Defence Research and Development Canada
Recherche et développement pour la défense Canada

Best Design Practices for Effective Use of N-Version Programming

Raphaël Khoury
Mario Couture
DRDC Valcartier

Abdelwahab Hamou-Lhadj
Concordia University

Defence R&D Canada – Valcartier

Technical Memorandum
DRDC Valcartier TM 2013-017
February 2013

Canada

Best Design Practices for Effective Use of N-Version Programming

Raphaël Khoury
Mario Couture
DRDC Valcartier

Abdelwahab Hamou-Lhadj
Concordia University

Defence R&D Canada – Valcartier

Technical Memorandum
DRDC Valcartier TM 2013-017
February 2013

Abstract

N-version programming is a software development paradigm that draws upon the concept of diversity to increase the reliability of software. The central idea is to independently produce multiple functionally equivalent versions of a program, and execute them in parallel. If the versions fail independently, then the probability of multiple versions producing a faulty output on any given input is very small; much lower than the failure probability of any single version. In this technical memorandum, we examine and contrast various experiments that have been performed to evaluate the benefits of this approach and draw some conclusions with respect to the most effective way that N-version programming can be utilized. We find that for diversity to be effective, it must be introduced in a targeted and informed manner and encompass several phases of the software's development.

Résumé

La programmation en N-versions (N-version programming) est un paradigme de programmation qui s'appuie sur le concept de diversité pour assurer la fiabilité des logiciels. L'idée clef est de développer de multiples versions fonctionnellement équivalentes et de les exécuter en parallèle. Si l'occurrence de failles dans chacune de ces versions exhibe une distribution indépendante de celle des autres versions, alors la probabilité d'un échec lors de l'exécution simultanée de toutes les versions est très faible. Dans ce mémorandum technique, nous examinons les résultats de diverses études académiques qui ont évalué les bénéfices de ce paradigme de cette approche, et synthétisons des conclusions quant à son utilisation efficace. En particulier, nous concluons que pour que l'usage de la diversité soit efficace, celle-ci doit être introduite d'une manière ciblée et informée, et ce, dans plusieurs phases du développement du logiciel.

This page intentionally left blank.

Executive summary

Best Design Practices for Effective Use of N-Version Programming

Raphaël Khoury; Mario Couture; Abdelwahab Hamou-Lhadj; DRDC Valcartier TM 2013-017; Defence R&D Canada – Valcartier; February 2013.

Introduction: N-version programming is a software development paradigm that draws upon the concept of diversity to increase the reliability of software. The central idea is to independently produce multiple functionally equivalent versions of a program, and execute them in parallel. If the versions fail independently, then the probability of multiple versions producing a faulty output on any given input is very small; much lower than the failure probability of any single version. In this technical report, we examine and contrast various experiments that have been performed to evaluate the benefits of this approach and draw some conclusions with respect to the most effective way that N-version programming can be utilized.

Results: A central concept in the evaluation of N-version architecture is that of independence of failure. Two instances fail independently if the occurrence of failure in one instance provides no information about the likelihood of failure of the other instance on the same input. It is from the presence of independence of failure that gains in reliability can be accrued from the use of N-version programming. However, we find that the independence of failure is not certain to be achieved simply by developing multiple instances independently. Indeed, for diversity to be effective, it must be introduced in a targeted and informed manner and encompass several phases of the software's development. We also find that the care taken to isolate the development team of the various versions is unwarranted.

Significance: The use of diversity can be an effective strategy to increase the reliability of software, but only if it is introduced in a targeted manner, aimed at maximizing the differences between the instances.

Future plans: Introducing diversity in the context of security, rather than reliability, is an important topic for future research. We also believe that a metric that would allow us to quantifying objectively how much diversity exists between two instances would allow a more precise evaluation of diverse architectures. Another useful avenue for future research is to develop a reasoning framework that will allow us to evaluate objectively how much reliability is gained by the introduction of diversity in an architecture in different manners.

Sommaire

Best Design Practices for Effective Use of N-Version Programming

Raphaël Khoury; Mario Couture; Abdelwahab Hamou-Lhadj; DRDC Valcartier TM 2013-017; R & D pour la défense Canada – Valcartier; février 2013.

Introduction: La programmation en N-versions (N-version programming) est un paradigme de programmation qui s'appuie sur le concept de diversité pour assurer la fiabilité des logiciels. L'idée clef est de développer de multiples versions fonctionnellement équivalentes et de les exécuter en parallèle. L'idée clef est de développer de multiples versions fonctionnellement équivalentes et de les exécuter en parallèle. Si l'occurrence de failles dans chacune de ces versions exhibe une distribution indépendante de celle des autres versions, alors la probabilité d'un échec lors de l'exécution simultanée de toutes les versions est très faible, bien moindre que la probabilité d'échec de chacune des versions qui la compose. Dans ce mémorandum technique, nous examinons et contrastons plusieurs études antérieures qui visaient à évaluer les bénéfices de cette approche, et tirons des conclusions quant à la manière la plus efficace d'utiliser la programmation en N-versions.

Résultats: Le concept d'indépendance des échecs a été identifié dans plusieurs études comme élément essentiel des architectures à N-versions. On dit de deux instances d'un logiciel qu'elles exhibent une indépendance d'échec si l'occurrence d'un échec dans l'une des instances sur une entrée donnée n'apporte aucune information quant à la probabilité d'échec de l'autre instance sur la même entrée. La présence d'indépendance d'échec dans une architecture N-versions assurerait une amélioration significative de la fiabilité. Néanmoins, nous trouvons qu'il n'est pas certain d'obtenir l'indépendance des échecs simplement par l'entremise du développement indépendant de multiples versions. En effet, pour que l'usage de la diversité soit efficace, celle-ci doit être introduite d'une manière ciblée et informée, et ce, dans plusieurs phases du développement du logiciel. Nous concluons aussi que le soin habituellement pris pour isoler les différentes équipes de développeurs de chacune des versions n'est pas nécessaire.

Importance: La diversité peut être une stratégie efficace pour assurer la fiabilité des logiciels, mais seulement si elle est introduite d'une manière ciblée, de telle sorte à maximiser les différences entre les instances.

Perspectives: Introduire la diversité dans l'optique d'assurer la sécurité des systèmes, plutôt que leur fiabilité, est un objectif important de recherche future. Nous croyons aussi qu'une métrique permettant de quantifier objectivement le niveau de diversité existe entre deux instances permettrait une évaluation plus précise des architectures en N-Versions. Une autre avenue de recherche intéressante serait de développer un cadre de raisonnement formel, permettant d'évaluer objectivement les gains en fiabilité obtenus quand la diversité est introduite de différente manière.

Table of contents

List of figures

List of tables

This page intentionally left blank.

1 Introduction

1.1 Motivation

Redundancy has long been used in engineering and hardware to increase reliability and fault tolerance when operating in an uncertain environment. The key insight is that even if one instance fails, an alternative redundant one is available to replace it.

This approach can also be adapted to the context of software development. However, since every identical instance of software will, in principle, behave in the exact same manner when exposed to the same situation, *diversity*, rather than simply redundancy, must be employed to avoid having the defect that caused the failure to propagate to other instances.

