Assuring Design Diversity in *N*-Version Software: A Design Paradigm for *N*-Version Programming

Michael R. Lyu
ECE Department
The University of Iowa

Algirdas Avižienis
Computer Science Department
University of California, Los Angeles

ABSTRACT

The N-Version Programming (NVP) approach achieves fault-tolerant software units, called N-Version Software (NVS) units, through the development and use of software diversity. To maximize the effectiveness of the NVP approach, the probability of similar errors that coincide at the NVS decision points should be reduced to the lowest possible value. Design diversity is potentially an effective method to get this result. It has been the major concern of this paper to formulate a set of rigorous guidelines, or a *design paradigm*, for the investigation and implementation of design diversity in building NVS units for practical applications. This effort includes the description of a most recent formulation of the NVP design paradigm, which integrates the knowledge and experience obtained from fault-tolerant system design with software engineering techniques, and the application of this design paradigm to a real-world project for an extensive evaluation. Some limitations of the approach are also presented.

1. Introduction

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

NVS systems are gaining acceptance in critical application areas such as aerospace industry, nuclear power industry, and ground transportation industry. The construction of such systems is still, however, done mostly in an *ad hoc* manner. In order to obtain a paradigmatic approach in applying the NVS techniques for fault-tolerant software systems, a joint project [3] (to be called "the Six-Language Project" throughout the text) was initiated at the UCLA Dependable Computing & Fault-Tolerant Systems (DC & FTS) Laboratory and at the Honeywell Commercial Flight Systems Division. This paper will describe an NVP design paradigm which was applied to conduct the Six-Language Project, and discuss evidences and lessons learned from the project to testify and revise the proposed paradigm.

2. A Design Paradigm for N-Version Programming

NVP has been defined from the beginning as "the independent generation of $N \geq 2$ functionally equivalent programs from the same initial specification[1]." "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[1]."

The research effort at UCLA 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 was revealed in several previous research activities[4], [5], [6], which have investigated a total of 81 programs in four different applications. This evolving methodology was most recently formulated in[7] as an NVS *design paradigm* by integrating the knowledge and experience obtained from both software engineering techniques and fault tolerance investigations. The word "paradigm," used in the dictionary sense, 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 NVX.

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 N versions of an NVS unit, and (3) to elaborate efficient error detection and recovery mechanisms. 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. Detailed descriptions of the incorporated activities and guidelines are presented in the following sections.

2.1 Determine Method of NVS Supervision and Execution Environment in System Requirement Phase

The NVS execution environment has to be determined early in the system requirement phase to evaluate the overall system impact and to obtain required support facilities.

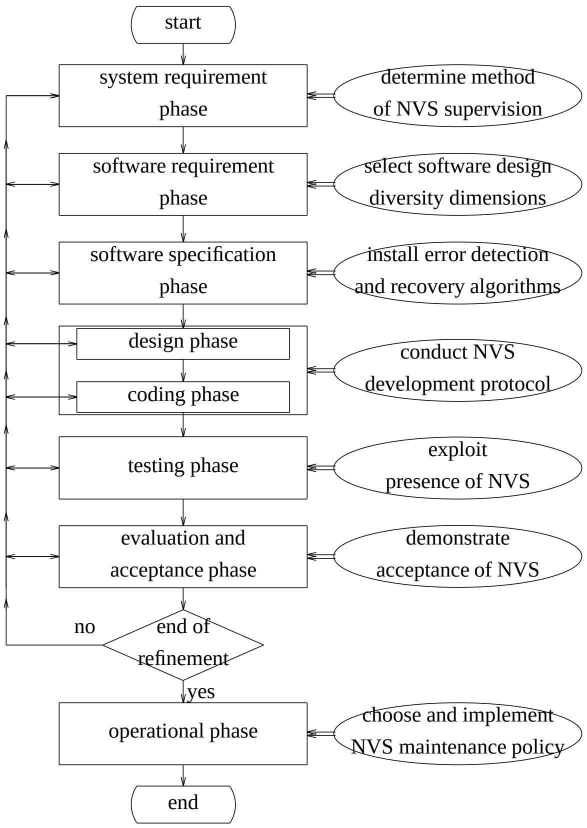

Figure 1: A Design Paradigm for N-Version Programming

(1) Decide NVS execution methods and required resources

The overall system architecture might be well defined during system requirement phase, at which time the software configuration items could be properly identified. This means the number of software versions and their interaction could be investigated and determined. Due to the cost of software development for multiple versions, the number of versions is not expected to be large at present. However, from dependability (including reliability and safety) viewpoints, at least two versions are required for single-failure-detection operations, and at least three versions are required for single-failure-correction/double-failure-detection operations. The current limitation is the lack of quantitative methods for an accurate decision.

