Empirical Evaluation of the Effectiveness and Reliability of Software Testing Adequacy Criteria and Reference Test Systems

Mark Jason Hadley

PhD

University of York
Department of Computer Science
September 2013

Abstract

This PhD Thesis reports the results of experiments conducted to investigate the effectiveness and reliability of '*adequacy criteria'* - criteria used by testers to determine when to stop testing. The research reported here is concerned with the empirical determination of the *effectiveness* and *reliability* of both tests sets that satisfy major general structural code coverage criteria and test sets crafted by experts for testing specific applications. We use automated test data generation and subset extraction techniques to generate multiple tests sets satisfying widely used coverage criteria (statement, branch and MC/DC coverage). The results show that confidence in the reliability of such criteria is misplaced. We also consider the fault-finding capabilities of three test suites created by the international community to serve to assure implementations of the Data Encryption Standard (a block cipher). We do this by means of mutation analysis. The results show that not all sets are mutation adequate but the test suites are generally highly effective. The block cipher implementations are also seen to be highly 'testable' (i.e. they do not mask faults).

Contents

Table of Tables

Table of Figures

Acknowledgements

I would firstly like to thank Professor John Clark for his support over the last six years. I would also like to thank Tim White for his review and discussions on this thesis. I would like to thank Karen Jones for her support.

I would also like to thank Defence Science Technology Laboratory (Dstl) who sponsored my PhD. The views in this thesis are those of the author and do not necessarily reflect the views of Dstl.

Author's declaration

I declare that this work is original research and written solely by the author.

Chapter 4 of this thesis contains material that was published in [Hadley & Clark 13].

1. Introduction

1.1 Background and Motivation

In software testing we often apply an *'adequacy criterion'* to determine when to stop testing. The adequacy criteria used differ depending on the type of software system and criticality of the software. Some authors like [Littlewood 93], [Littlewood 2011] and [Butler & Finelli 96] have urged use of a statistically quantifiable measure to determine when to stop testing. However, for high levels of reliability (e.g. 10^{-6}) the number of tests required is prohibitive.

In the civilian aerospace domain and increasingly on Ministry of Defence (MoD) and Department of Defense (DoD) programmes the standard DO-178B [178B][1] is applied. In this standard coverage criteria and functional testing are used to determine when to stop testing. At the software unit level, assuming the required coverage criteria have been achieved (i.e. Statement, Branch, and MC/DC) unit testing is deemed to have been completed and testing is stopped. However, if the satisfaction of such objectives is to be taken as an indication of the thoroughness of testing we may legitimately ask how *effective and reliable* are these coverage criteria as adequacy criteria?

In some domains, a standard means of gaining confidence in the operation of a system is to run it on a standard reference test set. Cryptography is an obvious example. It is usual practice for algorithm designers or international standardisation efforts to supply reference test sets against which developers can evaluate their implementations.

Passing all the tests is a strong form of evidence regarding the correctness of the implementation. However, we know of no independent assessment of these test sets. Thus, it is reasonable to ask: is passing all the tests in these sets a good stopping criterion?

Effectiveness is the ability of test sets to find faults, while reliability is a measure of the consistency across the test sets used to achieve the testing objective. For example, how reliable are two different test sets meeting the same coverage criteria? Do they detect or find different failures? Two test sets of the same size may each achieve 100% MC/DC coverage but have different fault finding capabilities. This thesis evaluates by empirical experiments two different

[1] It is noted that a new version [178C] of this document has now been published; however, the same coverage objectives are listed in both [178C] as in [178B].

types of adequacy criteria: test sets that meet general-purpose structural code coverage criteria and application specific test sets. We evaluate three widely used structural code coverage criteria: Statement, Branch and MC/DC. For the application specific test sets, we use three internationally used test sets for the Data Encryption Standard (DES) algorithm.

We have developed an automated testing framework that enables large-scale testing and determination of effectiveness (fault finding ability) of test sets. To evaluate the reliability of criteria we generate a very large test set that satisfies the criteria of interest with very significant redundancy. We then extract optimal (minimal size) test sets that satisfy the criteria and compare their effectiveness.

1.2 Research Objectives

The main objectives of this thesis are the following:

1. To measure the test effectiveness of the three coverage criteria (Statement, Branch and MC/DC) mandated by a widely used commercial airborne software standard for safety critical software D0-178B [178B] and its recent updated version D0-178C [178C].

2. To measure the reliability of those three coverage criteria by comparing the effectiveness of multiple minimal size tests sets meeting these criteria.

3. To measure the reliability of the three widely used coverage criteria used in the commercial airborne safety critical software e.g. [178B] and [178C] with test sets with a small degree of redundancy. To add redundancy we plan to combine the different optimum coverage test sets.

4. To measure the test effectiveness of three reference test sets developed to test a DES algorithm.

In all cases we believe the research is novel. While empirical studies in the past have examined the effectiveness of testing techniques, these are typically based upon a small number of injected or known faults on small programs and with all testing being manually driven. They have also generally focused on Statement coverage or Branch coverage. None of these empirical studies has used test automation for the generation of the test cases. A small

number of papers have examined test subset extraction; however, none has used repeated automated subset extraction to determine the *reliability* of criteria. Automated approaches based on heuristic optimisation algorithms from the artificial intelligence and operations research communities have generally been used to reduce the number of test cases required in the test set without impacting test effectiveness. In terms of evaluating known test reference systems, we are not aware of any known formal assessment of the ability of available test sets to find flaws in the implementation. This thesis sets out to provide one.

1.3 The Aims and Goals of the Thesis

Since enumerating tests to cover the whole input space is generally infeasible, we are faced with choosing some subset of possible tests. Some subsets will clearly find more faults then others. We are faced with the task of recognising such stronger subsets or those, which have greater chances of doing so. Since testing is expensive, we would wish to generate small efficient subsets and stop at that point.

The goal of this PhD thesis is to examine the effectiveness and reliability of two different types of adequacy criteria. To enable us to achieve this goal we present a testing framework and conduct experiments on a range of programs with different program properties. The framework is developed incrementally and supports the following key functionality:

- **Automated Testing** - This is so that we can generate large test sets efficiently and reduce/remove bias from our results.
- **Test Coverage** - We need to capture test coverage for each test case and for each coverage objective, i.e. Statement, Branch and MC/DC.
- **Mutation Injection** – Mutation testing serves as our means of determining fault-finding ability (effectiveness). We need to be able to inject a variety of different faults into our programs under test and measure for each test what mutants are killed by it.
- **Subset Extraction** – To evaluate reliability of criteria we need to be able to extract multiple coverage compliant test sets from much larger redundantly compliant test sets and compare their effectiveness.
- **Program Properties Determination** - A way of assessing different program properties of the programs under test so that these may be correlated with aspects of

our testing. Example properties might be numbers of branches or other complexity measures.

The development of the framework enables us to use freeware and bespoke components to undertake our experiments and to analyse the data generated from the execution of thousands of tests. While there are freeware components to undertake some of the functionality of the framework none provided a total solution consistent with the aims of this thesis. For example JUnit is a testing framework but does not provide code coverage metrics, fault injection or any form of test subset extraction capability. Running thousands of tests over thousands of mutants generated thousands of test log files. While analysis tools exist, it was relatively easy to generate bespoke tools to extract out the information required in the correct format and in very quick time using bespoke C# components.

The development of a testing framework allows empirical evaluation of the effectiveness and reliability of the adequacy criteria. There is a good deal of research into adequacy criteria and a large amount into automated testing. This thesis takes the view that the latter is a methodical enabler of the former and forms a distinctive feature of our approach.

1.4 Overview and Structure of the Thesis

This thesis documents the framework development activities and the experiments and results from experiments. The chapters of this thesis are as follows:

- **Literature Survey:** Reviews the current state of practice in software testing. It compares pairs of contrasting testing characteristics e.g. static with dynamic testing. Also detailed are the findings from a number of empirical studies. These studies evaluate testing techniques that are '*state of practice'* rather than '*state of the art*'. Their findings indicate no consensus on the effectiveness of any one technique, but do show a general consensus that software testing techniques are complementary and that combining techniques increases fault-finding ability. (Chapter 2)

- **The Research Test Framework and its Components:** Presents the framework developed to undertake empirical experiments to examine the effectiveness and reliability of different adequacy criteria. The framework covers test set generation, code coverage, mutant generation, test subset extraction, and test execution and mutation score calculation. (Chapter 3)

- **Testing Code Coverage Criteria:** Documents the experiments performed primarily on numerical recipes taken from [Press 92]. In addition to these numerical algorithms a number of sorting algorithms (bubble, heap, shell, insertion, merge, quick, and shell-sort) and other miscellaneous algorithms were used. (Chapter 4)

- **Testing Encryption Algorithms:** Presents the findings from the mutation analysis of a Java implementation of the Data Encryption Standard (DES) and a Java Big Integer implementation. For the DES Java program we applied different international standard test vectors to assess their effectiveness and reliability. (Chapter 5)

- **Conclusions:** Discusses the findings and contributions of this research and identifies further research work. (Chapter 6)

- **References:** Contains the bibliographic information for literature cited in this thesis. (Chapter 7)

2. Literature Survey

Software testing is the most widely used method for defect detection during software development and for gaining confidence in the developed product. The majority of the techniques detailed by [Myers 79] can still be seen today as being the '*state of good engineering practice*' in the software testing domain. However, how effective these techniques are and how reliable they are remain unresolved issues.

The only current scientifically based method of quantifying the reliability of software is through '*software life testing/software reliability growth models'* i.e. experience of operating the software in its operational environment for a pre-determined length of time. However, time and cost reasons make this generally impractical. With cloud based computing, multi-core processors and Graphic Central Processor Units (GCPUs), this is now possible and may in the future become more widespread. However, currently this is not the case. Therefore to achieve software release, it is necessary to have established assurance evidence from '*prior beliefs*': one such belief is that the development process has detected, and removed[2], as many defects as reasonably practicable.

Software is pervasive and modern society is now dependent upon it. With such dependencies come the need for the assurance of software reliability and a need to argue in as rigorous a manner possible how good the development analysis and testing processes are.

The overall aim of my thesis is to undertake a practical examination to investigate the '*effectiveness*' and '*reliability*' of different testing techniques. The aim of this literature review is to support this aim. It explores the software testing literature and details some fundamental concepts, techniques and approaches of software testing.

This literature review is split into 7 sections:

- Section 2.1 discusses some fundamental concepts and techniques. Also discussed in this section is where the techniques are applied in relationship to the development life-cycle and different phases of dynamic testing.

[2] Defects are often categorise by their severity and if defects severity is low or no impact these defects may not be removed due to time and cost pressures.

- Section 2.2 details the characteristics of different testing techniques and provides a comparison of them, e.g. static v dynamic. Section 2.2 finishes with a comparison of testing and debugging.

- Section 2.3 details software standards, process models and software measurements. Software standards, process models and metrics play an important role in the managerial aspects of software engineering and are necessary to ensure a level of quality is built into the developed product. However, as we will see in section 2.3, projects continue to fail, even when adopting so called 'best practice'.

- Section 2.4 discusses model based testing, reliability growth models and fault based adequacy techniques, e.g. fault seeding and mutation testing. It discusses the three common forms of mutation testing: weak, firm and strong. This section closes with a discussion on Propagation, Infection and Execution (PIE), used to measure software *testability*.

- Section 2.5 discusses the empirical studies and case studies. It shows that the empirical studies indicate no consensus of opinion in the effectiveness of the techniques applied. The case studies also show that testing consumes up to 50% of the development cost and application of formal techniques (e.g. symbolic execution) remains rare, despite several success stories.

- Section 2.6 examines the question '*when do we stop testing?*'

- Section 2.7 discusses different types of test sub-setting via heuristic search that could be used for test subset extraction.

2.1. Fundamental Concepts

2.1.1 Testing – the English Usage

An inspection of English dictionaries reveals definitions of the word 'test' of which [Collins 01] is typical:

1. to try (something) out to ascertain its worths, safety or endurance. 2. to carry out an examination on (a substance, material or system) in order to discover whether a particular substance, component or feature is present. 3. to make heavy demand on: his behaviour really tests my patience. 4. to achieve a result in a test which indicates the presence or absence of something. 5. a method, practice or examination to test a person or thing.

It is interesting to note that this definition is positive in the sense that 'testing', or a 'test', has an overall 'flavour' of a method to establish some property so as to ascertain something's worth.

2.1.2 Software Testing – a Technical Usage

In contrast, the definition of 'software testing' (hereafter 'testing') appears somewhat negative. It is now widely accepted that testing for the majority of software systems can show only the presence of faults in software: it cannot prove the absence of faults. (Dijkstra's famous quote is reportedly first spoken during a NATO Science Committee, Room, Italy 27-31 October 1969.)

As a result, the major aim of testing is to find faults, and 'testing' can be defined as the examination of software in order to detect faults.

Assurance, then, can be inferred from the ability of the testing techniques to find faults. Quite simply, our assurance claim is that, if our testing is effective and reliable, then it will have found the faults that it is reasonable and practical to do so.

2.1.3 Faults and Failures

There is only weak consensus in the literature of the necessary terminology and there are numerous terms for a software anomaly. For example, [Bezier 90] uses the word 'bug', but does not define it. 'Bug', 'defect' and 'fault' appear to be used interchangeably in the literature. This convention will be adopted herein.

However, there seems to be consensus that a 'failure' is a deviation of a system's behaviour from that expected. [Sommerville 95] and [Testing Glossary 04] define 'failure' as a '*deviation of the component or system from its expected delivery, service or result*'. A key issue is that this definition refers explicitly to a system (or component) and implicitly to its observable operation.

The [IEEE Glossary 1990] defines 'failure' as:

> *'The inability of a system or component to perform its required functions within specified performance requirements.'* ***Note:*** *The fault tolerance discipline distinguishes between a human action (a mistake), its manifestation (a hardware or software fault), the result of the fault (a failure), and the amount by which the result is incorrect (the error).*

This definition, too, explicitly refers to a system (or component) and, implicitly, to its observable performance. The definition also clearly refers to the differences between a mistake (a human action), a fault (the result of a mistake manifest in an artefact), a failure (observable misbehaviour of the artefact) and error (a measure of deviant behaviour). This Literature Review will use the word 'failure' as defined by the IEEE.

Finally, we note the definition implies a direct relationship (one of cause and effect) between a fault and a failure. However, this causal relationship may not be one-to-one. [Frankl et al 98] proposes the concept of 'failure regions', rather than individual faults. A failure region could contain several faults resulting in one observable failure. [Frankl et al 98] recommended changing use of 'testing has exposed a fault' to 'testing has exposed a failure'.

The discussion of software fault and failure definitions generally focuses on the functional and side steps non-functional issues. Non-functional properties, e.g. timing, could lead to a race condition for unprotected code. Therefore the software component may be functionally correct, but non-functionally incorrect. This is more than likely to be an architectural fault, and cannot be mapped to specific lines of code or even regions of code. Also the manifestation of this non-functional failure may only be observable under specific processor loading conditions and these types of non-functional failures are unlikely to be discovered by functional based testing.