This idea of a diverse environment was first described by Avizienis in [1], and takes the form of *N-version programming*. The guiding principle of this approach is to produce several distinct versions of a given software, and execute them in parallel with the same inputs. In case of a discrepancy between the outputs of the various instances, an output is chosen by majority voting. The intuition behind this is that while it may be impossible to produce a single flawless instance of any complex system, multiple instances of this system would normally exhibit different faults.

A recurrent goal in N-version programming is that failure between versions should be independent. Independence of failure can be formally defined in several ways (see for e.g. [2]), and captures the intuition that the faults occurring in each version are unrelated. In the presence of statistical independence of failure, the probability of two instances failing simultaneously (i.e. on the same input) is substantially smaller than that of the original programs, and the reliability of overall architecture can always be improved by the incorporation of additional diverse components [3][1]. Interestingly, it is, in principle, possible for an N-version architecture to have better-than-independence failure behavior if the incidence of failure between its components is negatively correlated [4].

While the study of software diversity for reliability dates back to Avizienis in 1985 [5], a new generation of researchers has recently revisited the idea of software diversity, but in a context of security rather than reliability and much work has already been done on this topic. For example, Gao et al. [6][7] propose architecture for intrusion detection, analogous to that of N-version programming, in which multiple versions of a system are run in parallel. An intrusion can then be detected by the abnormal divergence in the behavior of the multiple instances. In the same vein, other researchers have also argued that there are substantial benefits to the use of diversity in anti-virus software [8]. Schneider [9], and Littlewood et al. [10] discuss some of the issues involved in using diversity for security purposes. This line of research also intersects the emerging idea of breaking the *software monoculture,* defined as the tendency of having multiple connected computers running the same software [11]. Researchers drawing an analogy from biological systems have argued that the presence of a monoculture in a network exposes it to a substantial

[1] It is important to stress that considerable gains in reliability can be achieved through N-version programming even in the absence of independence of failure [15] . Independence of failure should then be seen as a desirable goal in the development process, rather than an essential property that must be met for N-version programming to be valuable.

security risk, as identical softwares can be compromised simultaneously by the same attack vector. In this context, diversity can also be employed to decrease the attacker's knowledge of the target system's implementation details, thus making it harder for him to engineer a successful intrusion.

The renewed interest in N-version programming motivates us to revisit earlier research on this topic. Much of our knowledge about how to build effective N-version architectures comes from experiments that have been conducted in academic or industrial settings. In this paper, we review some of these experiments and contrast their conclusions. The object of this paper is to synthesize the lessons learned from these experiments on developing reliable software, rather than to exhaustively survey all research related to software diversity. We further identify open questions and remaining challenges and suggest possible avenues of solution. While we chose to focus specifically on experiments aimed at the development of highly reliable systems as this was the main object of most of the experiments conducted with N-version architectures, we believe this study would be useful to researchers and practitioners working in any of the related fields of dependability, availability, reliability, or security.

1.2 Terminology

Throughout the remainder of this paper, we will use the following terminology:

Reliability is defined as the probability of a system or a component to perform its required functions under stated conditions for a specified period of time. In other words, reliability is the probability of failure-free software operation for a specified period of time in a specified environment. Reliability is itself a component of *dependability,* defined as the ability of a system to provide a service that can justifiably be trusted. In addition to reliability, dependability includes another of other desirable attributes, namely availability, safety, integrity and maintainability [12]. While the focus of this study is on reliability, it is important to stress that the use of N-version programming in software systems affects every component of dependability. Intuitively, we can assume that availability will improve in tandem with reliability, while the maintainability of the system will be decreased by the introduction of additional components.

Dependability intersects with *security*, the absence of unauthorized access to, or handling of, system state [13]. More specifically, security is defined as defined as the conjunction of availability, confidentiality and integrity. The connection between reliability and security has been observed in several contexts, and we will discuss the use of N-version programming for security purposes in section 4.

N-version programming is a programming paradigm that consists in independently generating N functionally equivalent programs. Each of the independently generated programs is termed a *version* or an *instance*. A system that contains an element of N-version programming is *N-version architecture*.

A *failure* occurs when a resource does not deliver the expected service i.e., the system does not behave as specified, or the specification itself does not adequately capture the intended behavior of the system [14]. The cause of a failure is a *fault.* A fault causes the system to deviate from its expected behavior, and instead enter an incorrect state, called an *error*. Once the error causes the

system to fail to deliver its intended service, a failure has occurred. *Fault tolerance* describes the capacity of a system to continue to provide correct service in the presence of faults. N-version programming can be seen in this context as a strategy to increase system fault tolerance.

Of particular interest in the context of N-version programming are *coincident failures*. Two failures present in two different instances are coincident if they both occur when the instances are fed the same input, indicating the possible presence of a common fault between the two instances. Observe that this definition does not require that both instances originate with the same fault or return the same erroneous output.

The remainder of this paper is organized as follows. In Section 2, we survey several experiments that have been conducted in an academic setting to evaluate the feasibility of using N-version programming to increase the reliability of systems. In Section 3, we analyze the results of these experiments to uncover the challenges and research opportunities. Concluding remarks are given in Section 4.

2 Current State of The Art

We have reviewed several studies that focus on diversity of N-version programming techniques. We found that the proposed approaches can be categorized based on the software layer in which diversity is introduced. We distinguish between four main layers: diversity of implementations, diversity of the programming languages, design diversity, and finally data diversity.

2.1 Diversity of Implementation

The choice of the layer or layers that are to be diversified is the most central question arising when developing an N-version architecture, as alternative choices differ greatly with respect to both cost and the level of failure independence that can be achieved. Generally speaking, experiments have shown that the earlier diversity is introduced in the development process; the more likely that the final product will exhibit independence of failure.

The most common strategy is to develop several instances from the same specification, and using the same programming language. This is, for instance, the strategy used by NASA [15], Campbell et al. [16], Shimeall et al. [17] and Knight et al. [18]. In the former experiment, software that determines the acceleration of a vehicle was coded 20 times by 20 teams of coders, in 4 universities. All teams proceeded using the same specification and worked in isolation. Results were not completely encouraging. For instance, in the NASA experiment, despite the fairly low rate of occurrence of failures of each version, the various instances exhibited a higher rate of coincident failures than would have been expected if failures were completely independent. As the authors starkly conclude: "*Coincident failures occurred at rates that greatly exceed the rates expected by chance under the assumption of independence*." And furthermore: *"The assumption if independence is clearly not justified"*.

Common faults leading to coincident failure between two or more systems seemed to have two causes: difficulties on the part of the programmers in manipulating the complex mathematical objects needed to solve the problem at hand, and misunderstandings of the specification. In the latter case, it must be stressed that ambiguities in the specification cannot be faulted for the coincident failures since in the worst case at most 6 of the 20 versions exhibited a given fault. Dissimilar faults causing coincident failure were also observed.