(2) Develop support mechanisms and tools

Generally speaking, a generic class of NVX forming the NVS execution support environment is favorable. The NVX may be implemented in software, in hardware, or in a combination of both. The basic functions that the NVX must provide for NVS execution are: (a) the decision algorithm, or set of algorithms; (b) assurance of input consistency for all versions; (c) interversion communication; (d) version synchronization and enforcement of timing constraints; (e) local supervision for each version; (f) the global executive and decision function for version error recovery; and (g) a user interface for observation, debugging, injection of stimuli, and data collection during N-version execution of application programs. The nature of these functions was extensively illustrated in the descriptions of the DEDIX testbed system[2].

(3) Comply with hardware architecture

Special dedicated hardware processors might have to be implemented or procured in advance for the execution of NVS systems, especially when the NVS supporting environments need to operate under certain stringent requirements (e.g., accurate supervision, efficient CPUs, etc.). The options of combining with hardware fault-tolerance for a hybrid configuration could also be considered[8], [9], [10].

In order to create enough sampling space under the budget constraint, it was decided that six versions would be generated in the Six-Language Project. Representatives from Honeywell extracted the information needed for the Six-Language Project from their original system specification and provided it in a System Description Document (SDD). Moreover, for the purpose of industrial-standard validation and verification, a Model Definition Document (MDD) was also supplied. This document described a mathematical Aircraft Model and a Square Wave Model. The former provided functions within the landing control loop but external to the application program to form closed-loop flight simulations, while the latter applied open-loop testing strategy with various stringent conditions to saturate the executions of the control laws in the application program.

2.2 Investigate Software Design Diversity Dimensions in Software Requirement Phase

The major reason for choosing design diversity is to eliminate the commonalities between the separate programming efforts, as they have the potential to cause related faults among the multiple versions.

(1) Compare random diversity vs. enforced diversity

Different dimensions of diversity can be applied to the building of NVS systems. Design diversity could be achieved either by randomness or by enforcement. The *random diversity*, such as that provided by independent personnel, leaves the dissimilarity to be generated according to individual's training background and thinking process. The diversity is achieved somewhat in an uncontrolled manner by this way. The *enforced diversity*, on the other hand, investigates different aspects in several dimensions, and deliberately requires them to be implemented into different program versions. The purpose of such *required diversity* is to minimize the opportunities for common causes of software faults in two or more versions (e.g., compiler bugs, ambiguous algorithm statements, etc.), and to increase the probabilities of significantly diverse approaches to version implementation.

(2) Derive qualitative design diversity metrics

There are four phases in which design diversity could be applied: the specification phase, the design phase, the coding phase, and the testing phase. Applicable dimensions of diversity include different implementors, different languages [11], different tools[12], different algorithms[13], and different software development methodologies (including phase-by-phase software engineering, prototyping, computer-aided software engineering, or even the "clean room" approach[14]).

A qualitative design diversity metric is proposed in Table 1. This assessment of diversity is an initial effort based on the experiences gained from previous experiments at UCLA[4], [5], [15], and published work from other sites[12], [13], [16], [17].

	implementors	languages	tools	algorithms	methodologies
spec.	higher	higher	lower–	higher+	lower
design	higher+	lower	lower	higher+	higher
coding	higher+	higher+	lower	higher	higher
testing	lower	lower–	higher	lower	higher+

Table 1: A Qualitative Design Diversity Metric

This table suggests that in the specification phase, using different implementors, languages or algorithms might achieve higher diversity than applying other dimensions. In the design phase, using different implementors, different algorithms, or different methodologies tends to be more helpful. All dimensions except tools are considered effective in the coding phase. Finally, investigation of different tools or methodologies might be more favorable in the testing phase. Moreover, to compare rows and columns in

Table 1, extra granularity is provided by using "+" (indicating "further" for "higher") and "–" (indicating "further" for "lower") signs. For example, diversity by using different implementors in testing (marked "lower") is considered lower than using them in the previous three phases, but that could still be higher than using different languages in testing phase, which is marked "lower–".