The definitions of software fault and failure lead to a distinction between different forms of software testing. Software reliability-based testing finds failures based upon an operational profile. In contrast, lower levels of testing, e.g. unit testing, tend to find defects. A defect does not necessarily result in a system failure (it may be 'masked'). As a consequence, lower levels of testing could discover a number of trivial faults, which give rise to no observable software failure. Limited resources (i.e. time and money) applied to testing may be consumed by finding trivial faults. Also by finding a number of trivial faults during testing, we may form a misleadingly positive assessment of the effectiveness of the testing undertaken.

2.1.4 Faults, Failures and Testing

A fault does not necessarily result in a failure: for example, a variable mistakenly set out-of-range may be re-assigned correctly before the incorrect value manifests itself. However, it must remain a fundamental assumption that a fault has the *potential* to cause a failure and, therefore, must be detected and corrected.

This distinction leads to a categorisation of the testing methods used to detect faults:

- those methods that find faults manifested as failures (that is, observable in dynamic operation);

- those methods that find faults by other means (that is, statically).

This Literature Review will use the terms 'dynamic testing' and 'static testing' respectively for the above two categories, and will use the term 'software testing' (or simply 'testing') to refer to both or either as the English sense requires.

2.1.5 Terminology in the Literature

We note that, in the literature:

- Dynamic testing is sometimes called software testing.
- Static testing is sometimes called software evaluation or static analysis.

For example, [Sommerville 95] defines testing as:

> '*exercising the program using data like the real data processed by the program. The existence of program defects or inadequacies is inferred from the unexpected system outputs*.'

Thus, Sommerville favours a strictly dynamic interpretation of testing.

On the other hand, [Storey 96] favours a wider, inclusive, view. Storey defines testing as:

> '*the process used to verify or validate a system or its components*.'

The software testing literature is divided over these two distinctions. Some authors e.g. [Harrold 00], [Demillo et al 87], [Binder 00], [Kaner 93], [Beizer 90] and [Myers 79] see testing as being strictly dynamic in the sense of Sommerville. Others like [Adrion et al 82], [Smith and Wood 87], [Gardiner et al 99] and [Hetzel 88] see testing as the wider, inclusive activity. Returning to [Storey 96], we see: '*static testing {is} investigating the characteristics of a system or component without operating it'*. [Hetzel 88] defines testing as '*Testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results.*'

[Demillio et al 87] clearly makes a distinction between software testing ('*exercising the software on representative test data under laboratory conditions to see if it meet the stated requirements')* and software evaluation ('*examines the software and the processes used during the development to see if the stated requirement and goals have been met)'*.

Unfortunately, some authors are not consistent with their use of the word 'testing'. For example, [Myers 79] defines testing as: '*the process of executing a program or system with the intent of finding errors'*. Myers then proceeds to define and discuss human testing or non-computer based testing, including program inspections, walkthroughs, and reviews.

For clarity, and where there might be confusion, this Literature Review will use the phrase 'dynamic testing' where the original author may have used the phrase 'software testing' in the exclusive sense; further, the phrase 'static testing' will be used instead of, and synonymously with, original authors' use of 'software evaluation'.

2.1.6 What is Tested and Where?

The 'where' and 'what' of testing depends on the type of software lifecycle applied and the level of rigour required for the development of the software system. Different software systems such as safety critical, safety related, mission critical, financial, office applications, gaming systems will have applied different levels of testing rigour depending on their criticality and their commercial sensitivity.

Lifecycles indicate the sequence of activities to be performed and provide a framework for software assurance activities to be applied and constructed in a disciplined manner. [Boehm 88] defines four types of software development models: Code and Fix, Waterfall, V model and

the Spiral model. Code and fix only includes two phases i.e. code and fix. Requirements, design, test and maintenance were undertaken after the code and fix phases. More structured process lifecycles were then developed containing sequential phases, i.e. requirements, leading to design, followed by code and test. 'Waterfall' and 'V Models' are examples of these types of sequential lifecycles that are the most commonly applied to software projects today.

With the V-model, the left side of the 'V' indicates the development activities; requirements definition, architecture design, detailed design. The right side of the 'V Model' shows the dynamic testing activities. Figure 1 illustrates this type of V-model, with through life verification. The solid lines indicate products, i.e. from the user requirements phase a User Requirements Document (URD) is developed. The Software Requirements (SR) is verified against the user requirements to verify that they contain the user requirements illustrated by the dashed lines. The verification activities on the left hand side are static and might include reviews, inspections, walkthroughs, formal inspection, architecture modelling, formal methods etc.

Table 1 indicates the testing activities, which could be performed at each life cycle phase. Dynamic test design, which includes test planning and test data generation, is undertaken while the left side phases are being undertaken but the actual dynamic testing activities occur on the right side. Good test design is one of the best 'bug' prevention techniques known according to [Beizer 90] and is the first goal of testing.

Testing and bug prevention are the two primary concepts embodied into two types of testing models defined by [Gelperin & Hetzel 88]. Figure 1 illustrates what [Gelperin & Hetzel 88] call lifecycle models which include analysis and review activities as part of wider Validation, Verification and Testing (VVT) activities. The second type of model is where test execution is the primary testing activity, which goes back to the code and fixes lifecycle philosophy.

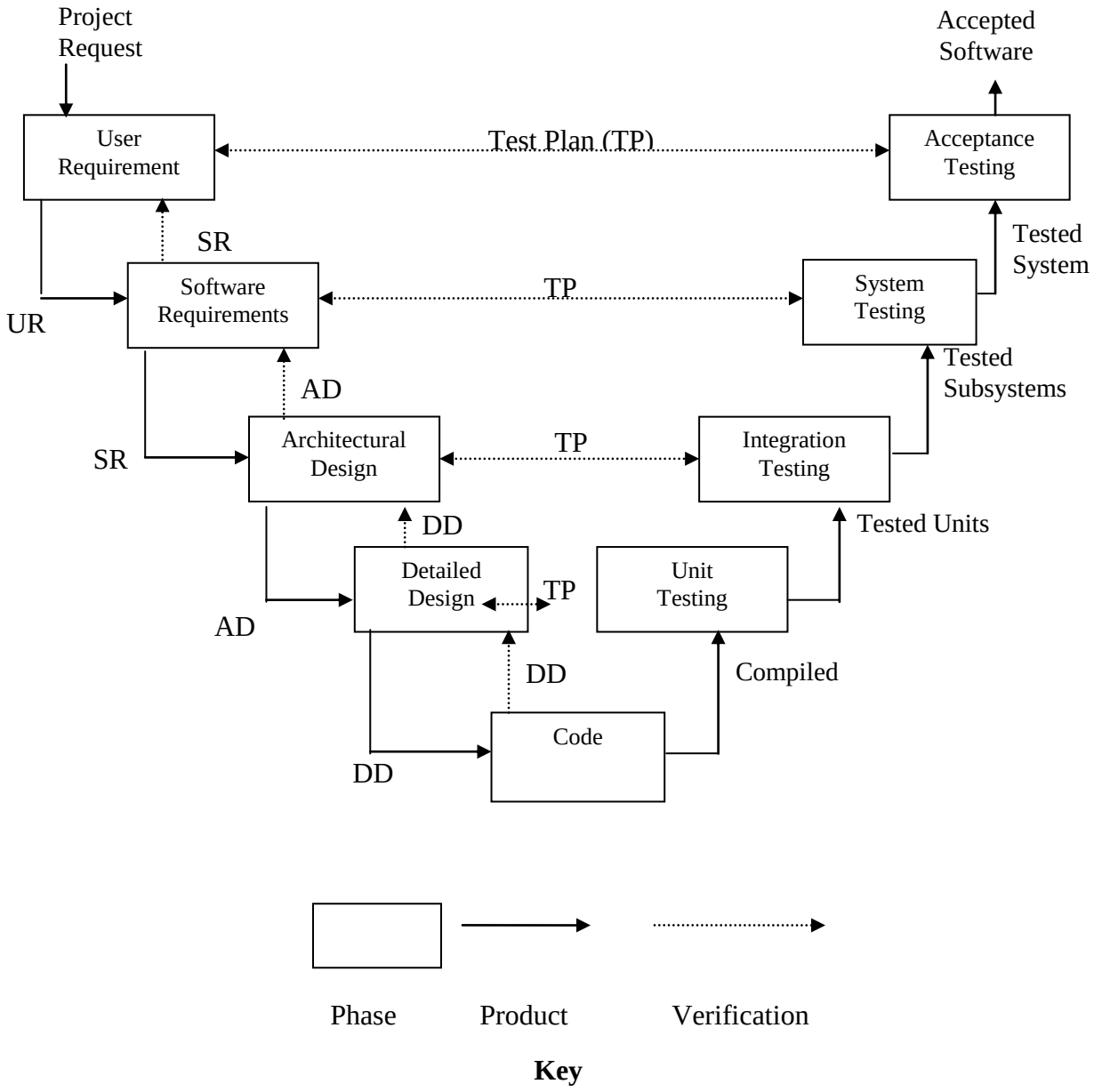

Figure 1: V-Model Life Cycle Verification Approach

Techniques/Phase	Requirements	Architecture Design	Detailed Design	Implementation/Unit	Software Integration	System Validation
Static						
Walkthroughs	■	■	■	■	■	
Reviews	■	■	■	■	■	
Checklist	■	■	■	■	■	
Fagan Inspection	■	■	■	■	■	
Active Reviews	■	■	■	■	■	
Formal Proofs		■	■	■		
Control Flow	■	■	■			
Data Flow	■	■	■			
Symbolic Execution			■			
Peer Reviews	■	■	■	■	■	
Desk Checking	■	■	■	■	■	
SAAM		■				
Stepwise Refinement		■	■	■		
Usage Base Reading						
Audits	■	■	■	■	■	■
Dynamic						
Requirement				■	■	■
Error Handling				■	■	■
Intersystem/Interface					■	■
Boundary Value Analysis				■	■	■
Cause and Effect Testing					■	■
Probabilistic/Reliability testing						■
Error Guessing/Random Testing				■	■	■
Mutation Testing i.e. Strong, Weak Trace, Interface, firm				■	■	■
Resource Testing i.e. CPU time and memory testing etc						■
Volume Testing						■
Usability Testing						■
Execution/Performance Testing				■	■	■
Recovery Testing						■
Operations/Environmental Testing						■
Security Testing						■
Equivalence Partitioning				■	■	■
Syntax Testing						■

Special Value Testing				■	■	■
Domain Based Testing (include both input and output domains)				■	■	■
State Transition				■	■	■
Decision				■	■	■
Structural Testing: Statement, Branch, Conditional, Expression, Path.				■		
Algebraic				■		
Axiomatic				■		
Perturbation Testing				■		
Design Based Functional Testing					■	
Complexity Based Testing				■	■	
Loop Testing				■		
Usage Based Testing						■
Control Flow Testing				■	■	
Data flow Testing				■	■	
Reliability Testing						■
Modelling						
Formal Methods	■	■	■			
Software Prototyping/Animation	■	■				■
Performance Modelling		■				
State Transition Modelling		■				
Petri Nets		■				
Data Flow Modelling		■				
Structured Diagrams		■				
Environmental Modelling					■	■

Table 1: Techniques by Phase
[In part taken from Storey 88]

2.1.7 Phases of Dynamic Testing: Unit, Integration, System and Acceptance Testing

Figure 1 indicates there exists a number of testing phases i.e. unit, integration testing etc on the right side of the V-Model. These tend to use dynamic testing approaches. The corresponding software artefact on the left side of the V-Model i.e. software requirements, architecture design, and detail design tend to be tested statically.

While it is clear what software artefacts correspond with acceptance testing and system testing i.e. User Requirements Document (URD) and Software Requirements Document (SRD), the same cannot be said about a software 'unit'. No consistent definition of 'software unit' is established in the software literature. [Def Stan 00-55 Issue 2] refers to unit testing but does

not define what a unit is. [Zhu et al 97] focuses on unit test coverage and adequacy but similarly does not define what a unit is.

[IEEE Glossary 92] indicates unit, module and component as interchangeable and defines them as:

1. *A separable testable element specified in the design of a computer software component.*
2. *A logically separable part of a computer program*
3. *A software component that is not subdivided into other components.*

However, as indicated in [IEEE Glossary 92] as of that date no formalization of unit, module or component have been established. The MIL-STD series of standards i.e. [MIL STD 498], [J-Std-16 95] and [MIL STD 12207] define a unit as:

> '*An element in the design of a software item, a component of that subdivision, a class, object, module, function, routine or database…Software Units in the design may or may not have a one to one relationship with the code and data entities (routines, procedures, databases, data, files, etc) that implements them or with the computer files containing those entities.'*

[Marick 95] discusses unit testing in terms of individual routines, but does not define what a routine is. [Marick 95] advocates against this form of testing based on cost grounds due to the need of generating software stubs and drivers to enable unit testing to be performed. [Marick 95] instead advocates sub-system testing i.e. a collective of individual units.

[Massa et al 96] defines module as '*as discrete and identifiable with respect to compiling, combining, with other units...also a logically separable part of a program'*. A unit is defined as '*A element specified in the design of a computer software component. A unit is composed of one or more modules'*. This would appear to suggest a module is an Ada function/procedure or C[3] function, or a unit could relate to an Ada Package or a C include file. [Beizer 90] defines '*A unit is the smallest testable piece of software'*, in that it can be compiled or assembled, linked and loaded.

[3] This could be a C, C++ or C# function or method.

[Sommerville 95] defines the unit testing as '*Individual components are tested to ensure that they operate correctly. Each component is tested independently, without other system components.*' [Story 96] defines module testing as the testing of small or simple functions. [Myers 79] defines module (unit testing) as the process of testing individual subprograms, subroutines or procedures in a program. [Binder 00] defines unit testing as '*testing individual software unit or groups of related units. A test unit may be a module, a few modules or a complete computer program.*'[4]

[Beizer 90] and [Beizer 84] also suggested that other attributes of a unit are being created by one programmer and containing only a couple of hundreds of lines of code. The lack of a clear definition for software testing is indicated by [Jorhensen and Erickson 94] who note that organisations which undertake unit testing have not defined what a software unit is.

Dependent units i.e. functions, procedures, object class or abstract data types can be integrated together to form what [Beizer 90] called components or what [Sommerville 96] called modules. Software integration testing focuses on integrating software components. A number of different approaches to integration testing have been proposed by a number of different authors and standards. [Beizer 84] focuses on four techniques; top-down, bottom up, big bang and backbone. The first two are frequently stated in the general software testing literature which also indicates their applicability, advantages and disadvantages e.g. [DeMillo et al 87], [Myers 79], [Dunn & Ullman 82] and [Binder 00].

Top-down and bottom-up use the structure of the program to integrate other units. Top down integration integrates units from the top of the program structure and progressively works down until all the units in the program have been integrated. Stubs are required to simulate the calling modules and pass test data back to that unit under test. Stubs can vary in their complexity from returning single to multiple values, to returning error conditions. Bottom up testing involves testing terminal units in the structure of the program first, i.e. units that call no other units. Drivers are required to stimulate that unit under test. Units are then built up incrementally working up the structure of the program from the terminal unit. Integration continues until all the components have been integrated to make up the whole system.

[4] [Binder 00] defines a unit test as: *A test that exercises a relatively small executable. In OOP an object is the smallest executable unit, but test messages must be sent to a method…A test unit may be a class, several related classes (cluster) or a executable binary file*.