Similar results were found by Knight et al. who developed 27 versions of a launch interceptor, all in Pascal, and from the same specification and subjected the resulting programs to one million input tests. The reliability of the 27 instances was very high, with 6 instances exhibiting no failure for any the one million inputs tested, and every other being successful for over 99% of inputs. There were, however, a number of cases in which multiple (up to 8) versions failed for the same input. Using a statistical analysis, Knight et al. showed that the occurrences of common failures were higher than would have been expected under an assumption of independence. They concluded categorically that: "*the assumption of independence of errors that is fundamental to the analysis of N-version programming* ***does not hold***" (emphasis in original). Indeed, about one half of all failures involve at least two instances.

In subsequent work [19], Brilliant et al. re-examined the results of this experiment and try to identify, amongst the possibilities listed above, the main cause of coincident failures. They found that there are a number of cases of faults that are not logically related (in the sense that they reflected the same or similar mistakes, and occurred in the processing of the same part of the problem), and yet produce coincident failures. This is explained by the fact that in these cases, both faults involve mishandling inputs that share a certain specific characteristic. It is not so much that the same mistakes were made in programming as that the inputs creates conditions which the programmers did not anticipate. Rather than logically related faults, the authors propose reasoning about input domain related faults, which occur when a given input value triggers certain execution paths. The extent of failure correlation thus depends on the proportion of inputs which lie inside the failure region. This result argues in favour of data-based diversity as seen in [20].

A similar experiment was conducted by the University of Iowa and the Rockwell/Collins Avionics Division [20]. Twelve programming teams of graduate students independently designed, coded, and tested 12 computerized airplane landing systems in C, from a single specification. The purpose of this experiment was to test a software development paradigm specifically tailored to the development of highly diverse N-version software. The results showed great benefits to using this development paradigm. Despite extensive testing, only two pairs of common faults were found between the 12 instance programs. Furthermore, 3-version architectures with output voting exhibited, on average a seven-fold improvement in reliability compared with single version, while deploying these same 12 programs in 5-version architectures yielded an average improvement in reliability by a factor of 7. When the architecture considered the timing of the occurrence of a failure, rather than simply contrasting outputs, the benefits of diversity where even more evident, with the 3-version architectures being on average 12 times more reliable than single versions, and no coincident failures detected in the 5-version architecture.

The final experiment in N-version programming at the implementation layer which we will examine is that performed by Shimeall et al. [17]. The goal of this experiment was to compare the efficiency of N-version programming against that of other fault-detection techniques, and determine if the cost incurred by developing multiple instances of the same software could be offset by a reduction in the costs of verification and validation.

Their experiment was performed using eight programs coded in Pascal from the same specification of a system that models the movements of military units. Each of the versions was subjected to five different fault detection or fault tolerance techniques, namely: code reading by step-wise abstraction, data flow analysis, runtime-assertions, functional testing and 3-version voting. The 3-version voting was conducted by subjecting each instance to 10 000 randomly generated inputs and checking the behaviour of the architecture for each of the 56 possible triplets.

Interestingly, the authors found that faults that were tolerated were not the same as those that were detected using traditional fault detection techniques. A total of 67 distinct faults were tolerated (by at least one triplet) but not detected (the total number of faults was not given). Conversely, only 24 of the 103 faults that caused coincident failures were detected by any of the fault detection techniques used. These results strongly suggest that N-version programming and fault detection should be seen as complementary tools rather than alternatives. The authors also

hypothesize that this result indicates that the faults that cause coincident failures are amongst the most difficult to detect.

Taken together, these experiments seem to play in favour of using multiple instances to increase reliability, despite the observed absence of failure independence. However, the absence of statistically verified failure independence is troublesome, as it indicates that the reliability gains associated with N-version programming are not as great as we could have hoped. It is thus necessary to investigate whether or not introducing diversity at another layer of software development would provide better results.

2.2 Diversity of Programming Languages

Since diversifying only at the level of the implementation alone is insufficient to ensure independence of failure between the instances, an added measure of diversity can be introduced by diversifying both the implementation and the programming language i.e., developing each instance in a different programming language. This is the strategy taken by Gmeiner et al. [22], Avizienis et al. [23][24] and Adams et al. [25]. In the latter case, N-version architectures built using the same programming languages were compared to architectures built using different programming languages.

In the experiment conducted by Avizienis et al., six teams of two developers each produced a flight simulator. Every team was working from the same specification, written in English, and each team was assigned a different programming language. The six programming languages chosen for the experiment were C, Pascal, Ada, Modula-2, Prolog and T. These six languages cover a broad spectrum of programming paradigms since two are procedural languages, two are object-oriented, one is logic programming and one is functional programming. A similar experiment had been performed by Gmeiner and Voges [22] in 1979. In this experiment, safety-critical software, a reactor safety system, was coded in three instances, from a single specification, using three different programming languages, namely IFTRAN, Pascal and PHI2.

While the hypothesis of failure independence was not formally tested, these experiments show that substantial improvements in reliability can occur through the use of this type diversity. Coincident failures were rooted in misunderstandings or ambiguities of the specification.

Adams et al.'s experiment is particularly revealing since it contrasted diversity introduced at the layers of implementation and programming language in a controlled setting and tested the hypothesis of independence. In effect, Adams et al. repeated the experiment from [18] using two sets of programs coded using two different programming languages, namely Modula-2 and PROLOG. As was the case in the experiments conducted by Avizienis et al. and Gmeiner et al., it was hypothesized that a high level of diversity could be achieved using these two languages since they are based upon different programming paradigms.

The experiment was conducted using six Modula-2 programs and 5 PROLOG programs, coding the same launch interceptor as was used in [18]. These programs were then subjected to 9878 input tests and in each case the output of each version was contrasted against that of a gold version to determine its level of reliability.

Adams et al.'s data shows that the hypothesis of independent failure is not warranted if two versions are coded using the same programming languages, with common failure occurring between one and two order of magnitudes more frequently than would be the case if failure was independent. However, their analysis shows that using two versions written using different programming languages increases the chances to achieve true independence of failure.

A recent investigation by van der Meulen et al. [26] adds credibility to these results. Their research was conducted with upwards of 36,000 programs submitted by students to a contest website. The programs were written in C, C++ and Pascal and computed a well-known mathematical formula. The programs were tested using a 2-Version approach with randomly selected pairs of programs. A failure is detected if the two instances returned different result. In a second phase, the same experiment was repeated for 61 different problems, with a combined total of 89,402 programs, in order to generate a statistically significant dataset. Interestingly, the authors found that the size of the pool of programs from which those in a 2-version architecture were drawn does not affect reliability. Van der Meulen et al. found that the effectiveness of this approach for the more unreliable programs is close to the independence of failure assumption. For more reliable programs, the 2-version architecture still brought improvements in reliability in the order of 100 on average. The authors also found that different programmers using different programming languages did tend to make different faults, leading to lower rate of coincident failures.

2.3 Diversity of Specification and Design

In several of the experiments discussed above, the principal causes of common faults were misunderstandings of the specification or outright errors in the specification. It is thus natural to ask if better results could be obtained by introducing diversity in the design phase of software development. This is the strategy that was used by Avizienis et al. [26] [5] and [27] in an experiment conducted at UCLA and by the PODS project on diverse software [28], a collaborative research project aimed, amongst other objectives, at evaluating the effectiveness of N-version programming.

