

Buidling Dependable Software for Critical Applications: Multi-Version Software versus One Good Version

PAUL TOWNEND JIE XU MALCOLM MUNRO

Dept. of Computer Science, University of Durham, DH1 3LE, England

{P.M.Townend, Jie.Xu, [Malcolm.Munro](mailto:Malcolm.Munro@durham.ac.uk)}@durham.ac.uk

Abstract

An increasing range of industries has a growing dependence on software-based systems, many of which are safety-critical, real-time applications that require extremely high dependability. Multi-version programming has been proposed as a method for increasing the overall dependability of such systems - however, the increased cost of using this approach may mean that this increase in dependability is not worth the extra expense involved. We describe an experiment undertaken in order to establish for the first time whether or not the multi-version method can offer increased dependability over the traditional single-version development approach when given the same level of resources. Three programs were developed independently to control a real-time, safety-critical system, and were put together to form a decentralized multi-version system. Three functionally equivalent single-version systems were also implemented, each using the same amount of development resources as the combined resources of the N -version system. The analytic results from this experiment showed that 1) a single-version system is much more dependable than any individual version of the N -version system, and 2) despite the poor quality of individual versions, the N -version method still results in a safer system than the single-version solution. Although these results could not be considered conclusive in the general sense and the experiment itself needed to be improved in several areas, it was evident that regarding the single-version method as a “seem-to-be” safer design decision for critical applications is not generally justifiable. We conclude by describing plans for a follow up study based on our initial findings.

1 Introduction

Software dependability is usually referred to as the property of a software system such that reliance can justifiably be placed on the service it delivers [LAP95], and subsumes the usual attributes of reliability, availability, safety, and security etc. These attributes have their own definitions. For example, software reliability is defined as the probability of *failure-free* software operation for a specified period of time in a specified environment, whilst software safety emphasizes the non-occurrence of *catastrophic* consequences on the environment of the software. As the role of software becomes more and more entrenched in everyday usage, software dependability has increasingly come to the foreground.

Although software faults affect the dependability of all types of software, they are of particular concern when developing safety-critical applications, where a single fault may result in a serious incident. Obvious examples of safety-critical software include aircraft flight systems and nuclear shutdown systems, but other examples also extend to more common applications, such as embedded systems within vehicles and domestic appliances, or indeed any system that controls significant amounts of power [STO96]. In all cases, the cost of failure is invariably high and there are numerous documented examples, many of which have resulted in the loss of human life [LAD99] and large scale economic risks. For example, the recall costs for faulty equipment in the consumer electronics field are extraordinarily high [HAT97].

Given the increased need for dependability, many software systems still have an unacceptably high level of faults. One argument put forward to explain the reasons for this lack of dependability is the lack of resources allocated to the design and development of software. Resources may be defined as the total means available to a company for increasing production or profit. It is suggested that given enough resources, software dependability can be greatly increased. This viewpoint has been called a delusion by some commentators, such as [HAT97], who argues that techniques that supposedly promote the goal of improved dependability have come and gone, whilst the defect density of software has remained similar for more than fifteen years. Even high-integrity systems which have had formal specification methods and extensive testing applied to them still have faults; the example cited in [HAT97] is of an air-traffic control system which, despite its thorough development, still had a defect density of 0.7 faults per thousand lines of code.

It is also the case that the development and maintenance overhead of software systems is increasing significantly. Pressman [PRE97] states that with the advent of object-oriented technologies and increased reuse of program components, the amount of system code that must be 'built from scratch' may decrease, but the overall size and complexity of systems continues to grow. It is therefore reasonable to assume that the incidence of faults within software systems will remain a problem for the foreseeable future, despite current advances in software engineering methodologies such as object-

oriented design – therefore alternative methods need to be investigated; one such method is that of fault tolerance [LYU95].

Fault-tolerant software allows faults to be detected and logged, without affecting the running of a system. According to [AVI85] the function of fault tolerance is: “*to preserve the delivery of expected services despite the presence of fault-caused errors within the system itself. Errors are detected and corrected, and permanent faults are located and removed while the system continues to deliver acceptable service.*” In this context, a system *failure* occurs when the delivered service deviates from what the system is aimed at (e.g. specification). An *error* is that part of the system internal state which is liable to lead to subsequent failure. A *fault* is the (hypothesized) cause of an error.

The two main approaches used by fault-tolerant systems are *Recovery Blocks* and *N-Version Programming*. Recovery blocks, discussed in detail in [RAN75][RAN95], work on the principle of *acceptance testing* and *diverse designs*, whereby a primary alternate is executed, and then an acceptance test is evaluated to provide adjudication on the outcome of this primary alternate. If the acceptance test fails, then the system reverts to its previous state, and the next alternate is executed, and so on; if the final alternate is executed and fails the acceptance test, then the system fails. However, it can be seen that there are some situations in which such a system is not appropriate – for example, in many safety-critical and real-time systems, it is simply not possible to “roll back” the system to its previous state. It is also the case that recovery-block systems are less predictable, as it is often impossible to know the duration of a given task, should one alternate fail and another be executed.

Multi-version programming, first proposed by [AVI77], works on the principle of independently implementing N versions of a program (channels), which are then executed *in parallel* with a single input in a multiprocessor or distributed processing environment (although conceptually, parallel execution is not necessary – channels may be executed separately and their results stored later for comparison). The outputs of the channels are compared by a voting system, which then forwards a single output based on the majority agreement of the channels. The major advantage of developing a multi-version system is that it can offer increased dependability over a functionally equivalent single-version system; the main disadvantage is that *such* systems require much greater resources than single-version systems, due to the increased cost required to develop and maintain N separate channels.