[Binder 00], includes the four [Beizer 84] techniques and adds an additional five techniques to what [Binder 00] calls 'Software integration testing patterns' as detailed in Table 2 below.

Pattern Name	Description
Big Bang	Try everything at the same time.
Bottom-Up	Integration by dependencies.
Top-Down	Integration by control hierarchy.
Backbone	Hybrid integration of subsystems.
Collaborations	Integration by cluster scenarios.
Layers	Integration by layered architecture.
Client/Server	Integration for client/server architecture.
Distributed Services	Integration for distributed architecture.
High Frequency	Build and test at frequent regular intervals.

Table 2: Software Integration Testing Patterns
(Directly quoted from [Binder 00])

System testing generally focuses on validating that the system meets it functional and non-functional requirements and tests the whole interaction of the integrated components. [Beizer 84] indicates the following activities for system testing:

- System level functional verification by the programming staff and/or QA.
- Formal Acceptance test plan design and execution thereof by the buyer or a designated surrogate.
- Stress Testing
- Load and performance testing
- Background Testing
- Configuration Testing
- Recovery Testing
- Security Testing

[Beizer 84] indicates that system testing includes acceptance testing which focuses on ensuring that the system meets the initial needs of the end users in its operational use. Acceptance testing is sometimes referred to as alpha testing. This can involve enabling end users to use the system to undertake activities or end users/procurers of the system providing the test data. Acceptance testing often occurs as part of system testing or as a separate phase.

[Myers 79] indicated a slightly different definition of system testing and indicates that functional testing is the place to test the functional requirements of the system. The purpose of system testing, according to [Myers 79], is to show that the program does not meet it objectives. The objectives are not derived from external specifications but generated from the user documentation and the program's original objectives. [Myers 79] defines 15 categories of test case design which may be applicable to programs, but not all may be applicable to every program. These include: facility, volume, stress, usability, security, performance, storage, configuration, compatibility/conversion, installability, reliability, recovery, serviceability, documentation and procedure testing.

2.2 Characteristics of Testing

2.2.1 Introduction

Different techniques test the software in different ways. This section highlights pairs of contrasting characteristics of test techniques:

- Static v Dynamic
- Functional v Structural
- Formal v Informal
- Manual v Automated

The section closes with a comparison between testing and debugging and notes that while they are related their aims are very different.

2.2.2 Static v Dynamic

Testing techniques can be categorised either static or dynamic. Static techniques do not execute the actual program. However some form of conceptual execution may take place, i.e. using modelling or constructs added to the program to analyse the program under review. Techniques like code inspection, review, formal proofs, control- and data flow analysis are all categorised as static. One of the advantages of these techniques over dynamic testing is that you do not have to wait until the source code artefacts have been developed and compiled to assess them. This enables errors in requirements, architecture and detailed design to be captured earlier in the development life cycle and has major cost and time implications. The other major advantage of static techniques over dynamic is that application of the static technique locates

the fault rather than merely observing a failure. For example code reading can locate where a variable has not been initialised before use or where a buffer overrun could occur due to an array size not being set to the correct length.

Domain testing, requirements testing, mutation testing[5], and structural testing are all forms of dynamic testing, since they require the program to be executed. This type of testing follows the traditional view of testing held by [Harrold 00], [Demillo et al 87], [Binder 00], [Kaner 93], [Beizer 90], [Sommerville 95] and [Myers 79]. Test cases are executed over the program and the results compared against the expected results. [Harrold 00] indicated three major advantages of dynamic testing:

- Testing activities are relatively easy to perform. Test case requirements can be derived from many different software artefacts i.e. requirements, source code, module interfaces etc. Test cases can be automated and instrumented so that information can be gained relating to the execution of the software under test to support the software validation.

- Testing can be performed in the expected environment. This can be in terms of testing the program on its target hardware and in its real world environment. This in turn provides confidence that the software will operate as intended.

- Much of the testing can be automated and test cases can be reused as the software evolves.

2.2.3 Functional v Structural

In functional testing test cases are generated without visibility of the internal structures i.e. the program is treated as a 'black box'. The outputs generated by the program are compared with the expected behaviour for those inputs as defined by the requirements/specifications. This comparison is based upon some form of oracle. The oracle indicates the expected behaviour for these inputs. The oracle results are then compared with the actual results. Since functional testing focuses on the requirements/specification it generally detects failures relating to incorrect implementation of requirements, i.e. domain based defects.

[5] Mutation testing is not really a testing technique, but measures the adequacy of a test set, via a mutation score (MS). Mutation testing will be discussed in detail in section 4.4.

In structural testing test cases are generated with visibility of the internal structures i.e. the software system is treated as a 'white box'. Structural testing focuses on discovering errors that have occurred during program implementation. Therefore, structural testing is generally applied to software units or at software integration. Beyond this level the number of possible software paths makes software structural testing infeasible for the majority of programs. This contrasts with functional testing that can be applied to all the dynamic testing phases in the development lifecycle.

One of the advantages of structural testing is that it enables you to reduce your test data selection i.e. reducing your input space by tailoring the test data to exercise the internal logic. Coupled to that, tests can be designed to exercise the internal logic in a more sophisticated way, to be more stringent and more cost effective, and to generate more complicated tests. However, with functional testing, the input space is much wider; and for the majority of programs it is infeasible to exhaustively test each possible input space. Therefore, the literature [Myers 79], [Bezier 96] attempts to address the infeasibility of exhaustive testing. Approaches include reducing the number of test cases needed by reducing the input space into sub-sets. For example, *equivalence partitioning* divides the input space into invalid and valid domains. Test data is generated by selecting data from these two domains. The assumption is that any test data selected from the same domain should lead to the same results. Some of the testing literature e.g. [Lestiennes & Gaudel 02] refers to this as the 'uniformity hypothesis'. For example all inputs in one domain might be expected to be handled the same way. If the code operates correctly for one such input it might be reasonable expected to work correctly for all such inputs.

To reduce the test data selection further, boundary value analysis is commonly applied. This selects test data normally just under, on and just over the boundary of a domain: the assumption is that values at the boundaries are often mishandled[6]. [Beizer 96] indicated that domain based errors in requirements account for 30% of all requirements errors. Equivalence partitioning and boundary value analysis are the most common functional testing techniques as discussed in Section 3.5.

[6] To supplement these techniques functional control flow testing is generally also applied since equivalence partitioning and boundary value analysis do not examine combinations of input data values.

Software structural testing is not a technique but a 'goal' of the test cases to achieve a required level of test coverage of the software structure. The aim of these coverage goals is to ensure that the program internal logic is exercised for defect detection and to gain confidence that the internal logical works as expected. One of the problems with structural testing is that it is often performed in a 'sterile' test harness environment with the sole aim of the test sets being to ensure coverage of the internal logic. When the software system is fully integrated together, the system's non-functional properties may make the earlier assumptions and results about the internal logic invalid.[7]

Avionic and software safety standards focus on obtaining a level of code coverage from the unit test cases. For example [178B] indicates statement, branch and Modified Condition Decision Coverage (MC/DC) test coverage goals. [Def Stan 00-55] defines the following coverage goals:

- Statement
- Branch
- Source code variables set to min and max value and to an immediate value
- Booleans executed to true and false
- Each variable of an enumerated type set to each possible value
- All loops executed with 0 and 1, immediate number of times and maximum, where semantically feasible
- Special case values , e.g. zero values

Structural based testing aims to ensure as a minimum that all the source code statements of a program are executed at least once and can highlight 'dead code' (code that cannot be executed). Therefore, structural testing ensures that only code which should be there is there. However, it cannot highlight missing logic, i.e. missing implementation of requirements. Therefore, we are dependent on functional testing to ensure that functions have been implemented correctly and structural testing to ensure that all statements as a minimum are exercised and no 'dead code' exists.

[7] This emphasizes the need for test harness environments - system rigs, iron birds (static aircraft rig) etc. - to be 'qualified', in some sense, as truly representative of the final target. The value of reliability-based testing (which finds failures) can be reduced if the test harness environment is not qualified.

2.2.4 Formal v Informal

The majority of the dynamic testing techniques are informal or heuristic in nature. Their creditability is based upon engineering knowledge on how effective these techniques are. However, dynamic testing has limitations:

- [Harrold 00], [Myers 79] and [Beizer 90] are among many in software testing literature who indicate that testing cannot show the absence of faults, only their presence;

- Testing cannot show that the software has certain qualities such as X, Y or Z; and

- Dynamic testing lacks sound theoretical foundation as [Howden 81] noted. To overcome these limitations formal methods have been used to show proof of correctness using mathematical reasoning, for either the whole or parts of the program. [Howden 78] divided formal methods into three categories: provable program classes, program verification and model checking.

Testing is an inductive activity as [Demillo 87] notes, where formal proof of correctness is a deductive activity. In other words, formal proofs attempt to show logically that the program is correct for all inputs; in contrast, testing demonstrates that a program with a given set of inputs will generate specific correct outputs, inferring it will do so for all inputs. In dynamic testing, observations are made and from these observations conclusions are drawn about the program correctness. The effectiveness of testing is dependent on the adequacy of the test set and in mathematical terms may not be valid unless the test set is reliable. [Goodenough and Gerhart 75] defined two requirements for an adequate test set in terms of 'Reliability' and 'Validity'. This will be discussed in Section 2.6.

Formal proofs allow the analyst to reason about the behaviour of a program over all valid inputs that the program is correct. Formal proofs can be generated from numerous formal languages; the most common include OBJ, CCS, Z, CSP, B, LOTOS and VDM. These have well defined semantics and calculi that enable formal proofs to be constructed and arguments to be generated to determine if the program is correct based upon its specification. [Adrion et al 82] notes that proof of correctness is the most complete static analysis technique and divide proof of correctness into formal and informal proofs. To enable formal proof of correctness the most common approach has been to insert annotations into the source code so that formal arguments

can be generated. Both SPARK and MALPAS use this type of approach. Both tools use pre, post and assertion conditions to enable the formal correctness arguments to be generated.

Formal proof of correctness suffers from a number of issues. Expressing requirements and design in a mathematical form necessary to enable formal proof has limited the extent of their application according to [Mazza et al 96]. Cost is also highlighted as a factor. [Howden 78] highlighted how time can be represented by a formal definition and the time required undertaking formal proofs. [Howden 78] illustrated how a theorem prover for a railway crossing design required 1000 pages of proofs and 3 man months of effort[8]. This is less an issue today due to tool support e.g. SPARK examiner. [Woodcock et al 09] conducted a survey of the use of formal methods across of a number of different applications has follows:

- The Transputer Project (microprocessor chips designed for parallel processing)
- Railway Signalling and Train Control (computerised signalling systems)
- Mondex Smart Card (Low value cash card)
- AAMP Microprocessors (Microprocessor widely used by Rockwell Collins)
- Airbus and the use of SCADE
- The Maeslant Kering Storm Surge Barrier (moveable barrier protecting Rotterdam from flooding)
- The Tokeneer Secure Entry System (meeting the Common Criteria requirements of Evaluation Assurance Level 5 (EAL5))
- The Mobile FeliCa" IC Chip Firmware (contactless IC card for electronic purses, travel tickets, door keys etc.)

Some of these projects like AMMP suffered from high cost i.e. 300 hours per instruction. According to [Woodcock et al 09], this was due to steep learning curve and the need to '*develop supporting application-oriented theories'*. One of the positive outcomes of AAMP was that methods were developed to handle complex microcode. The Airbus and the use of SCADE according to [Woodcock et al 09] had a number of positive advantages that included the following [quoted directly from [Woodcock et al 09]]:

[8] For safety critical or mission critical systems three months may be insignificant based upon an ALARP (As Low As Reasonably Practicable) argument.

(i) A significant decrease in coding errors: for the Airbus A340 project, 70% of the code was generated automatically.

(ii) Shorter requirements changes: the SCADE tool suite manages the evolution of a system model as requirements change, and in the Airbus A340 project, requirements changes were managed more quickly than before, with improved traceability.

(iii) Major productivity improvement: Airbus reported major gains, in spite of the fact that each new Airbus project requires twice as much software as its predecessor.

[Woodcock et al 09] concluded there had been a resurgence in the use of formal methods. The paper highlights significant interest in formal methods, showing the number of papers published based upon formal methods.

2.2.5 Manual v Automated

[Harrold 00], [Ng et al 04] and [Hetzel 88] all indicated that the cost of testing can account up to 50% of the development cost. A considerable percentage of this cost is due to the high labour intensity involved in testing and defect isolation, correction and re-testing. In an attempt to reduce this cost burden, software test automation has been proposed by some, e.g. [Denson et al 1999]. While cost may be the main driver towards testing automation, it is not the sole reason. One of the other major advantages of test automation is that it can remove the debate relating to the scope of regression testing required[9] since previous test sets can be repeated quickly and cost effectively to ensure confidence that the software system has not regressed. It also enables a sufficient quantity of test cases to be executed over the software to enable statistical testing.

Automated testing is performed by computer-based systems, whilst manual testing is performed by people. Static testing e.g. reviews, walkthroughs and inspections are typically manual activities. Automated testing normally focuses on the dynamic aspects of software testing and hence focuses on program artefacts once the source code has been developed. Automated testing, like manual testing, can be applied at different phases in the life-cycle and therefore on different software artefacts. Automated testing like manual testing incorporates the same

[9] You can relatively quickly re-run all the tests again without the need to be make arguments relating to the level of regression testing required based upon dependency diagrams, call graphs etc.

testing techniques but discharges the test points automatically, unlike manual testing, where the test points or violations are manually discharged. Therefore, the testing process activity should require less time and effort.

Test scripts contain steps or statements that can either be performed by the tester or undertaken automatically by the test script. The aim of these steps is to place the software in the desired condition/state, so that test points contained in the test scripts can be verified. Manual and automated test scripts contain start up and clean up sections so that test data can be loaded and the software can be placed in the required state. In manual test scripts the expected results are derived from the oracle and are stated in the test script. The tester performing that manual test verifies that the expected results occur. In automated testing, the expected results are automatically verified by the test script. This is normally undertaken by 'verification' or 'check' statements in the script. If the desired condition/state occurs inside the required time frame the test scripts automatically logs a test point pass, otherwise it logs a test point failure.

The recording of test results is one of the other distinctions between manual and automated testing. Manual testing records the results usually on the actual test script, with the tester manually marking each test point as being passed, failing, or noting any observation which deviates from the actual test script. The results of each test script are then manually tabulated in a test summary document, summarizing the results of all tests performed during that phase of the development life-cycle. These results are then placed under configuration control. Formal manual testing is often witnessed by the Quality Assurance (QA) representative of the supplier, and witnessed by a representative of the purchaser. This contrasts with automated testing, where the test scripts can be executed in batches, with each test script generating its own logged output file. These files are automatically checked to ensure that all test points pass and an automated test report summary is generated.

Test scripts can be generated manually or via recording of keystrokes and mouse button presses. The IBM Rational Robot tool records keystrokes and mouse button presses to automate Graphical User Interfaces (GUI) testing. The script can then be manually modified to insert verification statements in the script to ensure specific conditions/states occur. For example a dialogue box may contain several fields, for user input. The script could record the user keystrokes and mouse clicks, to test each data entry field to ensure that each data field will

accept or deny that data entry. The test data is generated from the Human Machine Interface (HMI) specification.