In the UCLA experiment, a single specification for an airport scheduler was written in English, and in the specification languages OBJ and PDL. 18 programs were then produced in PL/1, of which seven were constructed from the OBJ specification, five from the PDL specification and six from the English specification. The programs were tested with 100 inputs.

The percentage of good outputs ranged from 35% to 98%. 21 common faults have been identified. Of these, five were rooted in common specification errors; seven resulted from logic errors made by the programmers and nine from implementation errors. The 18 instances were then arranged into the 816 possible combinations of three programs, and execution in a 3–version architecture with majority voting.

The results of this experiment do not indicate that 3-versions built from three programs written from different specifications are more reliable than those built from programs written using the same specification. The authors did not advance an explanation of this somewhat surprising result. One possibility is that this results from the fact that specification errors, which were the

most frequent source of common faults between the instances in other experiments, were in this case the rarest source of common faults.

The experiment conducted as part of the PODS project [28] was even more thorough in introducing diversity in every step of the development process simultaneously. Three instances of a reactor over-consumption protection system were developed independently. Each development team produced its own software specification from a customer-supplied requirement specification, and then produced an implementation accordingly. Two teams used Fortran and the third used assembly code. Yet another layer of diversity was introduced by supplying each team with one of two possible power consumption calculation algorithms. Finally, the programs also differed with respect to the kind of testing that was applied to them during a verification phase.

The three programs were then tested against each other to detect residual faults by comparing outputs values. The three versions contained a total of seven faults, of which six were attributable to mistakes in the original customer specification, and the last related to ambiguity in one of the software specifications (written by the development team). There were two common faults, both of which arose from the customer specification.

Experiments in 3-version voting show substantial improvement in reliability over a single version architecture. Indeed, the failure rate of the N-version architecture was substantially lower failure rate than any of the single versions composing it, thought the rate of coincident failures is not reported.

Another interesting conclusion of the approach is that iteratively improving the component comprised in an N-version architecture does not result in correspondingly monotonic gains in reliability for the overall system. Instead, the majority vote failure rate seems to reach a series of "plateaux", and does not improve until several corrections have been brought to every version.

2.4 Data Diversity

As a final alternative, diversity can be introduced at the level of the data manipulated by software. This strategy was suggested by Ammann et al.[20]. For most complex systems, any given input can be expressed in a number of different but equivalent ways. For instance, the specifications have some tolerance when it comes to input values. Experiments show that for many faults, a given input will cause an error to occur even though another equivalent input value will not. Faults can thus be detected by inputting multiple re-expressions of the same value and correlating the results.

Each input is passed through a rewriting algorithm to generate a series of equivalent inputs. Alternatively, if it is not possible to find an equivalent input, the input could be distorted, and the distortion removed on the output. The set of input values for which a given program returns an invalid value is called the failure domain of this program. The possibility of using data diversity to increase reliability or tolerate faults is thus contingent on the capacity to generate alternative input values that lie outside of a given failure domain, even if the original input does not. Since failure regions vary greatly in size, the difficulty of successfully using this technique varies accordingly.

Ammann et al. conducted an experiment to evaluate the efficiency of this technique on a program that simulates the decision procedure of a hypothetical antimissile missile launch system. The experiment was performed for both 3-copies and 5-copies diversity, with voting performed by majority vote. Input data associated with seven known faults was used. In both cases, four faults were successfully tolerated and three were not.

An open question that deserves future attention is to identify exactly which faults are more likely to be tolerated using data diversity. Unfortunately, this method seems ill-suited to tolerate faults arising from misunderstandings of the specification, which is the main cause of coincident failure in other diversity based architectures.

One of the main benefits of introducing diversity at the level of the data is that multiple copies of the same program can be used (N-copies instead of N-versions). This greatly reduces the costs associated with the method. Furthermore, since the alternate inputs are automatically generated, it should be possible to experiment with a very high number of diverse instances without incurring prohibitive overhead costs. However, voting can be problematic since the various inputs may return different acceptable (equivalent) outputs, with no clear majority.

This strategy is not mutually exclusive with the other design diversity approach discussed above, and they could be used complementarily. It remains to be seen if the complementarily between these approach can be used to successfully tolerate faults that resist to either approach when they are used independently.

2.5 Summary of the Techniques

Table 1 summarizes how the experiments surveyed in this paper can be classified with respect to the following criteria. The level of communication allowed between development teams is not shown in this table since in every experiment, teams where allowed the same minimal level of communication.

- **The layer where diversity is introduced:** As can be seen in the discussion above, diversity can in principle be introduced at any layer of a program's development or usage. This choice is consequential in several respects, but most importantly w.r.t. whether or not failure independence between the instances is achieved.
- **The number of instances present in each diversified layer**: In the context of increasing reliability, N-version programming can be constructed for any value of N greater than two. Different Ns have been used as shown in the table.
- **The method used to correlate the instances:** This refers to the way instances of a diverse architecture are compared including input/output correlation and comparing a running instance with a gold version. A more precise method of comparison, for instance one that compares intermediate steps rather than simple outputs, is more likely to detect errors. Furthermore, a more precise comparison method might distinguish between multiple faults that reveal themselves in identical failures.
- **The method by which diversity is introduced in the development:** We distinguish between two methods by which diversity can be introduced in a given architecture. *Random diversity* occurs as a result of the unique experience and intuition of each programmer. If

this path is taken, then the specifications and programming instructions given to the developers of each instance should be minimal, thus giving maximal leeway to each developer or developing team to make design decisions. This is contrasted with *required diversity*, in which the developers of each instance are purposefully given different instructions, with the object of maximizing diversity between the instances.

- **The level of isolation of the teams developing the various instances:** In all experiments**,** a fairly rigid protocol was used to prevent any kind of collaboration between the developers of the various instances. Only form of collaboration was generally allowed: requests for clarifications about the specification were broadcast to all developers**.** In principle, other channels of communications could be considered permissible.

