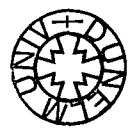
Assessing Multi-Version Systems Through Fault Injection

Paul Michael Townend

M.Sc. Thesis

Research Institute in Software Engineering,
Department of Computer Science,
University of Durham
UK

2001



ABSTRACT

Multi-version design (MVD) has been proposed as a method for increasing the dependability of critical systems beyond current levels. However, a major obstacle to large-scale commercial usage of this approach is the lack of quantitative characterizations available. Fault injection is used to help seek an answer this problem. Fault injection is a phrase covering a variety of testing techniques that can be applied to both hardware and software, all of which involve the *deliberate insertion of faults into an operational system to determine its response*. This approach has the potential for yielding highly useful metrics with regard to MVD systems, as well as giving developers a greater insight into the behaviour of each channel within the system. In this research, an automatic fault injection system for multi-version systems called FITMVS is developed. A multi-version system is then tested using this system, and the results analysed.

It is concluded that this approach can yield several extremely useful metrics, such as metrics related to channel sensitivity, channel sensitivity to common-mode error, program scope sensitivity, program scope sensitivity to common-mode error, error frequency distribution and common-mode error frequency distribution. In addition to this, the analysis of the multi-version system tested indicates that the system has an extremely low probability of experiencing common-mode error, although several key points in channel code are identified as having higher sensitivity to faults than others.

Copyright

Declaration

No part of the material offered has previously been submitted by the author for a degree in the University of Durham or in any other University. All the work presented here is the sole work of the author and no one else.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1 Introduction

1.1 Introduction

An increasing range of industries has a growing dependence on software-based systems. Many of these systems are critical systems developed for safety-critical, business-critical or mission-critical applications, and it can be seen that failure within such systems has the potential to be devastating.

Given the need for dependability, many software systems still have an unacceptably high level of faults. *Multi-version design* (MVD) has been proposed as a method for increasing the overall dependability of software systems above that of those developed using traditional approaches. However, a major obstacle to the large-scale commercial rollout of MVD systems is the lack of quantitative characterizations of the approach. These are difficult to assess, but important, as in most cases resource allocation cannot be done arbitrarily or carelessly [KIM00], and without relevant metrics, sensible resource allocations cannot be achieved.

It can therefore be seen that a concerted effort needs to be made to improve the level of empirical knowledge in regard to multi-version systems. This has been done to limited effect with traditional testing methods, but the area of fault-injection has been especially neglected [VOA97, CHE99].

Fault injection as an analysis tool has a number of benefits; for example, it can effectively simulate rare events that may not have been considered during a target system's testing phase, and is also a very good method for deriving metrics about a system. Currently however, most fault injection systems within the software engineering field have concentrated on the assessment of single version software, with little or no analysis tools for the detection of common-mode failures in multi-channel systems. [CHE99] states that "*as far as fault injection for diversity evaluation is concerned, the lessons from the literature are limited and of a general nature only.*"

1.2 Objectives

This research is centred around the design and development of an automated fault-injection system for the analysis of multi-version systems, in order to provide a method for

easily extracting useful metrics from such a system, as well as facilitating the testing process for MVD systems by identifying areas of code with a high sensitivity to common-mode failure. A fault-injection system is developed capable of parsing C and C++ source code, injecting faults, compiling the resulting code, automatically testing the code using user-specified tests, and logging the results. In addition, an existing factory simulation is re-written in C++ in order to allow the testing of an existing MVD factory control system to be performed much faster. The results outputted by the fault injector are then analysed in order to gauge the sensitivity of individual MVD channels to errors as well as their sensitivity to common-mode failure. This research also results in good non-commercial fault-injection being made available for future studies.

1.3 Organization of the Remainder of Dissertation

This chapter (**Chapter 1**) introduces an overview of the research area of this project, and details the structure of the rest of the document.

Chapter 2 introduces the basic definitions used throughout the thesis and gives a detailed definition of the concept of dependability. The traditional software engineering approach to developing software, software fault tolerant techniques such as recovery blocks and multi-version design, the controversy over multi-version design, and the cost factors of MVD systems are also discussed. The chapter concludes by discussing the need for more fault-tolerant metrics

Chapter 3 details the problems associated with traditional testing techniques, the background to fault injection, and the differences between fault injection and traditional testing techniques. A method for applying fault injection to MVD systems is also discussed.

Chapter 4 introduces the tool to be developed for this research. It goes on to detail the design and operation of the tool, its objectives, its limitations, and portability issues associated with it. The chapter ends with a detailed description of the development of the tool and the make-up of the test files used by it.

Chapter 5 describes in detail the factory production cell simulation used to test the effectiveness of the fault injection tool developed. The system requirements, operational details, and assumptions made by the production cell simulation are also discussed.

Chapter 6 gives an overview of the experiment performed using the fault injection tool. It details the re-development of the production cell simulation in C++, and describes the test data used during the experiment. The chapter concludes by describing the extra hardware used to combat the large amount of processing time required for each test.

Chapter 7 details the results of the experiment performed, together with an analysis as to what these results mean. The chapter concludes by examining issues that arose from the fault injection tool as a result of the experiment.

Chapter 8 gives the conclusions of the thesis, describes potential future work and research directions, and contains acknowledgements.

Chapter 2 The Need for Dependable Software

2.1 Basic Definitions

Before beginning a detailed discussion, it is first necessary to define a number of basic concepts that are related to the areas of dependability, fault tolerance and fault injection. These will be used throughout the whole thesis.

2.1.1 Software systems

A *system* may be viewed as a set of components interacting under the control of a design (which is itself a component of the system) [LEE90]. Components are themselves systems, and receive requests for service and produce responses; when a component cannot satisfy a request for service, it will produce an exception. This system model is recursive in that each component can itself be considered as a system in its own right and thus may have an internal design which can identify further sub-components.

2.1.2 Errors

An *error* can be defined as a discrepancy between a computed, observed, or measured value or condition and the true, specified or theoretically correct value or condition. Errors occur at run-time, when some part of the computer software enters an undesired state. They are therefore a property of the state of the system, and cannot be observed easily (unless special mechanisms are employed to record the occurrence of some types of events.)

2.1.3 Failure

A *failure* occurs when an error passes through the system-user interface and affects the service delivered by the system. A component failure results in a fault (1) for the system which contains the component and (2) as viewed by the other components with which it interacts; the failure modes of the failed component then become fault types for the components interacting with it.

2.1.4 Faults

A *fault* (also referred to as a *bug*) is a defect that has the potential of generating errors. It is a static notion, and the presence of a fault may lead to system failure.

In most cases, the fault can be located and removed; in some cases it remains a hypothesis that cannot be adequately verified (e.g. timing faults in distributed systems). It is important to note the distinction between error detection and fault location; an error shows the presence of a defect, but the underlying cause of this defect is only identified by a fault location process [HAL90]. This process is very much a problem-solving activity, but it can be tackled systematically (see [KER86]).

2.1.5 System design

A *system design* can be considered as the algorithm which is responsible for defining the interactions between components, establishing connections between components and the system environment, and for providing an supplementary processing for the system to achieve its required behaviour.

2.1.6 Design faults and component faults

A *design fault* is the failure of the system design algorithm to perform its intended function, whilst the failure of a system component to operate according to its specification is termed a *component fault.*

2.1.7 Related errors

A *related error* is a multi-version design specific conjecture whereby the probability of a version manifesting an error when another version has manifested an error is greater than the probability of the version manifesting an error on its own. This may lead to a higher probability of common-mode failure than would be the case if errors within versions were independent of each other.

2.2 Dependability

Traditional terminology, commonly used by both software engineers and hardware reliability engineers, is often inadequate when discussing software faults. Some of these traditional terms are defined below.

Reliability Reliability may be defined as the ability of a system to perform its required functions under stated conditions for a specified period of time.

Availability Availability is the degree to which a system or component is operational and accessible when required for use. This is often expressed as a probability.

Safety Safety is the non-occurrence of catastrophic consequences on the environment.

Confidentiality Confidentiality is the non-occurrence of the unauthorized disclosure of information.

Integrity Integrity is the degree to which a system or component prevents unauthorized access to, or modification of, computer programs or data.

Maintainability Maintainability has two forms :

1) The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment.
2) The ease with which a hardware system or component can be retained in, or restored to, a state in which it can perform its required functions.

The use of these terms is inadequate for several reasons - for example, design faults often lack any one useful categorization, whilst the actual identification of a particular aspect of a complex system design as being a fault may well be subjective. Also, depending on the circumstances, failures of interest could concern differing aspects of the service – e.g. the average real-time response achieved or the degree to which deliberate security intrusions can be prevented, etc. Hence, there is a need for a more general definition; ideally this should be properly recursive, in order to allow adequate discussion of problems that might occur at any level of a system.

This concept is known as *dependability* and was first proposed by Laprie in [LAP92]. Writing in [RAN95a], Laprie defines dependability as "*that property of a computer system such that reliance can justifiably be placed on the service it delivers. The service delivered by a system is its behaviour* as it is perceived *by its users*".

Dependability has three characteristics: attributes, means and threats. These are illustrated in figure 1.

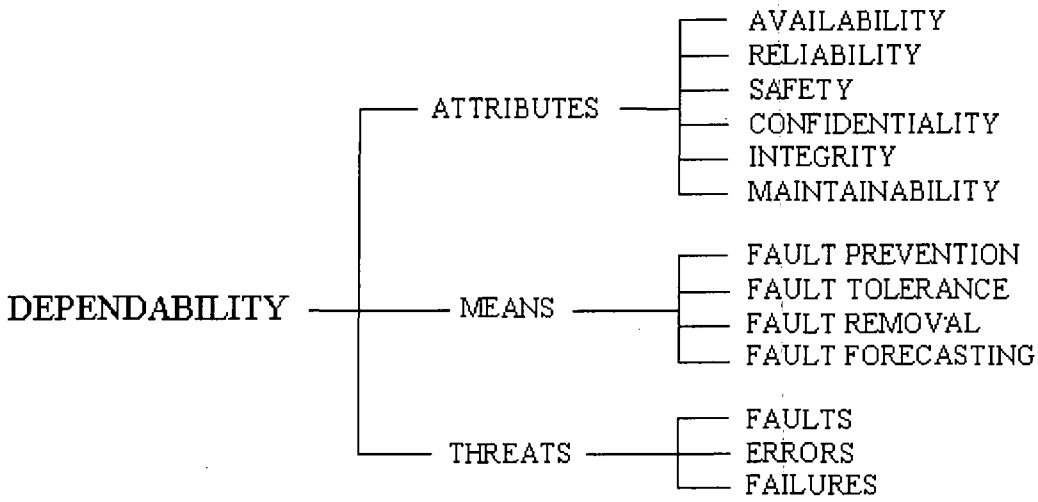


Figure 1 - Dependability

Dependability is a global concept, and subsumes the *attributes* of reliability, availability, safety, security, maintainability and confidentiality. These attributes enable the *properties* which are expected from a system to be expressed, and allow system quality resulting from the threats and means opposing it to be assessed. The *means* for dependability refer to methods and techniques that enable a system to provide the ability to deliver a service on which reliance can be placed, and confidence reached in this ability. The *threats* to dependability refer to undesired (but not necessarily unexpected) circumstances resulting from undependability.

Depending on the application, different emphasis may be placed on the various facets of dependability within a system; however, regardless of this, it can therefore be seen that dependability is not simply a synonym for reliability; rather, reliability is just one attribute of the overall concept.

2.3 The Need for Dependable Software

As the role of software becomes more and more entrenched in everyday usage, software dependability has increasingly come to the foreground. Although faults affect all types of software, they are of particular concern when developing safety-critical and real-time applications, where a single fault may result in a serious incident. Safety-critical software may be defined as *any software that can directly or indirectly contribute to the occurrence of a hazardous system state*. Obvious examples of this include aircraft flight systems and nuclear shutdown systems, but this definition also extends to more common applications, such as embedded systems within vehicles and domestic appliances, or indeed any system that

controls significant amounts of power [STO96]. The cost of failure within such systems is invariably high; there are numerous documented examples of such failure, many of which have resulted in the loss of human life [LAD99]. Given the increased need for dependability, many software systems still have an unacceptably high level of faults.

In an attempt to reduce this level of faults, the safety of relying on traditional development techniques has been questioned, and alternative development methodologies have been proposed. The vast majority of these 'alternative' methods fall into a category known as "Software Fault Tolerance". The question of *whether such alternative development methods result in a more dependable system* is multi-faceted and controversial, and is a question that this research seeks to further explore.

2.4 The "Traditional" Software Engineering Approach

Traditionally, software has been developed using a single variant approach – i.e. all the resources available for the development and implementation of a system (such as time to develop and the number of programmers) are concentrated on producing a single, dependable, "good" system.

This method addresses the "fault prevention" attribute of dependability, as it aims to prevent (as much as possible) the occurrence of program faults, through good design principles and implementation processes. It also addresses the "fault removal" attribute as it places an emphasis on thorough testing strategies with the aim of removing as many faults as possible.

Lack of dependability in such systems has been explained as due to lack of resources allocated to the design and development of software, such as the amount of time for implementation. This viewpoint suggests that given enough resources, software dependability will be greatly increased.

This viewpoint has been called a delusion by some commentators, such as [HAT97], who argues that different 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 15 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 it's thorough development, still had a defect density of 0.7 faults per thousand lines of code.

Current advances in the field of software engineering, such as object-orientation and software reuse strategies, attempt to increase the correctness and maintainability of software and thus reduce the number of undetected faults within systems. However, these approaches

cannot completely eliminate the risk of systems being developed with potentially serious and undetected faults. 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.

The advantage of this development approach is that it is a well-known and well-understood methodology, with a large number of supporting metrics [KIT90] that can be used to justify the approach to management. Perhaps the main disadvantage is that, due to the reasons given above, it is reasonable to assume that the incidence of faults within software systems will remain a problem for the foreseeable future.

Given this disadvantage, there is a need to investigate alternative approaches in order to investigate possible methods to reduce the potential amount of undetected faults within applications.

2.5 Software Fault Tolerance

The concept of *Software Fault Tolerance* [LYU95] has become increasingly recognized in recent years. Fault tolerant software allows errors to be detected and logged, without affecting the running of a system, and potentially offers great improvements in dependability over traditional development methods. [AVI85] describes the function of fault-tolerance:

"*...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.*"

There are two main approaches to software fault tolerance, depending on the goal of the system designer; these are either preventing a failure from leading to complete system disruption, or ensuring continuity of service. The aim of the former is to detect an erroneous task as soon as possible, and halt it to prevent error propagation – a technique often termed as *fail-fast* [GRA90]. The latter approach requires the use of *design diversity*; this is defined by [AVI86] as

"*....the production of two or more systems aimed at delivering the same service through separate designs and realizations.*"

The majority of fault tolerant methods use design diversity, and as such it is this approach that is of interest in this research. Two of the principle techniques in the area of design diversity are *Recovery Blocks* and *Multi-version Design*.

2.5.1 Recovery blocks

The *recovery blocks* technique is one of the earliest in fault tolerance, and was first introduced by [RAN75]. Recovery blocks work on the principle of *acceptance testing*; on entering a recovery block, system state is saved and a primary alternate is executed. An acceptance test is then performed to provide adjudication on the outcome of this primary alternate. If the acceptance test fails, then backward recovery is performed by the system reverting ("rolling back") to its previously saved, and the next alternate is executed. This may continue until either an alternate passes the acceptance test, or the final alternate is executed and fails the acceptance test. Should the final alternate fail, then the system will fail also. This is illustrated in figure 2. Recovery blocks can be nested, and so the raising of an exception from an inner recovery block can invoke recovery in an enclosing block.

The recovery block approach has a number of advantages. It is fault tolerant as errors discovered by the acceptance test can be detected, corrected and logged, and the approach can – if necessary - provide gradual degradation of a system, whereby each alternate runs a progressively smaller number of services in order to enable the system to pass an acceptance test. Also, provided the primary alternate does not fail, additional alternates will not be executed, and so the run-time overhead of recovery blocks can be minimal when compared to a single-variant system. There *is* a footprint, but tests by [SHR78a] [SHR78b] support the belief that recovery blocks do not impose any serious runtime and recovery data space overheads - the experiment showed that the run-time overhead ranged between 1 – 11% that of T1 (a program with no recovery facilities), provided the primary alternate did not fail. Should the primary alternate fail, the time to restore system state was up to 30% of T1.

However, the approach also has several disadvantages. For example, the success of recovery blocks rests to a great extent on the effectiveness of the error detection mechanisms used, especially (although not solely) the acceptance test. Should the acceptance test be faulty, alternates that are correct may be treated as though faulty, and faulty alternates may be treated as though correct. Also, there is a danger of what is called the 'Domino' effect. This can occur when a system of co-operating processes employs recovery blocks, as each process will continually establish and discard checkpoints, and may also need to roll-back to a previously established checkpoint. Should recovery and communication operations not be performed in a coordinated manner, then the rollback of a process can result in a cascade of rollbacks that could push all the processes back to their beginnings. Another potential

problem is finding a simple and highly reliable acceptance test that does not involve the development of an additional software version; the form of acceptance test depends on the application – for example, there may be a different acceptance test for each alternate, although in practice only one is usually used. This type of system is not considered appropriate for many real-time systems, as it is not feasible to simply 'roll back' the state of a system. Also, the nature of the system means execution time is unpredictable, as it depends on how many alternates fail the acceptance test. Alternates must not retain data locally between calls, otherwise the modules can become inconsistent with each other. The problem is more noticeable when attempting to design an alternate as an object. There is no guarantee that the state of the object is correctly modified unless the object is invoked each time, although [KIM84, KIM95] proposes *distributed recovery blocks* as a way of circumventing this limitation.

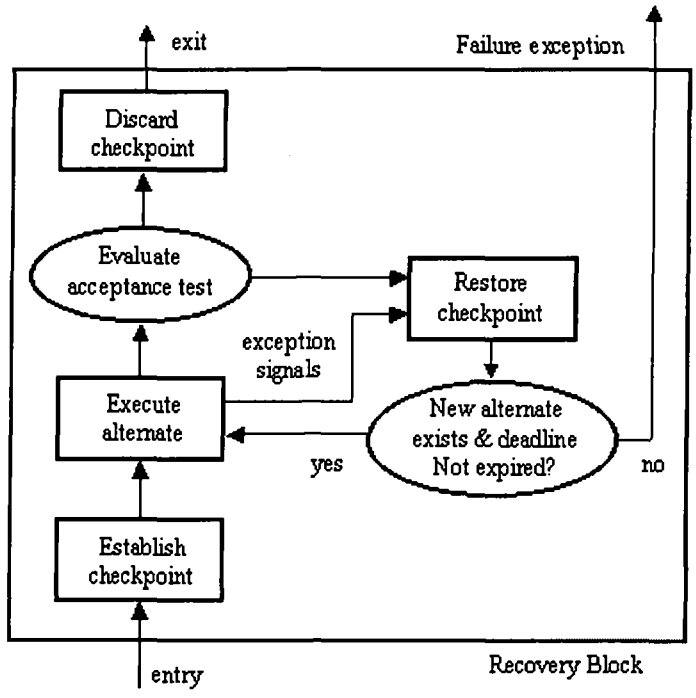


Figure 2 - Recovery block operation

Although the basic implementation of recovery blocks makes no provision for forward error recovery, this is possible, as described by [MEL77], whilst [CRI82] states that forward error recovery mechanisms can support the implementation of backward error recovery by transforming unexpected errors into default error conditions.However, this is very much application specific, and so it is often the case that the recovery block approach is

inappropriate for systems that require decisions to be made quickly (such as many real-time systems). Therefore, when such systems employ a fault tolerant approach, the most common methodology used is multi-version design.

2.5.2 Multi-version design

Multi-version design was first proposed by [AVI77]. It works on the principle of independently implementing *n* versions of a program (channels), which are then executed in parallel with a single input (although conceptually, parallel execution is not necessary - channels may be executed separately and their results later compared). The outputs of these channels are then compared under a voting system, which then forwards a single output based on the majority agreement of the channels [KNI86]. This is detailed in figure 3.