(3) Evaluate cost-effectiveness along each dimension

Since adding more diversity implies adding more resources, it is important to evaluate cost-effectiveness of the added diversity along each dimension. This evaluation will enable trade-off studies between cost and efficiency. Such evaluation might be application-dependent, and thus may need to be elaborated, possibly by several iterations of investigations. It is hypothesized that the main cost of NVP is dominated by the employment of extra implementors. Cost of adding other diversity dimensions should not be significant, especially when the resources in these dimensions are abundant (e.g., languages, tools). No concrete data is available to support or dispute this hypothesis, though.

(4) Obtain the final choice under particular constraints

After the above investigation, the final combination of diversity could be determined under particular project constraints. Typical project constraints include: cost, schedule, and required dependability performance. At the current stage, however, this decision might have to involve substantial subjective judgements, due to the lack of quantitative measures for design diversity and the resulting cost impacts. As more experiences and evidences are gained by researchers in this field, schemes for achieving an optimal solution prior to a project start might be available in the future.

In the Six-Language Project, it was decided that different algorithms were inappropriate for investigation due to tight accuracy requirements for numerical computations. Based on the availability of computer resources and language-knowledgeable programmers, applying different programming languages was considered as a cost-effective investigation in enforcing design diversity. Six programming languages of various programming style were chosen, consisting two widely used conventional procedural languages (C and Pascal), two object-oriented programming languages (Ada and Modula-2), one logic programming language (Prolog), and one functional programming language (T, a variant of Lisp). It was postulated that different programming languages would force people to think differently about the application problem and the program design, and to use different tools in their programming and testing activities, which could lead to significant diversity in the software development efforts.

2.3 Install Error Detection and Recovery Algorithms in Software Specification Phase

The specification of the member versions, to be called "V-spec," needs to state the functional requirements completely and unambiguously, while leaving the widest possible choice of implementations to the *N* programming efforts. Sufficient error detection and recovery algorithms have to be carefully designed and specified in order to detect related errors that could potentially lead to coincident failures.

(1) Prescribe the matching features needed by NVX

Each V-spec must prescribe the *matching features* that are needed by the NVX to execute the member versions as an NVS unit in a fault-tolerant manner[4]. The V-spec defines: (a) the *functions* to be implemented, the time constraints, the inputs, and the initial state of a member version; (b) requirements for internal *error detection* and *exception handling* (if any) within the version; (c) the *diversity* requirements; (d) the *cross-check points* ("cc-points") at which the NVX decision algorithm will be applied to specified outputs of all versions; (e) the *recovery points* ("r-points") at which the NVX can execute *community error recovery*[18] for a failed version; (f) the choice of the NVX *decision algorithm* and its *parameters* to be used at each cc-point and r-point; (g) the *response* to each possible outcome of an NVX decision, including absence of consensus; and (h) the prevention to the *Consistent Comparison Problem*[19].

(2) Avoid diversity-limiting factors

The specifications for simplex software tend to contain guidance not only "what" needs to be done, but also "how" the solution ought to be approached. Such specific suggestions of "how" reduce the chances for diversity among the versions and should be eliminated from the V-spec. Another potential diversity-limiting factor is the over-specification of cc-points and r-points. The installation of cc-points and r-points enhances error detection and recovery capability, but it imposes extra common constraints to the programs and might tend to limit design diversity. The choice of number of these points and their placements depend on the size of the software, the control flow of the application, the number of variables to be checked and recovered, and the time overhead allowed to perform these operations.

(3) Require the enforced diversity

The V-spec may explicitly require the versions to differ in the "how" of implementation. Diversity may be specified in the following elements of the NVP process: (a) training, experience, and location of programmers; (b) application algorithms and data structures; (c) software development methods; (d) programming languages; (e) programming tools and environments; (f) testing methods and tools.