2 One Good Version or Multi-Version Software?

There is much debate over how much of an improvement in dependability multi-version design offers over single-version design. Some researchers have concluded that the dependability of software developed using the N -version method increases dramatically; for example, Hatton’s 1997 analysis [HAT97], based on the Knight and Leveson experiment [KNI86] concludes that a three-channel version

of the system, governed by majority polling would have a dependability improvement ratio of 45:1 over a single version of the system.

However, such massive increases in dependability have been drawn into question; Knight and Leveson [KNI90] argue that these gains in dependability are under the assumption that no correlated failures occur in $N/2$ or more channels of the system. (A correlated failure occurs when two or more channels fail on the same input case, and the failure may cause these channels to give the same but incorrect response.) Numerous studies, beginning with [SCO84] have shown that this is simply not the case. Eckhardt and Lee's study [ECK85] has shown that even small probabilities of correlated failures can reduce the overall dependability of a multi-version system dramatically, and Leveson [LEV95] further argues that every experiment with the approach of using separate teams to write versions of the software has found that independently written software routines do not fail in a statistically independent way. Examples of this can also be found in [ECK91][KEL88]. It therefore appears to be the case that such massive dependability gains can only be assumed on a theoretical level. In real-world applications, the overall cost/dependability ratio is likely to be much lower for a multi-version system than the theoretical model may suggest. The factor of cost therefore becomes important, as the extra expense required to develop N versions of a system may not result in an equivalent increase in system dependability.

The cost of developing multiple versions can be N times the cost of developing one version as well as N times the cost of maintenance. Although arguments have been advanced that the increase in cost will be less than N [VOU90], [LEV95] argues that these rest on the assumption that some aspects of the software development process will not have to be duplicated; also, many aspects of the processing and outputs have to be specified with more detail in order to make the results comparable, thus requiring that the specification phase take more time and effort than usual. [MAC91] argues that it can be the case that an imperfect voter 3-version system will be less cost-effective than a single-version system, although it would of equal dependability. This assumes that all versions have equal development costs, whilst [LAP90] calculates that the cost increase of developing a 3-version system over a single-version system is at least 178% and can be as much as 271%. On average, such a system would be 225% more expensive, although the 3-version system is more dependable.

When the resources allocated to building an N -version system are severely limited, it can be the case that the final system is less dependable than an equivalent single-version system. This is because the dependability of an N -version system is directly related to the dependability of its individual channels. The overall system dependability cannot be improved if the individual versions are not themselves sufficiently dependable. However, the real problem that has yet to be explored is, given a reasonably realistic amount of development resources for critical applications, which development approach (either

single-version or N -version) will be more effective. Hatton's analysis [HAT97] puts the question simply: "Is it more cost effective to develop one exceptionally good software channel or N less good ones, and which is likely to lead to the more reliable system?"

3 A Safety-Critical Application and Experiment Settings

In order to investigate both development methods and determine which one produces a more dependable system for a given amount of resources, we need an appropriate application example. The factory production cell simulation [LOT96] was chosen as it is necessary to implement an application that is both small enough to construct within the resource budget allocated, and also complex enough to have the potential for faults to be present within the system code. It is also desirable for the application to be a real-time system, as such systems invariably involve high reliability and safety requirements. As [KEL91] states: "while complete fault removal in sequential software is difficult, it becomes effectively impossible in real-time, concurrent software. Current testing methodologies, while useful to detect and localize errors in sequential software, are inadequate when applied to highly concurrent and asynchronous real-time software."

The production-cell, detailed in Figure 1, consists of two conveyor belts, one of which delivers the raw units (blanks) into the system, and one of which moves the blanks out of the system once they have been fully processed. The unit also consists of four separate workstations, each of which has its own number. Depending on the type of a workstation, it can either be switched on and off by the controller software, or is permanently on. Two cranes are mounted on a racking which prevents them from both occupying the same horizontal position at the same time, and are used to transport blanks around the system. Each blank has its own bar-code, which will identify which workstations it needs to be placed in, and the minimum and maximum amounts of time that it could spend within each workstation. Blanks may be processed either in a specific order, or in any order, depending on the instructions in the bar-code. The production-cell simulation was implemented in Java, and could be run on a number of different operating systems.

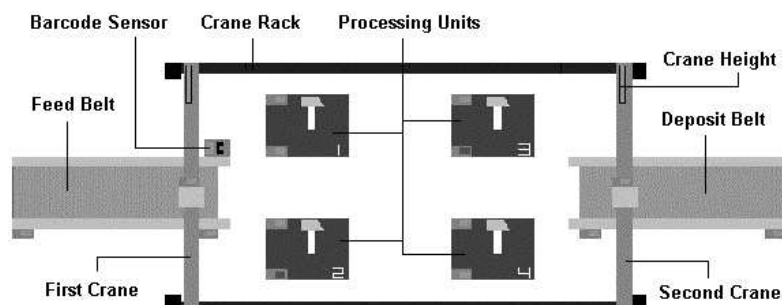


Figure 1 The flexible production cell [LOT96]

A software controller is needed to control the operations of the cell. It needs to allow the production-cell simulation to process up to two blanks at any one time, whilst ensuring that the blanks are processed correctly within the appropriate time constraints. It is also necessary to ensure that the system remains safe; for example, it is imperative to ensure that the two cranes never collide with each other, and that no blank is placed in a workstation that already contains a blank. Also, the feed belt needs to be controlled by the software in order to ensure that no more than two blanks enter the system at any given time, and that none falls off the end of the belt.