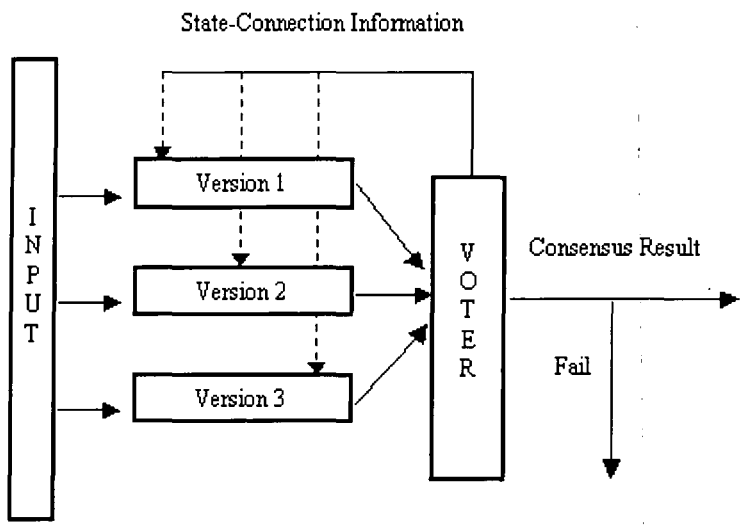


Figure 3 - A 3-version voter system

The multi-version approach has gained attention as a number of researchers have documented significantly increased levels of dependability within software developed using this methodology, e.g. [AVI89, HAT97] etc.

There is still much debate over how much of an improvement in dependability the approach offers over single variant design. Some researchers have concluded that the dependability of software developed using the multi-version methodology 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

variant of the system. This is not a new finding; earlier papers, such as [AVI84] have also argued that the approach produces highly dependable software.

2.5.3 The controversy over multi-version design

Such massive increases in dependability have, however, been drawn into question, and much debate has ensued; Knight and Leveson [KNI90] argue that these gains in dependability are under the assumption that there are no *correlated (common-mode) failures* within two or more channels of the system – in other words, no faults will occur in the same place and produce the same results. 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 faults can reduce the overall dependability of an N-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].

The voting software used in multi-version design must also be developed correctly and free of fault, otherwise the entire system can become unstable. An example of this is the NASA study of an experimental aircraft, which found that all of the software problems that occurred during flight testing were the result of faults found in the redundancy management system, and not the control software itself [MAC88].

Therefore, it 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 cost required to develop n versions of a system may not result in an equivalent increase in system dependability.

2.5.4 Cost factors of multi-version design

The cost of developing multiple versions is not n times the cost of developing one version, but also n times the cost of maintenance, which can be very high. Although arguments have been advanced that the increase in cost will be less than n [VOU90], Leveson [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 than usual, in order to make the results comparable, thus requiring that the specification phase take more time and effort than usual.

This therefore increases the overall algorithmic complexity of the project, which may again have an impact on the cost of the project as a whole.

[MAC91] argues, using a number of different calculations, that it can be the case that an imperfect 3-version voter system will be less cost effective than a simplex (i.e. single-version) system, although it would be as dependable; this assumes that all versions have equal development costs, whilst [LAP90] calculates that the cost of developing a 3-version system over a simplex system is at least 178% more costly, and can be as much as 271% more costly; on average, such a system would be 225% more expensive, although the 3-version system is more dependable.

It is not simply enough to implement *n* versions of a program if the resources allocated to that implementation are not substantial enough; the dependability of a multi-version system is directly related to the dependability of its individual channels. [KNI86] states:

"*...one might note that even in the hardware Triple Modular Redundancy (TMR) systems from which the idea of N-version programming arises, overall system reliability is not improved if the individual components are not themselves sufficiently reliable.*"

The emergence of software reuse libraries, whereby reusable software components may be bought and used to create large, dependable software systems very quickly, shows much promise for relatively cheap, fast creation of different channels within a 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. Although software libraries may help to drastically reduce the cost of developing N-version systems in the future, at present their impact on the cost of developing N-version systems is quite small.

It therefore appears to be the case that although an N-version system provides dependability that is at least equal (and usual superior) to that of an equivalent single version system, the cost is invariably higher.

2.5.5 Other FT methods based on RB and MVD

Although there are other fault tolerant methodologies, most are in some way based upon either the recovery block or the multi-version approach. For example, consensus recovery blocks [SCO85] and retry blocks [AMM87] both have their origins within the

recovery block approach, whilst acceptance voting [ATH89], *n* self-checking programming [LAP90] and n-copy systems [AMM87] are closely related to multi-version design.

2.6 The Need for Fault Tolerant Metrics

At present, there is very little empirical evidence as to which methodology (single-variant or fault tolerant) yields the most dependable system. The knowledge of which methodology is more dependable is very important – especially in industry – due to the increased cost associated with developing a fault tolerant system over a single-variant system.

Although much is known about assessing the dependability of single-variant systems [LAP95], lack of empirical evidence is especially acute when considering fault tolerant systems. For example, in a recent paper, [KIM00] states that "*effective, let alone optimal, resource allocation is not possible in the absence of quantitative characterizations of FT schemes*", and goes on to state that "*One can says that FT approaches not yielding to easy quantitative analyses are* unsafe to use. *Using such approaches is a blind exercise of an art.*"

This work seeks to develop a method for obtaining metrics from fault tolerant systems in order to better assess their dependability, and help build a more accurate dependability model for such systems. Systems that require the highest levels of dependability are invariably within the safety-critical domain, and are therefore usually real-time systems. Because of this, the use of recovery blocks is sometimes inappropriate (although schemes such as distributed recovery blocks [KIM95] help to address this problem), and so system designers frequently have to choose between multi-version design and the single-variant approach. Often, the single-variant approach is chosen due to the lack of empirical evidence regarding multi-version dependability - given the fact that multi-version systems may offer only a slight increase in dependability over single-variants, it is unknown whether the increased cost of developing such a system is worth the extra dependability gained. Therefore, this research will concentrate on developing a method for ascertaining the dependability of multi-version systems; derivatives of this method, such as n-copy and n self-checking systems will not be investigated, as these systems are less commonly applied in industry. Once a firmer understanding of the basic multi-version method is obtained, further investigations will be able to apply the technique developed to these systems.

2.7 Summary

This chapter begins by defining basic terms and concepts that will be used throughout the thesis, and gives a detailed definition of the concept of dependability. The traditional software engineering approach to developing software is then discussed, and both its advantages and disadvantages are explained. Software fault tolerant techniques such as recovery blocks and multi-version design are then discussed together with their respective advantages and disadvantages. The controversy over multi-version design is then described, and a discussion on the cost factors of MVD systems is given. The chapter concludes with the case for the need for more fault-tolerant metrics.

Chapter 3 Fault Injection

3.1 Problems with Traditional Testing

The vast majority of multi-version systems exist within the safety-critical domain. Within this domain, extremely high levels of dependability often need to be guaranteed; for example, [CHR94] states that the failure rate of these systems is usually required to be "*in the order of 10^{-8} - 10^{-10} failures per hour*". Unfortunately, it may be the case that traditional testing alone will not be able to adequately guarantee these levels of dependability. [HEC96] states that demonstrating that the failure rate of an item does not exceed x per hour requires "*approximately 1.5/x hours of test time under the most optimistic assumptions (no failures and a high risk test plan)*", and [BUT93] estimates that this would take thousands of years of testing to demonstrate (assuming one copy of software would be tested and one failure would be observed). Also, most multi-version systems are highly complex, and it is often infeasible to perform the enormous amount of test cases required to test every possible input and system state; according to [VOA95], "*the number of tests required for establishing high reliability are impractical if not impossible for software of even modest complexity*". Another weakness of traditional testing is that it often fails to exercise a systems response to rare (i.e. unlikely) events. A number of studies, such as [HEC93] and [HEC94] have shown that many failures in well-tested systems are caused by such events. The same data from these studies also shows that *multiple rare events* are almost the exclusive cause of the most critical failures in these systems.

Traditional testing may therefore never reveal any faults in such a system and it is a truism that non-exhaustive testing cannot reveal the absence of faults. This is a problem, as it not only means that a system's high levels of dependability cannot necessarily be guaranteed, but also makes comparisons between high-dependability single-version and multi-version systems extremely difficult.

3.2 Fault Injection

With this in mind, a different approach to testing is perhaps required. *Fault injection* has been proposed as an approach that addresses these limitations. Fault injection is a phrase covering a variety of testing techniques that can be applied to both hardware and software, all

of which involve the "*deliberate insertion of faults into an operational system to determine its response*" [CLA95]. Once this has been performed, an examination of the system for resulting errors and failures occurs, such as analysis of interactions between system components and of the resilience of the system against known faults. Fault injection is a "late life-cycle" software analysis [VOA98a] that can simulate human operator errors and observe their impact on the software as well as the *total* system. It is a technique that *complements*, but is not a substitute for, other verification and validation procedures.

3.2.1 Background of software fault injection

The idea of software fault injection is based upon hardware fault injection [CAR99], and originated in Mill's *fault seeding* approach in [MIL72], whereby an estimate of the number of faults in a system is made based upon how many injected faults are caught by the testing process. This was further improved using *stratified fault-seeding* [MOR88]. However, a number of other approaches have since been developed.

Fault injection is intended to yield three results: an understanding of the effects of real faults, feedback for system correction or enhancement, and a forecast of expected system behaviours [CAR99]. One of the major benefits of fault injection is its ability to test rare events and conditions, which, as discussed above, have been shown to be the cause of the majority of failures within safety-critical systems. [HEC96] states that "*The basic premise of the rare events approach is that well-tested software does not fail under routine input conditions, which means that failures must be triggered by unusual input data or computer states*". Such unusual input data and hardware states can easily be achieved with fault injection, and systems can be stress tested with large amounts of unusual conditions to garner their response. In this way, fault injection also helps to test the exception handling and redundancy management capabilities of a system, which are often overlooked by traditional testing.

Fault injection is also used to measure software *sensitivity*, or tolerance. Sensitivity is measured based upon a system's reaction to injections; high sensitivity means that injections frequently cause the system to produce undesirable outputs ("undesirable" is defined in either the system specification, requirements or defined software hazards [VOA97]). High sensitivity implies a lower tolerance for failure, and thus shows a system to have a greater risk of failure than a low sensitivity system.

Faults are introduced in one of two ways - either through direct alteration of code, or by the perturbation of data flows or control flows to achieve the effects of faults indirectly - and can be categorized based on when the faults are injected: either during compile-time or run-time [HSU97].

When altering program code, faults are typically created by either adding code to the code under analysis, changing the code, or deleting code. Code that is added to a program for the purpose of either simulating errors or detecting the effects of those errors is called *instrumentation code.* To perform fault injection, some instrumentation is always necessary, and is usually performed by a tool (although it can be added manually). Instrumentation code can be placed on top of input or output interfaces to the software, or directly into the logic of the software, and can be added to a variety of code formats, such as source code, assembly code, binary object code, etc. Typical injected faults include mis-timings, delays, missing messages, corrupted memory, faulty disk reads, logical errors, syntax errors and perturbation of variables. Faults can be injected in many ways and can address program state as well as communication and interactions.

There are two key approaches for instrumentation – code mutation and state perturbation. *Code mutation* [DEM78] occurs at compile-time and involves direct alteration of program code, attempting to reproduce potential human errors within code; this typically involves changing the syntax of existing code statements or modifying their logic in some way – an example of this is shown in figure 4. The main danger with code mutation is that of creating an *equivalent mutant*; this is a mutation that does not affect the output of the code in any way (i.e. has no semantic impact on the code base) and is hence meaningless. Mutation may also result in transient faults occurring - for example, in figure 4, one of the mutations shown (`A = A + A + 2;`) will only affect the value of A if A is not zero; this is also undesirable, and needs to be guarded against.

Suppose a program has the following code statement :

```
A = A + 2;
```

This statement can be mutated as follows :

```
A = A + A + 2;
```

or it could be mutated to :

```
A = A + 20;
```

etc. The code could also be deleted.

Figure 4 - An example of code mutation

State perturbation [VOA97] has the intention of forcefully modifying program states created by the original code, without mutating existing code statements. This is often

achieved through the use of *code insertion* whereby instrumentation code is added to a system in the form of function calls that modify internal program values (termed *perturbation functions*), but it can also be implemented by modifying input data or by the fault injector trapping exceptions generated by the system through the use of interrupts.

Perturbation functions are code instrumentation, and are typically applied to programmer-defined variables. They can change either the value of a variable to a value based upon the current value, or can change the variable to a value picked at random, independent of the original value. They may also return a constant replacement value, if it is suspected that any fault placed at that point in the code will result in one particular value regardless of what the current value is. When non-constant replacement values are used, the perturbation functions produce random values based upon the current value and a *perturbation distribution*, with non-constant perturbation distributions including all of the continuous and discrete random distributions.

Figure 5 shows an example of a perturbation function. The function, `newvalue(int a)`, randomly either increases a value by 40% or reduces it by 40%. Should this increase/decrease not affect the original value in any way, then the function returns the original value minus one. This perturbation function is then applied to a variable (in this case, an integer variable) in a desired part of the original code. For example, to modify the variable `a`, we simply add

```
a = newvalue(a);
```

to the original code.

Additionally, *faulty input data* can be passed into a system at run-time – either by the mutation of 'real' data or a false set of data. [VOA98a] suggests that faulty input data is the easiest form of fault to simulate correctly (i.e. in a way that reflects real errors that could occur naturally). Although state perturbation sometimes requires system code to be re-compiled, original code is not altered (i.e. instrumentation is added, but original code is not mutated) and injections occur at run-time – it can therefore be thought of as run-time based fault injection. The advantage of state perturbation is that the problem of equivalent mutants does not arrive, and all perturbations should affect system state.

Assume a function `equilikely(x,y)` that randomly returns either `x` or `y`.

```
int newvalue(int a)
{
   int counter = 1;
   int oldvalue = a;

   do
   {
        a = equilikely (oldvalue * 0.6, oldvalue * 1.4);
        counter ++;
   }
   while ((a == oldvalue) && (counter < 100));

   if ((counter == 100) && (a == oldvalue))
   {
        a = oldvalue - 1;
   }

   return a;
}
```

Figure 5 - An example of a perturbation function

3.2.2 Differences with traditional testing techniques

As stated earlier, fault injection *complements* traditional testing but does not replace it. Fault injection cannot be viewed as testing in the *traditional sense*, as traditional testing seeks to determine whether a system meets its stated requirements, and requires a definition of what the correct outputs of the system should be. Fault injection is generally incapable of determining correctness, as the act of injecting anomalies into code and/or data results in an altered state that may produce incorrect outputs with regard to the system requirements. It is therefore impossible to assert that the code *itself* produces incorrect output, but it can be asserted that the *modified* code produced incorrect output. [VOA98b] states that "*The main use of software fault injection is in demonstrating what sort of outputs software produces under anomalous circumstances.*"

Although software engineering practices attempt to predefine system behaviour in the event of anomalous conditions, testing invariably only looks at 'reasonable' anomalous conditions that are considered possible. Fault injection however, can often offer insight into a systems behaviour with the injection of unreasonable, highly unlikely conditions. Should a previously unconsidered anomaly be injected and cause the system to fail, then fault injection will have demonstrated that the system is highly sensitive to the problem it was forced to deal

with, and the system will need to be analysed in order to ascertain whether any related faults also exist.

3.2.3 Issues to consider

When considering how to deploy fault injection, two issues need to be addressed. The first is that of *simulation versus execution. Simulation* refers to the development of a model of a system, with faults introduced into the model rather than the system itself. This method is often slower to test, but easier to change. *Execution* refers to the process of injecting faults into a real system; this is often more useful for analyzing final designs, but is typically more difficult to modify afterwards.

The second issue is that of *invasive and non-invasive techniques*. A major problem with sufficiently complex systems – particularly time dependant ones – is that is may be impossible to remove the footprint of the testing mechanism from the behaviour of the system, independent of the fault injected. For example, a real-time communication protocol that would normally meet a deadline for a particular task may miss it because of the extra latency induced by the fault injection mechanism. Invasive techniques are those that leave behind such a footprint during testing, whilst non-invasive techniques are able to mask their presence so as to have no effect on the system other than the faults they inject.

These factors need to be considered when developing a fault injection strategy for a system, in order to gain the most useful results for the budget and type of system used.

3.3 Applying Fault Injection to Multi-Version Systems

Given the potential benefits of fault injection, it is surprising that the method has mainly been focused on assessment of single version software. [CHE99] states that "*as far as fault injection for diversity evaluation is concerned, this has not been achieved, and the lessons from the literature are limited and of a general nature only*".

The potential is great; by developing an automated system that can inject faults into different versions of a multi-version system, test the systems, and then repeat the process with another set of injected faults, it should be possible to build up a picture of the relationships between different versions with regard to common-mode failures. For a multi-version system, there are a total of

$$\sum_{R=1}^{N} \frac{N!}{(N-r)!\times r!}$$

combinations for fault injection to be applied to, where N is the number of versions taken r at a time. Given that N will usually be a small odd integer, such as 3, 5 or 7, this should not pose a problem.

For a more detailed analysis, it may also be possible to inject faults into individual functions within different versions of a multi-version system, in order to investigate possible relationships between disparate channels. For example, consider a 2-version system, each version containing 4 functions/procedures (see figure 6). Version 1 contains the set of functions {A,B,C,D} whilst version 2 contains the set of functions {E,F,G,H}. Faults are injected in each of the functions in turn, and the systems are analysed for common-mode failures following each injection. This is repeated for as many combinations of functions as possible. Should it be found that injecting faults (either similar or otherwise) into function A of version 1 and function H of version 2 causes a common-mode failure, then the analysis will have revealed a potentially unsafe relationship between these functions, even if the functions have no obvious connection. "Traditional" testing methods can then be fine-tuned to test these functions in more detail.

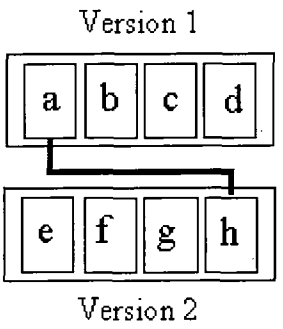


Figure 6 – Should faults injected into functions within individual versions lead to common-mode failure, then these functions can be seen to have a potentially unsafe relationship and need to be tested in more detail.

Fault injection can also assess the sensitivity of each version, on either a system-level or a function-level. Should any version or function within a version be highly sensitive, then further debugging/testing can be applied, in order to reduce the sensitivity and hopefully reduce the likelihood of a failure that could lead to a common-mode failure within the system.

Furthermore, fault injection provides a very good method for deriving metrics about a system, and could therefore help to provide quantitative characterizations for multi-version systems – [VOA95] states that "*fault-injection techniques are dynamic, empirical and*

tractable". Therefore, this approach will help to solve one of the problems highlighted by [KIM00], discussed in section 2.6.

This research therefore proposes to implement an automated fault injection system, designed to assist with the assessment of multi-version systems. This implementation is detailed in chapter 4.

3.4 Summary

This chapter details the problems associated with traditional testing techniques, and then goes on to detail the background of fault injection. Different methods of fault injection are discussed, and the differences with traditional techniques are examined. A method for applying fault injection to multi-version systems is then discussed.

Chapter 4 Implementation

4.1 FITMVS

The major goal of this research is to develop a non-commercial fault injector that will enable an automated fault injection process to be performed on multi-version systems, in order to produce valuable metrics, such as sensitivity measures and analysis of potential for common-mode failure. This system is called **FITMVS** (Fault Injection Tool for Multi-Version Systems). The remainder of this chapter discusses both the design and implementation of FITMVS.

4.2 The Design of FITMVS

FITMVS performs *data value perturbation*, whereby code modifying a particular variable's value is added to an existing system's code. Data value perturbation was chosen as by using this technique, FITMVS neatly avoids the *equivalent mutant* problem. This occurs when an injection (in the form of code mutation) is made that does not affect the output of the code in any way (i.e. has no semantic impact on the code base) and is hence meaningless. Instead, all injections made by FITMVS will alter system state in some way – whether trivial or otherwise. Data value perturbation also leads to a simpler parsing process, and hence allows for quicker development time.

The basic operation of FITMVS is to parse the code of each channel within a multi-version system, and then systematically inject faults into each scope within a specified source file, compile and execute the code, test the system against a user-created set of tests, log the results, revert the code back to its original state, and inject a fault into the next scope within the source file. This is continued until the last scope within the source file has had at least one injection applied to it. At the conclusion of running FITMVS, a multi-version system will therefore have had at least one injection made into each scope within its code, and will have been tested for each of these injections. This is explained in more detail in figure 7. The process by which this takes place can be split into three stages: system input, the automated process, and system output. These three stages are detailed below.

4.2.1 System input

User input to the FITMVS system is achieved by way of a menu-driven user interface, inside of a standard UNIX terminal window. When the program is initially

executed, the user is shown the main menu screen, which gives the user the option of starting the system immediately, editing the system's settings, or exiting the system. This main menu screen is shown in figure 8.

