frequencies in each program and to evaluate the similarity of program errors among different versions. The fault-injection testing procedure is:

- The fault removal history of each program was examined and each program fault was analyzed and recreated.
- Mutants[3] were generated by injecting faults one by one into the final version — from which they were removed; *eg*, a fault from program C is injected only into program C. Each mutant contains exactly one known software fault.
- Each mutant was executed by the same set of input data in the flight simulation environment to observe errors.
- The error characteristics were analyzed to collect error statistics and correlations. ◄

The 2 differences between our fault-injection technique and the mutation testing are:

- We used "real" mutants, *viz*, mutants injected with actual faults committed by programmers, instead of mutants with hypothesized faults.
- Our purpose was to measure the coverage of NVS in detecting errors during operation, not merely the coverage of the test data in detecting mutants during testing. When there are multiple realizations of the same application, test data are no longer the only means for fault treatment and coverage analysis. Study of the error correlations among multiple program versions offers another dimension of investigation in fault-injection testing and mutation testing.

Using the fault removal history of each version, we created a total of 96 mutants from the 12 program versions. Several error-measures were created.

Notation

i	serial number of mutant
n	number of mutants
x	software version
x_i	software version x with mutant i
$\boldsymbol{x}$	$x_1, \ldots, x_n$
m	population size (for the x_i)
τ	serial number of test data-set
$\lambda(x_i,\tau)$	error frequency function
$\mu(x_i,\tau)$	error severity function
$\sigma(\boldsymbol{x};\tau)$	error similarity function
$N_{\text{err}}(x_i,\tau)$	total number of errors
$N_{\text{exe}}(x_i,\tau)$	total number of executions
ϵ	allowed error deviation (for severity)
$\text{val}(x_i,\tau)$	computed (erroneous) value of mutant x_i running τ
$\text{val}(x,\tau)$	anticipated (corrected) value of software x running τ
$\mu^*(x_i,\tau)$	relative error: $\lvert 1 - \text{val}(x_i,\tau)/\text{val}(x,\tau)\rvert$
$C_n(\tau)$	safety coverage factor for an n-mutant system and τ.

$$\lambda(x_i,\tau) \equiv N_{\text{err}}(x_i,\tau)/N_{\text{exe}}(x_i,\tau)$$

Assumptions

1. Each mutant contains only one known fault.
2. Errors produced by each fault are always the same for the same test inputs [26].
3. ϵ is specified
4. The NVS supervisory system does not introduce errors which corrupt program execution. ◄

$$\mu(x_i,\tau) \equiv \begin{cases} 0, \text{ for } \mu^*(x_i,\tau) < \epsilon \\ 1, \text{ for } \mu^*(x_i,\tau) > 1 \text{ (includes run-time exceptions or no results)} \\ \mu^*(x_i,\tau), \text{ otherwise.} \end{cases}$$

The $\lambda(x_i,\tau)$ & $\mu(x_i,\tau)$ were measured for all the mutants in 100 flight simulation (0.5×10^6 executions) of each mutant program. Three types of relationships (for errors in two or more versions) are identified [27]:

[3]This word is used for lack of a better, more appropriate one.

- distinct: produced by faults whose erroneous results can be distinguished
- similar: two or more erroneous results that are within a small range
- identical: erroneous results are identical.

$$\sigma(x;\tau) \equiv \begin{cases} 0, \text{ if the majority of } \{x\} \text{ produce distinct errors in } \tau \\ 1, \text{ otherwise.} \end{cases}$$

The $\sigma(x;\tau)$ were measured for populations of two versions. Table 8 shows the reduced $\sigma(x;\tau)$ matrix for 2-mutant sets.[4] The two incidents of indistinguishable errors shown in table 8, *ie*, $\theta 4$ (mutant #4 of version #θ) — $\mu 1$ and r6 — $\lambda 7$, result from the two pairs of identical faults discussed in section 4.

TABLE 8
$\sigma(x;\tau)$ Matrix [reduced] in 2-Mutant Sets

σ	$\theta 4$	$\mu 1$	$\gamma 6$	$\lambda 7$
$\theta 4$	—	1	0	0
$\mu 1$	1	—	0	0
$\gamma 6$	0	0	—	1
$\lambda 7$	0	0	1	—

Analysis of 3-mutant sets is much more tedious since a 3-dimensional matrix is needed. However, it is similar to the analysis of 2-mutant sets. The error similarity function of the 3-mutant sets is also a sparse matrix, since we have not seen any common errors affecting more than two mutants.