Table 1: Summary of the experiments in diversity for reliability

Experiment	Layer	Source of diversity	Correlation	Number of Instances	Main Conclusions
Gmeiner and Voges [22]	Programming languages	Random	Input/output & intermediate steps	3	•The faults detected by correlating the N-versions are not the same as those detected using traditional fault detection techniques.
The Design Diversity Experiment [5][27]	Specification	Random	Input/output	18, arranged in 816 combinations of 3-versions	•There was no noticeable gain from introducing diversity at the level of the specification, rather than at the implementation. However, the number of common faults rooted in the specification seems diminished.
PODS [29]	Specification, programming language, algorithm and testing	Random	Input/output	3	•N-version is effective at reducing the rate of failure. •Improvements to the instances of an N-version architecture do immediately result in corresponding gains in reliability for the overall system. Instead, the failure rate reaches a series of "plateaux", and does not improve until several corrections have been brought to every version.
Knight and Levenson [18]	Implementation	Random	Input/output	27 copies, tested in 3-versions	•Common failure occurred at a rate far higher than would be expected under the assumption of independence. •"[T]*he assumption of independence of errors [...] programming* ***does not hold***."
6-language Experiment [23]	Programming languages	Random	Input/output	6, tested in both 3- and 5-versions	•Substantial improvements can occur through the use of N-version programming. •Reliability is increased by using a higher value of N.
NASA [15]	Implementation	Random	Consistency Relation	20	•"*Coincident failures occurred at rates that greatly exceed the rates expected by chance under the assumption of independence*." *"The assumption if independence is clearly not justified".* •Nonetheless, the use of N-version programming seems justified to increase reliability.
Shimeall and Levenson [17]	Implementation	Random	Input/output and a gold version	8, tested in 3-versions	•The faults causing common failure are not the same as those which are easier to detect using other fault detection methods. Conversely, faults that are tolerated may not be detected.
Adams and Taha [25]	Programming Language and Programming paradigm	Req.	Input/output	7 versions coded in 2 programming languages.	•Independently written programs using the same programming language do not exhibit independence of failure. •Using two versions written using different programming languages is sufficient to achieve true independence of failure.
Data Diversity [20]	Data	N/A	Input/output	N/A	•Exhibits a substantially lower cost compared with the other layers where diversity may be introduced.
Lyu and He [21]	Implementation	Random	Input/output, and time of occurrence of the fault	12 versions, arranged in both 3- and 5- versions	•Average improvements in reliability on the order of 7-fold for 3-version architectures with output voting and 12-fold for 5-version architecture. •Event greater gains in reliability occur when the timing of a fault is taken into consideration.
Van der Meulen and Revilla[26]	Implementation and Language	Random	Input/output	2-versions, chosen from a pool of 36,123 programs	•The failure rate exhibited by the 2-versions is close to independence of failure for unreliable programs, and the approach exhibits substantial improvements even for the more reliable ones. •The use of different programming languages is an effective way to increase the rate of coincident failures between the versions.

2.6 Synthesis of Findings

The main conclusions of this study can be synthesized in the form of influence diagrams. Such diagrams allow us to model how a given design choice or factor affects other factors or attributes, and to display this relation using a concise and visual representation. For the moment, we will limit our analysis to whether or not the impact is positive (+) or negative (-), and sometimes add a conditional predicate to this outcome. An objective of further research is to quantify at least some of the dependencies shown here.

As a starting example consider the introduction of redundancy in a system. The presence of multiple instances running in parallel increases the reliability of the system, since if one instance fails; alternative redundant one is available to replace it. However, the maintainability of the system is increased, as is its cost, since multiple instances must now be built and maintained. This is illustrated in figure 1.

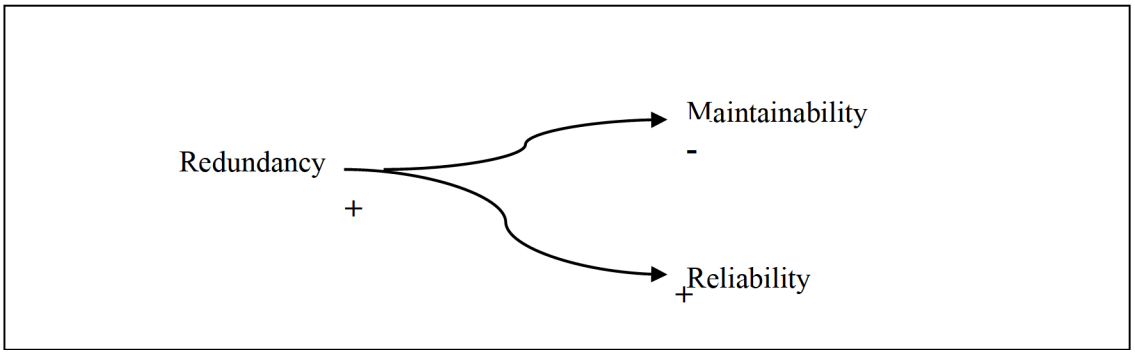

Figure 1 Redundancy and its outcomes

Both the maintainability and the reliability of a system correlate with its operating costs. The above figure can thus be amended to reflect this reality.

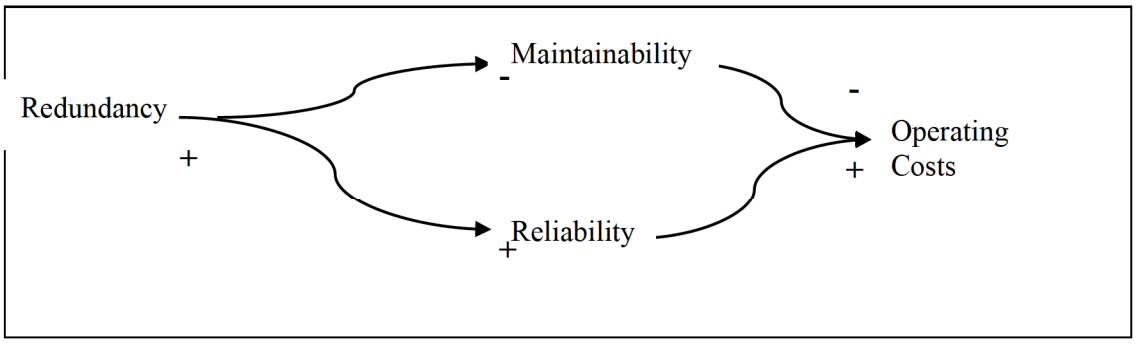

Figure 2: Redundancy and its outcomes.

Observe that two arrows enter the Operating costs node, one indicating a negative influence, and the other a positive one. Since we have not yet quantified these relations, the influence diagram does not inform us as to whether or not the benefits (with respect to costs) accrued from introducing redundancy offset the increased costs.

As discussed in above, redundancy is insufficient to ensure the security and reliability of software systems and diversity must be employed instead. For software systems, diversity affects maintainability and reliability in the same manner as redundancy does in hardware systems.

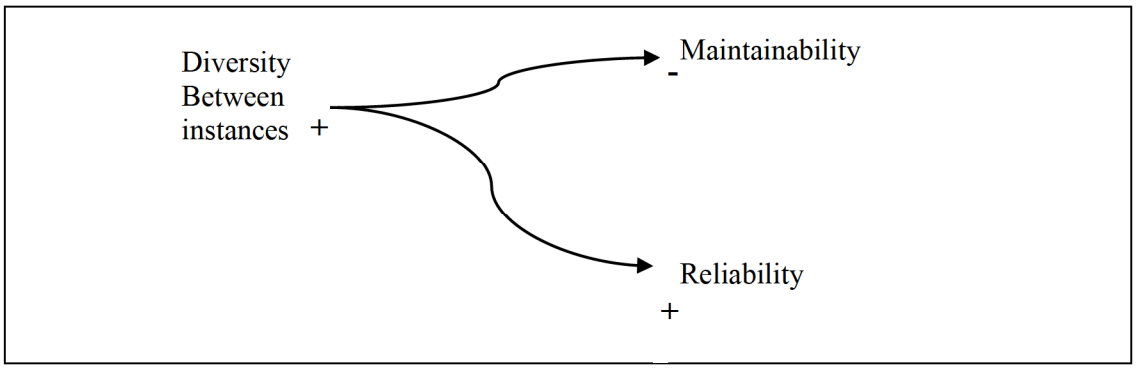

Figure 3: Diversity and its outcomes

Several strategies are used to maximize the degree of diversity between the multiple instances of an N-version architecture, since it is from this diversity that N-version programming draws its promise of greater reliability. In particular, great care is often taken to isolate the different teams developing each instance, in the belief that teams operating in isolation will make different design decisions. However, the literature on N-version programming is unclear with respect to whether or not this is an effective method of generating diversity. Indeed, programmers operating in isolation often tend to make similar design choices and even to make similar mistakes.