One of the major advantages of automated testing is that once the test scripts have been generated they can be run in batches or individually, and at any time. This therefore has a positive impact on time and cost in the long run and aids regression testing. However, test scripts can only be as good as the engineers developing those test scripts. This equally applies to manual test scripts. The upfront cost, including tool cost, training of staff to use the tool and generating the test scripts in the tool scripting language, have limited the level of automated testing currently seen as indicated by [Ng et al 04]. Due to the perceived high start-up costs of automated testing, it is generally only seen on large scale projects; where the life of the software systems is measured in decades and not years and even with this, automated testing is still rare.

While full automated testing still remains rare, there has been an increasing trend towards tool support to support code reviews and enforcement of coding standards. Tools like PC lint, Nunit test, Ada Analyzer, LDRA and PolySpace can review the code against pre-defined program verification models. These models check for violations against specific rule sets e.g. MISRA C coding standards. The tools indicate each programming violation. The violations are then discharged manually. One of the biggest issues relating to this type of tool aided testing is the number of violations (False Positives) generated by the tools and the time it takes to discharge these violations manually. The other issue is that false negatives may also be generated, which generate a false confidence in the software.

2.2.6 Testing v Debugging

The aims of testing and debugging are very different. The aim of testing is to show that a program contains errors, while the aim of debugging relates to the location and correction of the errors. [Telles and Hsieh 01] and [Araki et al 91] have similar definition of debugging, in that debugging is 'the process of understanding the behaviour of a system to facilitate the removal of bugs.' Defects can stem from many sources from simple programming errors to mis-conceptions in the design or requirement documentation. [Bezier 90] notes that debugging normally occurs after software testing. However, for static testing such as code reviews, error identification, i.e. error location, is part of the code review process. What is not normally part of the code review process is the error correction.

[Araki et al 91] proposes a debugging process model as shown in Figure 2. The error report could be from testing or from operational use and forms the bases of the initial hypothesis for the cause of the defect. During further investigation of the software artefacts, hypothesis selection occurs followed by verification of that hypothesis. [Araki et al 91] indicated four ways of verifying that hypothesis which included static, dynamic, semi-dynamic and program modification.

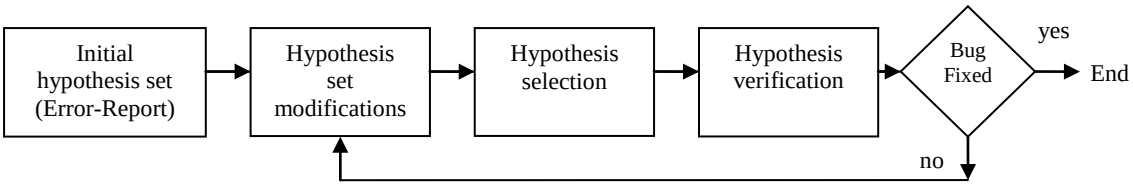

Figure 2: A Debugging Process Model

The debugging process is very much a cause and effect activity, whereby testing is less concerned with the causes of the defect and more concerned with detecting failures. Once debugging has established and verified the hypothesis and therefore established the causes of the error, this can be used in future test design (assuming that the defect was not detected by the actual testing).

Testing and debugging facilitate each other. A fine level of granularity of forward and backward traceability throughout the whole software lifecycle aids testing and debugging. Good testing has positive effects on debugging. Good requirements, good design and good well structured code all have positive impacts on testing and debugging. Good testing as [Beizer 84] notes helps debuggers eliminate fruitless defect hypotheses. Good testing attributes include: clear well-defined objectives, requirement traceability, initial test state clearly defined, followed by a pre-defined set of test steps and expected results. Good test cases enable repeatability and predictability of tests which can form the bases of the initial bug hypothesis. Well-defined and structured design and source code aid testing defect investigation. [Fenton & Ohlsson 00] noted that low cyclomatic complexity is a good predictor of the module attribute maintainability. The opposite is also true in that negative externalities in software artefacts, e.g. poor design, spaghetti code, and lack of requirements traceability, impact testability and debugging. As [Beizer 00] noted, good testing cannot turn bad design into good design.

Formal and informal frameworks for testing have been proposed e.g. [Gourley 83], [Myers 79], but as [Araki et al 91] and [Beizer 90] notes this is not the case for debugging. Debugging tools

in general focus on enabling programmers to observe and understand the program execution i.e. breakpoints, tracers. [Beizer 90] notes nine major differences between testing and debugging listed in Table 3.

No	Description
1	Testing starts with known conditions, uses predefined procedures and has predictable outcomes; only whether or not the program passes the test is unpredictable. Debugging starts from possible unknown initial conditions, and the end cannot be predicted, except statistically.
2	Testing can and should be planned, designed and scheduled. The procedures for, and duration of, debugging cannot be so constrained.
3	Testing is a demonstration of error or apparent correctness. Debugging is a deductive process.
4	Testing proves a programmer's failure. Debugging is the programmer's vindication.
5	Testing, as executed, should strive to be predictable, dull, constrained, rigid and inhuman. Debugging demands intuitive leaps, conjectures experimentation and freedom.
6	Much of the testing can be done without design knowledge. Debugging is impossible without detailed design knowledge.
7	Testing can often be done by an outsider. Debugging must be done by an insider.
8	Although there is a robust theory of testing that establishes theoretical limits to what testing can and can't do, debugging has only recently been attacked by theorist.
9	Much of testing execution and design can be automated. Automated debugging is still a dream.

Table 3: Differences between Testing and Debugging

[Zeller and Hildebrandt 02] defines a debugging approach called Delta Debugging based upon simplifying and isolation. In the simplification phase the failure causing the input is simplified by examining smaller configurations of the inputs. The inputs are reduced until the smallest possible input is established that still generates the failure. Isolation attempts to find a test case passing configuration by removing a particular part of that test case, so the test case does not fail. [Zeller and Hildebrandt 02] indicated that isolation is more efficient than simplification. If we have a large failure-inducing input, isolating the difference will pinpoint a failure cause much faster than minimizing the test cases. However, [Misherghi & Su 06] notes for large test cases, it may lead to worse running times because of the time spent testing the large configurations.

[Misherghi & Su 06] focuses on simplification and defines an approach called Hierarchical Delta Debugging (HDD) that is a technique exploring the input structure to minimise failure inducing inputs. The authors claim the HDD approach reduce the number of test cases required and produce a smaller number of output compared to the original delta debugging defined in [Zeller and Hildebrandt 02].

2.3. Software Standards, Process Models and Metrics

2.3.1 Introduction

This section focuses on software standards, process models and metrics and is split into three parts. Software standards have seen increased prominence over the past three decades as a means of solving the perceived 'software crises'. However, as we will see in section 3.5, software projects continue to fail in terms of being delivered on time, to budget and to quality, i.e. with appropriate delivered functionality. Standards are necessary but not sufficient. Also standards continue to evolve. Standards have in the past been very prescriptive, and in the majority of the cases remain so. However, [Def Stan 00-56 Issue 4] has moved away from a being a prescriptive standard to a more descriptive one, allowing the developer to determine what are the best practices and what evidence is required to ensure safety arguments and confidence in the delivered system.

From the 'standards revolution' of the 1980's, two process models started to merge in the late 1980's, SPICE and CMM. They were primarily developed for two reasons: firstly to enable the purchasing organisation to make assessment of the contracting agency and secondly to encapsulate whole lifecycle process development and improvement. We have since seen the expansion of software metrics in an attempt to quantify the level of quality in the developed software artefacts. However, we currently don't have any software measurements that relate directly to the level of overall system reliability.

2.3.2 Software Standards

Software engineering has grown from a collection of ad hoc practices to a disciplined and controlled set of rigorous activities defined by a well understood process model. Though no 'silver bullet' [Brookes 87] has been found, the constant striving to make software development a true engineering discipline has brought significant benefits. Contributions to this challenging task have come from a range of disciplines. Management has benefited from inputs from the sociological and psychology fields and program validation and verification has benefited from

'hard' computer science. However, software projects continue to fail. They are often over-budget, delivered late, or simply fail to work as required. To deal with such matters a concerted attempt has been made to control the development process more rigorously. One aspect of such control, aimed at enforcing consistency of good practice, is the adoption of well-grounded standards for software development. This does assume that the quality of software systems are heavily influenced by the quality processes used to acquire, develop and maintain that system.

Software engineering textbooks, from [Sommerville 95], to safety critical textbooks, [Storey 96] note the important role of software standards in achieving fitness for purpose and built in quality. [Storey 96] notes the role of standards as follows (quoted directly):

> *1. Helping staff to ensure that a product meets a certain level of quality.*
>
> *2. Helping to establish that a product has been developed using methods of known effectiveness.*
>
> *3. Promoting a uniformity of approach between different teams.*
>
> *4. Providing guidance on design and development techniques.*
>
> *5. Providing some legal basis in case of dispute.*

Standards play an important legal role, in addition to fitness for purpose. Not only do they form the basis of the contract between the purchaser and the provider, they also provide evidence in legal disputes to show that best practice has been applied. This is especially true for safety critical systems.

Standards have evolved over the past three decades and some are now commonplace, such as ISO 9001/2000 for Quality Assurance. A number of different software standards exist, developed by different organisations. These all contain similar elements but differences exist between them. There are a great number of such standards and one might legitimately ask which one should be applied in which circumstance.

Standards can be categorised as de-jure (official) and de-facto standards. De-jure standards broadly fit into three categories:

- Those developed by international organisations e.g. International Standards Organisation (ISO), Institute of Electrical and Electronic Engineers (IEEE).

- Those developed by domain specific institutions e.g. Avionic based standards such as Federal Aviation Regulations (FAR) and Joint Aviation Regulations (JAR).

- Those developed by government agencies e.g. for use in the defence domain - UK Defence Standards, US Military Standards.

De-facto standards include Microsoft Windows and Apple human interface guidelines.

In the past, the UK MOD has generated its own set of defence standards. However, currently the UK MOD is moving towards using international standards or domain specific standards rather than using bespoke defence standards.

Software standards have evolved over time. For example [Def Stan 00-55 Issue 2] mandated specific requirements for different criticalities of software. This standard mandated the use of formal methods for all safety critical software i.e. Safety Integrity Level (SIL) four. [Def Stan 00-55 Issue 2] has been superseded by [Def Stan 00-56 Issue 4]. However, [Def Stan 00-56 Issue 4] no longer mandates specific requirements for different SILs, but places the onus on the supplier of the software to determine what activities and evidence are required, commensurate with the criticality of the software. In doing so it places the emphasis on the purchaser e.g. MOD to determine the adequacy of that evidence.

The majority of the standards mandate software development requirements by using the word 'shall' in the requirement sentence. [178B] uses the word 'should'. However, all standards can be tailored, even mandated requirements can be tailed by agreement between the purchaser and provider of the software. Safety critical software standards such as [Def Stan 00-55], [178B] and [EN-61508] provide different requirements based upon different criticality of the software i.e. certain requirements do not apply for less critical software.

[Def Stan 00-55 Issue 2] defines four levels, referred to in this standard as Safety Integrity Levels (SILs). The criticality of the software is in numeric descending order from SIL4 being safety critical and focusing on loss of life i.e. would cause multiple loss of human life if a

software failure would occur. SIL3-SIL1 are lower safety integrity levels and would not result in multiple loss of human life if a software failure would occur. SIL 3 would result in a single loss of life, SIL2-SIL1 software would not result in any loss of life. [178B] uses a similar scale but uses levels in ascending alphabetical order, and focuses on failure conditions that would prevent safe operation of the aircraft. A level 'A' failure would prevent continued safe flight and landing, while a level E failure would have no effect on the aircraft operational capability or pilot workload.

The safety critical standards and non-safety critical software standards i.e. [Def Stan 05-95], [Mil Std 498] and [DOD 2167A] place a high importance on reviews on the right side activities of the V-Model for defect detection. These reviews focus on the review of requirements, architecture and detailed design. The level of rigour applied to software testing is one of the key differences between the safety critical and non-safety critical standards. The safety critical standards focus on process and software product evaluation whilst non-safety critical standards focus less on software product evaluation. What all these standards have in common is a continuation of V & V activities throughout the whole development lifecycle or what [Gelperin & Hetzel 88] called the 'lifecycle' model.

All the standards reviewed indicate their preference or requirements for independence between the development and testing, as the majority of the testing literature would support, e.g. [Myers 79], [Beizer 90]. However, [Gelperin & Hetzel 88] conflicts with this 'independent approach' and indicates that programmers and testers should work together so that a 'comprehensive' test set is developed. Due to increased complexity of software systems, software programmers are required to provide guidance to the test team in generating more complete and thorough test sets. [Gelperin & Hetzel 88] indicates that this also 'buys in' the test team to accepting responsibility for cost of failures discovered during testing.

[178B/C] focus on requirement based testing, since this technique has been '*found to be the most effective at revealing errors*' according to [178B/C]. To undertake requirement based testing, [178B/C] requires that test sets with normal and abnormal ranges are developed, which [178B/C] refers to as 'robustness testing'. [178B/C] defines testing requirements for different software testing phases, i.e. low-level software unit, software to software integration and software to hardware integration. In [Def Stan 00-55 Issue 2] have similar requirements for

unit, integration and system testing. Example system testing requirements are shown below (Quoted Directly from [Def Stan 00-55 Issue 2]):

> *a) all functions in the Software Requirement and the Software Specification have been executed;*
> *b) all numerical inputs and all outputs have been set to their minimum, maximum and an intermediate value;*
> *c) all booleans, inputs and outputs, have been set to both true and false values;*
> *d) all non-numerical outputs, including error messages, have been tested;*
> *e) all non-numerical inputs have been tested;*
> *f) all testable non-functional requirements, including timing and capacity have been tested.*

[Def Stan 00-55 Issue 2] states only one requirement for integration testing and that is to demonstrate the correctness of all interfaces. [178B] provides more specific requirements for software to software integration. [Def Stan 00-55 Issue 2] also indicates validation testing, defined in that standard as demonstrating that the Safety Related Software (SRS) operates in a safe and reliable manner under all conceivable operating conditions. Part of the requirement for validation testing refers to non-functional aspects of the software, including timing, accuracy, stability and error handling.

Both [178B] and [Def Stan 00-55 Issue 2] have similar structural coverage criteria. For [178B] level 'A' software the following are required to be undertaken by an independent test team[10]:

- Statement Coverage (SC)
- Branch/Decision Coverage (BC)
- Modified Condition/Decision Coverage (MC/DC)
- Data Coupling and Control coupling (DF and CF)

[Def Stan 00-55] defines its coverage criteria as follows (Quoted Directly [Def Stan 00-55 Issue 2]):

> *a) all source code statements and all source code branches;*

[10] [178B], [178C] defines independence in terms of verification as being performed by a different person other than the developer. A tool(s) may be used.

b) all source code variables set to minimum and maximum values as well as an intermediate value;

c) all source code booleans executed with true and false values;

d) all feasible combinations of source code predicates executed;

e) all source code variables of enumerated type set to each possible value;

f) all source code loops executed 0, 1, an intermediate number and maximum times, where this is semantically feasible;

g) special cases, for example source code variables and source code expressions which can take or approach the value zero.