(4) Protect the specification

The use of two or more distinct V-specs, derived from the same set of user requirements, can put extensive protection against specification errors. Two cases have been practically explored: a set of three V-specs (formal algebraic OBJ, semi-formal PDL, and English) that were derived together[5], [15], and a set of two V-specs that were derived by two independent efforts[20]. These approaches provide additional means for the verification of the V-specs, and offer diverse starting points for version implementors.

In the Six-Language Project, only absolutely necessary information was supplied to the programmers in the software specification. The diagrams describing the major system functions were taken directly from the original SDD, while the explanatory text was shortened and made more concise. A further enhancement to the specification was the introduction of *seven* cross-check points, placed right after each main computation unit, and *one* recovery point at the end of the last computation unit. In total, the instrumented application required 14 external variables (for input functions), 68 intermediate and final variables (for cross-check functions), and 42 state variables (for recovery function). The resulting specification given to the programmers was a 64-page English document.

Another characteristics of the Six-Language Project is that the "(in)consistent comparison problem"[19] *did not* exist in the application. This was due to two main reasons: (1) We did not vote final results on Boolean or integer values; the final, critical results that required consensus were always real numbers upon which cross-checking could be properly applied for error detection and recovery; (2) Multiple correct values were allowed for intermediate results during computation without limiting the potential for diversity in various implementation; however, as imposed by the specified algorithm, they would converge to the final results with tight numerical precision for an accurate comparison. Although there might be applications appearing unsuitable for NVS investigations, this automatic landing problem, a typical example in aeronautic industry, turned out to be appropriate.

2.4 Conduct NVS Development Protocol in Design and Coding Phase

In this phase, multiple programming teams (P-teams) start to develop the NVS concurrently according to a given software specification. The main concern herein is to maximize the isolation and independence of each version, and to smooth the overall software development efforts.

(1) Derive a set of mandatory rules of isolation

The purpose of imposing rules on the P-teams is to assure the *independent generation* of programs, which means that programming efforts are carried out by individuals or groups that do not interact with respect to the programming process. The rules of isolation are intended to identify and eliminate potential "fault leak" links between the P-teams. The development of the rules is an on-going process, and the rules are enhanced when a previously unknown "fault leak" is discovered and its cause pinpointed. Current isolation rules include: strict prohibition of any discussion of technical work between P-teams, widely separated working areas (offices, computer terminals, etc.) for each P-team, usage of different host machines for the software development, protection of all on-line computer files, and safe deposit of technical documents.

(2) Define a rigorous communication and documentation (C&D) protocol

The C&D protocol imposes rigorous control on the manner in which all necessary information flow and documentation efforts are conducted. The main goal of the C&D protocol is to avoid opportunities for one P-team to influence another P-team in an uncontrollable, and unnoticed manner. In addition, the C&D protocol documents communications in sufficient detail to allow a search for "fault leaks" if potentially related faults are discovered in two or more versions at some later time.

(3) Form a coordinating team (C-team)

The C-team is the keystone of the C&D protocol. The major functions of this team are: (a) to prepare the final texts of the V-specs and of the test data sets; (b) to set up the implementation of the C&D protocol; (c) to acquaint all P-teams with the NVP process, especially rules of isolation and the C&D protocol; (d) to distribute the V-specs, test data sets, and all other information needed by the P-teams; (e) to collect all P-team inquiries regarding the V-specs, the test data, and all matters of procedure; (f) to evaluate the inquiries (with help from expert consultants) and to respond promptly either to the inquiring P-team only, or to all P-teams via a broadcast; (g) to conduct formal reviews, to provide feedback when needed, and to maintain synchronization between P-teams; (h) to gather and evaluate all required documentation, and to conduct acceptance tests for every version.

In the Six-Language Project, all communications between the C-team and the P-teams were allowed only in standard written format for possible post mortems about "fault leaks." Electronic mail between the C-team and the P-teams has proven to be the most effective medium for this purpose. Moreover, to reduce the unnecessary information exchange, answers to a particular question from a P-team were only sent to the corresponding P-team. A message was broadcast to all P-teams only when strictly necessary (e.g., a specification update). This protocol has avoided a possible bothersome overload of the message flood, as was observed in[6]. Ratio of P-team members to C-team members was 4:1 in the Six-Language Project, in which the C-team has performed a satisfactory job.