On the other hand, tailored diversity occurs when different developers are given differing instructions with respect to how the problem at hand should be addressed. Diversity is thus forced in the system by design, rather than introduced randomly. In the optimal case, it is hoped that certain instances will be more likely to fail on those instances where other instances are more likely to succeed, thus maximizing the diversity and minimizing the occurrence of coincident failures.

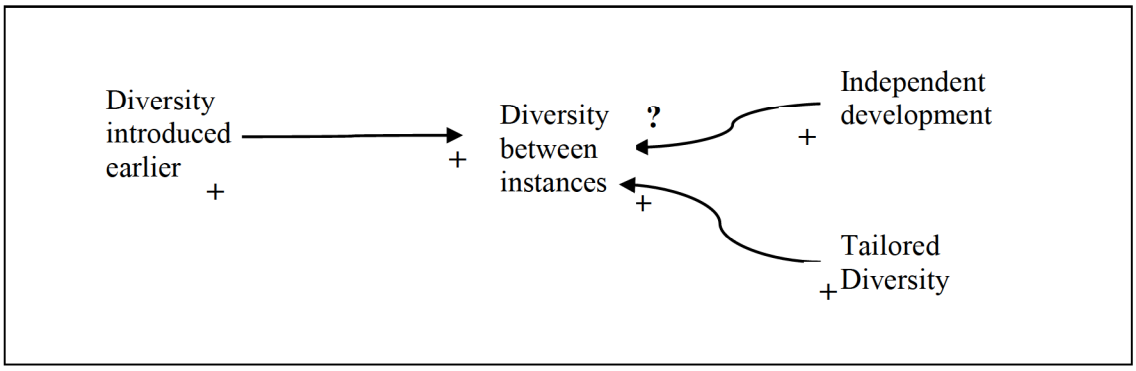

Figure 4: Impact of design choices on the level of diversity in an N-version architecture.

The final aspect to consider is that of the number of instances present in the architecture. Maintainability is predictably reduced propositionally to the increased number of instances. The question whether or not an increase in the number of instance leads to a linear increase in reliability is more subtle. Prior research seems to indicate that, in the presence of coincident errors, there exists a threshold up to which each additional instance does increase the reliability of the overall architecture. Beyond this threshold however, each additional instance may reduce the reliability of the system.

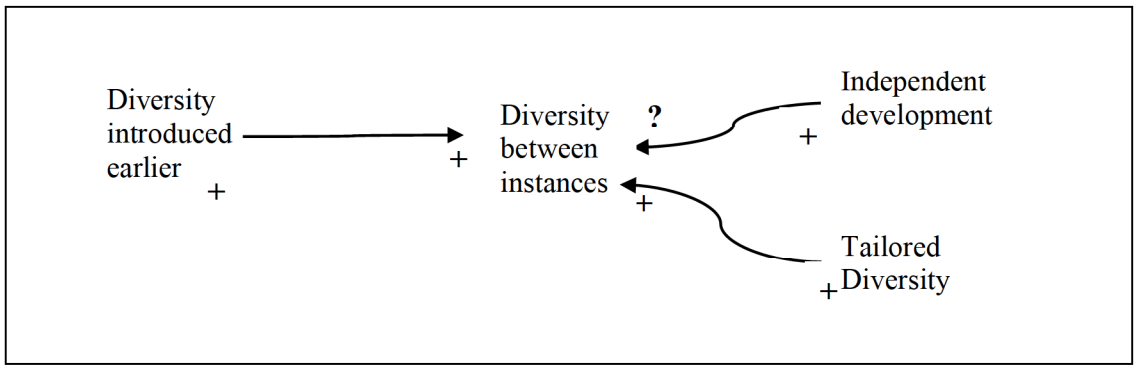

Figure 5: Impact of altering the number of instances present in an N-version architecture.

3 Challenges and Recommended Solutions

In this section, we elaborate on the main conclusions that can be drawn from surveying the various experiments that have been performed in using diversity and suggest perspectives for future research.

First, independence of failure has been seen by multiple authors as an important goal in the development of multiple versions. However, the experiments detailed above disagree with respect to whether or not independence of failure is achieved through independent development. Unfortunately, most of the experiments do not provide data about whether or not failure independence has been achieved. Table 2 summarizes the only results that are given.

Table 2: Independence of failure in experiments

Experiment	Layer	Independence of failure achieved
NASA [15]	Implementation	No
Adams and Taha [25]	Programming Language	Yes
Adams and Taha [25]	Implementation	No
Knight Leveson [18]	Implementation	No

As is clear from this table, different development teams working from the same specification may not achieve independence of failure. The use of alternative specifications and programming languages is thus clearly warranted. Introducing diversity at different layers however would also affect the cost and the risks due to the difficulty of the approach. There is therefore a need to study the trade-offs between accuracy of the approach, cost, and associated risks.

The sources of coincident failures are another important topic of discussion. In Table 3, we list the main causes of coincident failures that have been identified. The table also shows the experiments in which has been reported.

Table 3: Main causes of coincident failures

Source of Coincident failures	Experiment
Misunderstanding of the specification	[15][23]
Ambiguous or erroneous specification	[22][28]
Common programming error or unexpected input case	[15][17][18][25]

Each of these elements should be tackled independently. In the first two cases (misunderstanding and erroneous specifications), the number of faults arising in the specification could be effectively reduced through the use of formal or semi-formal specification languages and other formal and semi-formal methods. Tackling the third may prove more difficult and necessitates that every instance be developed with care. In fact, this result indicates that N-version programming, while useful to increase the reliability of systems, is not a substitute for good programming practices.

Another central question in any N-variant architecture is the number of instances that should be developed (i.e. the value of N). As discussed above, only one study, that of Avizienis et al. [23], experimented with several possible values. The choice of N raises several interesting questions: is increasing the value of N an effective way to increase the reliability of the overall architecture or to reduce the number of coincident failures? If this is the case, how can we balance the increase in reliability with the increase in costs associated with developing more instances of the desired software?

Partial answers to these questions are given in a theoretical study by Eckhardt et al. [2]. They show that in the presence of common faults, a higher number of instances are required to achieve the same reliability than would be the case if failures between the instances were completely independent. Furthermore, if the number of coincident failures is high, then there exists an optimal value of N, and increasing the number of instances in the system above N will lead to a *decrease* in reliability. This result indicates that preliminary research suggesting that highly reliable systems could be built from unreliable components simply by multiplying the number of components does not bear out. Instead, every instance comprised in an N-version architecture must be built with care if the overall architecture is to exhibit a high reliability. This confirms the result of [28] to the effect that improving the reliability of an N-version architecture necessitates improvement to every component.