[DOD 2167A], [Def Stan 00-95] and [Mil-Std 498] do not contain coverage criteria and focus on functional based testing.

2.3.4 Software Testing Metrics[11]

Process improvement models such as CMMI and international software standards e.g. ISO and IEEE have increased the demand for software measurement. Software metrics are not new. [Boehm et al 78] listed numerous software measurements to cover what they defined as the attributes that make up software quality e.g. reliability, testability, modifiability. More recent authors [Bache & Bazzana 94], [Kaner 00], [Arora 95], [Fenton & Pfleeger 97] have all proposed or examined software measurement in an attempt to measure specific quality attributes of software. [Bache & Bazzana 94] and [Fenton & Pfleeger 97] do not just recommend a list of metrics, but discusses data collection and measurement theory. [IEEE Std 982.1 2005] and [IEEE Std 982.1 1988] provide a detailed set of software metrics (16 and 39 respectively) which could be applied to software in an attempt to measure software reliability through and after development. CMMI, ISO and IEEE have broadened out software measurement to capture not only product measurement but also process. Metrics no longer just examine effectiveness of the techniques but also efficiency.

[Beizer 90] indicates three broad types of metrics: Linguistic; Structural; and Hybrid, as defined below [Quoted directly Beizer 90]:

[11] The term software metric was the traditional word to define a measurement tool in the domain of software metrics. However, since the late 1980's 'measures' have also been introduced into the software literature. [Melton 96] indicated two reasons for this: '*Metric is a well defined type of mathematical function (a metric takes two arguments and returns the distance between them) and that a software metric is not a metric.*' Secondly some authors wanted to stress that software metrics are defined according to the principles of measurement theory. Here we do not make this distinction and use the terms metric and measurement as interchangeable.

Linguistic *– Measure properties of a program or specification text without interpreting what that text means or the ordering of components of the text e.g. lines of code, number of executable statements, number of unique operators number of unique operands, total number of operators, total number of operands, total number of keyword appearances, total number of token etc.*

Structural *– Metrics based in structural relations between objects in the program i.e. control and data flow-graphs, number of links, nodes nesting depth etc.*

Hybrid *– A combination of structural and linguistic properties of a program or based on a function of both structural and Linguistic properties.*

For [Beizer 90] metrics are product based.

[Myers 79] divided metrics into two domains: product and process. [Myers 79] then sub-divided the product metrics into internal and external metrics.

External product metrics include:

- Product non-reliability metrics, assessing the number of remaining defects.
- Functionality metrics, assessing how much useful functionality the product provides.
- Performance metrics, assessing a product's use of available resources: computation speed, space occupancy.
- Usability metrics, assessing a product's ease of learning and ease of use.
- Cost metrics, assessing the cost of purchasing and using a product.

Internal product metrics include:

- Size metrics, providing measures of how large a product is internally.
- Complexity metrics (closely related to size), assessing how complex a product is.
- Style metrics, assessing adherence to writing guidelines for product components (programs and documents).

Process metrics include:

- Cost metrics, measuring the cost of a project, or of some project activities (for example original development, maintenance, documentation).
- Effort metrics (a subcategory of cost metrics), estimating the human part of the cost and typically measured in person-days or person-months.
- Advancement metrics, estimating the degree of completion of a product under construction.
- Process non-reliability metrics, assessing the number of defects uncovered so far.
- Reuse metrics, assessing how much of a development benefited from earlier developments.

[Ng et al 04] indicates that the most popular software testing metric was simple defect counting. However, from the organisations surveyed by [Ng et al 04] just over 50% of believed that software metrics had improved the overall quality of the software. This would appear to support earlier findings by [GOA 83] and [Gelperin & Hetzel 88]. [Gelperin & Hetzel 88] indicates a more positive trend, from a survey conducted from a testing conference. However, the results are biased and may not reflect the software testing community as a whole.

What is not clear from [Gelperin & Hetzel 88] and [Ng et al 04] is how these defect counts were used. Defects counts can be used as part of other software metrics. [Kaner 00] proposes numerous measures relating to defect counts. [Myers 79] proposes defect counts as part of stopping or halting criteria for testing. [Dunn & Ullman 82] illustrated a hypothetical example that uses defect counting to improve the efficacy of the testing strategy. The testing strategy used four phases of testing: Unit; Integration; Qualification; and System Testing. This example estimated the number of defects in the program captured in each phase. By using the following equation, [Dunn & Ullman 82] could determine the efficacy of testing at each phase and re-allocate of testing resources to other phases of the testing strategy to improve the efficacy of testing.

$$E = \frac{D_f}{D_f + D_r} \text{x}100$$

D_f = Defects Found
D_r = Defects remaining after the test

Two commonly applied software testing measures are the defect and fault density metrics as shown below. They have been used on the NASA space shuttle and are commonly applied on US based military software applications. These two metrics can then be used for more complex models and reliability modelling. Defect density is a coarse-grain metric, since it does not specifically relate to failures, while fault density relates faults to specific failures.

[Fenton & Pfleeger 97] defined defect density (DD) as:

$$DD = \frac{D}{KSLOC}$$

[Musa 99] defined fault density (FD) as (used on the space shuttle) :

$$FD = \frac{F}{KSLOC}$$

D = Defect
F = Fault
KSLOC = Thousand source lines of code which include executable and none-executable lines.

[Berling & Thelin 03] used defect detection to generate a 'measure for goodness' denoted by d_g to measure where defects could have been found in the development lifecycle. Where a defect is found at the earliest possible phase $d_g = 1$, when all defects are found in the last phase the value of $d_g = 0$. When the defects were found in phases between the first and last phase the d_g value is between 0 and 1. [Berling & Thelin 03] does not explain what d_g would be applied to phases between the first and last. One way would be to factor down the phases between the first and last.

One commonly applied software metric used during design to assess fault density is cyclomatic complexity (CC). The hypothesis was that the higher the CC the higher the fault density. [McCabe and Butler 89] proposed a CC of no more than 10. [Walsh 79] indicated a quantum jump in 'error counting' at CC of 11 compared to 10. More recent research by [Herzner et al 05] supported this hypothesis that CC is a good measure for fault probability. [Herzner et al 05] also indicated that detection of faults also deteriorates significantly as CC increases and the amount of effort required to detect these faults increases as CC increases.

Other researchers have shown less positive results from CC. [Fenton and Ohlsson 00] and [Lauterbach & Randall 89] have indicated that CC is not a good indicator of fault density. [Fenton and Ohlsson 00] indicated that LOC is a better indicator than CC, but low CC was beneficial for software maintainability. [Lauterbach & Randall 89] discovered that the

'buggiest code' were modules with some of the shortest LOC and lowest CC. A more fundamental problem with CC is that it tells you nothing about dataflow or dataflow problems inside your program.

Software standards e.g. [Def Stan 00-55 Issue 2] and [178B/C] both refer to some form of structural criteria such as statement, branch, MC/DC. However, [Marvick 00] indicated that between 30-50% of all errors relate to errors of omissions. This would appear to support [Howden 81b] findings which indicated the most common errors were due to missing logic and therefore, structural testing would not be capable of detecting such omissions.

Metrics can only be collected with the aid of tool support and automation. With the expansion of metrics so have the tools to support such metrics. The Cantata software testing tool for example supports over 100 software metrics. However, how effective these metrics are is a different question. The real questions are: firstly are we collecting and measuring the right software characteristics or are we measuring things we can measure due to tool support and the ease of collection? Secondly, how do we use this information to re-allocate the finite resources to improve overall software quality? Thirdly, what are the software quality metrics telling us about the overall reliability of the program? For example [Berling & Thelin 03] 'measure of goodness' tells us nothing about the reliability of the end program or the severity of the defects found in the different phases. We may find a number of defects in the early phases and none in the later phases due to poor effectiveness of the techniques applied, leading to a high 'measure of goodness', but the fielded program may have a very low reliability. On the other hand we may find a number of defects in the later phases with a high severity, resulting in a low 'measure of goodness' but a more reliable program.

2.3.5 Software Projects Continue to Fail

It is common for the US and UK governments not to procure software from organisations with a CMM or CMMI level less than 3. This was similar to the UK MOD requirement that mandates all procurements must be from organisations who are ISO9001/2000[12] certificated. This attempts to ensure a level of quality in the developed product.

While research would appear to indicate that CMM/CMMI and standards play an important role in ensuring a level of quality in software systems, software projects continue to experience

[12] There were exceptions to this rule. For example a one man manufacturer and sole supplier of hand crafted wooden propeller blades, the cost burden of ensuring ISO9001/2000 prohibited this.

time and cost overruns and do not deliver the original planned functionality. CMM spanned 1987 – 1997 when its development was abandoned to enable development of CMMI, which was initially released in 2000. [Def Stan 00-55 Issue 2] was released in August 1997 and [178B] was released in December 1992. However, studies show the increasing failures of software projects, for example:

- [Kazman 00] stated from a US Army study that only 2% of projects are used as delivered, while 29% are paid for and not delivered and 47% are delivered but not used.

- [Bosch 00] noted that a US Air Force command and control system was delivered one year late and at double the cost.

However, these problems are not just limited to US military projects. The UK MOD has similar results; as recorded in a number of UK government documents as indicated below:

- The Euro-fighter was 42 months late and 1,505 million pounds over the total cost as agreed in 1987 [Parliamentary Memorandum Appendix 8]. This was a 37 million pound increase in real terms from the previous year (2001 from 2000), due to computer hardware obsolescence.

- The Nimrod MRA 4 is late, over the original budget, and fewer aircraft are going to be delivered than originally planned (to reduce cost due to the complexity of integrating so many different components together) [UK Parliament, Defence Offet Obligations].

- The Royal Navy payment system had 12 months slippage and finally it was decided to abandon the project because of IT related issues [Appropriate Accounts 1989-89 Vol 1:Close 1- MoD]

- ALR 66 Radar Warning equipment was abandoned with a total loss of 13.4 millions pounds partly due to obsolescence. However, as [Departmental Resource Accounts] noted, the RAF identified numerous software faults that made the system inoperable.

Software standards and process models are ‘necessary but not sufficient’ in ensuring the development of high quality software systems. Only by direct product assessment via software

testing can the software product be truly assessed. Standards and processes do not provide what [Brooks 87] referred to as the 'silver bullet' to resolve the increasing number of software failures, but perhaps provide a stepping stone to enable consistent quality and aid product evaluation and learning by experience.

2.4. Model Testing, Reliability Growth Models and Fault Adequacy Techniques

2.4.1 Introduction

Model based testing (static and dynamic testing) enables testing to be performed earlier in the development lifecycle rather than waiting for the source code to be developed. For example, the software architecture is modelled and dynamic testing is performed against that architecture model. This enables dynamic testing to start earlier in the software development cycle. Static techniques can also be applied to assess the applicability of the model for different quality attributes e.g. maintainability, reliability etc. However, section 2.4.2 focuses on the dynamic testing aspects of model based testing.

Reliability growth models attempt to quantify the reliability of software by applying statistical measures to the observation of software failures.

Fault based adequacy techniques started with fault seeding in the early 1970's, in an attempt to estimate the number of bugs remaining in the software after testing. However, the informality of seeding faults was one of the major issues with fault seeding and in part lead to the development of mutation testing. Mutation testing formalised fault injection and measures the adequacy of the test set in detecting the injected mutants by the generation of a mutation score. Section 2.4.2 discusses model based testing and section 2.4.3 software reliability growth models, followed by a discussion on fault seeding and fault mutation techniques and finishes with a discussion on the PIE (Propagation, Infection and Execution) suite of techniques that are used to measure software *testability*.

2.4.2 Model Based Testing

Model based testing enables testing to start earlier in the software development lifecycle e.g. at the architecture level. This therefore may reveal faults earlier in the development lifecycle and reduce cost in fault detection and correction.

Model based testing is one of the more recent developments in the software testing domain, the issue is not covered in older VVT surveys e.g. [Adrion et al 82] and [Wallace & Fujii 89]. Is it also interesting to note that [Dunham 89] indicated that modelling and simulation would be one of the new developments in V & V in the 1990's.

[Richardson & Wolf 96] and [Jin & Offutt 01] proposed the use of an Architecture Description Language (ADL) to model the architecture. Dynamic test sets could then be executed over these models. Since these papers focus on architecture, they focus on how components and connectors interact.

[Jin & Offutt 01] use a form of CSP and Behaviour Graphs (BG) to model the proposed behaviour of the architecture. [Richardson & Wolf 96] and [Jin & Offutt 01] both focus on the structural aspects of the architecture. [Jin & Offutt 01] focus on the following architecture properties:

- Data Flow Reachability: A data element should be able to reach its target components through the connectors without having that data element value modified.

- Control Flow Reachability: The next element in the control thread is reachable.

- Connectivity: Examines to see if no next or no previous architecture element (component or connector) exists, i.e. dangling/disconnected components or connectors.

- Concurrency: Ensuring that interactions of components do not cause deadlock situations.

Five architecture based testing criteria are defined in [Jin & Offutt 01] (quoted directly):

- *Individual components interface coverage: Requires that the set of paths executed by T covers all Component_Internal_Transfer_Paths and Components_Internal_Sequencing_Paths for each component.*

- *Individual connector interface coverage: Requires that the set of paths executed by T covers all Connector_Internal_Transfer_Paths and Connector_Internal_Sequencing_Paths for each connector.*

- *All direct components to components coverage: Requires that the set of paths executed by T covers N_C paths, all C_N paths and all Direct_Compoent_to_Component_Paths.* (Where N = Component and C = Connector)

- *All indirect components to components coverage: Requires that the set of paths executed by T covers all Indirect_Component_to_Component_Paths.*

- *All connected components coverage: Requires that the set of paths executed by T covers all possible All_Connected_Component_Paths for all the components in the architecture.*

The [Jin & Offutt 01] paper refers to an experiment in which a small scale industrial program was injected with [Gacek & Boehm 98] based defects. Three different testing approaches were then applied in an attempt to detect the injected faults. These approaches included: the architecture-based testing method as discussed by [Jin & Offutt 01]; [Jin & Offutt 98] coupling based integration testing method; and a specification-based testing method. The conclusion is that architecture based testing performed better than the other two techniques in terms of faults detected but not in terms of faults detected per test.

[Richardson & Wolf 96] used Chemical Abstract Machine (CHAM) to formalise the architecture description. CHAM describes the architecture in terms of static components- 'molecules' and transformation by reactions. [Berry & Boudol 98] defines CHAM as (quoted directly from [Berry & Boudol 98]):

> *'States of a machine are chemical solutions where floating molecules can interact according to reaction rules. Solutions can be stratified by encapsulation subsolutions within membranes that forces reactions to occur locally.'*

[Richardson & Wolf 96] note (quoted directly from [Richardson & Wolf 96]):

> *'A CHAM architecture simulation would run the test cases over the architecture, resulting in the set of solutions generated together with the causal history and timing. Causality between solutions results from the execution of transformation rules. These casual dependencies demonstrate the architecture behavior and may show dynamic problems not easily revealed via static analysis.'*

[Richardson & Wolf 96] discusses a number of implementation and specification test criteria which include structural test criteria to exercise the system. The criteria include control flow and data flow coverage and fault based testing.