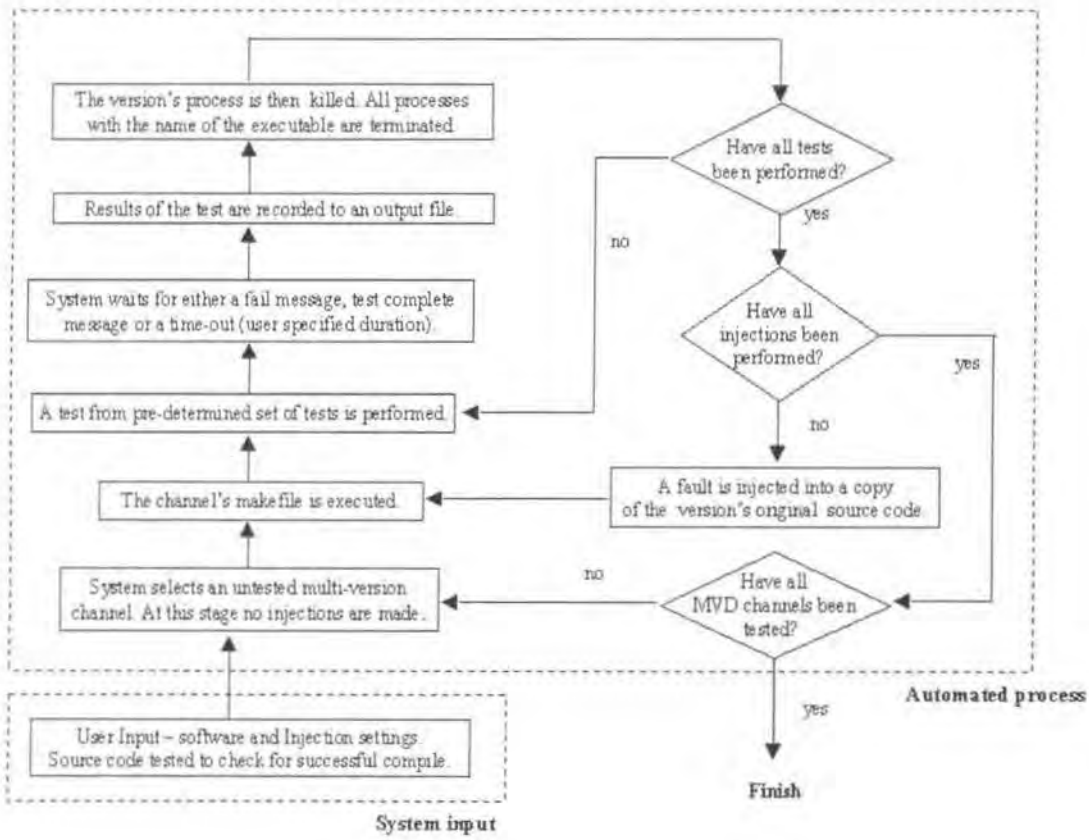


Figure 7 - FITMVS Operation Flow-Chart

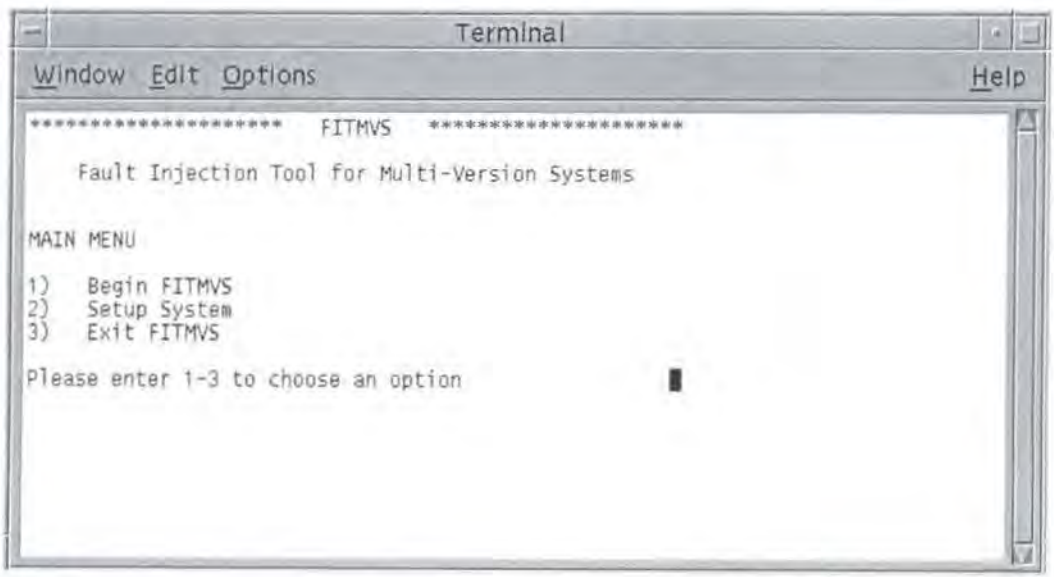


Figure 8 - FITMVS Main Menu screen

Selecting the first option – to start the system immediately – will result in the system running with whatever defaults have been hard-coded into it, and so this is only of real use if the user would like to start the system with minimal effort, and has placed their required settings into the FITMVS source code.

The system setup menu – shown in figure 9 – allows the user to decide the category of settings that they desire to alter, as well as allowing them to load previous settings and save the current settings. "Configure software versions" allows them to configure settings with regard to the names and locations of the multi-version system channels that are to be tested, whilst "Configure injection settings" allows the user to edit the way the system goes about its automatic task of injecting and testing faults within the software versions. When the user loads or saves settings, they are prompted for a filename, which is then used to either save settings to, or restore settings from.

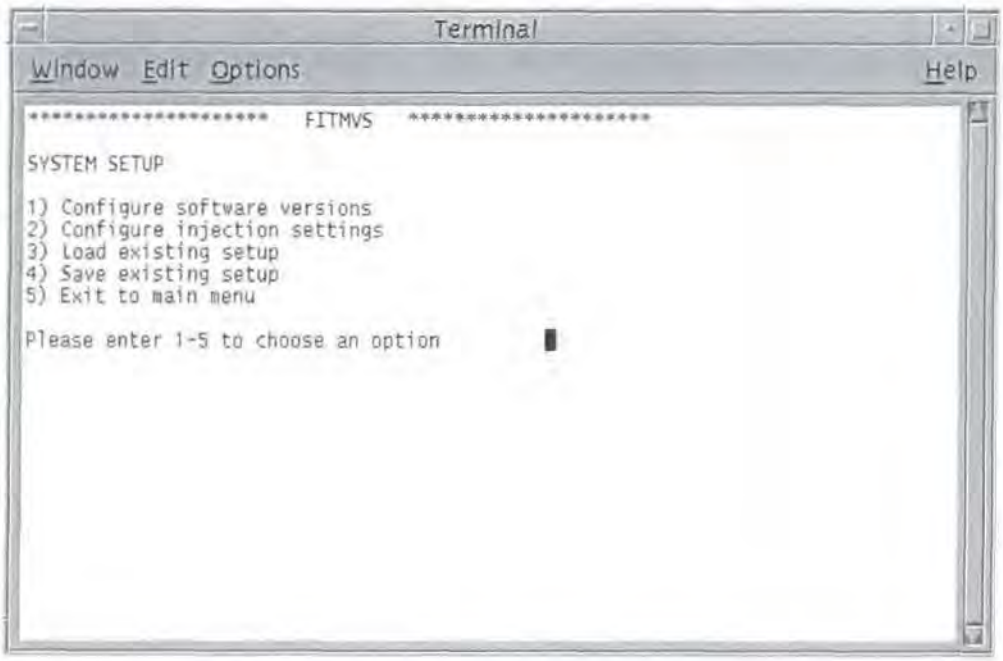


Figure 9 - FITMVS System Setup screen

The "Configure Software Settings" screen, shown in figure 10, displays the name (i.e. the name of the executable) of each MVD version that has been entered into the system, and allows users to add versions, remove existing versions or edit the version information. When a version is edited or removed, the user is prompted for the number of the version, as displayed on this screen. When a version is added, the user is prompted for the name of the version, and is then taken to the "Edit Version" screen, where further details about the version can be entered.

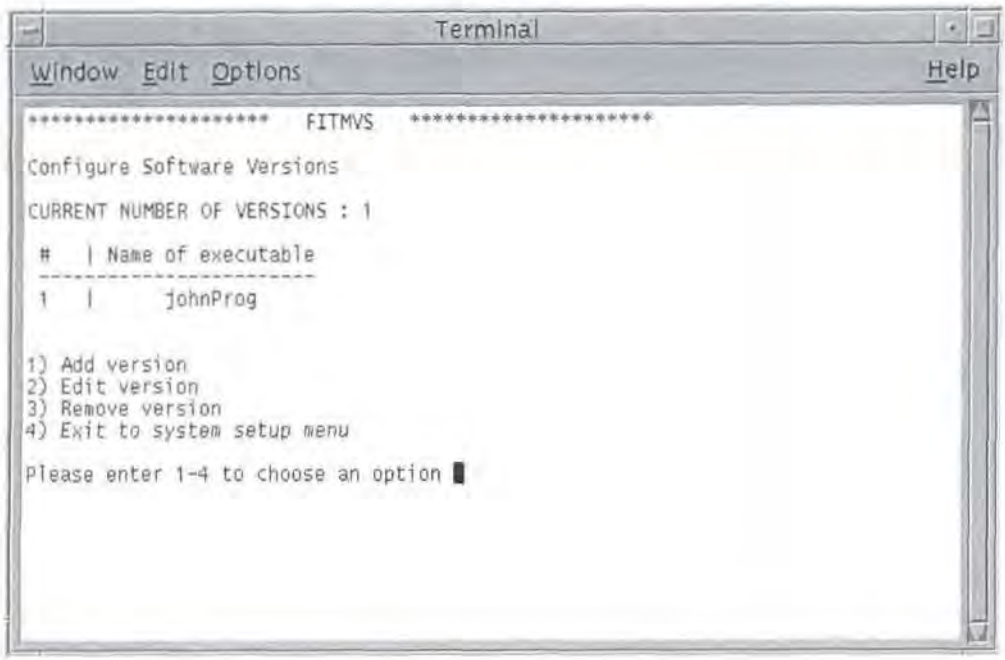


Figure 10 - FITMVS Configure Software Settings screen

The "Edit Version" screen (figure 11) allows the user to modify a number of aspects of the MVD version. The executable name of the version can be modified, as can the source directory of the version (i.e. the directory where the version source code is contained). The invocation command may be set if the version requires a special command to execute (for example, a batch file may be required, that starts other essential processes for the version to successfully execute).

The "Edit Related Processes" option refers to the names of processes that are related to the MVD version at execution time; when FITMVS kills (terminates) the MVD version (at the conclusion of each test performed), all processes listed in the related process list are also killed.

The "Edit Injectable Sources" option allows the user to specify the filenames of source code that FITMVS will inject faults into; selecting this option will take the user to another screen (shown in figure 12) and gives the user the option of adding or removing filenames from the list.

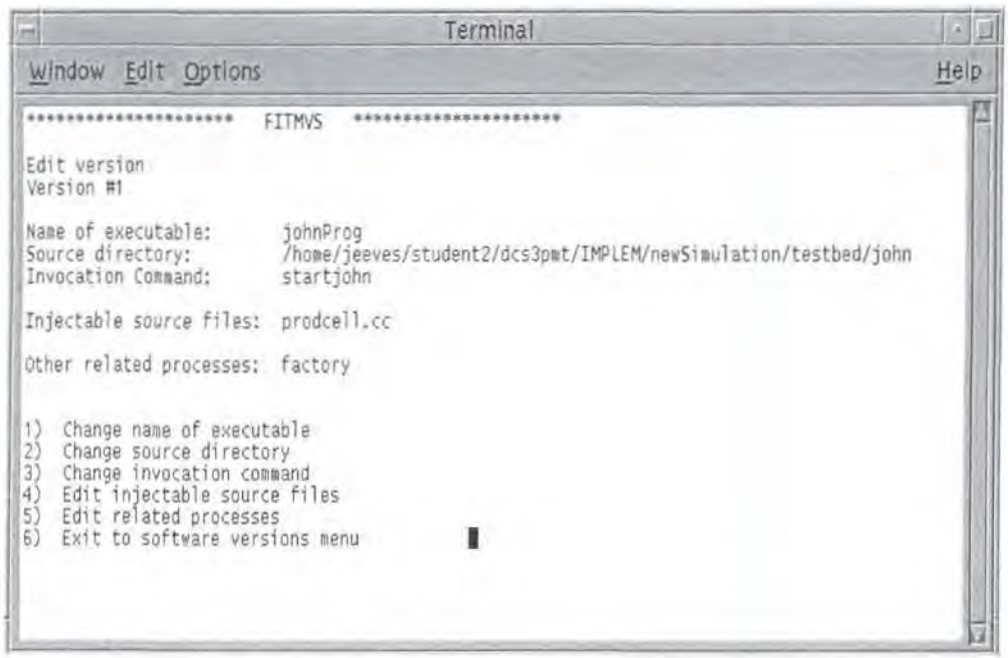


Figure 11 - FITMVS Edit Version screen

Figure 12 - FITMVS Edit Injectable Sources screen

A single screen handles all of the injection settings within FITMVS, and is reached from the main menu. This screen is detailed in figure 13. The "injections per scope" value sets how many times an injection/test cycle will be performed per scope in each source code file listed in the "Injectable Sources" list. The "minimum scope lines for injection" value refers to

the minimum number of lines that a scope within the source code must have before an inject/test cycle is performed on it; any scopes with fewer lines than this value are ignored. The perturbation distribution refers to the maximum amount by which a variable may be perturbed when a fault is injected; this number applies to both positive and negative values. For example, should the PD be set to 32768 then a perturbation function can be added to source code that increases or decreases a variable's value by no more than 32768.

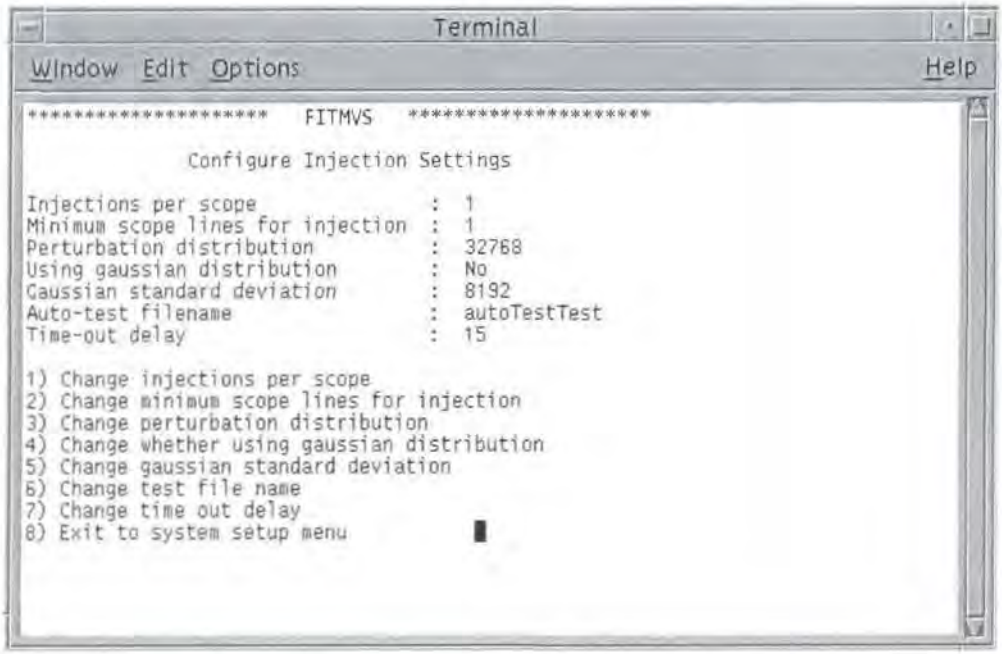


Figure 13 - FITMVS Configure Injection Settings screen

The "change whether using Gaussian distribution" option allows the user to specify whether the values by which variables are perturbed follow either a normal (i.e. every value is equally likely) distribution or a Gaussian distribution (for which the probability of a number being picked follows a bell-shaped curve). The "Gaussian standard deviation" value applies only when gaussin distribution is being used, and refers to the width of the Gaussian distribution curve. A Gaussian probability distribution is described as follows:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)}$$

μ = mean

σ = standard deviation

This results in a bell-shaped probability curve with a width based upon the standard deviation, with the probability of a number being selected increasing as the number becomes

closer to the mean value. In FITMVS, this mean value is always 0. An example Gaussian distribution is shown in figure 14. 99.9% of all values generated by the Gaussian function will fall within 4 standard deviations of the mean, and so FITMVS allows the user to specify a standard deviation of up to 25% of the perturbation distribution. For example, should the perturbation distribution be set to 32768, then the maximum allowed Gaussian standard deviation is 8192. This ensures that 99.9% of possible values outputted by the function will fall between +32768 and -32768. Any that fall outside of this value are rounded to the nearest maximum (either positive or negative). This means that the probability curve will have a small up-turn on large standard deviations, but this should be negligible.

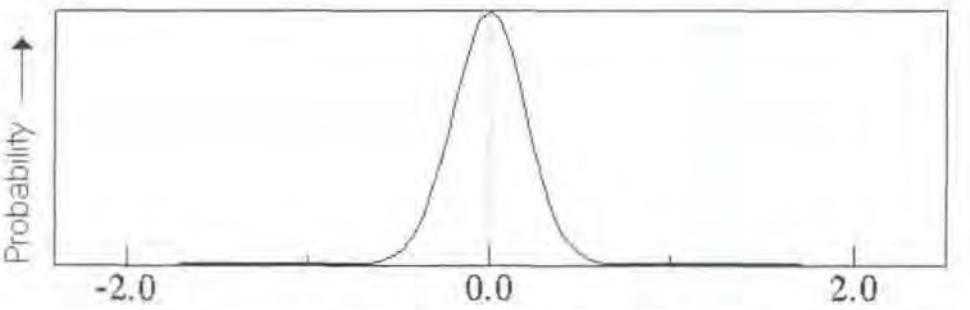


Figure 14 – A Gaussian probability distribution curve with a standard deviation of 0.2 and a mean of 0.0

The purpose of including Gaussian probability distributions in FITMVS is to further the scope for statistical analysis of data outputted by the system; varying the standard deviation will force FITMVS to perturb variables by different ranges, and so it may be of interest to see if a relationship between the size of perturbations (i.e. the standard deviation) and the sensitivity of a system exists.

The "Change test file name" allows the user to specify the filename of the test file that FITMVS reads when testing each MVD version. The "time out delay" value refers to the number of seconds that must pass without response during testing from the MVD version before FITMVS determines that a time-out has occurred.

4.2.2 The automated process

The automated phase of FITMVS is completed without any input from the user. The 'main loop' of this process operates by injecting a fault into the source code of a version, executing that version's makefile, and performing the tests listed in the test-set file on this version. Once each test has been performed, the system waits for either a fail message, test complete message or time-out from the version; this result is then logged in a results file specific for that version (either a "pass" or a fail description/number). The version's process is

then 'killed' by FITMVS using the standard UNIX `kill` system call on the version's process number, and the testing process continues with the next test in the set. When all tests have been completed, the source code of the version is reverted to its original state and a different injection is made, and the process is repeated; this is repeated until the specified number of injections have been performed for every eligible program scope.

This 'main' loop is repeated for each version in the multi-version system, with the first cycle of the process for each version performed without any injection in order to record 'baseline' results.

4.2.3 System outputs

When all the results for each version have been collected, an analysis can be performed based on the log file outputted by FITMVS at the conclusion of each injection cycle. This is shown in figure 15.

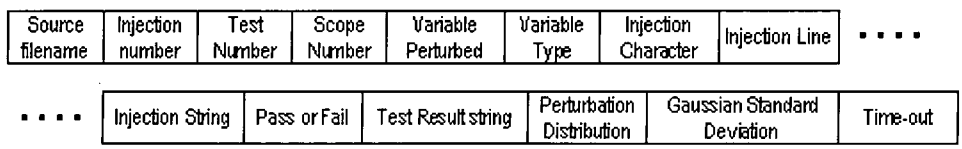


Figure 15 – The layout of the FITMVS log file

This consists of the filename of the source file being injected, the number of times an injection has been performed on that scope, the number of the test being performed, the number of the scope being perturbed, the name of the variable being perturbed, the type of the variable (int, float, etc.) being perturbed, the character and line of the source file that the perturbation function was injected at, whether or not the test was a pass or a fail (represented as 1 or 0 respectively), the message received from the target system following the conclusion of a test, the perturbation distribution of the injection, the standard deviation of the Gaussian function being used, and the number of seconds that the time out delay is set for.

4.3 Objectives of the System

Current tools for the implementation of fault injection in multi-version systems are rare, and of the few that exist (such as [VOA97]), all are commercial and thus inaccessible to

most researchers in the field; therefore, one of the implicit goals of this research is to make such a system available to the general academic community.