2.5 Exploit Presence of NVS in Testing Phase

An appealing and promising application of NVS is its reenforcement for current software verification and validation procedure during the testing phase, which is one of the hardest cores of any software system.

(1) Explore comprehensive verification procedures

For software verification, the NVS provides a thorough coverage for error detection since every discrepancy among versions needs to be resolved. Moreover, it is observed that consensus decision of the existing NVS may be more reliable than that of a "gold" model or version (usually provided by an application expert)[21].

(2) Enforce extensive validation efforts

NVP provides protective redundancy around requirement misinterpretations and specification ambiguities. Any single-version approach gets a single interpretation of the requirement, no matter how carefully a development procedure has been followed. Especially when the software development group is small, everyone can share a misunderstanding. The NVP approach forces the system requirements and the software specifications to be assessed from independent observations and viewpoints, making the validation effort more effective and more extensive.

(3) Provide opportunities for "back-to-back" testing

There is a possibility that two or three versions can be executed "back-to-back" in a testing environment, completing verification and validation concurrently with productive execution. However, there is a risk here. If codes are brought together prematurely, the independent programming efforts would be violated, and "fault leaks" might be created among the program versions. In any case, should this scheme be applied in a project, it must be done by a testing team independent of the P-teams (e.g., the C-team), and the testing results should never be revealed to a P-team, if they contain information from other versions that would influence this P-team.

Some experiences in NVS testing were obtained from the Six-Language Project: (a) a golden reference model, derived by a Honeywell expert, was less reliable than the consensus of multiple program versions in defining correctness of a computation[3]; (b) multiple teams around testing explored erroneous test cases effectively; (c) the pace in testing phase was accurately tracked and controlled by monitoring and comparing the progress of the multiple teams.

2.6 Demonstrate Acceptance of NVS in Evaluation Phase

Evaluation of the software fault-tolerance attributes of an NVS system is performed by means of analytic modeling, simulation, experiments, or combinations of those techniques. Many open issues are yet to be investigated.

(1) Define NVS acceptance criteria

The acceptance criteria of the NVS system depend on the validity of the conjecture that residual software faults in separate versions will cause very few, if any, similar errors at the same cc-points. These criteria might depend on various applications and their developing procedures, thus need to be carefully elaborated case by case.

(2) Provide evidence of diversity

Diversity requirements support the objective of reducing common programming errors, since they provide more natural isolation against "fault leaks" between the teams of programmers. Furthermore, it is conjectured that the probability of a random, independent occurrence of faults that produce the same erroneous results in two or more versions is less when the versions are more diverse.

(3) Demonstrate effectiveness of diversity

Another conjecture is that even if related faults are introduced, the diversity of member versions may cause the erroneous results not to be similar at the NVX decision. Therefore, evidence and effectiveness of diversity should be carefully identified and assessed[22].

(4) Make NVS dependability prediction

For dependability prediction of NVS, there are two essential aspects: the choice of suitable software dependability models, and the definition of quantitative measures[23], [24], [25], [26]. These aspects should also characterize the level of fault-tolerance present in the NVS system[27], [28]. Usually, the dependability prediction of the NVS system is compared to that of the single-version baseline system.

In the Six-Language Project, nine flight simulations engaging various flight modes were imposed on the six program versions before they were finally accepted. This represented a total of 18440 program executions. Parallel to this testing, a structural analysis for the multiple programs was conducted in the evaluation phase. The efforts of finding more faults and the search for evidence of structural diversity among these programs were the major concerns. An additional benefit of this analysis was that it necessitated a thorough C-team inspection by *code comparisons*, in which seven additional faults that were not caught by any tests or any P-team inspections were detected[3].

2.7 Choose and Implement an Appropriate NVS Maintenance Policy in Operational Phase

The key point to remember regarding NVS maintenance policy in this phase is to follow a philosophy consistent to the overall design paradigm.

(1) Assure and monitor NVX basic functionality

The basic functionality of NVX should be properly assured and monitored during the operational phase. Critical parts of the NVS supervisory system could themselves be protected by the NVP technique. Operational status of the NVX running NVS should be carefully monitored to assure basic functionality. Any anomalies should be recorded for further investigation in order to improve this NVP paradigm, which is an on-going effort aiming at achieving ultra-reliability (e.g., 10^{-9} failures per hour) for safety-critical software systems. Such stringent requirements could not be achieved without a progressive evolution of the underlining design process[29].