[Richardson & Wolf 96] focuses on specification based criteria, since specification based testing has as well *'defined control and data flow criteria as well as fault based criteria with regards to a specification's assertions'.* [Richardson & Wolf 96] notes that architecture is composed of data, processing and connecting elements and it is these elements and their complex relationships between elements that should be '*exercised to adequately test the architecture*'.

[Richardson & Wolf 96] defines a CHAM based criterion to test specifications based upon the CHAM architecture descriptions as follows (Italics indicate quoted directly):

1. Determine '*the set of structures to be covered*'.

2. Define paths through the architecture that will cover these structures. Each path will generate a set of solutions. (Solutions - combination of elements, which represent state of the CHAM.)

3. '*Define test data in terms of the architecture interactions with the outside world that would cause this set of solutions to be generated.*'

It then recommends a set of testing criteria for the CHAM model which includes 'all data elements', 'all processing elements', 'all connected elements', 'all transformations', 'all

transformation-systems' that '*require that all distinct paths or all non-repeating sequences of transformations from the initial solution to a stable solution, be tested*' and 'all data dependences'. [Richardson & Wolf 96] state that different tests cases could be developed to test different quality attributes as in '*performance, load, communication or identify missing functionality.'* [Richardson & Wolf 96] propose the use of an architecture test oracle for checking executed results with expected results. This could also be used to check for conformance of the architecture description to the implementation. To enable practical use of architecture testing requires mapping between the architecture and source code.

2.4.3 Reliability Growth Models

Numerous software reliability authors [Butler & Finelli 91], [Littlewood & Strigini 92] [Musa et al 87], [Jalote & Murphy, 04], and [Musa 89] have discussed how to obtain a statistical measure to quantify software reliability via use of reliability growth models. This method starts with the assumption that a program includes faults. During the testing process, these faults manifest themselves as observable failures and are detected and correction occurs[13]. The data is thus a sequence of successive inter-failure times that tend to increase (e.g. stochastically ordered), because of the growth in reliability due to fault removal. The logarithmic Poisson model proposed by [Musa 89] assumes that some faults cause more failures than others, and that on '*average the improvement in failure intensity with each fault correction declines exponentially as corrections are made'*.

Reliability growth models represent a kind of 'average' reliability growth since the frequency of perfect (good) fixes outweighs that of the imperfect (bad) fixes. Some reliability growth models assume perfect correction. [Brocklehurst & Littlewood 92] assessed the effectiveness of 8 software reliability growth models, which indicated that no one model represents a '*panacea*' and that reliability models need to be selected on a case by case basis and only modest claims can be made towards software reliability, due to the hours of testing required to satisfy higher reliability claims. Also, with the ever-increasing trend towards incremental software builds, a purist view might maintain that any change to the software creates a new program and therefore the evaluation of the program should start afresh even when a localised correction has occurred.

[13] [Musa 89] defines a static reliability growth model in which the software and the operational profile remain unchanged, i.e. software faults, discovered by an observable software failure are not corrected. However, this type of model is not the norm.

The major issue with reliability growth models is the time required to quantify ultra-reliability i.e. a failure rate $<10^{-7}$ makes the use of reliability growth models currently infeasible due to the 'unforgiving' law of diminishing returns as [Killer & Miller 91], [Littlewood 89] and [Butler & Finelli 96] indicated. For example: [Bertolino 96] noted that to demonstrate one failure per X hours of execution requires approximately 100X of testing as shown in Table 4.

Failure Rate	Hrs of Testing
10^{-3}	100000 (11 Years)
10^{-4}	1000000 (114 Years)
10^{-5}	10000000 (1141 Years)
10^{-6}	100000000 (11415 Years)
10^{-7}	1000000000 (114155 Years)

Table 4: Testing Hours Required to gain a Failure Rate using 10X

[Butler & Finelli 96] use the following equation[14] to gain similar results to Table 4 as shown in Table 5.

Failure Rate	No of Failures(n)	One 'Blob' of S/W (r)	r/n (Mean) $Dt = \mu_0 \frac{r}{n}$	Hrs of Testing Mean*Failure Rate	Years of Testing
10^{-3}	10	1	10	100000	11
10^{-4}	10	1	10	1000000	114
10^{-5}	10	1	10	10000000	1141

Table 5: Hrs Determined by Using [Butler & Finelli 91] Formula

[Littlewood & Strigini 92] used a simple failure rate growth model based upon Mean Time To Failure (MTTF) and perfect operation to reduce the number of hours required to quantify software as shown in Table 6.

[14] The larger the size of r and n the smaller the statistical estimation error. This use of the replacement reliability growth model i.e. failure detected, lead to fault identification and correction.

Failure Rate	Hrs of Testing Required
10^{-3}	4600
10^{-4}	46000
10^{-5}	460000

Table 6: Hrs Determined by Using [Littlewood & Strigini 92] Failure Growth Model

To qualify the Sizewell B reactor software, which originally had a failure rate requirement of 10^{-4}, Table 6 was used to determine the number of test cases required. Since the system was an on-demand based system, they were required to execute 46,000 test cases, rather than hours of testing without failure to quantify the required failure rate. The software outputs were compared with a developed oracle. An additional benefit of using reliability growth models for software reliability is that you can start to quantify when to stop testing, as [Musa 89] indicated.

[Musa 89] also proposed the use of a 'compression factor' that could be applied to reduce the number of hours required to qualify software. For example if a factor of 2 was applied to the hours of testing required in Table 6, this would mean only 23,000hrs would be required quantify a failure rate of 10^{-4}. This is based upon the assumption that software testing 'stresses' the software more than operational experience. How do you quantify this compression factor is unknown and not all authors would agree with this approach e.g. Littlewood.

Significant hours of testing would still be required even if a compression factor were applied to quantify relatively moderate reliability failure rates i.e. 10^{-3} to 10^{-6}. Thus far more testing is needed than is currently applied to the majority of software projects, either safety or non safety critical applications. Therefore, the use of reliability growth models is rare and the Sizewell B project is one of the few examples of their use. However, with distributed networks and multi-core processors, the feasibility of gaining the number of testing hours required to quantify the required reliability in the software may now become possible, even for ultra-reliability.

The other issue with reliability growth models is that they are often applied not to the software artefact on the target hardware but on models based upon the expected behaviour of the intended software system. However, while this may enable the quantity of testing to be performed and automated, the real question is how we qualify and make these models truly

representative of the real system and gain the Independent Safety Advisor (ISA) or the licensing authority acceptance of these models.

2.4.4 Fault Based Adequacy Techniques

Error seeding was proposed by [Gilb 77] and [Mills 72] to estimate the number of bugs remaining in the software after testing. Seeded bugs (artificial faults) were planted into the program just before the program was tested. The artificial faults aim to be representative of real world faults. The testers would be unaware of the seeded bugs and the quantity of them. Figure 3 can be used to estimate the number of bugs left in the system involving error seeding.

$$\text{Number of Bugs In System} = \frac{\text{Number of seeded bugs} * \text{Number of detected bugs}}{\text{Number of detected seeded bugs}}$$

$$\text{Bugs Remaining in the system} = \text{Number of seeded bugs in system - Number of detected bugs - Seeded bugs not detected.}$$

Figure 3: Fault Seeding

The main usage of error seeding has been mainly in terms of assessing the adequacy of a test set or attempting to assess software reliability. [Zhu et al 97] provide a brief summary of the technique and discussed it in terms of measuring the quality of the software testing i.e. adequacy criteria. [Davies 93] discussed error seeding in term of assessing software reliability; [Davies 93] indicates a number of advantages of fault seeding over traditional methods of software reliability. [Davies 93] indicates four main disadvantageous of this technique:

- Not all bugs are equal.

- There exists no equilibrium between easy to detect and hard to detect defects. Some bugs are easier to find than others. Planting easier to find bugs would result in a biased estimation of bugs remaining with the opposite being true if the seeded bugs are hard to find.

- One bug may hide another bug.

- Fixing bugs often leads to new bugs being created.

[Beizer 92] makes the same point relating the first and last bullet point in a generalization of software bugs. [Bezier 92] defines 10 bug severities from mild[15] to catastrophic and infectious. The second bullet point raises two issues. Firstly, what constitutes simple and hard bugs?[16] Secondly [Demillio 78] defined a hypothesis named the 'coupling effect', which linked simple errors to complex errors, which later has been verified by experiment by [Offutt 89]. Therefore, the second bullet point may not be valid. Another issue not indicated by [Davies 93] is the randomness of the seeding process. This also affects repeatability. [Zhu et al 97] noted similar limitations to error seeding as did [Davies 93] and to overcome them [Zhu et al 97] noted that fault mutation was developed which introduce faults in programs in a more systematic fashion.

Fault mutation was proposed by [Demillio et al 78] as a method of determining how well a program (P) is tested and therefore mutation testing is a measure of test set adequacy rather than a testing technique as [Demillio et al 87] indicated. Mutation testing mutates the original program by the use of mutation operators (MOs). An MO represents real world faults that are applied to the original problem. Table 7 illustrates some possible MO. [Voas & McGraw 98] detail numerous MOs and language specific MOs.

The test set is executed over the mutated program. The output (O) from the mutated program is then compared against the original program to see if the test set has distinguished the mutated program from the original program. Figure 4 illustrates this: the original P^0 is used as the program baseline. The program is executed against the existing test set resulting in O^0. The original program P^0 is mutated by selecting a mutation operation MO^1 which results in P^1. The same test set as applied to P^0 is then applied to P^1, leading to O^1. The outputs are then compared between P^o and P^1 i.e. O^0 and $O^{1.}$

[15] [Bezier 92] defines mild as being: '*The symptoms of the bug offend us aesthetically; a mis-spelled output or a misaligned printout'*.

[16] The definitions of simple and hard bugs are not defined by [Davies 93].

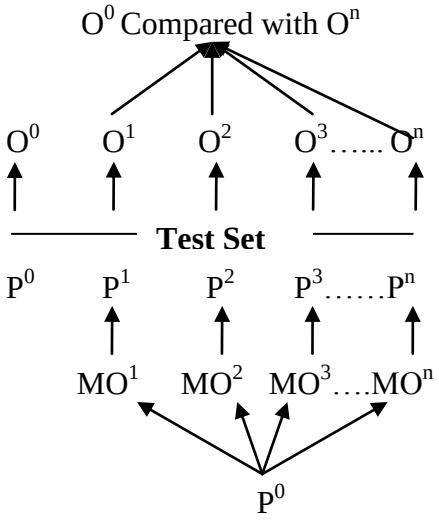

Figure 4: Mutation Testing

Mutants are said to be '*dead*' if the test set distinguishes them from the original program (i.e. there is at least one test case which gives rise to different observable results). '*Live*' mutants are those mutants that are not distinguished from the original program and require further investigation. [Delillio et al 78] defined live mutants as those mutants that give the same results as the original. This may be due to two reasons according to [Delillio et al 78] and [Woodward & Halewood 88]. Firstly the test set might simply be inadequate to detect the mutants; further tests could be created to kill these live mutants therefore improving the quality of the test set. Secondly it might be that no test case could ever distinguish the mutants and the original; the mutants are declared as '*equivalent*'. [Voas & McGraw 98] describes this latter issue as the '*fly in the ointment*' to mutation testing, since without manual intervention it usually cannot be determined if the mutant is equivalent. If there exist thousands of equivalent mutants this therefore impacts the effectiveness of mutation testing in assessing adequacy of the test set. [Zhu et al 97] indicated that equivalent mutants only normally account for a small percentage. [Offutt et al 93] claims equivalent mutant account for 8.8% of mutants. [Zhu et al 97] indicate that the major expense of mutant testing is perhaps the human cost in determining equivalent mutants.

To measure the adequacy of a test set on P a mutation Score (MS) as shown in Figure 5 is used. It measures the ability of a test set to distinguish mutants from the original program.

$$MS = \frac{D}{M - E}$$

D = Number of dead mutants
M = Number of total mutants
E = Number of equivalent mutants

Figure 5: Mutation Score

An MS of 1 indicates that the test set is fully adequate in detecting that class of non equivalent mutants, a low MS would indicate that the test set is poor in detecting those mutants and therefore, those class of faults may be present in P.

[Woodward & Halewood 88] highlighted the issue of how a dead or live mutant is defined. This therefore, has implications in what results should be used to compare the original and the mutated program. One method would be to compare the original and mutated program outputs i.e. screen or files. The original proposed method used a character-by-character comparison [Lipton 78]. A less stringent criterion of comparing non-blank characters was also proposed. [Woodward & Halewood 88] indicated two extreme cases to highlight the issue of live and dead mutants. If a program generated no output, all mutants would remain alive. While inserting output statements i.e. print statements in the appropriate places in the program every mutant may be seen to be dead. Therefore the definition of the mutant being alive or dead is dependent on the granularity of capturing when the comparison takes place. The granularity of observation of the mutation is one of the reasons for different forms of mutation testing.

Some of the early pioneers of mutation testing e.g. [Demillio 78], [Hamlet 77] applied one mutant operator to the original program, and then compared the final output of the 'whole' executed program. This mutation approach is termed as '*strong mutation*'. However, one of the main disadvantages is the large number of mutants generated. [Howden 82] estimated for an n-line program the number of mutants is of order n^2. [Offutt et al 93] noted similar results: a 78-line Fortran program generated 7435 mutants. [Offutt et al 93] also indicated that the majority of the computational cost occurs when running the mutants against the test cases. [Offutt et al 93] refers to studies done by [Budd 80] which indicated the number of mutants is approximately proportional to the '*number of data references times the size of the set of data objects.*' For a 7-line function this included 44 mutants. For these reasons [Offutt et al 93] proposed selective mutation based upon studies of simple programming errors based upon an

early study by [Mat 91], who established a set of 22 MOs for the Fortran 77 programming language as shown in Table 7.

Mutation Operator (MO)	Description
AAR	Array reference for array reference replacement
ABS	Absolute value insertion
ACR	Array reference for constant replacement
AOR	Arithmetic operator replacement
ASR	Array reference for scalar variable replacement
CAR	Constant for array reference replacement
CNR	Comparable array name replacement
CRP	Constant replacement
CSR	Constant for scalar variable replacement
DER	DO statement end replacement
DSA	.DATA statement alterations
GLR	GOTO label replacement
LCR	Logical connector replacement
ROR	Relational operator replacement
RSR	RETURN statement replacement
SAN	Statement analysis
SAR	Scalar variable for array reference replacement
SCR	Scalar for constant replacement
SDL	Statement deletion
SRC	Source content replacement
SVR	Scalar variable replacement
UOI	Unary operator insertion

Table 7: Mothra Mutation Operators
(Quoted Directly from [Offutt et al 93])

[Mat 91] further proposed the omission of Scalar variable replacement (SVR) and Array reference for scalar variable replacement (ASR) MO's due the number of mutants generated by those MO's. [Offutt et al 93] proposed extending this to omitting the next prevalent MO. [Offutt et al 93] named this N-selective mutation.