Three programmers were required to develop the control software, and implement an N -version channel and a single-version system each. The controller software could be implemented in any language, on any platform, and development was to be separated into three equal time phases. The first time phase was used by each programmer to develop a working version of the controller software; at the end of this phase, the three programs were used as channels in the N -version system. The second and third time phases were then used by each programmer to refine their respective versions and to exploit various methods and tools for improving the dependability of those versions. There is no differentiation between these two phases – they are merely noted in order to demonstrate that the additional time spent developing the N -version systems was twice that of the original development time. These additional time phases resulted in each of the single-version systems having had the same amount of resources spent on them as the 3-version voter system. In the real world, this would encompass a number of different elements, such as system cost and employee wages. For this experiment, it was decided that the resource measured would be time; in other words, the accumulative resources (time) allocated to the development of the N -version system would be the same as that allocated to each single version. At the conclusion of the third time phase, an extensive testing plan was implemented in order to ascertain the dependability of the single-version systems and the 3-version system, and an analysis was then performed to determine which of the methods had resulted in the most dependable software.

As many experiments have found that correlated faults can drastically reduce the overall dependability of an N -version system (e.g. [KNI86][ECK91]), a conscious decision was made to make the development of each channel as diverse as possible; these channels could be developed using different languages such as C++ and Java, and each system was implemented under a different operating system. As the controller software does not require any graphical output, none of the platforms used for the development of the controller software were necessarily expected to produce graphical output; however the platform used by the production-cell simulation requires the graphical environment capable of running Java applications. The simulation and the controller software communicate via a first-in-first-out pipe mechanism, with communications being sent as ASCII text.

4 Design and Implementation

In order to specify the expected behaviour of an N -version system, it is essential that a single requirements document be developed, so as to ensure consistency between channels; failure to do this can result in discrepancies between channels that cause the voter system to fail [KNI91]. Whilst consistency is essential, it is important not to over-specify such a system, as this can compromise the diversity of each channel [AVI89].

Unlike many other experiments of its type, this study does not make use of a pre-written requirements document; rather, the document's production is considered to be part of the development process for which resources are allocated, and was produced jointly by the three programmers, following several design meetings. Apart from this, there was no further interaction between programmers.

4.1 Development

The process of building the N -version system took place entirely within the first time phase, over a period of five weeks. In order to encourage diversity, each channel was developed using different development environments; these are shown in Table 1. As with the Knight and Leveson experiment [KNI86], no single software engineering methodology was imposed on the programmers - each was allowed to develop their program using whatever method they saw fit.

Channel	Operating System	Language
A	Sun Solaris	GNU C++
B	Red Hat Linux	GNU C++
C	Microsoft Windows NT	Java

Table 1 Operating system/language combinations for each channel

All three channels were implemented using object-oriented techniques; in the case of the Java channel, this was unavoidable, whilst both C++ channels took advantage of class structures but are probably best described as semi-object-oriented. One of the most noticeable differences between the two languages is the ability of Java to support multi-threading – something not supported in ANSI C++. This leads to channel *C* employing a multi-threaded algorithm to control the system, whilst the channels written in C++ employ single-threaded algorithms. It was hoped that this diversity would further reduce the probability of correlated faults or common-mode failures between the channels.

At the end of development, the difficult decision was made to forsake the implementation of a voter system, due to the inadequate amount of time remaining for the programmers. However, like many existing experiments of investigating N -version systems, the presence of a voter system is not essential for our experiment. We are primarily interested in the relative dependability of each N -version channel compared to its single version cousin.

The three channels completed in the first time phase were stored for inclusion in the 3-version system. Each programmer was required to use the additional two time phases of development time to further refine their system, including the possible use of formal methods, in order to make them as dependable as possible.

4.2 Testing

There are only a finite number of possible situations in which the systems can be in (given that a maximum of two blanks can be processed by the systems at any one time) and these are specified as scenarios within the requirements document. The set of tests is therefore formulated based upon these situations.

The set of tests is divided into four scenarios. The first scenario tests each system's ability to process a single blank through the production cell, without a second blank being introduced into the system. The second scenario is designed to assess the ability of each system to handle situations in which both blanks finish and require transportation to the deposit belt at the same time. The third scenario is designed to test the ability of each system to handle situations in which there are two blanks in the system, both of which require transportation to other workstations at the same time. The fourth scenario tests each system's ability to handle situations in which two blanks are in the system – one of which requires moving to the deposit belt, and the other of which requires moving to another workstation within the system. In each of these scenarios, all possible configurations of blank are tested. The entire set of tests is repeated with each possible combination of blank ordering (e.g. preserved order and non-preserved order), in order to rigorously ensure that each system is tested fully. The results of each test are either marked as "Pass" or given a failure number. A channel achieves a "Pass" whenever it processes a blank successfully through the system; this is judged to be whenever a blank is correctly processed through all specified workstations – in the correct order where necessary – and placed on the deposit belt. Whenever a channel fails to meet the criteria, it is adjudged to have failed. The nature of the failure is recorded along with a failure number, in order to ensure that should two channels fail with the same input, then the nature of the failures can be analyzed for a possible common-mode failure.