Based on the 3 *Error Functions* we can obtain another reliability-related quantity, safety coverage, which is important for assessing the effectiveness of fault-tolerant systems. The safety coverage factor is defined as [28]:

$$\Pr\{\text{error detection or recovery} \mid \text{a fault has manifested itself}\}.$$

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.

The main contribution of the failures of NVS schemes is the similar or identical error among multiple program versions. We use the mutants (generated in the tests) as the sample-space to represent the condition: A fault has been created. Since this fault is manifested in errors during program execution, the probability that it is covered is related to its *severity* and its *similarity* to errors of other program versions. If we assume an equal distribution on the mutant population, then the safety coverage factor is:

$$C_n(\tau) \equiv 1 - [1/\binom{m}{n}] \cdot \Sigma_{1 \le x_1 \le \ldots \le x_n \le m} \sigma(x;\tau) \cdot \mu(\text{median}\{x\},\tau).$$

Example 1

Let: τ = 100 data sets, $n = 1$.

$$C_1(\tau) = 1 - [1/\binom{96}{1}] \cdot \sum_{x_1=1}^{96} \sigma(x_1;\tau) \cdot \mu(x_1,\tau)$$

$$= 1 - [1/96] \cdot \sum_{x_1=1}^{96} \mu(x_1,\tau) = 0.437.$$

Example 2

Let: τ = 100 data sets, $n = 2$.

$$C_2(\tau) = 1 - [1/\binom{96}{2}] \cdot \sum_{x_2=x_1+1}^{96} \sum_{x_1=1}^{95} \sigma(x_1,x_2;\tau) \cdot \mu(x_1,\tau)$$

$$= 1 - 0.0002193 \cdot (1 \cdot 0.02 + 1 \cdot 0.056) = 0.99998$$

Example 3

Let: τ = 100 data sets, $n = 3$.

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

Example 4

Let: τ = 100 data sets, $n > 3$.

$$C_n(\tau) = 1.00000.$$

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

These safety coverage factors indicate an enormous improvement of NVS over 1-version software, which was similarly observed in UCLA/H6LP [29]. The coverage defined and measured here is limited to the particular mutant population and the 100 specific test data sets. Many of the faults injected in these mutants could, perhaps, be detected by an ordinary 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. Consequently, sampling from this fault population is valid, and the resulting measures are useful evidence for the effectiveness of NVS methodology to the assigned application.

7. COMPARISONS WITH OTHER PROJECTS

This project, guided by an evolving L-DP, revealed some major differences in the underlying process and the resulting

[4]The complete matrix is 96×96, but since it is sparse (most entries are zero), it was reduced by removing many of the 0 entries.

product compared with similar NVS projects. Table 9 compares this project and 3 other NVS projects: Knight-Leveson [16], NASA [17], and UCLA/H6LP [18].

The program quality in our UI&RI project and the associated NVS reliability improvement were superior to those in previous projects. In evaluating the differences among these NVS development projects, our UI&RI project learned from previous projects in many ways:

- A more diversified background of students (from academia and industry, in both electrical engineering and computer science) and working environment.
- A specification which was less error-prone and less ambiguous.
- A better error detection and recovery mechanism.
- A coordinator who was more familiar with the NVS development protocol.
- A better validation policy to enforce specification updates (part of L-DP revisions).
- A smoother team communication procedure.
- An initial testing under a designated NVX environment (part of L-DP revisions).
- A more thorough & elaborate acceptance procedure for program versions.

These improvements were due to the refined L-DP and the experience in conducting the process associated with this L-DP.

ACKNOWLEDGMENT

We are pleased to thank the following people who participated in this UI&RI 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 Trans. Software Engineering*, vol SE-11, 1985 Dec, pp 1491–1501.

[2] A. Avižienis, Lyu, Schütz, Tso, Voges, "DEDIX 87 - A supervisory system for design diversity experiments at UCLA", *Software Diversity in Computerized Control Systems* (*ed* U. Voges), 1988, pp 129–168; Springer-Verlag.