In an attempt to improve mutation efficiency e.g. reduced computational cost [Howden 82] developed '*weak mutation*'. This is based upon mutating simple components in the program. Simple components included references to variables, arithmetic expressions and relations and boolean expressions. Weak mutation is localised to that component. [Woodward & Halewood 88] described weak mutation as change and undo immediately before and after each single

execution of a component, whereas strong mutation testing is described as change and undo before and after the execution of the whole program. [Howden 82] indicated that weak mutation MOs could be applied simultaneously, instead of requiring separate program execution runs as in strong mutation. Weak mutation requires that tests sets execute the original and mutated components to distinguish between the original and mutated components i.e. compute a different value between the mutated and original program. Different values would be generated for different components. The total MS for an entire program may compute the same. Conversely the entire program may compute the same MS, but different component values. Therefore a high MS score does not ensure that classes of errors generated from the mutant operators are avoided.

To overcome the weakness of weak mutation and to improve efficiency compared with strong mutation testing, [Woodward & Halewood 88] derived '*firm mutation*' which has a wider scope than weak mutation but lesser scope than strong mutation testing. Firm mutation testing is where simple errors are introduced into a program which persists for one or more executions, but unlike strong mutation, not for the whole program execution. Trace mutation testing is referenced by [Demillio 87] to [Brooks 80] whereby instead of comparing programs output, program traces are compared. However, there would appear to be little further research in this area.

[Delamaro 01] applies mutations for integration testing using call graphs to establish the interface calls and injecting faults into the interfaces. The paper focuses on the 'C' programming language and examines four ways in which C programs pass data from software units. These are: the parameters passed by value, parameters passed by reference, global variables and return values. Faults were injected based upon these four approaches. These interface mutation operators were then applied to a '*Unix Sort Utility*' as a practical case study example. Statistical evidence of the results is provided, generated from the results of injecting the mutations and executing the tests. The results according to [Delamaro 01] indicated that Interface Mutation (IM) can be applied with a relative small number of mutations, showing test case adequacy. However, the costs of undertaking IM may prohibit the use of it.

Mutation testing is based around two assumptions. The first being the '*competent programmer hypothesis'* that programmers create programs that are very nearly correct. Secondly there

exists a '*coupling hypothesis*' between simple faults and complex faults, and so tests that reveal simple faults will also reveal more complex ones. [Demillo et al 78] defined it as:

'Test data that distinguished all programs differing from a correct one by one simple errors is so sensitive that it also implicitly distinguishes more complex errors.'

Therefore, you do not need to generate complex test cases to find complex bugs, since by finding simple faults you find complex faults. [Demillo 87] notes that these two assumptions have been studied from the theoretical and experimental viewpoints. [Offutt 89] undertook an experiment that used simple faults to generate complex faults. Simple faults were modelled by using MOs. Complex faults were modelled by applying multiple mutations to the program simultaneously i.e. 2 order mutants were combining two MO's with n order being n MO's. The results supported the coupling effect. Later [Offutt 92] conducted an additional experiment but this time examining the mutation coupling effect. That is:

> *Complex mutants are coupled to simple mutants in such a way that a test data set that detect all simple mutants in a program will detect a large percentage of the complex mutants.*

Again [Offutt 02] results supported this hypothesis. However, [Perry 00] noted that studies have been undertaken for these two hypotheses and indicated that these two hypotheses do not hold. However, [Perry 00] does not provide any references to support this. [Marick 91] partly supports [Perry 00]'s view relating to the coupling effect. [Marick 90] pointed out that [Offutt 89] uses simple faults to generate complex faults, which only accounts for a small percentage 8% of real world faults. [Marick 90] found that 47% of faults related to omitted code and 22% of faults related to complex faults in terms of omitting code and changes in code. [Marick 90] noted that strong mutation testing may have low effectiveness and low cost effectiveness. This was based on [Marick 90] findings that only 23% of the faults examined in programs generated by existing mutations and only 40% of the faults generated are typically made by programmers. The overall conclusion made by [Marick 90] is that there is no evidence against it or for it. (This relates to weak mutation testing.)

2.4.5 Execution, Infection and Propagation (PIE)

[Voas 91 et al], [Voas 92], [Hamlet & Voas 93] and [Voas & McCraw 98] proposed using weak mutation as the infection technique as part of the Execution, Infection and Propagation (PIE)[17] analysis to assess software testability i.e. the ability of a program to hide or reveal faults[18]. PIE analysis 'identifies locations in a program where faults, if they exist are more likely to remain undetected during testing'. Locations are assignments, input statements, output statements or a condition parts of a 'while' or 'if' statement.

PIE is made up of three phases, with each phase generating a separate probability estimate for each location. How to generate these three probability estimates are detailed in [Voas 92], [Voas et al 91] and [Voas & McCraw 98]. Execution analysis records the locations executed by each test set input i.e. inserting a data logging statement in each location to record that location execution. As [Voas et al 91] indicated '*each execution estimate is a function of the program and an input distribution*'. Infection analysis uses a set of semantically significant mutants. A mutant from the set is selected and the code is mutated and executed numerous times. The data state from the original program is compared with the data state of the mutated program to see if the data state has been infected. The result is an infection estimate for each mutant at each location. [Voas 92] defines program data state as a set of mappings between all variables and their values at the point of execution. The propagation analysis estimate the probability of that infected data state propagates to the program output. [Voas et al 91] indicated that this is done by repeatedly perturbing the data state that occurs after some location, by amending one live variable in the data state in each execution. A live variable is a variable that could affect the output of a program. For example a dead variable would be a variable that is defined but not referenced. Therefore, this would have no impact on the output of the program. The propagation estimate is calculated by examining how the force changes in the data state affects the program output. [Voas & McCraw 98] indicate that a high probability scores indicate a high degree of testability and low scores the opposite.

[Voas and McCraw 98] indicated that PIE algorithms generate a large quantity of data from the different analysis and different locations. Therefore, [Voas et al 91], [Voas 92] and [Friedman 95] provided ways of collapsing this data into a precise testability metric. The probabilities

[17] [Voas 92] indicate that it should be called EIP but PIE is more memorable.

[18] [Hamlet & Voas 93] define testability as in terms of the PIE model as '*The testability of a Program P is the probability that if P contain faults(s), P will fail under test.*'

from PIE are used to generate a sensitivity testing location indicator, which indicate how much testing is required to reveal faults at each location. [Voas et al 91] indicated a method to calculate a location sensitive is by multiplying the locations execution estimate, minimum infection estimate and minimum propagation estimate. A sensitivity score of one would indicate the program would fail on the first test if a fault were present. A sensitivity of 0.01 would indicate that 100 tests would be required at that location to reveal a fault, if one existed. A program sensitivity score could then be generated for the probability scores obtained from each location. [Voas 92] did note a word of caution relating in the use of PIE. If locations are touched infrequently this impacts the infection and propagation analysis. Therefore, [Voas 92] proposed ignoring such locations and indicated that PIE is '*only viable for frequently executed locations*'.

A critique of PIE undertaken by [Al-Khanjari et al 02], who in general supported the finding of Voas, in that programs with low (zero) or high sensitivity as low or high testability. However, [Al-Khanjari et al 02] did note that final results are 'quite sensitive' to minor variations in the parameters applied to the calculations. [Al-Khanjari et al 02] also indicated that automation is essential to enable the use of PIE due to its 'complex sophisticated calculations'.

[Voas 03] propose the use of 'Interface Propagation Analysis' (IPA), for assessing Commercial-Off-The-Shelf (COTS) applications. Since the source code is not normally available, anomalies are injected into the data feeds that connect the binary components. This requires access to the interfaces between the binary formatted components. For example DLLs, calls to the Operating System (OS), calls to an external database etc. Once the anomalies have been injected, IPA observes how these anomalies may have been propagated and how forms of anomalies may have been created. [Voas 03] noted '*this form of fault injection is useful for impact analysis and robustness testing.*' But [Voas 03] also noted that due to the high cost of performing software fault injection at the source code level the majority of software fault injection applied today is at the interface level.

2.5. What do Empirical and Case Studies tell us about Software Testing?

This section focuses on empirical studies that examine the effectiveness and efficiency of software testing techniques. To complement these empirical studies and to provide a 'real world' experience of software testing techniques, case studies by [Selby 90], [Berling and

Thelin 03] and [Ng, et al 04] have also been examined. Primary empirical studies over the last four decades by [Myers 78], [Basili & Selby 87], [Lauterbach, Randall 89], [Kamsties & Lott 95], [Wood et al 97], [Laitenberger 98], [Runeson, Andrews, 03], [Andersson et el 03], [Jursito, Vegas 03] and [So et al 02] show no consensus on the effectiveness or the efficiency of the software testing techniques applied. The [Runesson et al 06] secondary empirical study encapsulates the results from some of these primary empirical studies and indicates the same result.

Table 8 summaries the majority of the empirical studies reviewed and shows the following:

- The techniques and programming language used.
- Lines of code (LOC) of the sample programs used to assess the testing techniques.
- Complexity of the sample program.
- Time allocated to each controlled experiment.
- Subjects used i.e. students or professionals.
- Number of defects in each sample program and results.

The techniques investigated in the majority of the primary empirical studies tend to be '*state of practice'* rather than '*state of the art*'. The majority of the papers applied functional testing based upon equivalence partitioning, boundary value analysis, some form of structural coverage criteria and code reading. Other black box functional based testing techniques as detailed in [Beizer 96] and [Binder 00] such as control flow, data flow and state based testing were not applied. Only [Basili and Selby 87] provides specifications for the software artefacts directly in the paper. Therefore in the majority of cases it was not possible from the papers reviewed what functional testing techniques may have been more effective and efficient than equivalence partitioning and boundary analysis. Structural coverage criteria varied from the simplest form such as statement coverage as applied by [Basili and Selby 87] to more stringent coverage criteria as applied by [Kamsties & Lott 95], i.e. branch, multiple-condition, loop and relational-operator coverage. Only one of the empirical papers applied data flow analysis [So et al 02].

The empirical studies in part replicate the findings in the [Ng 04] survey of software testing practices of 65 organisations in Australia. The survey indicated that black box testing comprising boundary value analysis and random testing was the most commonly applied technique, adopted by 29 out of 65 organisations. However, this survey also noted that 18 out

of 65 organisations applied dataflow analysis, and 3 out of 65 applied mutation testing. However, no organisations reported the use of symbolic analysis, e.g. SPARK or MALPAS.

A case study by [Selby 90] investigated a collection of NASA systems and a manufacturing system that indicated bias of test sets toward defects distributions. [Selby 90] discovered that for NASA systems the first 35% of testing detected 55% of all failures and the last 35% of testing detected 12% of all failures. In the manufacturing system, the first 15% of test cases detected 67% of the high severity failures and 50% of all failures. The last 33% of test cases detected the last 10% of high severity failures. [Selby 90] also concluded the following: (Quoted directly from [Selby 90]):

- *Multiple fault detection and testing phases may result in a significant increase in reliability or none at all.*

- *Composite measures of system reliability did not adequately reflect reliability at the function or compound level.*

- *Developers were biased towards portions of systems that would be heavily tested.*

- *Fault proneness of reused or modified components was 74 percent less than that of newly developed components.*

- *Systems with more reused software had lower components development effort, but not lower component fault proneness.*

[Berling & Thelin 03] discovered during their case study of V & V activities at Ericsson in Sweden that programmers often correctly implemented the code even though the requirements were incorrect, since programmers had more domain understanding. However, [Berling & Thelin 03] discovered that this lead to lower level design issues such as timing and interface faults.

[Runeson et al 06] and [Laitenberger 98] attempted to generate direct comparisons of the controlled experiments from other empirical studies. [Laitenberger 98] noted that fault defect

rates were observed between 30 – 50% from the papers[19] reviewed in that paper. Papers like [Kamsties & Lott 95] and [Jursito & Vegas 03] follow the same methodology and use similar programs as [Basili & Selby 87], but they are different. [Kamsties & Lott 95] include an additional step for fault isolation i.e. detection of the fault and time required to undertake this step. The C programming language is used instead of Fortran or Simpl-T. [Kamsties & Lott 95] structural coverage criteria include branch, multiple-condition, loop and relational-operator coverage, [Jursito & Vegas 03] use branch and statement coverage, [Basili & Selby 87] used only statement coverage. [Myers 78] did not define specific functional or structural coverage criteria. Other papers like [Laitenberger 98], [Lauterbach & Randall 89], [So et al 02] do not use the same type of programs as used by [Basili & Selby 87]. [Lauterbach & Randall 89] use post developed code from USAF and DOD and [So et al 02] used a Launch Interceptor Programs (LIP). Time allocated to the techniques also varies from paper to paper. Some papers like [Kamsties & Lott 95] and [Jursito & Vegas 03] apply no specific time limits, while other papers like [Runeson & Andrew 03], [Andersson et al 03] define maximum amounts. [Jursito & Vegas 03], [Kamsties & Lott 95], [Basili & Selby 87] and [Woods et al 97] all use code reading via step abstraction. [Myers 78] does not. [Basili & Selby 87] code contains no comments, while [Kamsties & Lott 95] contains a small number of lines of comments.

The programs and programming languages applied in the studies vary. [Myers 78] used a formatting program in PL/I program based upon an Algol program, while [Basili & Selby 87], [Kamsties & Lott 95] and [Jursito & Vegas 03], both contain a similar formatting program, [Basili & Selby 87] was developed in Fortran and [Kamsties & Lott 95], [Woods et al 97] and [Jursito & Vegas 03] was developed in C. [Lauterbach & Randall 89] did not define the programming language applied, while Pascal was used in [So et al 02]. None of the papers reviewed focus on the issue or even discuss the issue of software programming language and if different language characteristics affect the effectiveness and efficiency of the testing techniques applied. [Cullyer et al 91] discussed a number of issues relating to the selection of programming languages for safety critical projects and the need for a well defined sub language for safety related projects (a safe subset). All the empirical papers reviewed ignore such issues.

One interesting observation by [Lauterbach & Randall 89] was that the buggiest software units were those units with low complexity and smallest LOC counts. However, [Lauterbach & Randall 89] does not discuss the type of faults found. [Jursito & Vegas 03] examines the

[19] These papers included: [Hetzel 76], [Myers 79], [Basili & Selby 87], [Kamsties 03] and [Wood et al 97].

location of the fault in the source code listing in relationship to the detection of that fault i.e. in the upper or lower quartile. The findings indicated that the location of the fault in the source code listing makes no difference in fault finding. A number of the papers ([Basili & Selby 87], [Woods et al 97], [Kamsties & Lott 95] and [Jursito & Vegas 03]) indicated that the effectiveness of the techniques is affected by the program being tested. However, these papers only considered a small number of different programs i.e. maximum of 3. [Perry 00] however, defined 16 different types of software programs[20] and each type may exhibit similar types of defects.

[Selby 86] is one of the few papers reviewed to indicate a bug severity when post reviewing a manufacturing based system. [Thelin & Berling 03] define a bug severity based around the need for a new software build and secondly the number of days to fix the bug. [Lauterbach & Randall 89] refers to bug severity for future investigation in their paper, but do not provide one for the defects discovered in their papers. Other papers like [Basili & Selby 87], [Kamsties & Lott], [Jursito & Vegas 03], [Woods et al 97], [Myers 78], [Laitenberger 98], [Runeson & Andrew 03] and [Andersson et al 03] just ignored the issue. [Beizer 90] defined 10 different defect severities. [IEEE 1044] defined 5 (Urgent; High; Medium; Low and None). Defect severities are important but are subjective. Severity of defects may be seen very differently from the software developer and end user prospective. Similarly a technique that discovers more critical failures than less critical ones may be better. Furthermore, it may be possible to work around minor defects.