For the sake of brevity, the full set of tests performed on each system is not included in this paper; however, in order to demonstrate how the testing process is performed, the following tables show the tests, results and fault descriptions generated by the single blank tests. Table 2 details the configuration of each blank that is passed through the system; *P* stands for "Preserved Order" and *NP* stands for "Non-preserved order". Failure numbers that are shaded represent situations in which a channel processes a blank through the system in an incorrect way, but does not fail. Failures such as this are important as they can be hard to detect without constant supervision of the system. Details of the failures are listed in Table 3.

Blank Configuration	Channel A		Channel B		Channel C	
	P	NP	P	NP	P	NP
1-2-3-4	Pass	Pass	Pass	Pass	Pass	19
1-2-4-3	Pass	1	Pass	Pass	Pass	19
1-3-2-4	Pass	Pass	Pass	Pass	Pass	19
1-3-4-2	Pass	2	Pass	Pass	Pass	19
1-4-2-3	Pass	3	Pass	Pass	Pass	19
1-4-3-2	Pass	4	Pass	Pass	Pass	19
2-1-3-4	Pass	Pass	Pass	Pass	Pass	19
2-1-4-3	Pass	5	Pass	Pass	Pass	19
2-3-1-4	Pass	Pass	Pass	Pass	Pass	19
2-3-4-1	Pass	6	Pass	Pass	Pass	19
2-4-1-3	Pass	7	Pass	Pass	Pass	19
2-4-3-1	Pass	8	Pass	Pass	Pass	19
3-1-2-4	Pass	Pass	Pass	Pass	Pass	19
3-1-4-2	Pass	9	Pass	Pass	Pass	19
3-2-1-4	Pass	Pass	Pass	Pass	Pass	19
3-2-4-1	Pass	10	Pass	Pass	Pass	19
3-4-1-2	Pass	11	Pass	Pass	Pass	19
3-4-2-1	Pass	12	Pass	Pass	Pass	19
4-1-2-3	Pass	13	Pass	Pass	Pass	19
4-1-3-2	Pass	14	Pass	Pass	Pass	19
4-2-3-1	Pass	15	Pass	Pass	Pass	19
4-2-1-3	Pass	16	Pass	Pass	Pass	19
4-3-1-2	Pass	17	Pass	Pass	Pass	19
4-3-2-1	Pass	18	Pass	Pass	Pass	19

Table 2 Results from single blank tests with the three *N*-version channels

#	Comment
1	Follows path 1-2-3 and then reports workstation 2 time limit exceeded.
2	Follows path 1-3-4-2 and then reports workstation 1 time limit exceeded.
3	Follows path 1-2 and then reports workstation 1 time limit exceeded.
4	Blank is processed through the system, but has a path of 1-2-3-2
5	Follows path 2-1-3 and then reports that workstation 2 time limit exceeded.
6	Follows path 2-3-1 then reports that workstation 0 time limit is exceeded.
7	Follows path 2-1 and then reports workstation 0 time limit exceeded.
8	Processed through the system, but has a path of 2-1-3-1
9	Processed through the system, but has a path of 3-1-1-2
10	Follows path 3-2-1 and then reports workstation 0 time limit exceeded.
11	Processed through the system, but follows path 3-2-1-2
12	Processed through the system, but follows path 3-1-2-1
13	Moves to workstation 1 and then reports workstation 0 time limit exceeded.
14	Moves to workstation 1 and then reports workstation 0 time limit exceeded.
15	Processed through the system, but follows path 1-2-3-1
16	Processed through the system, but follows path 1-2-1-3
17	Processed through the system, but follows path 1-3-1-2
18	Processed through the system, but follows path 1-3-2-1
19	Falls into infinite loop before the first blank is picked up by crane 1

Table 3 Faults occurring during the single blank *N*-version tests

The test results of each channel developed in the context of the N-version system and its corresponding single version solution are detailed in Table 4. ($P \times P$ refers to tests involving two blanks of preserved order, $NP \times NP$ to tests involving two blanks of non-preserved order, $P \times NP$ to tests where the first blank is set to preserved order and the second blank is set to non-preserved order, and $NP \times P$ to tests where the first blank is set to non-preserved order and the second blank is set to preserved order.)

	Test Description	P x P	NP x NP	P x NP	NP x P	# of Faults	# of Tests	Fault Probability
N-Version Channel A	Single blank	0	18	N/A	N/A	18	48	37.5%
	Scenario 1	0	0	0	0	0	24	0.00%
	Scenario 2	11	26	29	21	87	224	38.83%
	Scenario 3	0	12	12	0	24	144	34.56%
	Total					129	440	25.22%
N-Version Channel B	Single blank	0	0	N/A	N/A	0	48	0.00%
	Scenario 1	1	0	0	1	2	24	8.33%
	Scenario 2	0	0	1	2	3	224	1.33%
	Scenario 3	0	2	0	0	2	144	1.38%
	Total					7	440	1.59%
N-Version Channel C	Single blank	0	24	N/A	N/A	24	48	50.00%
	Scenario 1	0	6	6	6	18	24	75.00%
	Scenario 2	26	56	56	56	194	224	86.60%
	Scenario 3	0	36	36	36	108	144	75.00%
	Total					344	440	78.18%
Single Version Channel A	Single blank	0	0	N/A	N/A	0	48	0.00%
	Scenario 1	0	0	0	0	0	24	0.00%
	Scenario 2	7	5	5	4	21	224	9.37%
	Scenario 3	1	0	0	0	1	144	0.69%
	Total					22	440	5.00%
Single Version Channel B	Single blank	2	0	N/A	N/A	2	48	4.16%
	Scenario 1	0	0	0	0	0	24	0.00%
	Scenario 2	2	0	0	0	2	224	0.89%
	Scenario 3	0	0	0	0	0	144	0.00%
	Total					4	440	0.91%
Single Version Channel C	Single blank	0	0	N/A	N/A	0	48	0.00%
	Scenario 1	0	0	0	0	0	24	0.00%
	Scenario 2	11	12	8	7	38	224	16.96%
	Scenario 3	8	9	8	8	33	144	22.91%
	Total					71	440	16.14%