In all cases which we have studied, the multiple instances were developed in isolation, with an expectation this would increase the amount of diversity between them. While this hypothesis is

intuitive, it, like any other, must stand the test of experiment before it is validated and adopted. In this respect it is important to recall that the main causes of common faults in several experiments were misunderstandings of the specification, and difficulty in dealing with unexpected conditions of the input value. Allowing discussions between the programmers with respect to these points would reduce the occurrences of such faults while not unduly restricting the amount of diversity between the instances.

It is clear from the experimental data that enforced diversity achieves better results than random diversity. Indeed, even when given broad liberty as to how to implement a given specification, programmers often made similar design choices, thus leading to a reduction in the amount of diversity of the overall architecture, and often to an absence of failure independence. A higher level of diversity is thus achieved by imposing different choices (such as alternative programming languages or algorithms) on each development team. The experiments surveyed above indicate that diversity at the level of the specification and algorithms are particularly desirable.

Pushing this conclusion further, the results of the experiments we surveyed strongly argue in favor of *tailored* diversity[10], in which the different specifications given to the developers of each instance are calculated so that one instance is more likely to fail in those cases where another is more likely to succeed. Such an approach to the design and selection of diverse components has been longstanding practice in hardware fault tolerance, and should be applied to software diversity as well.

Throughout this paper, we have discussed the issue of "increasing the amount of diversity" between instances, echoing similar language in present in the original articles we surveyed. However, very little headway has been made in quantifying objectively how much diversity exists between two instances. Lyu et al. [29], propose to use software metrics such as number of lines of code and number of modules as a rough guide to how different two instances of the same software are. However, these techniques do not take into account the control flow of the running instances, which reveals important insights in the way the instances behave. A future direction should therefore be correlating the behavior of instances based on execution traces and other types of run-time information. We believe that this would allow a more objective comparison.

While this paper is chiefly concerned with research aimed at developing reliable software, concurrent thread of research has sought to harness the potential of diversity to improve the security of software [10]. There are several parallels between these two problems. For instance, faults causing reliability failures can also be exploited by an attacker to gain unauthorized access to the system. However, several important distinctions must also be stressed, for instance, as observed in [9], important security attributes such as confidentiality cannot be improved by replication. Adapting the paradigm of N-version programming to the issue of security will thus necessarily involve the development of an alternative reasoning framework, more suited to achieving security goals.

Another useful avenue for future research is to develop a reasoning framework that will allow us to evaluate objectively how much reliability is gained by the introduction of diversity in an architecture in different manners. How can we compare, for example, an N-version architecture consisting of N instances, built from the same specification, but using different programming languages with another architecture in which the versions are developed from different specifications but using the same programming language?

To answer this question, we first need to investigate pragmatic ways to measure how much diversity exists between two instances of an N-version architecture. In this respect, it is important to note that only some aspects of an architecture are diversified. For example, the instances may share the same specification or the same operating system, but differ on the level of the source code and system libraries. Diversity metrics would guide further research in determining the optimal layers where diversity should be injected.

4 Conclusion

In this paper, we discussed the various strategies for implementing the N-version programming paradigm to inject diversity in a software system. The surveyed studies vary mainly depending on the software layer in which diversity is introduced. Though an important objective of diversity is to achieve independence of failure, most surveyed studies did not clearly show that this objective was attained. We also discussed in this paper the main challenges along with research opportunities of N-version programming. The latter include the necessity to study ways to select an optimal N for diversity to be effective, the challenges related to the method by which diversity is injected, the ways various instances of a diversified environment need to be correlated, and the need to conduct cost-effectiveness analysis of a diversity solution. Despite these challenges, we believe that N-version programming has good potential of becoming the design solution of choice for making critical systems more reliable.

We would also like to point out that reliability is just one of several attributes describing two interconnected goals of dependability and security. The other attributes are availability, safety, integrity maintainability and confidentiality. Each of these attributes imposes constraints and objectives to system developers, and diversity may play a role in all cases. It is certain that the lessons learned in studying the use of diversity in the context of reliability can be transferred to the use of diversity to ensure the other attributes mentioned above.

References

[1] A. Avizienis. The N-Version Approach to Fault-Tolerant Software. 1985, IEEE Transactions on Software Engineering 11 (12), pp. 1491--1501.

[2] D. E. Eckhardt, Jr. and L. D. Lee. A theoretical basis for the analysis of redundant software subject to coincident errors. NASA Technical Memorandum 86369, 1985.

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

[4] D. Partridge and W. Krzanowski. Distinct Failure Diversity in Multiversion Software. University of Exeter, U.K., 1997.

[5] A. Avizienis. The N-Version Approach to Fault-Tolerant Software. 1985, IEEE Transactions on Software Engineering (TSE) 11(12), pp. 1491-1501.

[6] D. Gao, M. K. Reiter and D. Song. Behavioral Distance Measurement Using Hidden Markov Models. In Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID 2006), Hamburg, Germany, September 2006

[7] D. Gao, M. K. Reiter and D. Song. Behavioral Distance for Intrusion Detection. In Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005), Seattle, WA, USA, September 2005

[8] I. Gashi, et al. An Experimental Study of Diversity with Off-the-Shelf AntiVirus Engines. In Proceedings of The Eighth IEEE International Symposium on Networking Computing and Applications (NCA). Cambridge, Massachusetts , USA. 2009. pp. 4-11.

[9] F. B. Schneider. Beyond traces and independence. Dependable and Historic Computing. Essays Dedicated to Brian Randell on the Occasion of His 75th Birthday, Lecture Notes in Computer Science, Vol. 6875 (Cliff Jones and John Lloyd, eds). Springer Verlag, 2011, 479--485.

[10] B. Littlewood and S. Lorenzo. Redundancy and Diversity in Security. In Proceedings of the 9th European Symposium on Research in Computer Security (ESORICS 2004), Sophia Antipolis, France, September, pp. 423-438, Springer-Verlag, Lecture Notes in Computer Science 3193, 2004.

[11] D. Williams et al. Security through Diversity: Leveraging Virtual Machine Technology. IEEE Security & Privacy. 2009,7 (1), pp. 26--33.

[12] ANSI/IEEE. Standard Glossary of Software Engineering Terminology. STD-729-1991, 1991

[13] A. Avizienis, J.-C. Laprie, B. Randell, Fundamentals of Dependability,Tech Report UCLA CSD Report no. 010028, LAAS Report no. 01-145, Newcastle University Report no. CS-TR-739, 2001.

[14] A. Avizienis and J. Kelly. Fault Tolerance by Design Diversity: Concepts and Experiments. 1984, Computer 17(8), pp. 67 - 80.

[15] D. E. Jr. Eckhardt et al. An experimental evaluation of software redundancy as a strategy for improving reliability. NASA, 1990.

[16] R. Campbell et al. Preliminary design of the redundant software experiment. NASA tech report, 1985.