The FITMVS system itself has five objectives. The first objective is to help identify areas of code that might lead to common-mode failure - when the automated fault injection process has finished, FITMVS logs can be analyzed and common-mode failures discovered, together with the location and type of faults injected to cause them. This will enable the user to ascertain which areas of code in each version of the multi-version system – when faulty – will combine to cause common-mode failure. Subsequent testing can then place a greater emphasis on proving the correctness of these areas, in order to minimize the risk of common-mode failures arising.

The second objective of FITMVS is to identify any channel of a multi-version system that shows a high sensitivity to injected faults; from this analysis, it will be possible to identify which MVD versions are most "at risk" in the event of an error occurring, and hence perform corrective maintenance on that version. The third objective of FITMVS is related to this; namely, by analyzing the number of errors resulting from faults injected into each program scope, the sensitivity of each scope will be determined, thus giving developers more insight into what areas of code need most attention. Areas of code with high sensitivity invariably has a much greater risk of failure than a low-sensitivity area, and so any highly sensitive areas revealed by the fault injection process may then be re-examined and changes made in order to increase their resilience.

The fourth objective of FITMVS is to calculate the probability that the complete MVD system will fail with a common-mode failure, should a fault be injected into each version; this metric should help to give much needed empirical data into the relative value of MVD systems. The fifth objective of FITMVS is to establish which errors manifest themselves most often when a fault is injected into a MVD channel.

4.4 Limitations of the System

FITMVS in its initial conception has a number of limitations, although these are largely implementational. Initially, the system will only be developed to analyse and inject faults into C and C++ source code; however, the actual parser used by the system will be modular, and so further language support will have the potential to be added in future versions. The parser itself will be limited, again due to time constraints, and therefore complex mutations will not be possible. Initially, the system will be designed to simply add perturbation of data values to

code, rather than any form of mutation, although this again will be modular, with the potential for code mutation functionality to be added in the future.

4.5 Portability Issues

FITMVS is written in ANSI C++ and should therefore be portable to most UNIX and Linux systems. However, the shared memory functionality and the mechanism used to kill processes mean that some modification will be required for the system to work in alternative operating systems, such as Microsoft Windows. Despite this, these changes should not be too difficult to make.

4.6 The Development of FITMVS

4.6.1 The parser component

The actual development of the FITMVS system took place over 6 weeks. The first four weeks of this time was dedicated to the creation of a parser capable of parsing C and C++ code and producing a parse tree as its output. The parser itself is quite simple, and records the name and return type of each variable within each code scope. In addition to this, the position in the code of each variable's definition and first assignment are also stored.

The parse tree is a linked list of type `ScopeRecord`. Each `ScopeRecord` object contains information in regard to a program scope – it's start and end position, and the number of its parent scope (should it be a nested scope). It also contains two linked lists; one of type `variableRecord` and one of type `injectRecord`. `VariableRecord` contains data with regard to each variable that exists within the scope – the position of its definition, the position of its first assignment, whether or not is assigned within the current `scopeRecord` object, its name and its type. Each `variableRecord` object is unique to each `scopeRecord` object, and so a variable declared early in the code may be represented in multiple variableRecord objects. The `injectRecord` object is used for storing records of injections made into each scope in order that no duplicate injections are made; this is not used in the initial parsing function of FITMVS. Figure 16 details the make-up of the `injectRecord`, `variableRecord` and `scopeRecord` objects, and figure 17 shows the overall parse tree structure. The parser component of FITMVS was written as a stand-

alone module, and hence can be used by any application to produce a parse tree like that illustrated in figure 17.

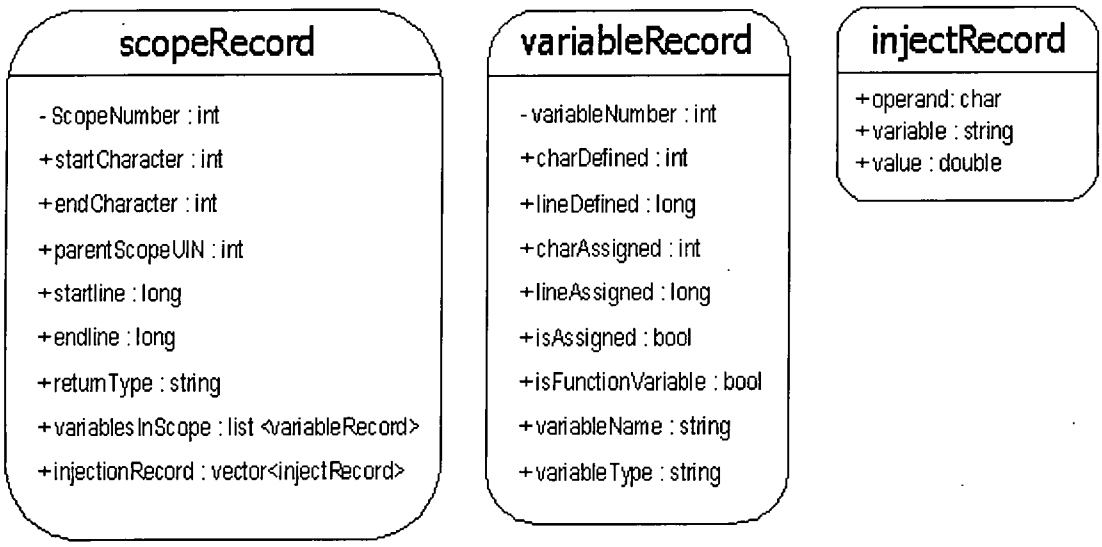


Figure 16 - The `scopeRecord`, `variableRecord` and `injectRecord` objects

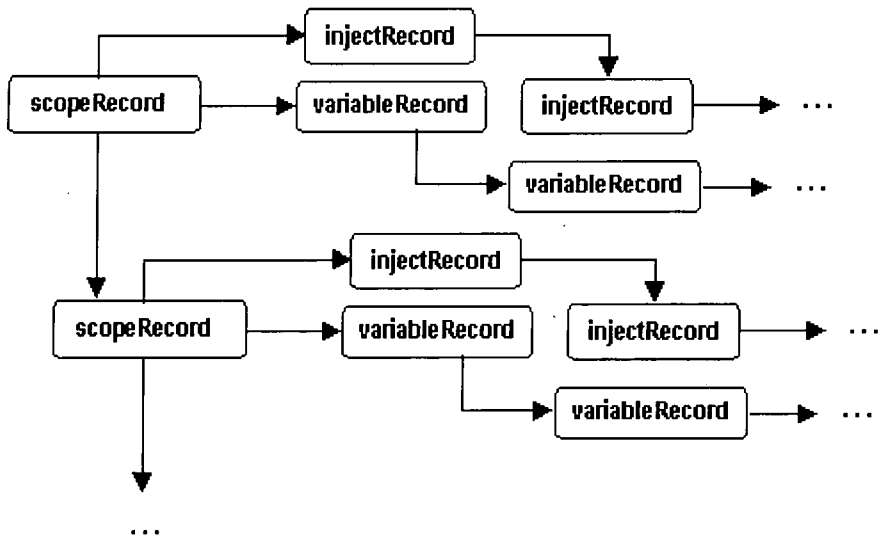


Figure 17 - The structure of the parse tree generated by the parser component of FITMVS

The performance of the parser module is satisfactory, especially when considering that the parser is only used once for every channel in the MVD system. In order to provide a rough guide to the exact performance of the parser, an automated test was created, whereby a 1000 line program was parsed, and then appended to itself (to form a 2000 line program) and re-parsed. This cycle was continued until the program had reached 100,000 lines in size, with the amount of time to parse being recorded in each cycle. The result of these tests is shown in figure 18.

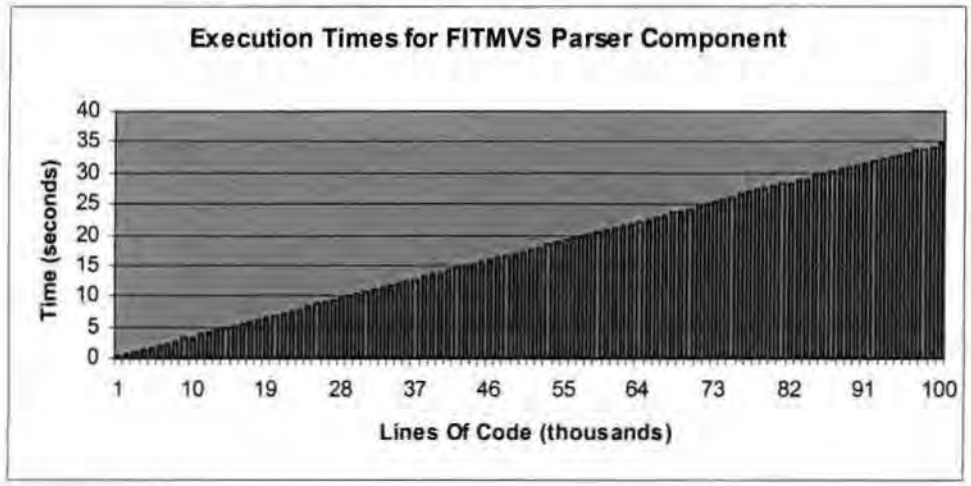


Figure 18 - Parse Times for different sized programs

As can be seen, the relationship between time taken to parse and the size of the target program was linear; this was expected, as the parser was essentially parsing the same additional code on each test. The main purpose of the test was to determine whether the structure of the parse tree or the amount of data being placed on the system stack would cause parse times for realistic-sized programs to be adversely affected. Fortunately, this does not appear to be the case.

4.6.2 Auto-testing functionality

After the parser component was completed, the auto-testing functionality of FITMVS was implemented. This consisted of the shared memory mechanisms, functionality to decode and send test messages from a specified test file, mechanisms for checking if a process has timed-out, and mechanisms for process termination.

The GNU shared memory libraries were used to create two classes – `sharedMemoryClient` and `sharedMemoryServer`. These classes are designed to

provide a simple interface between the FITMVS system (using `sharedMemoryServer`) and the target MVD system (using `sharedMemoryClient`). The mechanism for determining whether or not a time-out has occurred simply uses these shared memory objects to check whether the target (MVD) system has written to the shared memory space. If such a write does not occur within an amount of time specified by the user, the process termination mechanisms are enforced. These work by simply redirecting the output of the standard UNIX `ps` tool through a `grep` statement designed to filter out all processes that are not related to the process requiring termination. The output from this is then re-directed to a file, from which process numbers are extracted and terminated using a `kill -9` command. Overall, this stage of development took approximately 1 week of time.

4.6.3 The main fault injector and user interface components

With the completion of the parser and auto-testing routines, the development of the main fault injector component of FITMVS was relatively simple, and only required 3 days of development time. The injector's main duty is to analyze the parse tree for each program scope and calculate whether an injection should be performed; if so, then a variable stored within that scopes variable list is selected at random, and a perturbation function is placed within the program code at the either the start of that particular scope, or immediately after the variable is first assigned within the scope (if applicable).

The final major development process was the creation of the user interface. Due to time constraints, a graphical user interface was not pursued; indeed, it would be unwise to spend valuable time on such a display when the FITMVS system is still in a proof-of-concept stage. Instead, the user-interface consists of a series of text-based menus, and input from the user is entirely keyboard-based. The user interface routines were required to be portable between UNIX platforms and terminal types, and therefore some re-writing of standard C functions such as `kbhit()` was required; however, despite this, the user interface modules took only 3 days of development time to complete.

4.6.4 Changes required to the target system

Before FITMVS can be used, a number of preparations must be made in regard to the target system (i.e. the multi-version system to be tested), in order for the automated process to function correctly. A standard header file containing the `sharedMemoryClient` object

must be included in the MVD system code in order for the system to be able to communicate through shared memory to FITMVS.

This header file also contains the functions `FITMVS_pass()`, `FITMVS_fail(string)` and `FITMVS_confirm()`. These functions will send a 'test passed' message, a 'test failure' message (with a failure description), and a message confirming that the current data in the shared memory space has been received, respectively. It is through this use of shared memory that FITMVS will be able to record the results of tests performed. The only exception to this is if a version does not report a result within a given amount of time; should this occur, FITMVS will terminate the target systems process and record a `TIMEOUT` message. A `FITMVS_getMessage()` function is present, and automatically reads the shared memory and returns the contents as a string to the MVD system.

A `FITMVS_reset()` function is also present and will also have to be added to each target system. This function may involve significant changes between different software systems. Essentially, the goal of the function is to reset the state of the target system back to its initial state; should this prove difficult to do, then the function should send a `KILL_SYSTEM` message to the fault injector in order for the target system's process to be terminated and re-started.

An aim of FITMVS is to make the process of adapting an existing system in the way described above as easy as possible; this is why most of the function calls needed are pre-written and available in a header file which can then be inserted into the target system's code. Where necessary, the user will then be able to modify the pre-written functions in order to best represent the target system.

4.6.5 The test-set file makeup

The process of parsing and applying the data values specified by the test-set file is left to the user to implement, with a partially written function included in the standard FITMVS header file that all target systems will need to include. Each line of the test-set file constitutes a test; this takes the form of the name of the test data, followed by the values appropriate for this data, separated by commas and enclosed within brackets. Each data element is delimited with a semi-colon. The form of the test file therefore resembles:

```
VariableName(value,value,...);VariableName(value,value,...); etc.
```

Once the target system has parsed this data and entered it appropriately, a TEST_RECEIVED message is sent to FITMVS and the test is considered to have started, with FITMVS waiting for the test result to be transmitted through shared memory. Should no response come within a specified time-out period, then the target system will be considered to have timed-out.

4.7 Summary

This chapter introduces the Fault Injection Tool for Multi-Version Systems (FITMVS). It goes on to detail the design and operation of FITMVS, the objectives of the system, the limitations of the system, and portability issues. The actual development of each major component of FITMVS is then discussed. The changes that need to be made to target systems are detailed, and the chapter concludes by describing the make-up of the test files used by FITMVS.

Chapter 5 Application Case Study

5.1 Factory Production Cell Case Study

In addition to the development of FITMVS, it is also necessary to select an appropriate MVD application to test. Because of the large implementation time required to develop the FITMVS system, it is prudent to select an existing MVD system for which source code is available. It is also desirable for the application to be a real-time system, as real-time systems invariably involve high reliability and safety requirements. To this end, a system previously researched by the author [TOW01a, TOW01b] was chosen.

The application is the controller system for a simulation of a flexible factory production cell (figure 19). The production-cell 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 the workstation, it can either be switched on and off by the controller software, or is permanently on. Two cranes, mounted on a racking which prevent them from both being in the same X position at the same time are used to transport blanks around the system. Each blank has its own bar-code, which identifies which workstations it needs to be placed in, and the minimum and maximum amounts of time that it can spend within each workstation. Blanks can be processed either in a specific order, or in any order, depending on the instructions in the bar-code.

The controller software is required to allow the production-cell simulation to process up to two blanks (units) 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. Further safety requirements include both cranes being returned to safety positions whenever they are not in use, and ensuring that blanks are not left in workstations for longer than their maximum stipulated time. 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 fall off the end of the belt.

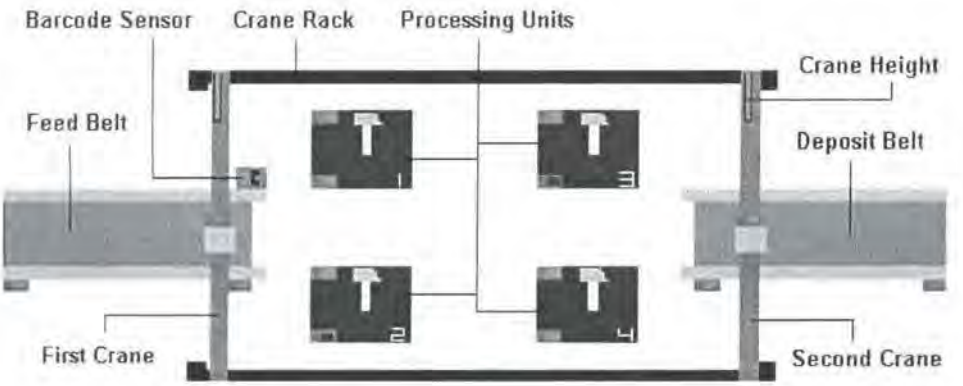


Figure 19 - Diagram of Flexible Production Cell

5.2 System Requirements

The MVD system comprises of three separate channels, two of which are developed in C++, and one of which is developed in Java. Each channel of the MVD controlled system was developed independently, to a rigorous specification document in order to ensure that the diverse channels followed the same rules and procedures in given scenarios, and with minimum contact between programmers. Although the specification document was rigorous, at the same time it was important to ensure that the system specification is not over-specified; therefore, the specification stated functional requirements clearly and unambiguously, whilst leaving the widest possible choice of implementations. Over-specification at the requirements stage has been a criticism of several past experiments [AVI89], as the reduced levels of diversity increase the probability of correlated faults, and hence reduce the overall dependability of the resulting multi-version system.

The system requires real-time processing, and is safety-critical, and therefore requires an extremely high level of dependability. As many experiments have found that correlated faults can drastically reduce the overall dependability of a multi-version system – e.g. [KNI86, ECK91] – a conscious decision was made to make the development of each channel as diverse as possible.

5.2.1 Assumptions

Within the requirements document, a number of assumptions are made about the working environment of the controller software and simulation. These are listed in figure 20.

#	Assumption
1	The system has no more than one feed belt and one deposit belt.
2	There is no set value or limit for the number of items passing through the system.
3	There are only two types of workstation as described in the task description document.
4	The setup has only two possible configurations; those of one crane and two workstations, and of two cranes and four workstations.
5	Each item has a minimum and a maximum time that it can spend in each workstation.
6	Items may have a maximum amount of time in which they can be in the system for.
7	The maximum time that a blank may spend inside a system may not be less than the total minimum time that it must spend in each of the workstations.
8	Each workstation can only process a blank once before it is removed from the system.
9	A blank may only be placed on the deposit belt if no blank is detected there.
10	The deposit belt is not controlled by the control program.
11	The gripper has only two vertical positions. It can only retrieve blanks while in the lower position, and it can only move horizontally without colliding with workstations and belts while in the upper position.
12	If a blank has a set order of processing, the system must process the blank in that order.
13	The cranes must never be at the same X position.
14	The cranes may not move unless the gripper is in the upper position.
15	A blank may only be placed in a workstation if the sensor reports that it is free.
16	The magnet may only be enabled or disabled while in its lower position.
17	The magnet may not be disabled while carrying a blank unless the gripper is both in its lower position and above either a belt or a workstation.
18	The feed belt must be turned off if the end-belt sensor reports a blank.
19	Every blank passing through the system has a bar code, and all bar codes are correct.
20	All blanks are set a distance apart so the system can distinguish between separate blanks
21	Every blank introduced to the system by the feed belt or present in the system at the start of operation must also leave the system via the output belt.

Figure 20 - Assumptions made regarding the controller software's working environment

5.2.2 Operational environment

The production-cell simulation is implemented in Java, and can be run on a number of different operating systems (although it is primarily designed for use on UNIX systems). The controller software did not require any graphical output, and so none of the platforms used for the development of the software were required to produce graphical output. The controller software itself is used only to produce output files for use by the production-cell simulation; therefore there is no minimum speed requirement on any system using the controller software.

5.2.3 External interfaces & data flow

The production-cell simulation and the controller software communicate via a first-in-first-out pipe mechanism, with communications being sent as ASCII text. For example, a message to the production-cell simulation consists of a header, the message body and a terminator. The header consists of an open squared bracket - [- followed by a linefeed. The

terminator consists of a closing square bracket -]. Figure 21 demonstrates the format of messages bodies.

Message	Description
PortalX*n* p	Move crane *n* to horizontal position p.
PortalY*n* p	Move crane *n* to vertical position p.
MagnetOn*n*	Switch on the magnet on the crane *n*.
MagnetOff*n*	Switch off the magnet on the crane *n*.
PortalDown*n*	Move crane *n* to the down position.
PortalUp*n*	Move crane *n* to the up position.
FeedBeltOn	Switch on the feedbelt.
FeedBeltOff	Switch off the feedbelt.
CodeSensorOn	Activate the code sensor.
WorkStationOn*n*	Switch on workstation *n*. This command is ignored by type 2 devices.
GetState	Requests the production-cell simulation to return the current state.

Figure 21 - Simulation inputs

5.2.4 Logging format

The controller software is required to log its activity in a file. These options (and the logging file name) are specified as a command line argument. The logging file is designed to be used to compare results between different controller versions, and to see where errors have been made.

A log entry is made for every program cycle where information is received from the simulator, or when a decision was made. Accordingly, the log records all information received from the simulation and the results of any decisions made by the controller. Figure 22 shows the format of the log data for each program cycle.