Table 4 Results of testing all the software versions

5 Analysis

When viewing the three channels as an N -version system, the actual number of faults within those channels is less important than would otherwise be the case in a single-version system; what is of most importance is whether any of these faults lead to potential common-mode failures, which cause a voting system to forward incorrect data. It is therefore necessary to analyze all reported faults in order to see whether or not common-mode failures exist. This analysis is carried out by investigating each group of tests performed on the channels, and isolating all test inputs that cause more than one channel to fail.

Of most concern is the possibility of a common-mode failure that results in the N -version system processing a blank incorrectly; failures that result in the voter being unable to reach a consensus opinion and failures that cause the voter to perform an operation that results in the failure of the system

as a whole are slightly less serious, as in either scenario, human operators would be alerted that the system was in a fail state. Malformed blanks however, will not necessarily be noticed, and such failures may remain undetected.

The *N*-Version Systems

The initial analysis of the three *N*-version channels indicates that channel *A* and channel *C* suffer from unacceptably high rates of failure, with only channel *B* dependable enough to pass the acceptance test that is necessary before any channel can be entered into a dependable *N*-version system. This is shown in Table 5.

Channel	Probability of Failure
A	25.22%
B	1.59%
C	78.18%

Table 5 The probability of failure among the *N*-version channels

As can be seen, the vast majority of failures are caused by channel *C*; this is largely due to the channel's inability to process blanks of non-preserved order, which automatically causes the channel to fail in 75% of all test inputs.

Although the distribution of failures amongst the three channels is quite diverse, it is the case that specific groups of tests lead to more failures than others, with tests involving non-preserved order blanks causing the majority of anomalies; tests using two preserved order blanks cause only 8% of all reported failures, whilst tests using two non-preserved order blanks cause 38% of all reported failures. Tests involving blanks of mixed ordering lie between the two extremes. Although the majority of non-preserved order failures are caused by channel *C*, this type of blank also results in the majority of failures discovered in channel *A*, with only 9% of the channel's failures occurring when both blanks are set to preserved order. The likely reason for tests involving blanks of non-preserved order producing more faults than similar tests involving blanks of preserved order is that the algorithm required to process non-preserved order blanks is more complex, and so there is greater scope for failure within program code. However, it is important to note that these results must not be taken at face value; as channel *C* produces a large number of failures, the relatively few failures found in channel *A* and Channel *B* make very little difference to the overall pattern of results.

Altogether, the 1320 tests performed on the three channels reveal a total of 480 faults. This means that for any given test on a single channel, there is a 36.36% probability of it resulting in a failure. However, at the conclusion of testing, only two common-mode failures have been discovered. This implies that even though two of the channels are unacceptably undependable, the overall *N*-version system will only unknowingly forward an incorrect decision in 0.45% of tested possible situations.

Figure 3 shows the overall pattern of faults found during testing. The large number of single faults can be mainly attributed to the inability of channel *C* to handle blanks of non-preserved order. Although the diagram appears to show that tests that cause more than one channel to fail tend to occur in groups, it must be remembered that as channel *C* fails in over 78% of tests and channel *B* fails in less than 1.6% of tests, these groups of failures are mostly due to faults occurring in channel *A*, rather than general faults that occur between channels. It must be also remembered that tests which cause two channels to fail are not necessarily common-mode failures; overall, only two common-mode failures have been found during testing. Unrelated failures occurring within two or three of the channels could be identified by a result-based voting mechanism, and human operators could be alerted.

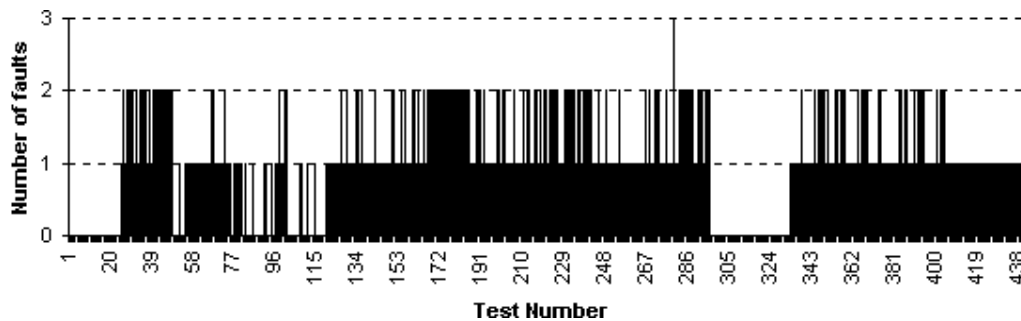


Figure 3 The results of all tests performed on the *N*-version channels

Of all 440 different situations tested, only 89 result in none of the three channels failing, with 218 resulting in one channel failing. Altogether 132 of the tested situations result in 2 of the channels failing, and only 1 situation leads to all three channels failing; this is illustrated in Figure 4 below. The large number of single faults can be explained by the poor performance of channel *C*, but despite this, the *N*-version system will detect and tolerate 214 faults. This gives the overall *N*-version system a failure probability of 30.22% for any input; this is more than 6% lower than the average possibility of failure between the three channels. Even with this large failure rate, the *N*-version voter system will detect 99.55% of faults produced by the three channels; as only two common-mode failures were detected, only 0.45% of tests will result in the *N*-version voter system unknowingly producing an incorrect output.