(2) Keep the achieved diversity work in the maintenance phase

It is postulated that patching the software, as has been widely used in industry, might more easily reveal the existence of faults by exhibiting dissimilarities among the independently generated software versions. This would be a valuable feature of NVS units since such a patching technique could create the potential of high risks in an originally well (and possibly elegantly) designed single version software. An observation from the Six-Language Project was that it appeared extremely difficult to inject similar faults which were hard-to-detect in the six programs[30]. This was due to the achieved diversity among the programs.

(3) Follow the same paradigm for modification and maintenance

As for the modification and maintenance of the NVS unit, the same design paradigm should be followed, i.e., a common "specification" of the maintenance action should be "implemented" by independent maintenance teams. The potential cost for such a policy is by no means cheap, but it is hypothesized that the induced extra cost in maintenance phase, comparing with that for single software, is a factor relatively lower than the extra cost factor in development phase. This is due to two reasons: (a) the achieved NVS reliability is supposedly higher than that of a single version, leaving fewer costly operational failures to be experienced; (b) when adding new features to the operating software, the existence of multiple program versions should make the testing and certification tasks easier and more cost-effective. These tasks usually share a larger portion in maintenance phase than in development phase.

The configuration of the operational flight simulation system in the Six-Language Project consisted of single or multiple lanes of the control law computation, obtained from the six accepted program versions, and the pre-programmed Airplane Model. The Airplane Model computed the response of an airplane to each elevator command, with a landing geometry model describing the deviation relative to a glide slope beam. Outputs of this model was fed back to each lane for a subsequent round of execution. In order to provide a set of inputs to the Airplane Model that create larger variation magnitudes, and thereby force off-nominal software operating conditions, random turbulence in the form of vertical wind gusts was introduced. Moreover, these testing facilities could replaced by the Square Wave Model to form an open-loop testing configuration without feedback, for the purpose of boundary value analyses.

During the operational testing phase, 1000 flight simulations, or over five million program executions, were conducted. Table 2 shows the errors encountered in each single version, while Table 3 shows different error categories under all combinations of 3-version and 5-version configurations. Note that the discrepancies encountered in the operational testing were called "errors" rather than "failures" due to their non-criticality in the landing procedure, i.e., a proper touchdown was still achieved at their presence.

version	size (l.o.c.)	total executions	number of errors	error probability
ada	2256	5127400	0	.0000000
c	1531	5127400	568	.0001108
modula-2	1562	5127400	0	.0000000
pascal	2331	5127400	0	.0000000
prolog	2228	5127400	680	.0001326
t	1568	5127400	680	.0001326
average	1913	5127400	321	.00006267

Table 2: Errors in Individual Versions

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

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

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

From Table 2 we can see that the average error probability for single version is .00006267. Table 3 shows that for all the 3-version combinations, the error probability concerning reliability is .00002857 (categories 3 and 4), and that for safety is .00002652 (category 4). This is a reduction of roughly 2.3. In all the combinations of 5-version configuration, the error probability for reliability is .000003413 (category 5; Two of the three errors are coincident, resulting in no-decision), a reduction by a factor of 18. This probability becomes zero in the safety measurement.

It is cautioned against interpreting these numbers as the expected dependability improvement of NVS over single-version software. The coincident errors produced by the Prolog and T programs were all caused by *one identical fault* in both versions, which was due to the ignorance of a slight specification update that was made very late in the programming process. This fault manifested itself right after these program versions were put together for the flight simulation. To eliminate causes for this type of faults in the future, the corresponding amendment to the NVP design paradigm is to deliberately request confirmation and validation for late specification changes in the C&D protocol, and to cautiously conduct multiple-version verification testing as part of the acceptance criteria. Had this fault been eliminated in the operational testing, categories 3, 4 and 5 for both 3-version and 5-version configurations in Table 3 would have been all zero, resulting in perfect dependability figures.

2.8 Refine by Iterations

Notice that some of the described stages occur at progressively later times, but backtracking from a given stage to its previous one may occur at any time. Alteration of requirements arising from use, revision of specification, change in environment, and erroneous implementation may interrupt the flow of the normal design paradigm or spawn sub-processes having their own life cycles. This flexibility might allow the proposed paradigm to be tailored for other software engineering development models (e.g., the spiral model[31]).