Item	Conditional	Description
[	No	Start of log item
time	No	Time in milliseconds since the start of the controller.
<STATUS>	Yes	Start of status
status	Yes	The unformatted feedback from 'GetState'
</STATUS>	Yes	End of status
[OUTPUT]	Yes	Start of output section
controller output	Yes	Output of controller: exactly the same as the output given to the production-cell simulation.
[/OUTPUT]	Yes	End of output section
]	No	End of log item

Figure 22 - Format of controller log

5.2.5 General crane operation

It is necessary to make assumptions regarding crane operation within the factory production cell. These are listed in figure 23.

#	Assumption
1	The two cranes can move simultaneously
2	When a crane has no job to perform, it will move back to the safety position
3	The safety position is defined as X1, Y2 for crane 1 and X8, Y2 for crane 2
4	Crane 1 will deal with workstations 1 and 2
5	Crane 2 will deal with workstations 3 and 4

Figure 23 - Assumptions made about the cranes

If the movement of a crane results in it colliding with the other crane, then it is required to move back to the appropriate safety position defined in figure 23, until the other crane is in a position which will not result in collision.

5.2.6 Movement of blanks

When a single blank enters the system, the controller is simply required to process the blank in the order stipulated by its bar-code, within specific time limits. If there are two blanks in the system then one of five scenarios described in figure 24 will occur.

#	Scenario
1	Both blanks needed to be transported to the deposit belt.
2	Both blanks needed to be moved to other workstations.
3	One blank needed to be moved to the deposit belt, the other to another workstation.
4	Only one blank needed to be moved to another production cell or the deposit belt.
5	Neither needed to be moved.

Figure 24 - Scenarios when there are two blanks in the system

5.2.7 Both blanks need to be moved to the deposit belt

If both blanks need to be moved to the deposit belt, then crane one will move to its safety position, whilst crane two collects the blank with the lowest maximum processing time(max_i) and moves it to the deposit belt. Crane two will then repeat the process with the

other blank. If both blanks possess the same max_i then the blank that is in the station with the lowest ID number will be moved first.

5.2.8 Both blanks need to be moved to other workstations

When blanks are on opposite sides of the production cell and need to be moved to the opposite workstation, it is specified that the following should be done; crane 1 will pick up the blank in station 1 or 2, and crane 2 will pick up the blank in station 3 or 4. Both cranes are then moved to their target stations and will both then deposit their blanks. This is shown in figure 25.

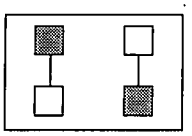


Figure 25 - example situation of blanks on opposite side of the production-cell

There are several possibilities for the movement of blanks in this scenario, and the requirements document specifies the procedure to follow for every combination of workstations, in order to make sure the different versions will make the same decisions.

5.2.9 One blank needs to be moved to deposit belt, the other to another workstation

In this case crane 1 will return to its safety position and crane 2 will move to pick up the blank which needs to be removed from the system. Crane one will then move and deposit the remaining blank in the desired station; crane 2 will then move its blank to the deposit belt.

5.2.10 Only one blank needs to be moved to another workstation or the deposit belt.

In this case, the controller will move the relevant blank to its target destination as if it is the only blank within the system; should its target destination be unavailable, the relevant crane will pick up the blank and move to its safety position until the target destination becomes free; it will then deposit the blank appropriately.

5.2.11 Neither needs to be moved

If neither blank needs to be moved, the controller software will perform no action.

5.2.12 Belt Control

The feed and deposit belts within the system also need to have their behaviour specified. Figure 26 lists the assumptions that are made about them.

#	Assumption
1	The feed belt is initialised to be off.
2	The feed belt should be switched off immediately after the feed belt sensor is activated.
3	At the same time that the feed belt is switched off, the code sensor should be activated.
4	Once a blank has been lifted from the feed belt (crane is in upper position), the feed belt should be started in order to move the next blank up to the code sensor.
5	The deposit belt is controlled separately and will always be running.
6	A blank may only be placed on the deposit belt if the deposit belt indicator shows that another blank is not already there.
7	If the above situation occurs, the crane should stay in the upper position at the deposit belt place until the sensor has indicated that the belt is free. The blank should then be immediately lowered and then immediately released.

Figure 26 - Assumptions about the feed belt and deposit belt

5.3 Summary

This chapter describes in detail the factory production cell simulation that is to be used to test the effectiveness of the FITMVS system; and describes the system requirements, operational details, and assumptions made by the production cell simulation.

Chapter 6 The Experiment Performed

6.1 Overview of the Experiment Performed

In order to assess the effectiveness of the FITMVS system, it is necessary to apply FITMVS to an existing MVD system; the system chosen is that of the factory production cell discussed in chapter 5. One of the current limitations of FITMVS is that it is only able to partially parse Java source code, and so the two MVD channels written in C++ were used to form a 2-version system for purposes of this experiment.

This experiment seeks to use the FITMVS system to perform injections on each program scope in both channels; following each injection, FITMVS will automatically compile the perturbed channel and test it against a set of tests specified below. One complete run of injections through a target channel is referred to as an "injection cycle". Altogether, a total of 25 injections cycles are applied to each channel during the experiment, with 5 injection cycles being performed for Gaussian distributions with standard deviations of 8192, 4096, 2048, and 1, as well as for a normal distribution. All tests will be performed with the perturbation distribution set to 32768. At the end of each injection cycle, the resultant log files produced by FITMVS are saved and analyzed; these list every single injection and test performed, together with the results of the test. From analysis of these log files, a picture of overall sensitivity to fault is created for each channel.

6.2 Re-development of the Factory Simulation

The major difficulty with testing the factory controller system with FITMVS is that the actual simulation itself is written in Java, and is both slow, unstable, and difficult to adapt to automatic testing (i.e. automatic entry of test data). In order to maximize the number of tests that could be performed on the system, it was decided that the entire simulation must be re-written. It should be noted that this in no way affects the MVD controller system - merely the simulation that it controls.

The simulation was therefore re-written entirely in C++. The new simulation includes all shared memory libraries and routines necessary for communication with FITMVS, as well as allowing for test data to be entered automatically. In addition, the new system executes many times faster than the original Java version; unfortunately, due to the real-time nature of the simulation, the MVD controller channels often process blanks whilst measuring

processing time based on the hardware timer, and so the time taken per test is only reduced by approximately 83%, from an average test time of 60 seconds to an average test time of approximately 10 seconds (although this depends on the actual minimum processing time values set for each blank). It was not possible to increase the interrupt rate (i.e. speed) of the hardware timer, as the SPARCstations used to test FITMVS are multi-user machines.

6.3 Test Data

As previously discussed, the MVD channels perform processing based upon the hardware timer, and so each test performed requires several seconds to execute. Although time values can be set to 0 seconds, it is desirable to retain minimum and maximum deadlines within the test data as the temporal domain is very important when considering real-time systems, and it is of interest to see if temporal faults are triggered during the injection testing. Due to the number of injection-cycles that are to be performed, the number of tests per injection have to be kept to a minimum otherwise the amount of time required to perform the tests will be too great.

With this in mind, a total of 5 tests are used. These are chosen to cover as broad a range of situations are can be expected with such a small test set. The setup of the tests is as follows :

6.3.1 Test 1 (single blank)

Maximum Time in System for blank 1 (ms) : 9000

Blank 1 – preserved order				
Workstation	1	2	3	4
Min (ms)	1000	1000	1000	1000
Max (ms)	3000	3000	3000	3000

6.3.2 Test 2 (single blank)

Maximum Time in System for blank 1 (ms) : 10000

Blank 1 – non-preserved order				
Workstation	2	3	1	4
Min (ms)	2000	1000	2000	1000
Max (ms)	3000	2000	4000	3000

6.3.3 Test 3 (two blanks)

Maximum Time in System for blank 1 (ms) : 7000
Maximum Time in System for blank 2 (ms) : 9000

Blank 1 – non-preserved order				
Workstation	1	3	4	2
Min (ms)	1000	1000	1000	1000
Max (ms)	3000	3000	3000	3000

Blank 2 – non-preserved order				
Workstation	2	1	3	4
Min (ms)	1000	1000	1000	1000
Max (ms)	3000	3000	3000	3000

6.3.4 Test 4 (two blanks)

Maximum Time in System for blank 1 (ms) : 9000
Maximum Time in System for blank 2 (ms) : 10000

Blank 1 – preserved order				
Workstation	1	4	2	3
Min (ms)	1000	1000	1000	1000
Max (ms)	3000	3000	3000	3000

Blank 2 – non-preserved order				
Workstation	4	3	2	1
Min (ms)	1000	1000	1000	1000
Max (ms)	3000	3000	3000	3000

6.3.5 Test 5 (two blanks)

Maximum Time in System for blank 1 (ms) : 7000
Maximum Time in System for blank 2 (ms) : 10000

Blank 1 – non-preserved order				
Workstation	3	2	4	1
Min (ms)	1000	1000	1000	1000
Max (ms)	3000	3000	3000	3000

Blank 2 – non-preserved order				
Workstation	1	4	2	3
Min (ms)	1000	1000	1000	1000
Max (ms)	3000	3000	3000	3000

The actual contents of the test file used to set up these tests is shown in figure 27.

```
BLANK(1,2,3,4,1000,1000,1000,1000,3000,3000,3000,3000,t,9000)
BLANK(2,3,1,4,2000,1000,2000,1000,3000,2000,4000,3000,f,10000)
BLANK(1,3,4,2,1000,1000,1000,1000,3000,3000,3000,3000,f,7000);BLANK(2,1,3,4,1000,1000,1000,1000,3000,3000,3000,3000,f,9000)
BLANK(1,4,2,3,1000,2000,1000,2000,3000,3000,2000,3000,t,9000);BLANK(4,3,2,1,3000,1000,2000,2000,3000,3000,4000,3000,f,10000)
BLANK(3,2,4,1,1000,1000,1000,2000,3000,2000,2000,4000,f,7000);BLANK(1,4,2,3,1000,1000,3000,2000,3000,4000,5000,4000,f,10000)
```

Figure 27 - Contents of the test file used to test the MVD factory system

6.4 Processing Time

Due to the large amount of time expected for the completion of each injection cycle, it is desirable to speed up the testing of the MVD system by executing FITMVS on multiple machines simultaneously. Therefore, a total of 14 different SPARC workstations will be used for testing; FITMVS will run on each system simultaneously (running an identical copy of the MVD channel software). When an injection cycle on a machine finishes, another can be started on the machine if necessary.

Output from FITMVS is in the form of a log file, which can be directly imported into a Microsoft Excel spreadsheet. This spreadsheet can then be used to analyse the results of each injection cycle, and should allow for relatively quick analysis.

6.5 Summary

This chapter gives an overview of the experiment performed using the FITMVS system. It details the re-development of the factory simulation in C++, and describes the test data used during the experiment. The chapter concludes by describing the extra hardware used to combat the large amount of processing time required for each test.

Chapter 7 Results and Analysis

7.1 Overview of Results

The experiment was performed over a period of one week. At the conclusion of the experiment, a total of 21,211 tests were performed; this is in contrast with the 4,320 tests performed manually on the MVD system in [TOW01a, TOW01b] – an increase of more than 490%. This was in large part due to the automatic testing mechanisms that were put into place. Each complete run of FITMVS took approximately 2 hours on Channel A of the MVD system, and 4 hours on Channel B of the MVD system, and the overall amount of processing time was approximately 150 hours, equating to 6 and a half days of continuous processing (although it must be remembered that much of this processing was done across multiple SPARC workstations).

The difference in processing time between the two MVD channels is explained due to the fact that Channel B has a greater number of code scopes than Channel A (131 scopes as opposed to 73), and so a larger number of injections and subsequent tests were performed on Channel B.

7.2 Output of FITMVS Log Files

The amount of data produced by FITMVS was very pleasing, with a total of more than 875 pages of Microsoft Excel-readable logs produced from the 25 injection-cycles performed on both channels. As described in chapter 4, each line of these log files states the source filename of the injected code, the number of the injection, the test number, the scope number, the name of the variable perturbed, the type of the perturbed variable, the character and line number within the source file where the perturbation function was placed, the injection string itself, whether or not the test was successful (1 indicates success, 0 indicates failure), the test result message, the perturbation distribution, the standard deviation of the gaussian distribution and the time-out interval of the test. Due to size considerations, not even a single log file can be produced in its entirety; however, figure 28 and figure 29 show an example of the data collected.

Thu Aug 16 14:32:08 BST 2001
Minimum lines for injectable scope: 1
Time-out delay: 15
Gaussian distribution: Yes
Perturbation distribution: 32768
Standard Deviation: 8192

components.cpp	1	1	1	t	long	19	17	t = t + 7152;	1	Test passed	32768	8192	15
components.cpp	1	2	1	t	long	19	17	t = t + 7152;	1	Test passed	32768	8192	15
components.cpp	1	3	1	t	long	19	17	t = t + 7152;	1	Test passed	32768	8192	15
components.cpp	1	4	1	t	long	19	17	t = t + 7152;	1	Test passed	32768	8192	15
components.cpp	1	5	1	t	long	19	17	t = t + 7152;	1	Test passed	32768	8192	15
components.cpp	1	1	2	yPos	int	25	37	yPos = yPos + 8234;	0	Test failed: Crane one dropped blank - Factory::checkCraneMagnets()	32768	8192	15
components.cpp	1	2	2	yPos	int	25	37	yPos = yPos + 8234;	0	Time out	32768	8192	15
components.cpp	1	3	2	yPos	int	25	37	yPos = yPos + 8234;	0	Test failed: Crane one dropped blank - Factory::checkCraneMagnets()	32768	8192	15
components.cpp	1	4	2	yPos	int	25	37	yPos = yPos + 8234;	0	Test failed: Crane one dropped blank - Factory::checkCraneMagnets()	32768	8192	15
components.cpp	1	5	2	yPos	int	25	37	yPos = yPos + 8234;	0	Test failed: Crane one dropped blank - Factory::checkCraneMagnets()	32768	8192	15
components.cpp	1	1	3	yPos	int	25	37	yPos = yPos + 9713;	0	Test failed: Crane one dropped blank - Factory::checkCraneMagnets()	32768	8192	15
components.cpp	1	2	3	yPos	int	25	37	yPos = yPos + 9713;	0	Time out	32768	8192	15
components.cpp	1	3	3	yPos	int	25	37	yPos = yPos + 9713;	0	Test failed: Crane one dropped blank - Factory::checkCraneMagnets()	32768	8192	15
components.cpp	1	4	3	yPos	int	25	37	yPos = yPos + 9713;	0	Test failed: Crane one dropped blank - Factory::checkCraneMagnets()	32768	8192	15
components.cpp	1	5	3	yPos	int	25	37	yPos = yPos + 9713;	0	Test failed: Crane one dropped blank - Factory::checkCraneMagnets()	32768	8192	15
components.cpp	1	1	13	temp	int	6	145	temp = temp + -6420;	1	Test passed	32768	8192	15
components.cpp	1	2	13	temp	int	6	145	temp = temp + -6420;	0	Time out	32768	8192	15
components.cpp	1	3	13	temp	int	6	145	temp = temp + -6420;	0	Time out	32768	8192	15
components.cpp	1	4	13	temp	int	6	145	temp = temp + -6420;	0	Test failed: Workstation 1 - blank exceeded time limit. Factory::checkWorkstations()	32768	8192	15
components.cpp	1	5	13	temp	int	6	145	temp = temp + -6420;	0	Time out	32768	8192	15
components.cpp	1	1	14	destination	int	2	149	destination = destination + 7274;	1	Test passed	32768	8192	15
components.cpp	1	2	14	destination	int	2	149	destination = destination + 7274;	1	Test passed	32768	8192	15
components.cpp	1	3	14	destination	int	2	149	destination = destination + 7274;	1	Test passed	32768	8192	15
components.cpp	1	4	14	destination	int	2	149	destination = destination + 7274;	1	Test passed	32768	8192	15
components.cpp	1	5	14	destination	int	2	149	destination = destination + 7274;	1	Test passed	32768	8192	15
components.cpp	1	1	15	workTmp	int	5	155	workTmp = workTmp + -446;	1	Test passed	32768	8192	15
components.cpp	1	2	15	workTmp	int	5	155	workTmp = workTmp + -446;	1	Test passed	32768	8192	15
components.cpp	1	3	15	workTmp	int	5	155	workTmp = workTmp + -446;	0	Test failed: Workstation 2 - blank exceeded time limit. Factory::checkWorkstations()	32768	8192	15
components.cpp	1	4	15	workTmp	int	5	155	workTmp = workTmp + -446;	1	Test passed	32768	8192	15
components.cpp	1	5	15	workTmp	int	5	155	workTmp = workTmp + -446;	0	Time out	32768	8192	15
components.cpp	1	1	17	destination	int	4	161	destination = destination + -982;	1	Test passed	32768	8192	15
components.cpp	1	2	17	destination	int	4	161	destination = destination + -982;	1	Test passed	32768	8192	15
components.cpp	1	3	17	destination	int	4	161	destination = destination + -982;	0	Test failed: Workstation 1 - blank exceeded time limit. Factory::checkWorkstations()	32768	8192	15

Figure 28 - Extract of FITMVS output for Channel A

Thu Aug 16 18:24:28 BST 2001
Minimum lines for injectable scope: 1
Time-out delay: 15
Gaussian distribution: Yes
Perturbation distribution: 32768
Standard Deviation: 8192

prodcell.cc	1	1	2	command	string	37	28	command = command + 18307;	1	Test passed	32768	8192	15
prodcell.cc	1	2	2	command	string	37	28	command = command + 18307;	1	Test passed	32768	8192	15
prodcell.cc	1	3	2	command	string	37	28	command = command + 18307;	1	Test passed	32768	8192	15
prodcell.cc	1	4	2	command	string	37	28	command = command + 18307;	1	Test passed	32768	8192	15
prodcell.cc	1	5	2	command	string	37	28	command = command + 18307;	1	Test passed	32768	8192	15
prodcell.cc	1	1	3	tm2	bool	33	73	tm2 = false;	0	Time out	32768	8192	15
prodcell.cc	1	2	3	tm2	bool	33	73	tm2 = false;	0	Time out	32768	8192	15
prodcell.cc	1	3	3	tm2	bool	33	73	tm2 = false;	0	Test failed: Crane two dropped blank - Factory::checkCraneMagnets()	32768	8192	15
prodcell.cc	1	4	3	tm2	bool	33	73	tm2 = false;	0	Test failed: Crane two dropped blank - Factory::checkCraneMagnets()	32768	8192	15
prodcell.cc	1	5	3	tm2	bool	33	73	tm2 = false;	0	Test failed: Crane two dropped blank - Factory::checkCraneMagnets()	32768	8192	15
prodcell.cc	1	1	4	command	string	7	36	command = command + -1924;	1	Test passed	32768	8192	15
prodcell.cc	1	2	4	command	string	7	36	command = command + -1924;	1	Test passed	32768	8192	15
prodcell.cc	1	3	4	command	string	7	36	command = command + -1924;	1	Test passed	32768	8192	15
prodcell.cc	1	4	4	command	string	7	36	command = command + -1924;	1	Test passed	32768	8192	15
prodcell.cc	1	5	4	command	string	7	36	command = command + -1924;	1	Test passed	32768	8192	15
prodcell.cc	1	1	5	end	bool	7	41	end = false;	1	Test passed	32768	8192	15
prodcell.cc	1	2	5	end	bool	7	41	end = false;	1	Test passed	32768	8192	15
prodcell.cc	1	3	5	end	bool	7	41	end = false;	1	Test passed	32768	8192	15
prodcell.cc	1	4	5	end	bool	7	41	end = false;	1	Test passed	32768	8192	15
prodcell.cc	1	5	5	end	bool	7	41	end = false;	1	Test passed	32768	8192	15
prodcell.cc	1	1	6	end	bool	7	46	end = true;	1	Test passed	32768	8192	15
prodcell.cc	1	2	6	end	bool	7	46	end = true;	1	Test passed	32768	8192	15
prodcell.cc	1	3	6	end	bool	7	46	end = true;	1	Test passed	32768	8192	15
prodcell.cc	1	4	6	end	bool	7	46	end = true;	1	Test passed	32768	8192	15
prodcell.cc	1	5	6	end	bool	7	46	end = true;	1	Test passed	32768	8192	15
prodcell.cc	1	1	7	command	string	7	52	command = command + -7551;	1	Test passed	32768	8192	15
prodcell.cc	1	2	7	command	string	7	52	command = command + -7551;	1	Test passed	32768	8192	15
prodcell.cc	1	3	7	command	string	7	52	command = command + -7551;	1	Test passed	32768	8192	15
prodcell.cc	1	4	7	command	string	7	52	command = command + -7551;	1	Test passed	32768	8192	15
prodcell.cc	1	5	7	command	string	7	52	command = command + -7551;	1	Test passed	32768	8192	15
prodcell.cc	1	1	8	command	string	7	58	command = command + 5533;	1	Test passed	32768	8192	15
prodcell.cc	1	2	8	command	string	7	58	command = command + 5533;	1	Test passed	32768	8192	15
prodcell.cc	1	3	8	command	string	7	58	command = command + 5533;	1	Test passed	32768	8192	15

Figure 29 - Extract of FITMVS output for Channel B

7.3 Sensitivity Metrics

Despite the large quantity of raw results, it is possible to derive a large number of different metrics and analyses. One of these metrics is that of *sensitivity*; this is the percentage probability that a channel will fail to successfully pass a test after a fault is injected into it. For example, in one injection cycle, 295 tests were performed on Channel A, of which 45 resulted in either a failure or a timeout. Therefore, the sensitivity of the channel to a fault in that particular injection cycle is (100 / 295) x 45 = 15.25424%.