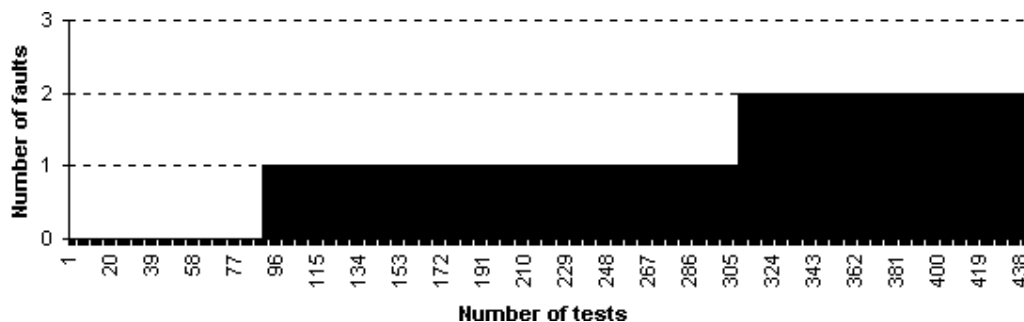


Figure 4 The distribution of faults in the *N*-version channels

The enforced diversity of operating system and platform resulted in diverse code; for example, the channel written in Java uses a multi-threaded approach to performing tasks, whilst the two C++ channels employ single-threaded algorithms. However, as the two common-mode failures suggest, the diversity of code does not necessarily lead to diverse distributions of faults. It is interesting to note that both common-mode failures are as a result of entirely unrelated faults within two of the channels; these faults result in the simulation moving into the same unrecoverable state, but one channel causes the system to fall into an infinite loop, whilst the other channel simply ceases to execute.

The Single-Version Systems

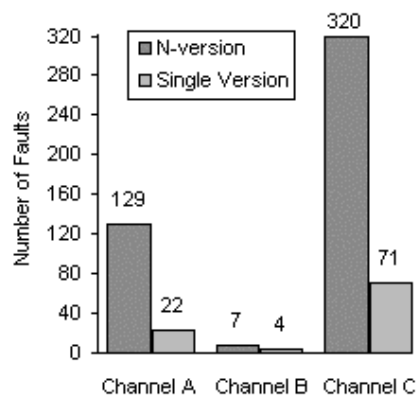


Figure 5. The number of faults found during testing for each system

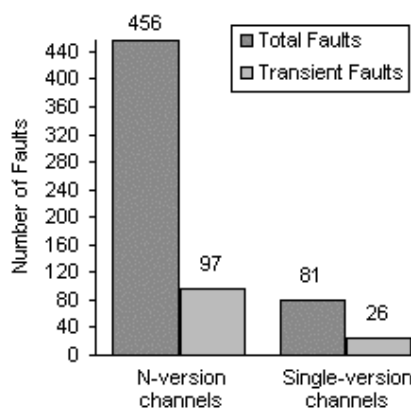


Figure 6. The number of faults found during testing, and the number of them transient

All three single version systems exhibit increased levels of dependability over their corresponding *N*-version systems, with the number of faults found in each system dropping significantly, as shown in figure 5 opposite.

Whilst the failure rates of channel *A* and channel *C* are still unacceptably high, both systems exhibit vastly improved dependability, with failure rates of 5.00% and 16.14% respectively, compared to the failure rates of 25.22% and 78.18% experienced with the *N*-version systems on which they are based. Channel *B* also exhibited increased dependability, with its failure rate reduced to 0.91%, although all of the faults found in this channel were transient and so repeated testing may produce different statistics. Interestingly, the proportion of transient faults discovered while testing the systems remains much the same, as is shown in Figure 6, and so whilst 55 repeatable faults were discovered among the single versions, the remaining faults are caused by timing anomalies with the Java simulation, and are therefore difficult to replicate with certainty.

One of the greatest areas of interest is that of how many undetected failures occurred in the single-version systems. There are two possible kinds of undetected failure; a blank can be left within the system instead of being extracted, or a blank can be processed through the system incorrectly. The occurrences of these are detailed in Table 6 below. As can be seen, *all three* single-version systems have an undetected failure rate that is greater than the 0.45% rate recorded in the *N*-version system.

The occurrences of these are detailed in Table 6 below. As can be seen, *all three* single-version systems have an undetected failure rate that is greater than the 0.45% rate recorded in the *N*-version system.

	Channel A	Channel B	Channel C
Blank in System	3	0	3
Blank incorrectly processed	3	4	20
Total Undetected failures	6	4	23
Undetected failure rate	1.36%	0.90%	5.23%

Table 6 Undetected failures in the single-version systems

6 Evaluation

The goal of this experiment is to focus on the cost side of implementing N -version and single-version systems, rather than to produce ultra-dependable systems. It can be seen that in this experiment, two of the three channels developed for the N -version system have very poor dependability. However, despite the poor quality of components, the N -version system will still theoretically perform more dependably than either of the two poor quality channels, and more importantly, will only unknowingly forward an incorrect decision in 0.45% of test inputs. Therefore, whilst the N -version system fails more often than the single-version systems derived from channel A and channel C , the number of undetected failures is still lower than either of these two systems, and so the N -version system has a higher level of *safety* than either single-version system mentioned above, although the single-version system B is more dependable than the N -version system in the value domain.