3. Conclusions

Although at first considered as an impractical competitor of high-quality single-version programs, N-version software has gained some significant acceptance in academia and industry in the past few years. Since more and more critical systems are software-intensive or software-embedded, the trustworthiness of software is the principal prerequisite for the building of a trustworthy system. At present, N-version software might be an attractive alternative that can be expected to provide a higher level of trustworthiness and security for critical software units than test or proof techniques without fault tolerance. The ability to guarantee that any software fault, as long as it only affects minority members of an N-version unit, will be tolerated without service disruption may by itself be a convincing reason to adapt N-version software as a safety assurance technique for life-critical applications. The main focus of the proposed NVP design paradigm attempts to promote this ability.

In summary, this research has made the following contributions:

1. An NVP design paradigm has been formulated, applied, and evaluated. The proposed design paradigm, which integrates software engineering techniques and NVP design diversity guidelines and rules, could provide a fundamental model for the practical development of NVS.
2. The design paradigm has been used during the entire life cycle of the UCLA/Honeywell Six-Language Project. This project served as an experimental means to executing and evaluating the proposed paradigm. In reviewing the objectives of the design paradigm, we believe that all of them were properly addressed, if not completely accomplished, by the experiment. All perceivable causes of related design faults were eliminated, and causes of the only two pairs of identical faults were identifiable and readily removable. The resulting amendment to the paradigm, as the lessons learned from this project, is to add extra guidelines in Section 2.3 for the production of graphical specification, in Section 2.4(2) for the confirmation of every specification update in the C&D protocol, and in Section 2.6(1) for the inclusion of multiple version verification testing as part of the NVS acceptance criteria.
3. The design paradigm tries to explore and support the idea of design diversity, and intends to prevent commonalities that could produce related software faults. The effectiveness of this design paradigm was shown by the experimental result that identical faults in two versions have occurred only twice in the Six-Language Project, comparing with a total of 93 faults found in the six software versions during the whole project life cycle. Identical faults involving more than two versions have never been observed. Moreover, in a mutation testing study which investigated all the 93 known faults[7], errors caused by every non-identical fault among program versions were *all* distinguishable and properly detected by the provided fault-tolerant mechanisms.

As the final concluding remark, it is obvious that coincident failures are indeed the Achilles' heel of NVS, and the main goal of the proposed design paradigm is to avoid them by two levels of treatment: (a) investigate design diversity to prevent possible *identical* faults, and (b) install error detection and recovery algorithms to handle potential *similar* errors. The advancement of the NVP technique could happen only when these two aspects are properly addressed and documented, and we hope that the proposed paradigm can serve as an evolving basis subject to public revision and amendment in order to achieve such an advancement.

Acknowledgement

The authors wish to thank the program committee member, Dr. J. Lala, for his valuable suggestions and observations for the revision of this paper. Comments from the program committee and the reviewers are also highly appreciated.

References

1. A. Avižienis and L. Chen, ''On the Implementation of N-Version Programming for Software Fault-Tolerance during Program Execution,'' in *Proceedings COMPSAC 77*, pp. 149-155, 1977.
2. A. Avižienis, ''The N-Version Approach to Fault-Tolerant Software,'' *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, pp. 1491-1501, December 1985.
3. 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.
4. L. Chen and A. Avižienis, ''N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation,'' *Digest of 8th Annual International Symposium on Fault-Tolerant Computing*, pp. 3-9, Toulouse, France, June 1978.
5. A. Avižienis and J.P.J. Kelly, ''Fault-Tolerance by Design Diversity: Concepts and Experiments,'' *Computer*, vol. 17, no. 8, pp. 67-80, August 1984.
6. J.P.J. Kelly, A. Avižienis, B.T. Ulery, B.J. Swain, M.R. Lyu, A.T. Tai, and K.S. Tso, ''Multi-Version Software Development,'' *Proceedings IFAC Workshop SAFECOMP'86*, pp. 43-49, Sarlat, France, October 1986.
7. M.R. Lyu, *A Design Paradigm for Multi-Version Software*, Ph. D. Dissertation, UCLA Computer Sci-

ence Department, Los Angeles, California, May 1988.