This calculation is performed for each injection cycle performed on both channels; these results are shown in figure 30. Each row represents a complete injection-cycle; "procName" refers to the name of the channel, "PD" refers to the perturbation distribution, "G-SD" refers to the standard deviation of the Gaussian distribution (if applicable), and "Sensitivity" is the percentage chance of a test failing as a result of a fault being added. The final two columns in each table refer to the standard deviation of the sensitivity values (not to be confused with the Gaussian distribution's standard deviation) and the average sensitivity for each set of 5 injection-cycles respectively.

As can be seen, there is a clear distinction (i.e. no overlap) between the average sensitivity values of the two channels; channel A has a sensitivity of approximately 20%, whilst channel B has a sensitivity of approximately 14.5%. The standard deviation of the sensitivity results for both channels is small, with channel A having a standard deviation of 1.3 and channel B of 0.3; it can therefore be seen that both channel's sensitivity values are relatively accurate. These sensitivity measures fit in well with what is already know about the dependability of the two channels as a result of previous studies [TOW01a, TOW01b]; namely, that channel A is error-prone (failing in approximately 25% of all possible situations), whilst channel B is far more dependable (failing on approximately 1.5% of all possible situations).

However, there appears to be no pattern amongst the sensitivity results for individual injection cycles performed within the channels themselves. Although tests using Gaussian distributions with different standard deviations were performed, it can be seen that for this application, the differences in sensitivity for each set of tests are very similar and clearly overlap when the standard deviations of the results are taken into account. It therefore appears to be the case that either the different distributions have no bearing on the sensitivity of the MVD system tested, or the number of tests performed is not great enough to establish the resolution necessary for identifying a possible relationship. Diagrams showing the average sensitivity for each set of five injection cycles performed in each channel are shown in figure 31.

procName	PD	G-SD	Sensitivity	SD	Average
Channel A	32768	8192	15.25424		
Channel A	32768	8192	24.91909		
Channel A	32768	8192	19.34426		
Channel A	32768	8192	19.6825		
Channel A	32768	8192	20	3.43229	19.84002
Channel A	32768	4096	24.29022		
Channel A	32768	4096	20.63492		
Channel A	32768	4096	25.23077		
Channel A	32768	4096	19.0476		
Channel A	32768	4096	15.9874	3.80095	21.03818
Channel A	32768	2048	21.93548		
Channel A	32768	2048	16.19048		
Channel A	32768	2048	21.29032		
Channel A	32768	2048	16.129		
Channel A	32768	2048	14.9206	3.26069	18.09318
Channel A	32768	1	20.73579		
Channel A	32768	1	19.72318		
Channel A	32768	1	25.53846		
Channel A	32768	1	17.74194		
Channel A	32768	1	23.49206	3.08736	21.44629
Channel A	32768	None	21.84615		
Channel A	32768	None	20.96774		
Channel A	32768	None	16.8254		
Channel A	32768	None	21.5873		
Channel A	32768	None	21.63009	2.1194	20.57134

SD of SDs:	0.628739061
Average SD:	3.140140299
SD of Averages:	1.319274278
Average of averages:	20.1977996

procName	PD	G-SD	Sensitivity	SD	Average
Channel B	32768	8192	12.7778		
Channel B	32768	8192	14.07407		
Channel B	32768	8192	16.11111		
Channel B	32768	8192	12.5925		
Channel B	32768	8192	17.037	1.98848	14.5185
Channel B	32768	4096	14.81481		
Channel B	32768	4096	13.33333		
Channel B	32768	4096	16.48148		
Channel B	32768	4096	16.1111		
Channel B	32768	4096	13.4328	1.46295	14.8347
Channel B	32768	2048	13.40782		
Channel B	32768	2048	14.62963		
Channel B	32768	2048	12.40741		
Channel B	32768	2048	15.92593		
Channel B	32768	2048	15.9259	1.55296	14.45934
Channel B	32768	1	11.66667		
Channel B	32768	1	16.11111		
Channel B	32768	1	15.58442		
Channel B	32768	1	10.37037		
Channel B	32768	1	16.48148	2.81666	14.04281
Channel B	32768	None	13.72913		
Channel B	32768	None	16.2963		
Channel B	32768	None	11.50278		
Channel B	32768	None	15.95547		
Channel B	32768	None	16.85185	2.22374	14.86711

SD of SDs:	0.548860549
Average SD:	2.008960344
SD of Averages:	0.33463212
Average of averages:	14.5444908

Figure 30 - Sensitivity results for both MVD channels

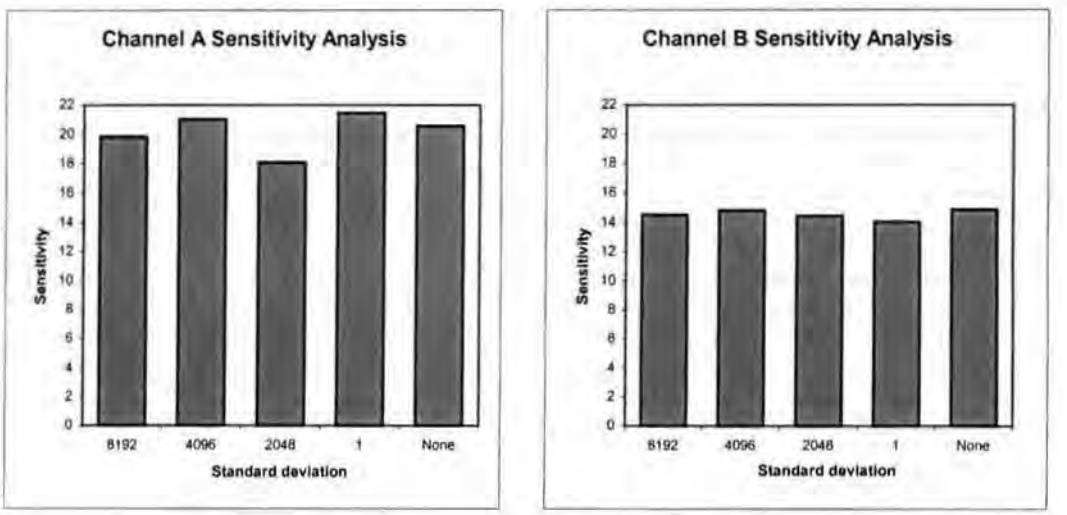


Figure 31 - Sensitivity results for each set of injections

7.4 Sensitivity to Common-mode Failure

Although these results are of interest, of even more interest is gauging the sensitivity of the channels within the overall MVD system to common-modal failure. This is calculated by analysing the results of each injection-cycle, identifying the scope of each test that resulted in error, and categorizing this based on the error description; this was done for both channels. For each channel, the number of failures for each error description were then calculated as a percentage of the total number of tests performed on that channel. It is important to note that because each channel has a different number of scopes, *the total number of tests performed on each channel are different.* For each error type, the percentages for each channel were divided by 100 and multiplied together to gain the percentage chance of common-mode failure for that error type within that injection-cycle. By collating these resultant common-mode probabilities, the overall probability of a common-mode failure occurring within that injection-cycle as a result of faults being injected can be discovered.

The results for every injection cycle are shown below; header of each table lists the standard deviation passed into the Gaussian function for that particular test (or "none" if a normal distribution was used), the name of the channel, and the number of tests performed on that channel (in brackets). Following this, the first column in each table lists the error that was observed, and the second and third columns refer to the number of the code scope in which injections were made to cause the error; the number in brackets in the second and third columns is the percentage probability that this error will occur on any given injection within the injection cycle. The fourth column gives the probability value (between 0 and 1) that the relevant error will manifest itself following injections in both channels; this is calculated by :

$$Pab = \frac{Pa}{100} \times \frac{Pb}{100}$$

where *Pa* is the percentage chance of channel A failing with the relevant error, *Pb* is the percentage chance of Channel B failing with the relevant error, and *Pab* is the probability (between 0 and 1) that both channels will fail with the relevant error at the same time. The bottom row in the table gives the sum of the probabilities calculated in column 4; this is the overall probability that for any random injection into channel A and channel B, the same error will manifest itself in both (i.e. a common mode failure). This value is multiplied by 100 and placed in brackets to give the percentage figure.

For example, the following table is the result of an injection cycle that perturbed variables based on a Gaussian distribution with a standard deviation of 8192, with Channel A having been subjected to 295 tests, and Channel B subjected to 540 tests:

Standard Deviation: 8192	Channel A (295)	Channel B (540)	
Crane One dropped blank	25, 25, 25, 25, 25, 25, 25, 25 **(2.71186%)**	13,13, 58, 58, 58 **(0.92592%)**	0.00025
Workstation 1 – blank exceeded time limit	13, 17, 26, 26, 26, 26, 28, 60 **(2.71186%)**	52, 53, 54, 63, 64, 73 **(1.11111%)**	0.00030
			0.00055 (0.055%)

In this test, two errors with the potential for common mode failure were discovered; one error manifests itself by crane one dropping a blank, and the other manifests itself by a blank placed in workstation 1 exceeding its time limit. For Channel A, the first error - "Crane One dropped blank" – was seen 8 times, all as a result of injections into scope 25 of the Channel's code. As 295 tests had been performed, this leads to a *100 / 295 x 8 = 2.71186%* chance that this error will be seen on any given injection. The same error was seen in Channel B five times; two times following an injection into scope 13 and three times following an injection into scope 58. This leads to a *100 / 540 x 5 = 0.92592%* chance of the error being seen on any given injection. The overall probability that following random injections both channels will manifest the same error is therefore

$$\frac{2.71186}{100} \times \frac{0.92592}{100} = 0.00025$$

The same process is repeated for the other error with the potential for common-mode failure – "Workstation 1 – blank exceeded time limit", where there is a 2.71186% chance that the error will be seen in Channel A on any given injection, a 1.11111% chance that the error will be seen in Channel B on any given injection, and an overall probability of 0.00030 that the error will be seen in both channels when random injections are made into each channel. When the two probabilities for common-mode failure are summed together, an overall probability for common-mode failure of 0.00055 is established. By multiplying this by 100, an overall percentage probability of 0.055% is obtained. The results for all other injection cycles performed are listed in appendix A, in the same format.

This data is summarized in figure 32. As can be seen, these results are very promising; out of more than 20,000 tests performed, despite faults being injected into the system, the probability of common-mode failure occurring is only approximately 0.049% with a standard deviation of approximately 0.035, with the "best" result being a probability of 0.005% and the "worst" results being a percentage chance of common-mode failure of 0.115%. However, it is important to remember that the results collected from the FITMVS

system do not allow for any non-independence of error weightings (i.e. related errors in two separate channels) to be taken into consideration.

G-SD	% chance of CMF	SD	Average
8192	0.055		
8192	0.114		
8192	0.0275		
8192	0.036		
8192	0.115	0.04227	0.0695
4096	0.051		
4096	0.075		
4096	0.095		
4096	0.049		
4096	0.005	0.033734	0.055
2048	0.016		
2048	0.026		
2048	0.071		
2048	0.008		
2048	0.024	0.024536	0.029
1	0.02		
1	0.015		
1	0.0532		
1	0.096		
1	0.0845	0.036645	0.05374
None	0.108		
None	0.028		
None	0.021		
None	0.028		
None	0.0195	0.037713	0.0409

Overall average: **0.049628**
Overall SD: **0.035284**

Figure 32 - Overall analysis of common-mode failure

Although for this experiment there is no obvious way to generate related errors amongst diverse channels, should we assume that doing so results in the probability for common-mode failure increasing by 20 fold (slightly more than the factor [HAT97] hypothesized for the [KNI86] experiment) then results still seem to be promising – with a worst-case probability of $0.115 \times 20 = 2.3\%$ chance of common-mode failure should a random fault be injected in each channel.

It is important to note, however, that this analysis of potential common-mode failure does not take into account any tests that resulted in a time-out; in other words, a situation in which both channels fail to reply within the expected period of time is not regarded as common-mode failure. This is due to the sheer volume of timeouts reported; for Channel A,

a total of 1220 time-outs occurred from 7812 tests performed, whilst Channel B produced a total of 1291 time-outs from 13399 tests. Figure 33 details the percentage probability of a timeout occurring on a given test for each injection cycle. The first column refers to the standard deviation of the Gaussian distribution (if applicable), the second and third columns detail the percentage probability of a test resulting in a time-out for Channel A and Channel B respectively, and the fourth column shows the probability that both channels will time-out on any given test.

SD	Channel A %	Channel B %	% Common Timeout
8192	7.79661	7.40741	0.57753
8192	12.13115	8.33333	1.01093
8192	20.06472	8.88889	1.78353
8192	16.19048	9.44444	1.52910
8192	15.55556	12.22222	1.90123
4096	17.46032	11.85185	2.06937
4096	18.61199	9.44444	1.75780
4096	14.76923	9.81481	1.44957
4096	14.92063	10.55556	1.57496
4096	14.10658	10.63433	1.50014
2048	19.67742	10.80074	2.12531
2048	18.38710	8.14815	1.49821
2048	12.69841	10.66667	1.35450
2048	14.83871	11.85185	1.75866
2048	12.69841	11.66667	1.48148
1	13.04348	9.46197	1.23417
1	21.84615	6.85185	1.49687
1	16.26298	11.48148	1.86723
1	9.35484	5.92593	0.55436
1	16.50794	8.51852	1.40623
Normal Distribution	12.00000	11.11111	1.33333
Normal Distribution	14.92063	6.86456	1.02424
Normal Distribution	18.70968	9.83302	1.83973
Normal Distribution	19.68254	10.74074	2.11405
Normal Distribution	17.55486	8.53432	1.49819
Standard Deviation:	3.41766	1.74360	0.41147
Average:	**15.59161%**	**9.64219 %**	**1.50962%**

Figure 33 - Analysis of time-out probabilities

In order to determine whether or not each time-out result will cause a common-mode failure, it would be necessary to look-up the injection in the FITMVS log file, manually perform this injection on the channel source code, compile and execute that code, and then manually observe the operation of the channel up to the point where a time-out occurs. Even if this process were to only take 5 minutes, this would still require $2511 \times 5 =$ 12,555 minutes (209.25 hours) of testing time, which is not feasible for this experiment.

However, as can be seen from figure 33, the average probability of both channels timing out on a given test is 1.50962%. If we are to assume that all time-outs lead to common-mode failure (an extremely unlikely assumption), then summing this probability with the average probability of common-mode failure shown in figure 32 would still lead to an average probability of common-mode failure of only 1.559248% (ignoring any weighting for related errors).

7.5 Sensitivity to Error of Each Program Scope

In addition to measures with regard to sensitivity and common-modal failure, the FITMVS log results also give an indication as to the sensitivity to error of each scope within the source code tested. Figure 34 shows the number of errors detected following injections into each scope in the two channels tested; this is created by grouping together all the rows of each FITMVS log file that contained an error message, and then creating a histogram graph based upon the scope number of the injection. This analysis does *not* include time-outs.

These results are of interest as they reveal that certain program scopes are far more prone to error (and are hence far more sensitive) than other scopes. A good example of this is scope 51 in channel B, responsible for a total of 105 reported errors. This metric is very useful as it provides a picture of the sensitivity of each channel's source code that can be assessed very quickly. By identifying scopes of special sensitivity and testing/coding them to behave more robustly, it should be possible to reduce the overall sensitivity of each channel significantly.

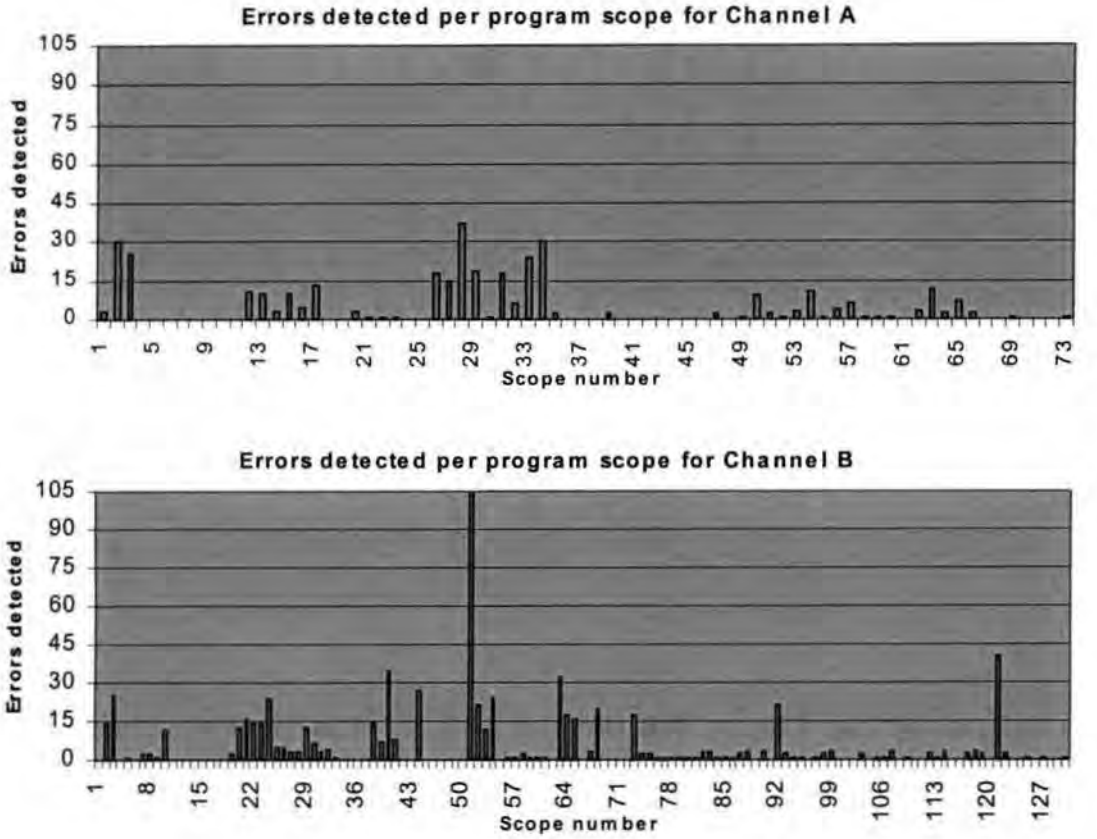


Figure 34 - Errors detected per program scope for both channels tested

The same analysis can also be performed to see which scopes have a high sensitivity toward common-mode failure; this is shown in figure 35. This is created by grouping together all errors in each FITMVS log file that had the potential for common-mode failure as described in section 7.4.

As can be seen, the results of this analysis show little change from the results displayed in figure 34, but may help in further refining the overall picture of each channel's sensitivity. It should be noted that the scale of the vertical axis in figure 34 and figure 35 is different, as the set of errors with the potential to be common-mode failures was smaller than that of set of all errors.