It must be stressed that when calculating the dependability of the N -version system, we assume that all of the channels that pass a test perform the same operations. Unfortunately, one of the criticisms of N -version design is that of the *consistent comparison problem*. As [KNI91] explains, “*There is no guarantee that N -version systems will make the same decision and ultimately make outputs that - although correct - are not suitable for the decision algorithm*”. Unlike other studies, no pre-written requirements document was provided; it was left to the programmers to collectively develop their own documentation as part of the development process for creating the N -version system. This had profound consequences later on in the development process, as documentation was rushed and at times ambiguous. The three channels often successfully pass a test, but each channel goes about the task in a different way; for example, whilst channel A moves the two cranes simultaneously, channel C only moves the second crane after the first crane has reached its destination. The factory system being controlled requires discrete instructions to be executed, and so there is no scope for inexact matching like that performed at UCLA/H [AVI88b]. When this problem is taken into account, the actual dependability of the N -version system would likely be worse than its theoretical level if an appropriate voter system were developed.

Another potential problem with the system is its high granularity. In the specification, it was decided that voting would take place before each crane movement, and so voting has to take place almost constantly, which increases the scope for faults due to the consistent comparison problem. When viewed from the point of view of attempting to produce ultra-dependable software, this leads the experiment to come under

the same criticisms as those made of the [KNI86] experiment by [AVI88a], namely that the lack of rigor in the design paradigm resulted in many of the later weaknesses with the N -version system.

The single-version systems all perform much better than their N -version channel counterparts, with much higher levels of dependability recorded. However, system A and system C still possess unacceptably high levels of faults, due largely to mistakes in their overall design. The maintainability of the single-version systems suffers, as each system had been essentially completed by the end of the N -version development period, and all further improvements and modifications had to be made without affecting the overall working of the system algorithms. Modifications tended to consist of blocks of code inserted into the system to handle specific situations, rather than any changes to the systems overall design. When larger changes were attempted, other parts of the system often ceased to function correctly, and so the original system design remained largely the same, in spite of faults. Further maintenance of the single-version systems is likely to further complicate matters, and in several areas, complete re-writes of the system code will be required in order to ensure that the system works in a predictable way.

7 Conclusions

It appears to be the case that *for this experiment*, the traditional single-version approach produces more dependable systems in terms of overall numbers of faults and failures discovered, with the level of granularity and an ambiguous requirements document resulting in a poor quality N -version system. However, the N -version system results in fewer *undetected failures* occurring than in a single-version system, despite the poor quality of its individual channels. Therefore, although the single-version systems produce fewer faults, the N -version system can still be seen as being potentially the *safer* of the two approaches. It is evident that regarding the single-version method as a “seem-to-be” safer design decision for safety-critical applications, and for critical applications in general, is not generally justifiable for a given amount of resources.

It is also the case that should the amount of development time allocated be longer, then the dependability of the N -version system is likely to be much higher, but there is no guarantee that the quality of the single-version systems will increase significantly likewise. It is not the case that N -version systems will always suffer from the high rates of failure experienced in this study, as the dependability of the N -version system rises greatly as the quality of its individual channels increases. Furthermore, as new technologies emerge, it may be the case that high quality channels can be created at quite low cost. For example, software reuse libraries or COTS components can be employed to create several channels that are functionally identical, but constructed with different versions and combinations of components. This shows much promise for the relatively cheap, fast creation of different channels within an N -version system; however, at present, although such software libraries exist, their price has yet to reach an acceptable level and the number of components available is still quite limited.

No previous experiment has ever made a comparison of single-version and N -version systems when both have been given fixed resources; this experiment is therefore the first of its type, and so there are naturally many weaknesses that need to be resolved in follow-up studies. The size of the software developed is unrealistic compared to real-world applications, but is certainly greater than that of [KNI86] and [AVI88b]. To develop more realistic software systems would be extremely difficult without commercial backing, and would be unjustified given that no previous experimental data of its type is available. The lack of a pre-written requirements documentation is another potential issue; although assigning the task of writing the requirements documentation in part of the development time is perhaps more realistic, the resulting documentation contained several faults and ambiguities. This is in contrast to other studies, which have supplied their programmers with pre-written requirements documentation, and so potentially leads to increased scope for mistakes within each channel. Related to this is the issue of using the second and third time phases to further test and enhance each individual system; should no N -version channel have been required, then each programmer would have had much more time to develop a rigorous and thorough specification, and the resulting single-version systems may have been more dependable. This issue needs to be addressed, and when the experiment is followed up, single-version and N -version systems will be developed separately. Finally, as with the [KNI86] experiment, no specific software engineering methods were imposed on the programmers; all the programmers involved were undergraduates taking a senior-level, advanced software engineering course, and the method of software development was left for them to decide. This leads to the same criticism that [AVI89] and [JOS88] made of the Knight and Leveson experiment, although in fact all three channels were implemented using object-oriented techniques. Nevertheless, it is possible that the programmers' lack of experience in developing N -version systems further exacerbates the differences in operation between the three channels, and so in order to guard against accusations of a causal programming process, it may be advisable to enforce more rigorous methods, such as the NVP process developed at UCLA.

Acknowledgments

This work was supported partially by the EPSRC Flexx project and has benefited from discussions with the members of the Durham *Distributed Systems Engineering* group. The authors would like to thank John Cullen and Alastair Davies for their contributions to the experiment.