[3] A. Avižienis, L. Chen, "On the implementation of n-version programming for software fault-tolerance during program execution", *Proc. COMPSAC 77*, 1977, pp 149–155.

[4] A. Costes, C. Landrault, J. C. Laprie, "Reliability and availability models for maintained systems featuring hardware failures and design faults", *IEEE Trans. Computers*, vol C-27, 1978 Jun, pp 548–560.

[5] A. Grnarov, J. Arlat, A. Avižienis, "On the performance of software fault-tolerance strategies", *Digest 10th Ann. Int'l Symp. Fault-Tolerant Computing*, 1980, pp 251–253; Kyoto, Japan.

[6] D. E. Eckhardt, L. D. Lee, "A theoretical basis for the analysis of multiversion software subject to coincident errors", *IEEE Trans. Software Engineering*, vol SE-11, 1985 Dec, pp 1511–1517.

[7] A R. K. Scott, J. W. Gault, D. F. McAllister, "Fault tolerant software reliability modeling", *IEEE Trans. Software Engineering*, vol SE-13, 1987 May, pp 582–592.

[8] B. Littlewood, D. Miller, "Conceptual modeling of coincident failures in multiversion software", *IEEE Trans. Software Engineering*, vol 15, 1989 Dec, pp 1596–1614.

[9] V. F. Nicola, A. Goyal, "Modeling of correlated failures and community error recovery in multiversion software", *IEEE Trans. Software Engineering*, vol 16, 1990 Mar, pp 350–359.

[10] J. C. Laprie, K. Kanoun, "X-ware reliability and availability modeling", *IEEE Trans. Software Engineering*, vol 18, 1992 Feb, pp 130–147.

[11] L. Chen, "Improving software reliability by n-version programming", *PhD Dissertation*, ENG-7843, 1978 Aug; UCLA Computer Science Department, Los Angeles, California.

TABLE 9
Comparison of Several NVS Projects

Item	Knight-Leveson	NASA	UCLA/H6LP	UI/RI
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/team	n.a.	150	58	20
average size, with comments (lines)	≈ 600	2175	1782	2558
average size, uncommented (lines)	n.a.	1246	1209	1564
number of inherent faults	n.a.	162	93	96
number of faults during operation	45	26	11	1 (9 after AT1)
inherent faults/KLOC	n.a.	6.5	18	8.1
accepted-version faults/KLOC	> 2.8	1.0	2.1	0.05 (0.48 after AT1)
number of ident/sim faults	8	7	2	2
max. span of ident/sim faults	4	5	2	2
3-version improvement*	n.a.	2 to 5 (est)	2.3 (by time)	13 (by time)
5-version improvement*	n.a.	8 to 20 (est)	18 (by time)	∞ (by time)

n.a. = not available; ident/sim = *identical* or *similar*; est = estimated
*improvement is the factor by which the failure probability is reduced, compared to 1-version.