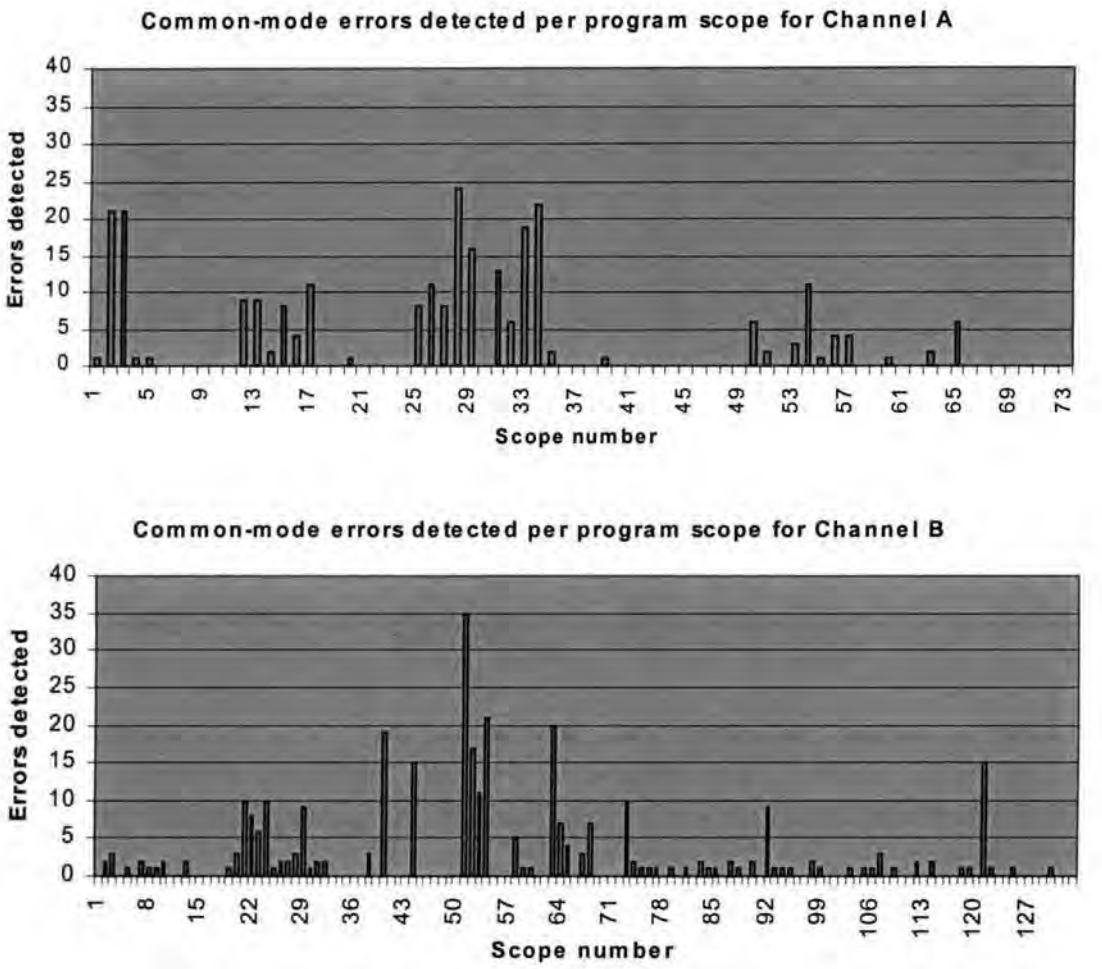


Figure 35 - Common-mode failures detected per program scope for both channels tested

7.6 Error Frequency Analysis

Finally, an analysis was performed to see what types of error occurred most frequently in each channel. This is created by grouping together all the rows of each FITMVS log file that contained an error message, sorting them based on the error description, and then counting each group of errors. The results of this analysis are shown in figure 36.

This analysis reveals that some specific types of error occur far more often than others; in Channel A, error types 6, 8, and 12 occur with most frequency, whilst in Channel B, error types 1, 9, 13, 19 and 20 occur with most frequency.

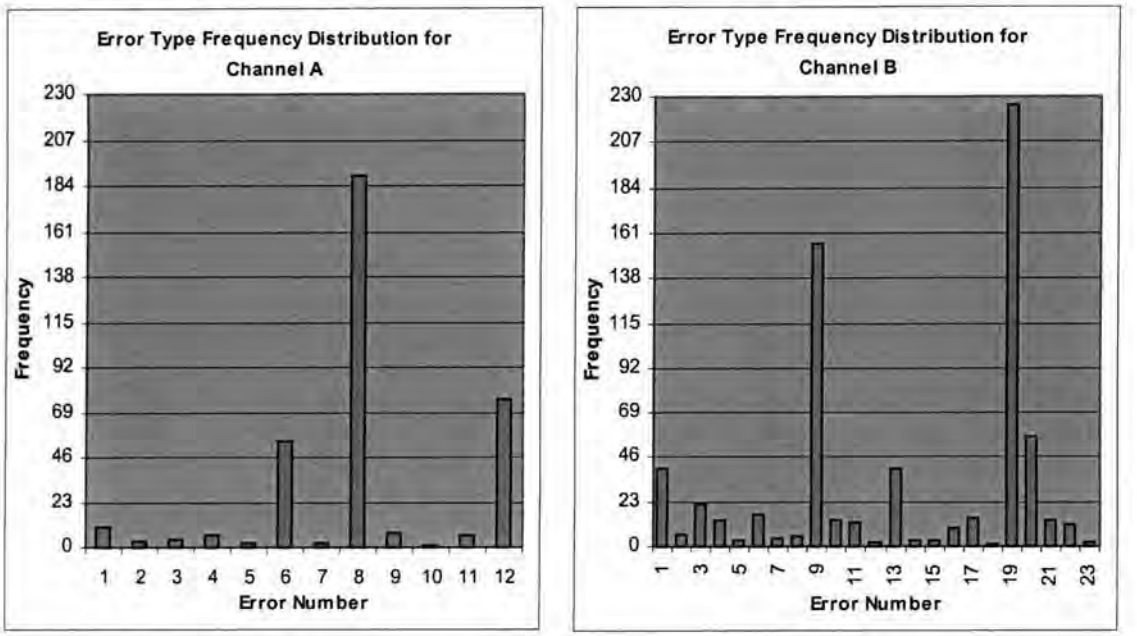


Figure 36 - Error type frequency for both MVD channels

A detailed breakdown of each error type is given for both channels in figure 37 and figure 38.

#	Error Description	Frequency
1	Blank in WS One picked up before minimum time elapsed	10
2	Blank in WS Three picked up before minimum time elapsed	3
3	Blank in WS Two picked up before minimum time elapsed	4
4	Blank passed through system, but exceeds maximum system time	6
5	Blank processed at too few workstations	2
6	Crane one dropped blank - Factory::checkCraneMagnets()	54
7	Crane two dropped blank - Factory::checkCraneMagnets()	2
8	Workstation 1 - blank exceeded time limit. Factory::checkWorkstations()	189
9	Workstation 2 - blank exceeded time limit. Factory::checkWorkstations()	7
10	Workstation 3 - blank exceeded time limit. Factory::checkWorkstations()	1
11	Workstation 4 - blank exceeded time limit. Factory::checkWorkstations()	6
12	Workstation used more than once	76

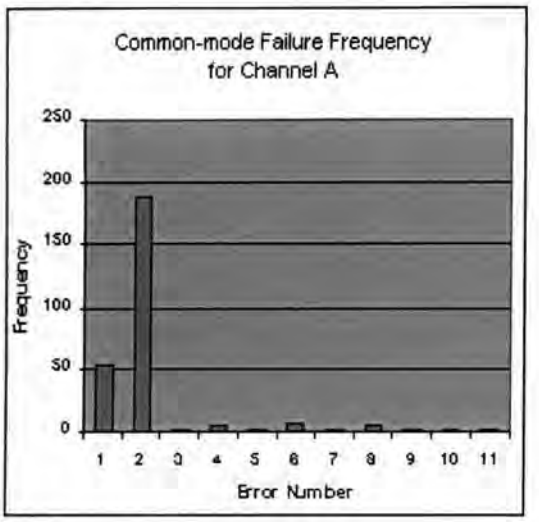
Figure 37 - Error type frequency breakdown for Channel A

#	Error Description	Frequency
1	Blank in WS One picked up before minimum time elapsed	40
2	Blank in WS Three picked up before minimum time elapsed	6
3	Blank in WS Two picked up before minimum time elapsed	22
4	Blank passed through system, but exceeds maximum system time	13
5	Blank processed at more than 4 workstations	3
6	Blank processed at too few workstations	16
7	Blank put back down on the end of the feedbelt	4
8	Blanks processed out of order	5
9	Crane one dropped blank - Factory::checkCraneMagnets()	156
10	Crane one has put a blank into workstation 1. It already has a blank in it	13
11	Crane one has put a blank into workstation 2. It already has a blank in it	12
12	Crane one has put a blank into workstation 4. It already has a blank in it	2
13	Crane two dropped blank - Factory::checkCraneMagnets()	40
14	Crane two has put a blank into workstation 1. It already has a blank in it	3
15	Crane two has put a blank into workstation 2. It already has a blank in it	3
16	Crane two has put a blank into workstation 3. It already has a blank in it	9
17	Crane two has put a blank into workstation 4. It already has a blank in it	14
18	hasBlankExceededLimit: blank inside illegal workstation	1
19	Workstation 1 - blank exceeded time limit. Factory::checkWorkstations()	226
20	Workstation 2 - blank exceeded time limit. Factory::checkWorkstations()	57
21	Workstation 3 - blank exceeded time limit. Factory::checkWorkstations()	13
22	Workstation 4 - blank exceeded time limit. Factory::checkWorkstations()	11
23	Workstation used more than once	2

Figure 38 - Error type frequency breakdown for Channel B

From this analysis, the system developer may wish to more thoroughly exercise exception handling mechanisms related to these errors, in order to increase the safety of the system as much as possible. It will also be possible to use an analysis such as this to rank common-mode failures by their *severity* and also count the number of common-mode failures that result in system failure, thus providing more MVD metrics

A related analysis to the one mentioned above is to assess the frequency of common-mode failures in the two channels; that is, the frequency of errors with the potential to lead to common-mode failure. This is shown in figure 39. Figure 40 details a breakdown of the common-mode failure frequency data.



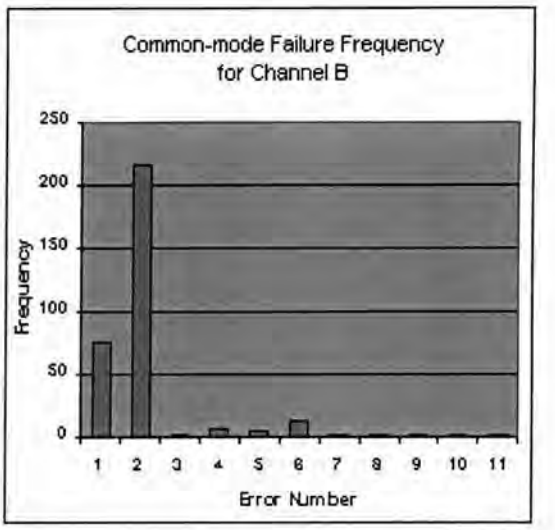


Figure 39 - Common-mode failure Frequency in Channel A and Channel B

#	Error Description	Channel A Frequency	Channel B Frequency	TOTAL
1	Crane One Dropped Blank	54	75	129
2	WS 1 - blank exceeded time limit	189	215	404
3	Blank passed through system but exceeds Max Sys Time	2	1	3
4	Blank in WS1 picked up before min time elapsed	4	7	11
5	Blank in WS2 picked up before min time elapsed	2	5	7
6	WS 2 - blank exceeded time limit	6	12	18
7	Blank processed at too few workstations	2	2	4
8	Workstation used more than once	4	2	6
9	WS 3 - blank exceeded time limit	1	1	2
10	WS 4 - blank exceeded time limit	1	1	2
11	Blank in WS3 picked up before min time elapsed	1	1	2

Figure 40 - Common-mode failure type frequency breakdown for Channel A and Channel B

It can also be seen that two of the errors – "Crane One Dropped Blank" and "Workstation 1 – Blank Exceeded Time Limit" – occur frequently in both channels. The exact reason as to why this is the case will require further investigation at the source code level, but nevertheless this analysis gives developers extremely useful information to investigate.

7.7 Issues with FITMVS Arising from the Experiment

At the conclusion of the experiment, a number of limitations with the current FITMVS system were apparent. The system does not recognise objects, and hence can only perturb primitive variable types, not objects or class variables. An example of this is shown in figure 41.

```
void function()
{
      int a;
      long b;
      Object theObject = new Object();

      // FITMVS can perturb either primitive
      // variable, such as :

      A = A + 43;

      // or

      B = B + 20;

      // but does not recognize objects and so
      // could not, for example, do as follows:

      theObject->variable = theObject->variable + 20;

}
```

Figure 41 - Code example of what FITMVS can and cannot perturb

The reason for this is the lack of sophistication in the FITMVS parser components, stemming from the lack of development time available. In an age of object-oriented technologies, this is obviously an issue that will need to be addressed in the future, as many potential perturbations were ignored by the system and an even greater insight into the two channels may have been missed.

Another issue to arise as a result of the experiment is that of the "time-out problem". In order to resolve whether or not time-outs will produce common-mode failure across channels, it is currently the case that the user must manually study the FITMVS log file, manually perform the specified injection, and then manually evaluate the execution of the channel. Although this is possible for a small number of time-outs, as noted in section 7.4, an

average of just over 10% of tests performed resulted in a time-out, and so such a manual analysis is unfeasible. An investigation is therefore needed into alternate methods for analysing time-outs between channels.

Perhaps the most profound problem of all is the inability of FITMVS (and perhaps the fault injection approach as a whole) to accurately model non-independence of failure. Every experimental analysis of MVD systems has shown that the probabilities of channels in MVD systems failing are not dependant of each other, although no research appears to have been performed on modelling this relationship between channels. Due to the fact that different channels will have different variable names, different structures, different functions and different objects, it is simply not possible to insert the "same" fault into more than one channel (unless perturbing input data). Therefore, all injections performed are completely independent of each other and so a non-independence relationship cannot be established between channels.

7.8 Summary

This chapter details the results of the experiment performed, together with an analysis as to what these results mean. Examples of the FITMVS log files produced by the experiment are shown, analyses are performed to give channel sensitivity analysis, channel sensitivity to common-mode failure, program scope sensitivity analysis, program scope sensitivity to common-mode failure, error frequency distribution analysis and common-mode failure frequency distribution analysis. The chapter concludes by examining issues that arose with the FITMVS system as a result of the experiment.

Chapter 8 Conclusions and Future Work

8.1 Conclusions

The primary goal of this research has been to develop a system capable of automatically injecting faults into an MVD system and then testing the system for its behaviour. Such a system is desirable as multi-version design has been proposed as a method for increasing the dependability of critical systems beyond current levels, but lack of quantitative characterizations is a major obstacle to large-scale commercial usage of the approach. The technique of fault injection provides much potential for generating large numbers of metrics. Fault injection is a "late life-cycle" software analysis that can simulate human operator errors and observe their impact on the software as well as the total system. It is a technique that *complements*, but is not a substitute for, other verification and validation procedures. By developing a fault injection system (FITMVS), it was hoped to provide a method for generating large amounts of data about both an MVD system as a whole, as well as its constituent channels.

The result of this has been very successful, and as a result, not only has a valuable tool for the production of detailed metrics into MVD systems been produced, but extremely useful metrics about a known MVD system have been produced also. The automated nature of the FITMVS system has also allowed for a much greater number of tests to be performed than might otherwise have been the case (21,211 tests automatically performed compared to the 4,320 tests performed manually over a much greater time period in a previous study). The following analyses can be produced using the FITMVS system:

1) *Channel Sensitivity Analysis*. This metric allows the user to gauge how likely a channel within an MVD system is to fail when a fault is injected into it. The user may then wish to invest more resources in channels with a high sensitivity to faults.

2) *Channel Sensitivity to Common-mode failure*. This metric is related to channel sensitivity analysis, but applies to the MVD system as a whole. This analysis is useful as it helps to refine dependability estimates for a MVD system by giving the user an indication of how likely the system is to

fail through common-mode failure, assuming a single random fault is injected into each of its constituent channels.

3) *Program Scope Sensitivity Analysis.* This analysis generates a graph showing the number of errors that were produced following injections into each scope within a channel's source code. This allows the user to assess at a glance which scopes are more sensitive to faults than others; the user may then wish to either perform increased tests on these scopes, debug them, or introduce more effective exception-handling routines in them.

4) *Program Scope Sensitivity to Common-mode failure Analysis.* This analysis is similar to the program scope sensitivity analysis, and produces a graph showing the number of errors with the potential for common-mode failure that were produced following injections into each scope within a channel's source code. A user may find this analysis helpful in assessing which scopes are in most urgent need for maintenance (assuming that the MVD system will be able to handle non-common-mode failures generated by scopes). This analysis may also be extremely useful in future research investigating the exact causes of the related-error phenomenon.

5) *Error Frequency Distribution Analysis.* This analysis measures the number of occurrences of each type of error reported during the course of testing by FITMVS. This analysis can help the user to detect which errors occur most frequently when a fault is present, and allows them the opportunity to allocate more resources to the development of exception-handling routines for these errors and/or investigate why the errors are so common.

6) *Common-mode Failure Frequency Distribution Analysis.* This analysis is similar to the error frequency distribution analysis, but measures the number of occurrences of each type of potential common-mode failure reported during testing. This enables the user to develop more effective exception-handling routines for the MVD system as a whole.

The MVD system tested was a trivial example, but nevertheless, the results gained are extremely satisfactory as a proof-of-concept, and show great promise, with the sensitivity to potential common-mode failure in particular being surprisingly low, whilst the sensitivity metrics for each channel appear to confirm earlier tests [TOW01a, TOW01b] into their

relative dependabilities which established channel B as being the more dependable channel. The program scope metrics were successful in establishing specific scopes in both channels with disproportionate sensitivity results, whilst the error type frequency analysis revealed a number of errors that were far more common than others when faults were injected into either channel. The common-mode failure type frequency analysis was also very useful, as it isolated two types of error that were by far the most likely to occur in the event of a common-mode failure.

The MVD system chosen as a test example required several seconds to perform each test, and so the total number of tests that could be performed was limited; other applications may not have this speed restriction, and hence much higher numbers of tests may be performed and the resulting statistics may have a more fine-grained resolution.

As has been stated earlier, the FITMVS system is very much a proof-of-concept system, but the potential for improvement in the future is great. The current system provides a method for extracting the much needed quantitative characterizations that are required by the fault-tolerant distributed-computing community [KIM00] and can therefore be considered to be very much a success.

8.2 Future Work

There is great potential for future work both on the implementation of FITMVS and the application of FITMVS. On the implementation side, perhaps the most pressing need is for a better parser. The current parser within the FITMVS system cannot handle objects, and can only parse C and C++. Improvements in the parser should also allow for a wider choice of possible injections; currently the FITMVS system only supports data value perturbation; however, one possible goal in the future is to provide the possibility of code mutation as well. Changes to the parser may include further work on the existing parser, or the replacement of the existing parser with a ready-made/commercial parser. Another improvement to the system would be the implementation of an analysis component; currently the system outputs a very detailed log file, but the actual metrics and analyses of this file have to be done semi-manually (the log file is tab-delimited and should import into most modern spreadsheet applications). By giving the user the option of automatic analysis of the output logs, the overall time taken to gain results should be much reduced.

There are also a number of promising research directions in which FITMVS may be helpful. The most profound of these is an investigation into related errors; currently, there is no understanding as to the relationship between errors and common-mode failures. By using the analyses offered by FITMVS, it may be possible to investigate relationships between

scopes that are more likely to cause common-mode failure, and perform reverse engineering to gain a greater understanding of the underlying causes. It is also of interest to analyse the results of FITMVS on other MVD systems in order to see if there are any underlying patterns or trends in the data extracted. Although the automated testing mechanism has increased the number of tests that were able to be performed significantly, the fact that the MVD system tested waited on the system timer prohibited a truly large number of tests from being performed, and therefore an alternative MVD system that does not wait on the system timer will enable a more rigorous analysis.

8.3 Acknowledgements

My thanks to my supervisor Jie Xu, and my mum and dad for all their support.

Appendix A

This appendix lists the results of the analysis for common-mode probabilities performed on every injection cycle. The format of these results is detailed in section 7.4.