References

- [AVI77] A. Avizienis and L. Chen, "On the implementation of N -version programming for software fault-tolerance during execution," in *Intl. Conf. Comput. Soft. & Appl.*, New York, pp.149-155, 1977.
- [AVI85] A. Avizienis, "The N -version approach to fault-tolerant software" *IEEE Trans. Soft. Eng.*, vol. SE-11, no. 12, pp.1491-1501, 1985.
- [AVI88a] A. Avizienis and M.R. Lyu, "On the effectiveness of multi-version software in digital avionics," in *AIAA/IEEE 8th Digital Avionics Systems Conference*, San Jose, pp. 422-427, Oct. 1988.

- [AVI88b] A. Avizienis, M.R. Lyu and W. Schuetz, "In search of effective diversity: a six-language study of fault-tolerant flight control software," in *18th Int. Symp. Fault-Tolerant Comput.*, pp.15-22, Tokyo, 1988.
- [AVI89] A. Avizienis, "Software fault tolerance," in *IFIP XI World Computer Congress '89*, San Fransisco, Aug. 1989.
- [ECK85] D.E. Eckhardt and L.D. Lee, "A theoretical basis for the analysis of multi-version software subject to coincident errors," *IEEE Trans. Soft. Eng.*, vol. SE-11, no. 12, pp.1511-1517, 1985.
- [ECK91] D.E. Eckhardt, A.K. Caglayan, J.C. Knight, L.D. Lee, D.F. McAllister, M.A. Vouk, and J.P.J. Kelly, "An experimental evaluation of software redundancy as a strategy for improving reliability," *IEEE Trans. Soft. Eng.*, vol. SE-17, no. 7, pp.692-701, 1991.
- [HAT97] L. Hatton, "N-version design versus one good version," *IEEE Software*, vol. 14, no. 6, pp.71-76, 1997.
- [JOS88] M.K. Joseph, *Architectural Issues in Fault-Tolerant Secure Computing Systems*. Ph.D. Dissertation, Dept. of Computer Science, UCLA, 1988.
- [KEL88] J.P.J. Kelly, D.E. Eckhardt, A.K. Caglayan, J.C. Knight, D.F. McAllister, and M.A. Vouk, "A large scale second generation experiment in multi-version software: description and early results," in *18th Intl. Fault-Tolerant Comput.*, pp.9-14, 1988.
- [KEL91] J.P.J. Kelly, T.I. McVittie and W.I. Yamamoto, "Implementing design diversity to achieve fault tolerance" *IEEE Software*, pp.61-71, 1991.
- [KNI86] J.C. Knight and N.G. Leveson, "An experimental evaluation of the assumption of independence in multi-version programming," *IEEE Trans. Soft. Eng.*, vol. SE-12, no. 1, pp.96-109, 1986.
- [KNI90] J.C. Knight and N.G. Leveson, "A reply to the criticisms of the Knight and Leveson experiment" *ACM Software Engineering Notes*, Jan. 1990.
- [KNI91] J.C. Knight and P.E. Amman, "Design fault tolerance," in *Software Reliability and Safety*, (eds. B. Littlewood and D. Miller), Elsevier Applied Science, pp. 25-49, 1991.
- [LAD99] P. Ladkin et al, "Computer-related incidents with commercial aircraft," <http://www.rvs.uni-bielefeld.de/publications/Incidents/>, 1999.
- [LAP90] J.C. Laprie, J. Arlat, C. Beounes, and K. Kanoun, "Definition and analysis of hardware- and software-fault-tolerant architectures," *IEEE Comput.*, vol. 23, no. 7, pp.39-51, 1990.
- [LAP95] J.-C. Laprie, "Dependability - its Attributes, impairments and means" in *Predictably Dependable Computing Systems*, (eds. B. Randell etc.), Springer-Verlag, pp.3-24, 1995.
- [LEV95] N.G. Leveson, *Safeware: system safety and computers*. Addison-Wesley-Longman, New York, 1995.
- [LOT96] A. Lötzbeyer and R. Mühlfeld, "Task description of a Flexible Production Cell with real time properties," Internal FZI Tech. Report, Karlsruhe, ftp://ftp.fzi.de/pub/PROST/projects/korsys/task_descr_flex_2_2.ps.gz, 1996.
- [LYU95] M.R. Lyu, *Software Fault Tolerance*. John Wiley & Sons, 1995.
- [MAC91] D.F. McAllister and R.K. Scott, "Cost modelling of fault-tolerant software," *Journal of Information and Software Technology*, vol. 33, no. 8, pp.594-603, Oct. 1991.
- [PRES97] R.S. Pressman, *Software Engineering: a practitioners approach*, 4th edition, Addison-Wesley, 1997.
- [RAN75] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Soft. Eng.*, vol. SE-1, no. 2, pp.220-232, 1975.
- [RAN95] B. Randell and J.Xu, "The evolution of the recovery block concept," in *Software Fault Tolerance*, (ed. M.R. Lyu), John Wiley & Sons, pp.1-22, 1995.

[SCO84] R.K. Scott, J.W. Gault, D.F. McAllister, and J. Wiggs “Experimental validation of six fault-tolerant software reliability models” *IEEE Fault Tolerant Computer Systems*, vol. 14, pp.102-107, 1984.

[STO96] N. Storey, *Safety Critical Computer Systems*, Addison-Wesley-Longman, 1996.

[VOU90] M.A. Vouk, “Back-to-back testing” *Information and Software Technology*, vol. 32 , no. 1, pp.34-45, 1990.