[12] L. Gmeiner, U. Voges, "Software diversity in reactor protection systems: An experiment", *Proc. IFAC Workshop SAFECOMP'79*, 1979 May, pp 75-79; Stuttgart, Germany.
[13] C. V. Ramamoorthy, Mok, Bastani, Chin, Suzuki, "Application of a methodology for the development and validation of reliable process control software", *IEEE Trans. Software Engineering*, vol SE-7, 1981 Nov, pp 537-555.
[14] J. P. J. Kelly, A. Avižienis, "A specification oriented multi-version software experiment", *Digest 13th Ann. Int'l Symp. Fault-Tolerant Computing*, 1983 Jun, pp 121-126; Milan, Italy.
[15] P. G. Bishop, Esp, Barnes, Humphreys, Dahl, Lahti, "PODS - A project of diverse software", *IEEE Trans. Software Engineering*, vol SE-12, 1986 Sep, pp 929-940.
[16] J. C. Knight, N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming", *IEEE Trans. Software Engineering*, vol SE-12, 1986 Jan, pp 96-109.
[17] D. E. Eckhardt, Caglavan, Knight, Lee, McAllister, Vouk, Kelly, "An experimental evaluation of software redundancy as a strategy for improving reliability", *IEEE Trans. Software Engineering*, vol 17, 1991 Jul, pp 692-702.
[18] A. Avižienis, M.R. Lyu, W. Schütz, "In search of effective diversity: a six-language study of fault-tolerant flight control software", *Proc. 18th Ann. Int'l Symp. Fault Tolerant Computing*, 1988 Jun; Tokyo, Japan.
[19] M. R. Lyu, "A design paradigm for multi-version software", *PhD Dissertation*, 1988 May; UCLA Computer Science Dept, Los Angeles, California.
[20] M. R. Lyu, A. Avižienis, "Assuring design diversity in n-version software: a design paradigm for n-version programming", *Proc. 2nd Int'l Working Conf. Dependable Computing for Critical Applications*, 1991 Feb, pp 89-98; Tucson, Arizona.
[21] M. R. Lyu, "Software requirements document for a fault-tolerant flight control computer", ECE55:195 Project Specification, 1991 Sep, p 64; Iowa City, Iowa.
[22] M. A. Vouk, McAllister, Caglayan, Walker Jr., Eckhardt, Kelly, Knight, "Analysis of faults detected in a large-scale multi-version software development experiments environment", *DASC'90 Proceedings*, pp 378-385.
[23] S. S. Brilliant, J. C. Knight, N. G. Leveson, "Analysis of faults in an n-version software experiment", *IEEE Trans. Software Engineering*, vol 16, 1990 Feb, pp 238-247.
[24] T. A. Budd, R. J. Lipton, F. G. Sayward, R. A. DeMillo, "The design of a prototype mutation system for program testing", *Proceedings NCC*, 1978, pp 623-627.
[25] W. E. Howden, "Weak mutation testing and completeness of test sets", *IEEE Trans. Software Engineering*, vol SE-8, 1982 Jul, pp 371-379.
[26] J. D. Musa, A. Iannino, K. Okumoto, *Software Reliability - Measurement, Prediction, Application*, 1987; McGraw-Hill.
[27] A. Avižienis, J.-C. Laprie, "Dependable computing: from concepts to design diversity", *Proc. IEEE*, vol 74, 1986 May, pp 629-638.
[28] W. G. Bouricius, W. C. Carter, P. R. Schneider, "Reliability modeling techniques for self-repairing computer systems", *Proc. 24th Nat'l Conf. ACM*, 1969, pp 295-383.
[29] M. R. Lyu, "Software reliability measurements in n-version software execution environment", *Proc. 1992 Int'l Symp. Software Reliability Engineering*, 1992 Oct, pp 254-263; Raleigh, North Carolina.

AUTHORS

Dr. Michael R. Lyu; Bell Communications Research; MRE-2D363; 445 South Street; Morristown, New Jersey 07960 USA.

Michael R. Lyu (S'84, M'88) is a Member of Technical Staff at the Software Process and Methods Research Group at the Bell Communications Research (Bellcore). He received his BS in Electrical Engineering in 1981 from the National Taiwan University, Taipei; his MS in Electrical & Computer Engineering in 1984 from the University of California, Santa Barbara; and his PhD in Computer Science in 1988 from the University of California, Los Angeles. Dr. Lyu was a Member of Technical Staff at Jet Propulsion Laboratory, California Institute of Technology, Pasadena from 1988 to 1990. He was an Assistant Professor at the Electrical & Computer Engineering at the University of Iowa from 1990 to 1992. His research interests include software reliability, software engineering, fault-tolerant computing, and distributed systems.

Yu-Tao He; Department of Electrical and Computer Engineering; University of Iowa; Iowa City, Iowa 52242 USA.

Yu-Tao He (S'92) received the BE in Electrical Engineering from Tsinghua University, Beijing in 1990. He is pursuing an MS in Electrical & Computer Engineering at the University of Iowa, Iowa City. He was the Testing Manager in Prisma Software Company, Cedar Falls in 1992 May to August. His research interests are software engineering, software testing, fault-tolerant computing, parallel and distributed systems. He is a student member of the IEEE and Association of Computing Machinery.

Manuscript TR92-301 received 1992 May 6; revised 1992 December 1.

IEEE Log Number 07619 ◄TR►