Standard Deviation: 8192	**Channel A (315)**	**Channel B (540)**	
Workstation 1 – blank exceeded time limit	12, 13, 14, 15, 17, 34, 34, 50, 50, 56 **(3.17460%)**	44, 52, 53, 54, 63, 121 **(1.11111%)**	0.00035
Workstation 2 – blank exceeded time limit	34 **(0.31746%)**	65, 92 **(0.37037%)**	0.00001
			0.00036 (0.036%)

Standard Deviation: 8192	**Channel A (305)**	**Channel B (540)**	
Blank in Workstation 1 picked up before minTime elapsed	29, 29, 29 **(0.98360%)**	19, 25, 52, 95, 106, 112 **(1.11111%)**	0.00010
Blank in Workstation 2 picked up before minTime elapsed	29 **(0.32786%)**	32, 81 **(0.37037%)**	0.00001
Crane One dropped blank	2, 2, 2, 2, 3, 3, 3, 3 **(2.62295%)**	31, 31, 32, 51, 51, 51, 51, 51 **(1.48148%)**	0.00038
Workstation 1 – blank exceeded time limit	12, 13, 27, 27, 27, 27, 34, 34, 34 **(2.95081%)**	7, 7, 30, 38, 44, 52, 53, 54, 63, 68, 73, 121 **(2.22222%)**	0.00065
			0.00114 (0.114%)

Standard Deviation: 8192	**Channel A (309)**	**Channel B (540)**	
Blank in workstation 1 picked up before time limit expired	28 **(0.32362%)**	44 **(0.18518%)**	0.000005
Crane One dropped blank	2 **(0.32362%)**	29, 29, 40, 40, 40, 40, 40, 40, 51, 51, 51, 51, 51 **(2.22222%)**	0.00007
Workstation 1 – blank exceeded time limit	15, 54, 54, 54, 56 **(1.61812%)**	24, 24, 24, 52, 54, 63, 94 **(1.29629%)**	0.00020
			0.000275 (0.0275%)

Standard Deviation: 8192	Channel A (315)	Channel B (540)	
Workstation 1 – blank exceeded time limit	15, 17, 28, 28, 28, 28, 33, 33, 33, 33, 34, 34, 51 **(4.12698%)**	24, 24, 24, 29, 29, 52, 53, 54, 58, 63, 64, 83, 84, 98, 98 **(2.77777%)**	0.00114
Workstation 2 – blank exceeded time limit	34 **(0.31746%)**	65, 92, 121 **(0.55555%)**	0.00001
			0.00115 (0.115%)

Standard Deviation: 4096	Channel A (315)	Channel B (540)	
Workstation 1 – blank exceeded time limit	12, 17, 28, 28, 28, 31, 31, 31, 31 **(2.85714%)**	44, 52, 54, 73, 93, 107, 114, 119, 121 **(1.66666%)**	0.00047
Workstation 2 – blank exceeded time limit	15, 28 **(0.63492%)**	64, 121 **(0.37037%)**	0.00002
			0.00049 (0.049%)

Standard Deviation: 4096	Channel A (319)	Channel B (536)	
Crane one dropped blank	2, 3 **(0.62695%)**	51 **(0.18656%)**	0.00001
Workstation 1 – blank exceeded time limit	12, 35 **(0.62695%)**	44, 52, 54, 114 **(0.74626%)**	0.00004
			0.00005 (0.005%)

Standard Deviation: 4096	Channel A (315)	Channel B (540)	
Workstation 1 – blank exceeded time limit	13, 16, 32, 32, 34, 34, 34, 57 **(2.53968%)**	22, 24, 24, 24, 44, 54, 65, 68, 112, 121,130 **(2.03703%)**	0.00051
			0.00051 (0.051%)

Standard Deviation: 4096	Channel A (317)	Channel B (540)	
Crane One dropped blank	2 **(0.31545%)**	40, 40, 40, 40, 40, 51, 51, 51, 51, 51 **(1.85185%)**	0.00005
Workstation 1 – blank exceeded time limit	12, 14, 15, 17, 31, 31, 31, 31, 34, 51, 56 **(3.47003%)**	2, 2, 22, 22, 22, 44, 53, 54, 63, 68, 92 **(2.03703%)**	0.00070
			0.00075 (0.075%)

Standard Deviation: 4096	**Channel A (325)**	**Channel B (540)**	
Crane One dropped blank	2, 2, 2, 2, 3, 3, 3 **(2.15384%)**	21, 21, 51, 51, 51, 51, 51 **(1.29629%)**	0.00027
Workstation 1 – blank exceeded time limit	12, 26, 32, 33, 34, 34, 34, 54, 54, 54, 54, 56 **(3.69230%)**	27, 44, 52, 54, 63, 64, 67, 73, 92, 107 **(1.85185%)**	0.00068
			0.00095 (0.095%)

Standard Deviation: 2048	**Channel A (315)**	**Channel B (540)**	
Workstation 1 – blank exceeded time limit	15, 16, 28, 28, 28, 28, 29, 34, 34, 34 **(3.17460%)**	3, 3, 9, 53, 54, 63, 68, 73, 79, 92, 103, 107 **(2.22222%)**	0.00070
Workstation 2 – blank exceeded time limit	17 **(0.31746%)**	8, 27 **(0.37037%)**	0.00001
			0.00071 (0.071%)

Standard Deviation: 2048	**Channel A (310)**	**Channel B (540)**	
Workstation 1 – blank exceeded time limit	17, 57 **(0.64516%)**	53, 54, 63, 64, 73, 74, 109 **(1.29629%)**	0.00008
			0.00008 (0.008%)

Standard Deviation: 2048	**Channel A (315)**	**Channel B (540)**	
Workstation 1 – blank exceeded time limit	3, 3, 3, 4, 5 **(1.58730%)**	5, 20, 20, 20, 44, 52, 54, 63 **(1.48148%)**	0.00023
Workstation used more than once	26, 26 **(0.63492%)**	122 **(0.18518%)**	0.00001
			0.00024 (0.024%)

Standard Deviation: 2048	**Channel A (310)**	**Channel B (537)**	
Workstation 1 – blank exceeded time limit	12, 29, 29, 29, 29, 35, 57 **(2.25806%)**	54, 63, 64, 68 **(0.74487%)**	0.00016
			0.00016 (0.016%)

Standard Deviation: 2048	**Channel A (310)**	**Channel B (540)**	
Workstation 1 – blank exceeded time limit	13, 28, 28, 28, 28, 33, 33, 33, 33 **(2.90322%)**	44, 52, 54, 63, 105 **(0.92592%)**	0.00026
			0.00026 (0.026%)

Standard Deviation: 1	**Channel A (289)**	**Channel B (540)**	
Blank processes at too few workstations	50 **(0.34602%)**	38 **(0.18518%)**	0.000006
Crane one dropped blank	2 **(0.34602%)**	51 **(0.18518%)**	0.000006
Workstation 1 – blank exceeded 1	26, 33, 33, 33, 55 **(1.73010%)**	21, 21 ,21, 23, 23, 23, 28, 28, 28, 52, 54, 60, 63, 67, 68, 121 **(2.96296%)**	0.00051
Workstation used more than once	63, 63 **(0.69204%)**	64 **(0.18518%)**	0.00001
			0.000532 (0.0532%)

Standard Deviation: 1	**Channel A (310)**	**Channel B (540)**	
Workstation 1 – blank exceeded time limit	26, 26, 26, 28, 28, 28, 27, 29, 29, 29, 34, 34, 34, 54, 54, 54, 54, 57 **(5.80645%)**	21, 23, 24, 52, 54, 68, 73, 92, 121 **(1.66666%)**	0.00096
			0.00096 (0.096%)

Standard Deviation: 1	**Channel A (315)**	**Channel B (540)**	
Blank processed at too few workstations	53 **(0.31746%)**	26 **(0.18518%)**	0.000005
Workstation 1 – blank exceeded time limit	12, 12, 29, 29, 29, 29, 50, 53, 65, 65, 65, 65, 65 **(4.12698%)**	10, 10, 53, 58, 67, 73, 77, 87, 88, 121, 125 **(2.03703%)**	0.00084
			0.000845 (0.0845%)

Standard Deviation: 1	**Channel A (299)**	**Channel B (539)**	
Blank passed through system but exceeds max time	39, 65 **(0.66889%)**	118 **(0.18552%)**	0.00001
Workstation 1 – blank exceeded time limit	13, 32, 33, 33 **(1.33779%)**	22, 29, 29, 29, 38, 59, 63, 121 **(1.48423%)**	0.00019
			0.00020 (0.02%)

Standard Deviation: 1	**Channel A (325)**	**Channel B (540)**	
Crane One dropped blank	2, 2, 2, 2, 3, 3, 3, 3, 3 **(2.76923%)**	51, 51, 51 **(0.55555%)**	0.00015
			0.00015 (0.015%)

Normal Distribution	**Channel A (310)**	**Channel B (539)**	
Workstation 1 – blank exceeded time limit	16, 17, 31, 32, 32 **(1.61290%)**	21, 21, 21, 26, 52, 53, 63 **(1.29870%)**	0.00020
Workstation 2 – blank exceeded time limit	13 **(0.32258%)**	65, 92, 121 **(0.55658%)**	0.00001
			0.00021 (0.021%)

Normal Distribution	**Channel A (315)**	**Channel B (540)**	
Workstation 1 – blank exceeded time limit	13, 16, 27, 27, 27, 27 **(1.90476%)**	21, 29, 29, 44, 52, 53, 54, 63 **(1.48148%)**	0.00028
			0.00028 (0.028%)

Normal Distribution	**Channel A (319)**	**Channel B (539)**	
Blank in workstation 3 picked up before minimum time elapsed	28 **(0.31347%)**	85 **(0.18552%)**	0.000005
Blank in workstation 2 picked up before minimum time elapsed	28 **(0.31347%)**	63, 121, 121 **(0.55658%)**	0.00001
Crane one dropped blank	2, 3 **(0.62695%)**	23, 23, 40, 40, 40, 40, 51, 51, 51, 51, 51 **(2.04081%)**	0.00012
Workstation 1 – blank exceeded time limit	15 **(0.31347%)**	3, 44, 52, 54, 63, 73, 76, 83, 87, 90, 99 **(2.04081%)**	0.00006
			0.000195 (0.0195%)

Normal Distribution	**Channel A (315)**	**Channel B (539)**	
Workstation 1 – blank exceeded time limit	13, 17, 33, 33, 33, 33 **(1.90476%)**	22, 22, 22, 44, 53, 54, 63, 92 **(1.48423%)**	0.00028
			0.00028 (0.028%)

Normal Distribution	**Channel A (325)**	**Channel B (540)**	
Crane One dropped blank	2, 2, 2, 2, 3, 3, 3, 3 **(2.46153%)**	40, 40, 40, 40, 40, 51, 51, 51, 51 51 **(1.85185%)**	0.00045
Workstation 1 – blank exceeded time limit	15, 17, 17, 20, 31, 31, 31, 31, 50, 50, 53 **(3.38461%)**	44, 52, 54, 63, 73, 74, 75, 90, 92, 121 **(1.85185%)**	0.00062
Workstation 3 – blank exceeded time limit	33 **(0.30769%)**	121 **(0.18518%)**	0.000005
Workstation 4 – blank exceeded time limit	1 **(0.30769%)**	44 **(0.18518%)**	0.000005
			0.00108 (0.108%)

References

[AMM87] P.E. Ammann, J.C. Knight, "Data Diversity: an Approach to Software Fault Tolerance", in Proc. Seventeenth International Symposium on Fault-Tolerant Computing, p.122-126, Pittsburgh, 1987

[ATH89] A. Athavale, "Performance Evaluation of Hybrid Voting Schemes", M.S. Thesis, North Carolina State University, Department of Computer Science, December 1989

[AVI77] A. Avizienis, L. Chen "On the implementation of N-version programming for software fault tolerance during execution" *Proc. IEEE COMPSAC 77* p.149-155 November 1977

[AVI84] A. Avizienis, J. Kelly "Fault Tolerance by Design Diversity. Concepts and Experiments" *IEEE Computer* - Vol. 17 - No. 8 - August 1984 p. 67-80

[AVI85] A. Avizienis, "The N-version Approach to Fault-Tolerant Software" - *IEEE Transactions on Software Engineering* - vol. 11 1985

[AVI86] A. Avizienis, J.C. Laprie, "Dependable computing: from concept to design diversity," in *Proceedings of the IEEE*, p. 629-638, 1986

[AVI89] A. Avizienis, "Software Fault Tolerance" - *IFIP XI World Computer Congress '89* - San Fransisco - August 1989

[BUT93] R. W. Butler, G. B. Finelli, "The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software", *IEEE Transactions on Software Engineering*, vol SE19 no. 1, p. 3-12, January 1993

[CAR99] J. V. Carreira, D. Costa, J.G. Silva, "Fault Injection Spot-checks Computer System Dependability", IEEE Spectrum, p. 50-55, August 1999

[CHR94] J. Christmansson, Z. Kalbarczyk, "An Approach to Experimental Evaluation of Different Data Diversity Schemes", *Predictably Dependable Computing Systems second year report*, p. 685-716, September 1994

[CLA95] J.Clark, D. Pradhan, "Fault Injection: A Method for Validating Computer-System Dependability," in IEEE Computer, p. 47-56, June 1995

[CHE99] L. Chen, J. Napier, J. May, G. Hughes, "Testing the Diversity of Multi-version Software using Fault Injection," in Proc. Of the Safety and Reliability Society Symposium: Advances in Safety and Reliability, June 1999

[CRI82] F. Cristian, "Exception Handling and Software Fault Tolerance", IEEE Transactions on Computers, 31(6):531-540, 1982

[DEM78] R.A. DeMillo, R.J. Lipton, F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer", *IEEE Computer*, 11(4), p. 34-41, April 1978

[ECK85] D.E. Eckhardt, L.D. Lee "A theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors" - IEEE Transactions on Software Engineering - Vol SE-11 - No. 12 - December 1985 - p. 1511-1516

[ECK91] D.E. Eckhardt et al, "An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability" IEEE Transactions on Software Engineering Vol. 17 1991 pp 692-702

[GHO99] A. K. Ghosh, J. M. Voas, "Inoculating Software for Survivability", in Communications of the ACM 42(7), p. 38-44, July 1999

[GRA90] J. Gray, "A census of Tandem system availability between 1985 and 1990," in IEEE Transactions on Reliability, p.409-432, 1990

[HAL90] P. Hall, "Defect Detection and Correction" in "Software Reliability Handbook", Ed. P. Rook, Elsevier Science Published Ltd., 1990

[HAN95] S. Han, K.G. Shin, H.A. Rosenberg, "Doctor: An Integrated Software Fault-Injection Environment for Real-Time Systems," in Proc. Of the Second Annual IEEE Int. Computer Performance and Dependability Symp., pp/ 204-214, IEEE 1995

[HAT97] L. Hatton, "N-version Design Versus One Good Version" - *IEEE Software* Vol.14 No.6 1997 pp 71-76

[HEC93] H. Hecht, "Rare Conditions – An Important Cause of Failures", *Proc. COMPASS'93*, Gaithersburg MD, June 1993

[HEC94] H. Hecht, P. Crane, "Rare Conditions and their Effect on Software Failures", *Proc. of the 1994 Reliability and Maintainability Symposium*, p. 334-337, January 1994

[HEC96] H. Hecht, M. Hecht, "Qualitative Interpretation of Software Test Data", *Computer-Aided Design, Test and Evaluation for Dependability Workshop*, Beijing, China, July 1996

[HSU97] M-C. Hsueh, T. K. Tsai, R.K. Iyer, "Fault Injection Techniques and Tools", IEEE Computer, April 1997, p. 75-82.

[JAL00] P. Jalote, "Fault Tolerance in Distributed Systems", Prentice Hall, Englewood Cliffs NJ, 2000

[KEL88] J. Kelly, D.E. Eckhardt, A. Caglayan et al, "Large Scale Second Generation Experiment in Multi-Version Software: description and early results" *IEEE Fault Tolerant Computer Systems* Vol 18 - 1988 pp 9-14

[KER86] E. Keravnou, L. Johnson, "Competent Expert Systems", Kogan Page, London, 1986

[KIM84] K.H. Kim, "Distributed Execution of Recovery Blocks: An Approach to Uniform Treatment of Hardware and Software Faults", in Proc. 4th Int. Conf. On Dist. Comp. Sys, p. 577-632, San Francisco, May 1984

[KIM95] K.H. Kim, "The Distributed Recovery Block Scheme", in [LYU95], p. 189-209, 1995

[KIM00] K.H. Kim, "Issues Insufficiently Resolved in Century 20 in the Fault-Tolerant Distributed Computing Field," in Proc. 19th IEEE Symposium on Reliable Distributed Systems, October 2000 p. 106 - 115

[KIT90] B. Kitchenham, "Software Development Metrics and Models", in "Software Reliability Handbook", Ed. P. Rook, Elsevier Science Published Ltd., 1990

[KNI86] J. Knight, N. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multi-version Programming" *IEEE Transactions on Software Engineering* - vol. 12 1986 pp 96-109

[KNI90] J. Knight, N. Leveson, "A reply to the criticisms of the Knight and Leveson experiment" ACM Software Engineering Notes - January 1990

[LAD99] P. Ladkin et al, "Computer-related Incidents with Commercial Aircraft" http://www.rvs.uni-bielefeld.de/publications/Incidents/ 1999

[LAP90] J-C. Laprie et al, "Definition and Analysis of Hardware- and Software- Fault-Tolerant Architectures", *IEEE Computer vol. 23 no. 7*, July 1990

[LAP92] J-C. Laprie, (ed.). "Dependability: Basic concepts and terminology – in English, French, German, Italian and Japanese," in "*Dependable Computing and Fault Tolerance*" Vienna, Austria, Springer-Verlag, p. 265, 1992

[LAP95] J-C. Laprie, "Software Reliability and System Reliability", in [LYU95], p. 27-69, 1995.

[LEV95] N. Leveson, "Safeware: System Safety and Computers" - Addison-Wesley-Longman - New York - 1995

[LYU95] M.R. Lyu, "Software Fault Tolerance" - John Wiley & Sons - Chichester - UK - 1995

[MAC88] D. A. Mackall, "Development and Flight Test Experiences With a Flight-Crucial Digital Control System" *Technical Report NASA Technical Paper 2857* - National Aeronautics and Space Administration - Dryden Flight Research Facility - November 1988

[MAC91] D.F. McAllister, R.K. Scott, "Cost Modelling of Fault-Tolerant Software" - *Journal of Information and Software Technology* Vol. 33 no.8 October 1991 - p. 594-603

[MEL77] P.M. Melliar-Smith, B. Randell, "Software Reliability: the Role of Programmed Exception Handling", SIGPLAN Notices, 12(3):95-100, 1977

[MIL72] H.D. Mills, "On the statistical validation of computer programs," IBM Federal Systems Division, Gaithersburg, MD, Red. 72-6015, 1972

[MOR88] L. J. Morell, J. Voas, "Infection and Propagation Analysis: A Fault-Based Approach to Estimating Software Reliability", Technical Report WM-88-2, College of William and Mary in Virginia, Department of Computer Science, September 1988

[PRE97] R.S. Pressman, "Software Engineering: A Practitioners Approach", 4th edition, Addison-Wesley, 1997

[RAN75] B. Randell, "System structure for software fault tolerance," in *IEEE Transactions on Software Engineering*, 1(2):220-232, 1975

[RAN95a] B. Randell et al, "Dependability - Its Attributes - Impairments and Means" - *Predictably Dependable Computing Systems* - Springer-Verlag 1995 p. 3-24

[SCO84] R.K. Scott, J.W. Gault, D.F. McAllister, J. Wiggs, "Experimental Validation of Six Fault-Tolerant Software Reliability Models" *IEEE Fault Tolerant Computer Systems* Vol. 14 - 1984 - pp 102-107

[SCO85] R.K. Scott, J.W. Gault, D.F. McAllister, "The Consensus Recovery Block", *in Proc. Total System Reliability Symposium*, p. 74-85, 1985

[SHR78a] S. K. Shrivastava, "Sequential pascal with recovery blocky", *Software Practice and Experience*, 8:177 – 185, 1978

[SHR78b] S.K. Shrivastava, A.A Akinpelu, "Fault-tolerant sequential programming using recovery blocks", in *Proc. Eighth International Symposium on Fault-Tolerant Computing*, p. 207, Toulouse, 1978

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

[TOW01a] P.Townend, J. Xu, M. Munro, "Building Dependable Software for Critical Applications: N-version design versus one good version," in Proc. 6th IEEE Intl. Workshop on Object-Oriented Real-Time Dependable Systems, p. 103-110, 2001

[TOW01b] P.Townend, J. Xu, M. Munro, "Multi-Version Software versus One Good Version: A further study and some results," in Proc. IEEE/IFIP International Conference on Dependable Systems and Networks, Goteborg, July 2001

[TSA96] T.K. Tsai, R.K. Iyer, "An Approach to Benchmarking of Fault-Tolerant Commercial Systems," in Proc. 26th Ann. Int. Symp. Fault-Tolerant Computing, p. 314-323, IEEE, Los Alamitos, CA, 1996

[VOA95] J. Voas, K. Miller, "Using Fault Injection to Assess Software Engineering Standards", in *IEEE Int. Soft. Eng. Standards Symp. 1082-3670*, p. 139-145, 1995

[VOA97] J. Voas, A. Ghosh, F. Charron, L. Kassab, "Reducing Uncertainty about Common-Mode Failures", in *Int. Symp. On Reliability Eng. 1071-9458,* p. 308-323, 1997

[VOA98a] J. Voas, "Analyzing Software Sensitivity to Human Error", Int. Journal of Failure and Lessons Learned in Information Technology Management, 2(4), December 1998

[VOA98b] J. Voas, "Software Fault Injection: Inoculating Programs against Errors", Wiley Computer Publications, New York, 1998

[VOU90] M.A. Vouk, "Back-to-Back Testing" - *Information and Software Technology* Vol. 32 - No. 1 1990 p. 34-45