[17] T. Shimeall and N. Levenson. An Empirical Comparison of Software Fault Tolerance and Fault Elimination 1991, IEEE Trans. Software Eng. 17(2), pp. 173-182.

[18] J. Knight, and N. Levenson. An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. 1986, IEEE Trans. Software Eng.

[19] S. Brillant, J. C. Knight, and N. Levenson. Analysis of Faults in an N-Version Software Experiment. 1990, IEEE Transactions on Software Engineering (TSE), 16(2), pp. 238-247.

[20] P. Ammann, and J. C. Knight. Data Diversity: An Approach to Software Fault Tolerance. 1988, IEEE Trans. Computers, pp. 418--425.

[21] M. R. Lyu and Y. He. Improving the N-Version Programming Process Through the Evolution of a Design Paradigm. IEEE Transactions on Reliability. 1993, Vol. 42, 2, pp. 179-189.

[22] L. Gmeiner, L. U. Voges Software diversity in reactor protection systems: an experiment. In Proceedings of the IFAC Workshop SAFECOMP 1979.

[23] A. Avizienis, M. R. Lyu and W. Schutz. In Search Of Effective Diversity: A Six-Language Study Of Fault-Tolerant Flight Control Software. Tokyo, Japan. Proceedings of the 18th International Symposium on Fault-Tolerant Computing (FTCS-18) 1998. pp. 15-22.

[24] M. R. Lyu, and A. Avizienis. Assuring Design Diversity in N-Version Software: A Design Paradigm for N-Version Programming. In Proceedings 2nd IEEE International Working Conference on Dependable Computing for Critical Applications, Tucson, Arizona, February 18-20 1991, pp. 89-98.

[25] J. M.Adams, and A. Taha. An experiment in software redundancy with diverse methodologies. In Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences. Kauai, HI, USA pp. 83-90.

[26] A. Avizienis. Design diversity: an approach to fault tolerance of design. AFIPS Press, 1984. In Proceedings if the AFIPS National Computer Conference. Las Vegas, Nevada, California pp. 163-171.

[27] A. Avizienis and J. P. L. Kelly. A specification-oriented multi-version software experiment. In Proceedings of the Thirteenth International Symposium on Fault Tolerant Computing (FTCS 13) Milan , 1983.

[28] P. Bishop et al. PODS—A project on diverse software.. 1986, IEEE Trans. Softw. Eng.12(9), pp. 929--940.

[29] M. R. Lyu, J.-H. Chen, and A. Avizienis. Software Diversity Metrics and Measurements. In proceedings of the Sixteenth Annual International Computer Software and Applications Conference (COMPSAC '92), Chigaco,. IL USA. 1992. pp. 69-78.

Glossary

Coincident Failure

The occurrence of a failure on more than one instance of the components of an N-version architecture, on a given input.

Dependability

The ability of a system to provide a service that can justifiably be trusted.

Diversity

Fault-tolerance strategy based on the intuition of providing multiple different instances of a component and of contrasting their outputs.

Error

The deviation in system state caused by a fault, and leading to a failure.

Failure

The occurrence of a situation where the target system does not deliver the expected service. This can occur because the system does not behave according to its specification, or because the specification itself does not adequately capture the intended behavior of the system.

Fault

The adjusted cause of a **Failure**.

Fault Tolerance

The capacity of a system to continue to provide correct service in the presence of faults.

Independence of Failure

The property of an N-version architecture in which the occurrence of a failure in one instance on a given input does not provide any information with respect to the probability of failure on the other instances on this same input.

Instance

Each of the independently generated programs used in an N-version architecture. Also called version.

Maintainability

The probability of performing a successful repair action in a given time.

N-Version Programming

A programming paradigm that consists in independently generating N functionally equivalent programs.

Redundancy

Fault-tolerance strategy based on the intuition of providing multiple identical instances of a component of switching to one of the remaining instance in case of failure.

Reliability

The probability of a system or a component to perform its required functions under stated conditions for a specified period of time.

Security

The absence of unauthorized access to, or handling of, system state.

Version

See **Instance**.

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATOR (The name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.)

Defence R&D Canada – Valcartier
2459 Pie-XI Blvd North
Quebec (Quebec)
G3J 1X5 Canada

2. SECURITY CLASSIFICATION (Overall security classification of the document including special warning terms if applicable.)

UNCLASSIFIED
(NON-CONTROLLED GOODS)
DMC A
REVIEW: GCEC JUNE 2010

3. TITLE (The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S, C or U) in parentheses after the title.)

Best Design Practices for Effective Use of N-Version Programming

4. AUTHORS (last name, followed by initials – ranks, titles, etc. not to be used)

Khoury, R.; Couture, M.; Hamou-Lhadj, A.

5. DATE OF PUBLICATION (Month and year of publication of document.)	6a. NO. OF PAGES (Total containing information, including Annexes, Appendices, etc.)	6b. NO. OF REFS (Total cited in document.)
February 2012	40	29

7. DESCRIPTIVE NOTES (The category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

Technical Memorandum

8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the research and development – include address.)

Defence R&D Canada – Valcartier
2459 Pie-XI Blvd North
Quebec (Quebec)
G3J 1X5 Canada

9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.)	9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.)

10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.)	10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)
DRDC Valcartier TM-2013-017	

11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification.)
Unlimited

12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in (11) is possible, a wider announcement audience may be selected.))
Unlimited

13. ABSTRACT (A brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual.)

N-version programming is a software development paradigm that draws upon the concept of diversity to increase the reliability of software. The central idea is to independently produce multiple functionally equivalent versions of a program, and execute them in parallel. If the versions fail independently, then the probability of multiple versions producing a faulty output on any given input is very small; much lower than the failure probability of any single version. In this technical memorandum, we examine and contrast various experiments that have been performed to evaluate the benefits of this approach and draw some conclusions with respect to the most effective way that N-version programming can be utilized. We find that for diversity to be effective, it must be introduced in a targeted and informed manner and encompass several phases of the software's development.

La programmation en N-versions (N-version programming) est un paradigme de programmation qui s'appuie sur le concept de diversité pour assurer la fiabilité des logiciels. L'idée clef est de développer de multiples versions fonctionnellement équivalentes et de les exécuter en parallèle. Si l'occurrence de failles dans chacune de ces versions exhibe une distribution indépendante de celle des autres versions, alors la probabilité d'un échec lors de l'exécution simultanée de toutes les versions est très faible. Dans ce mémorandum technique, nous examinons les résultats de diverses études académiques qui ont évalué les bénéfices de ce paradigme de cette approche, et synthétisons des conclusions quant à son utilisation efficace. En particulier, nous concluons que pour que l'usage de la diversité soit efficace, celle-ci doit être introduite d'une manière ciblée et informée, et ce, dans plusieurs phases du développement du logiciel.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

Software reliability, System design, N-version programming, fault-tolerance

Defence R&D Canada

Canada's Leader in Defence
and National Security
Science and Technology

R & D pour la défense Canada

Chef de file au Canada en matière
de science et de technologie pour
la défense et la sécurité nationale

www.drdc-rddc.gc.ca