8. K.H. Kim, ‘‘Distributed Execution of Recovery Blocks: An Approach to Uniform Treatment of Hardware and Software Faults,’’ in *Proceedings IEEE 4th International Conference on Distributed Computing Systems*, pp. 526-532, San Francisco, California, May 1984.

9. J.-C. Laprie, ‘‘Hardware-and-Software Dependability Evaluation,’’ in *Proceedings 11th World IFIP Congress*, pp. 109-114, San Francisco, California, September 1989.

10. J. Lala, L. Alger, S. Friend, G. Greeley, S. Sacco, and S. Adams, ‘‘Study of A Unified Hardware and Software Fault Tolerant Architecture,’’ Report No. 181759, NASA Contract No. NAS1-18061, January 1989.

11. U. Voges, ‘‘Use of Diversity in Experimental Reactor Safety Systems,’’ in *Software Diversity in Computerized Control Systems*, ed. U. Voges, pp. 29-49, Springer-Verlag/Wien, Austria, 1988.

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. 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.

14. M. Dyer, ‘‘Certifying the Reliability of Software,’’ in *Proceedings Annual National Joint Conference on Software Quality and Reliability*, Arlington, Virginia, March 1-3 1988.

15. 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.

16. T. Anderson, P.A. Barrett, D.N. Halliwell, and M.R. Moulding, ‘‘Software Fault Tolerance: An Evaluation,’’ *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, pp. 1502-1510, December 1985.

17. P. Traverse, ‘‘AIRBUS and ATR System Architecture and Specification,’’ in *Software Diversity in Computerized Control Systems*, ed. U. Voges, pp. 95-104, Springer-Verlag/Wien, Austria, 1988.

18. K.S. Tso and A. Avižienis, ‘‘Community Error Recovery in N-Version Software: A Design Study with Experimentation,’’ *Digest of 17th Annual International Symposium on Fault-Tolerant Computing*, pp. 127-133, Pittsburgh, Pennsylvania, July 1987.

19. S. S. Brilliant, J. C. Knight, and N. G. Leveson, ‘‘The Consistent Comparison Problem in N-Version Software,’’ *IEEE Transactions on Software Engineering*, vol. 15, no. 11, pp. 1481-1485, November 1989.

20. 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.

21. J.P.J. Kelly, D.E. Eckhardt, A. Caglavan, J.C. Knight, D.F. McAllister, and M.A. Vouk, ‘‘A Large Scale Second Generation Experiment in Multi-Version Software: Description and Early Results,’’ *Proceedings The Eighteenth International Symposium on Fault-Tolerant Computing*, Tokyo, Japan, June 27-30 1988.

22. J. J. Chen, ‘‘Software Diversity and Its Implications in the N-Version Software Life Cycle,’’ , Ph.D. Dissertation, UCLA Computer Science Department, Los Angeles, California, 1990.

23. J.-C. Laprie, ‘‘Dependability Evaluation of Software Systems in Operation,’’ *IEEE Transactions on Software Engineering*, vol. SE-10, no. 6, pp. 701-714, November 1984.

24. 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.

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

26. P. Thevenod-Fosse, ‘‘Software Validation by Means of Statistical Testing: Retrospect and Future Direction,’’ in *Proceedings the First International DCCA Working Conference* , pp. 15-22, Santa Barbara, California, August 1989.

27. D.E. Eckhardt and L.D. Lee, ‘‘A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors,’’ *IEEE Transaction on Software Engineering*, vol. SE-11, no. 12, pp. 1511-1517, December 1985.

28. 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.

29. J. C. Laprie and B. Littlewood, ‘‘Quantitative Assessment of Safety-Critical Software: Why and How?,’’ in *Proceedings Probabilistic Safety Assessment and Management Conference* , Beverly Hills, California, February 1991.

30. M.K. Joseph, *Architectural Issues in Fault-Tolerant, Secure Computing Systems*, Ph. D. Dissertation, UCLA Computer Science Department, Los Angeles, California, May 1988.

31. B. W. Boehm, ‘‘A Spiral Model of Software Development and Enhancement,’’ *IEEE Computer*, pp. 61-72, May 1988.