The empirical studies indicated that for a single defect detection method they achieve a fault detection rate of between 25-50% for inspection and 30-60% for testing (functional and structural). Therefore, the validity of the results from these empirical studies is either flawed, (i.e. other factors are involved in increasing fault detection effectiveness) or software is released containing numerous software faults. [Myers 78], [Selby 86] and [Wood et al 97] indicated that the techniques applied are complementary while [Laitenberger 98] indicates this is not the case. [Myers 78] discovered that defect detection increases between 26-84%[21] when

[20] Batch, Event Control, Process Control, Procedure Control, Advanced Mathematical Models, Message Processing, Diagnostic Software, Sensor and Signal Processing, Simulation, Database Management, Data Acquisition, Data Presentation, Decision and Planning aid, Pattern and Image Processing, Computer System Software, Software Development Tools.

[21] This range was calculated by using the data provided in [Myers 79] paper.

the techniques are combined. [Wood et al 97] showed an increase of between 23-58%[22] in fault detection when combining testing techniques over the best single technique. These empirical study findings led the [Runeson et al 06] secondary study to indicate that primary and secondary defect detection plays a critical role in defect detection. However, [Selby 90] indicated multiple levels of fault detection may or may not increase software reliability. Industrial practice in general uses multiple levels of fault detection on both sides on the V-Model during the whole software development life cycle. In doing so the majority of the empirical studies reviewed, ignore this fact.

[Selby 86] indicated that the six combined techniques [23] detected 17.7% more program faults on average than did the three single techniques[24], which was a 35.5% improvement in fault detection. However, [Laitenberger 98], compared code reading followed by structured based testing. The results indicated that code reading detected 38% of errors, which [Laitenberger 98] claims is consistent with other empirical studies findings. However, structural testing detected 9% of errors which according to [Laitenberger 98] is well below the range indicated by other empirical studies i.e. 34-55%. Therefore, [Laitenberger 98] concluded code reading and structure coverage detected the same type of defects and therefore these two techniques are not complementary.

Numerous papers in the testing literature e.g. [Myers 78], [Beizer 84], [Binder 00], [Nakajo & Kume 91], [Lutz 92], [Perry 87] have stressed the importance of interface faults, both at the software integration level or at the system level i.e. software and hardware integration. [Nakajo & Kume 91] and [Lutz 92] both separated interface faults into three board categories: 'program', 'human' and 'process'. [Nakajo & Kume 91] noted that 56.9% of all program faults were interface faults. [Lutz 92] noted that 36% of safety critical program faults were interface faults for the Voyager space program and 19% for the Galileo space program. [Lutz 92] noted the primary cause of safety critical interface faults were misunderstandings in the hardware interface specifications with 67% for the Voyager and 48% for Galileo. Process flaws such as

[22] This range was calculated by using the data provided in [Woods 97] paper.

[23] Two individuals code reading; one individual code reader and one individual functional testing (Equivalence partitioning and boundary value analysis); one individual code reading and one individual structural testing (100% statement coverage), two individual functional testing, one individual functional testing and one individual structural testing and two individual structural testing.

[24] Code Reading, Functional Testing (Equivalence partitioning and boundary value analysis) and Structural Testing (100% statement coverage).

flaws in control system complexity, i.e. interface not adequately identified or understood, accounted for 70% of interface faults for Voyager and 85% for Galileo, of which 56% and 87% were safety critical related interface faults. This was down to two key factors, in that the interface specification was not documented or not communicated to other teams.

Only a limited number of software-software interface faults are focused on in the empirical studies. They focus on the actual fault and not the actual causes. The empirical studies avoid hardware to software integrations faults. [Perry 87] defined 15 interface fault taxonomies. [Selby & Basili 87] and [Myers 78] faults would only have covered 2 out of the 15 categories defined by [Perry 87] i.e. Data structure alteration and inadequate error processing. The empirical papers reviewed would barely cover a handful of the interface faults detailed in the testing literature i.e. [Perry 87], [Myers 79], [Binder 00] and [Bezier 90].

[So et al 02] indicated that voting (back to back testing[25]) was the most effective in defect detection but noted that this approach would not be applied due to cost. However, [Selby 86], [Woods et al 97] and [Myers 78] all indicated that defect detection effectiveness increased when the same technique is applied by independent individuals. [Myers 78] also indicated that this is cost effective since the different individuals detected different defects. The reasons why this may be the case are unclear. However, one reason may be the experience of the subjects involved in the empirical studies. [Selby 86] indicated that experience plays a critical role in increasing fault detection effectiveness. [Basili & Selby 87] used professional and student subjects, while [Kamsties & Lott 95] used students with programming experiences and [Jursito & Vegas 03] used subjects with little programming experience, [Woods et al 97] also used students. [Basili & Selby 87], [Lauterbach & Randall 89] and [Myers 78] are the only papers reviewed which use professional programmers and not only students. [Lauterbach & Randall 89] noted that subjects are more important to defect detection than the techniques applied. While [Basili & Selby 86] do not make this specific point, they did highlight different subjects gained different results, with professionals being more efficient than students. These 'softer' issues related to software testing i.e. experience, may account for the variance in the technique effectiveness and require further research as [Runesson et al 06] indicated.

[25] [So et al 02] subjects tested a number of different versions of the program. Back-to-back testing entails a cross-comparison of outputs obtained from functional equivalent software components.

The empirical studies show no consensus on the effectiveness of the individual testing techniques. These techniques should be seen as state of practice rather then state of the art. However, they are still commonly applied today as [Ng et al 04] illustrated. More formal methods such as symbolic execution e.g. SPARK and MALPAS are rarely used outside the safety critical domain. The empirical studies indicated that the effectiveness of the techniques is dependent on a number of factors including the type of program and the expertise of the tester. The empirical studies focus on functional errors and ignore other factors that would impact findings faults, for example testability, non-functional properties, type of programming language applied, coding standards, fault masking etc. Real world applications all include some kind of non-functional requirements, such as safety, security, reliability, performance etc. This in part due to the small size of the programs used in the empirical studies experiments that do not permit such issues to be examined.

The testing techniques have shown limited effectiveness when applied in isolation but have been shown to be more effective when applied in combination. However, even here there is no complete consensus since [Laitenberger 98] illustrated that code inspection followed by structural coverage is not complementary and somehow code inspection inhibits structural coverage or in other words two structural based techniques find the same defects. The problem even with the empirical studies that combine the individual techniques is they do not take into account that testing is normally applied to multiple phases in the development cycle; therefore, these empirical studies results do not match reality.

Author	Techniques Applied	Language	LOC	Complexity	Time	Subjects	Defects	Results
[Myers 78]	CR, FT, ST	PL/I	63	-	No specific time limit, but subjects recorded the time spent undertaking the testing	59 IBM subjects with varies experience	15	CR>ST>FT None of the techniques are very good in isolation, but used in combination
[Basili & Selby 87]	CR (Stepwise Abstraction), ST (SC), FT (EQ & BVA)	Simpl – T & FORTRAN	4 programs 145-365 48-144 ELOC	18-57	No specific time limit over the three days	32 Professionals & 42 junior and senior students	6-12	Prof CR>FT>ST Seniors CR=FT>ST Juniors CR=FT=ST
[Lauterbach & Randall 89]	CR, ST (BT), FT, RT	Not Listed	29-680	6-91	Allocated time and actual time recorded	4 Experiences testers	-	CR best for CSU BT best at CSC
[Runeson & Andrews 03]	CR, ST (BT)	C	Used two programs 190-208	27 for each program. 9 Procedures	2 hours for each technique	27 students	9 defects in each program	ST>CR Techniques are complementary
[Andersson et el 03]	UBR (Design) UBT	-	Used two different programs	-	3hrs 45mins 2hrs 45 mins	51 fourth year master level students	13 and 14 faults	UBR>UBT
[Wood et al 97]	CR, FT, ST (BT)	C	200 Used on three	-	3 hrs each exercise	47 students - have completed 2 years of	8-9 faults in each program	ST>FT>CR Combining the techniques lead

Author	Techniques Applied	Language	LOC	Complexity	Time	Subjects	Defects	Results
			different programs			programming classes.		to the best results
[Laitenberger 98]	CR, ST (BT, LC, MCC, RC)	C	132	-	2 hrs	20 students		Techniques are not complimentary
[Jursito & Vegas 03] Replication 1	CR (Step wise Abstraction), ST (SC, DC), FT (EQ & BVA)	C	200 Used on three different programs	-	No time limit	196 Students - Little experience	9 in each program	FT>ST>CR
[Jursito & Vegas 03] Replication 2	CR (Stepwise Abstraction), ST (SC, DC), FT (EQ & BVA)	C	200 Used on three different programs	-	No time limit	196 student – Little experience	7 in each program	FT=ST>CR
[Kamsties & Lott 95] Replication 1	CR (Stepwise Abstraction), ST (BT, MCC, LC, RC), FT (EQ & BVA)	C	235-251 Used on three different programs	-	Minimum of 3hrs no other time limit specified	27 students	6-9 faults	FT=ST=CR However, FT was best for efficiency
[Kamsties & Lott 95]	CR (Stepwise	C	235-251	-	Minimum of 3hrs no	23	6-9 faults	FT=ST=CR

Author	Techniques Applied	Language	LOC	Complexity	Time	Subjects	Defects	Results
Replication 2	Abstraction), ST (BT, MCC, LC, RC), FT (EQ & BVA)		Used on three different programs		other time limit specified			However, FT was best for efficiency

Table 8: Empirical Studies Result Summary

Table 8: Key

ST – Structured Testing, FT – Functional Testing, CR – Code Reading, BT – Branch Testing, MCC – Multiply Condition Coverage, SC – Statement Coverage, EQ – Equivalent Partitioning, BVA – Boundary Value Analysis, DC – Decision Coverage, RT – Random Testing, UBR – Usage Based Reading, UBT – Usage Based Testing, LC – Loop Coverage, RC – Relational Coverage, BC – Branch Coverage.

2.6 When Can We Stop Testing?

2.6.1 Introduction

This section focuses on when to stop software testing. For Airborne Software that follow [178B] and [178C] software unit testing is stopped when the appropriate level of source code or object code coverage has been achieved. The effectiveness and reliability of these and other stop testing criteria is the central focus of this thesis. Our four effectiveness and reliability research objectives are restated below:

1. To measure the test effectiveness of the three coverage criteria (Statement, Branch and MC/DC) mandated by a widely used commercial airborne software standard for safety critical software D0-178B[178B] and its recent updated version D0-178C [178C].

2. To measure the reliability of those three coverage criteria by comparing the effectiveness of multiple minimal size tests sets meeting these criteria.

3. To measure the effectiveness and reliability of the three widely used coverage criteria used in the commercial airborne safety critical software e.g. [178B] and [178C] with test sets with a small degree of redundancy. To add redundancy we plan to combine the different optimum coverage test sets.

4. To measure the test effectiveness of three reference test sets developed to test a DES algorithm.

This section provides an overview of research into adequacy criteria in the software testing literature.to provide context for our research objectives. The literature is primarily focussed on general purpose coverage criteria, with extraordinarily little on the use of domains specific test suites.

2.6.2 Can We Stop Testing Now?

A number of criteria have been proposed and [Goodenough and Gerhart 75] attempted to formalise this question by defining two requirements for a testing criterion: 'reliability' and 'validity'. However, [Weyuker & Ostrand 80] showed that these two requirements were not independent, since '*a criterion is either reliable or valid for any given software*'. Therefore, the testing literature has moved away from the [Goodenough and Gerhart 75] criterion, to focusing on the use of adequacy criteria as '*stopping rules'* for software testing and as a measurement of test quality. For example stopping testing after 100 percent statement coverage can be seen as a stopping rule or adequacy

criterion. This section examines the issues surrounding when to stop testing and different criteria that have been proposed.

Testing should be stopped when the goals of testing have been achieved. However, these testing goals can vary in strength and generally focus on the program structures or specification properties. For example different structural coverage criteria vary in difficulty. The statement coverage criterion is the easiest to achieve but is seen in the testing literature as the weakest structural coverage criterion as [Myers 79] illustrated. Therefore branch/decision structural coverage criteria are often seen as the minimum coverage criteria for most programs.

[Perry 00] indicated three major failures of testing:

- Not setting testing goals before testing starts.
- Testing at the wrong phase in the life cycle.
- Ineffective testing techniques.

[Perry 00] attempted to illustrate the optimal level of testing by using a modified economics supply and demand curve illustrated in Figure 6. Line cc in figure 6 show the cost of testing and line dd the number of defects found. The optimal extent of testing and defects discovered is at point E or equilibrium between test cost and defects discovered via testing. At any point to the left of E there exists under testing, defects discovered during testing outweigh the cost of testing. At any point to the right of E the cost of testing outweighs the defects discovered.

Figure 6 has considerable appeal but is an over-simplification. The shape of the defect line could be very elastic i.e. very responsive to the testing applied or inelastic i.e. have little effect with the testing. It is highly likely that the testability of programs varies between programs. A simple defect and cost curve does not take into account different criticalities of software. Whilst more rigorous V & V activities applied to safety critical software, than to computer gaming software, the shape of the defect and cost lines for different criticality software systems would be very different. Figure 6 also assumes that defects are all equal. This is not the case, as [Beizer 84] clearly indicated, with different defects having different consequences and impacts.

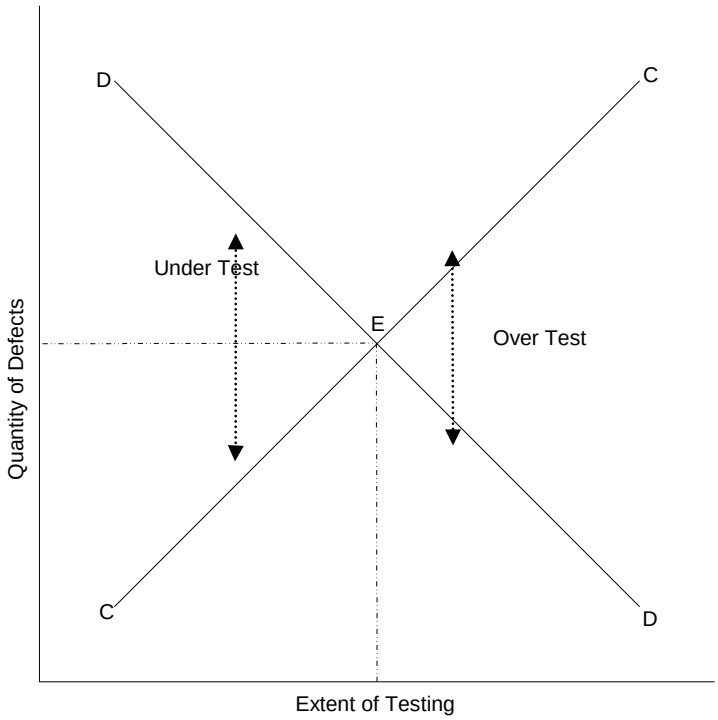

Figure 6: Generic Demand and Cost Curve
(Quoted Directly from [Perry 00])

Defect detection is affected by the quality of test cases as [Myers 79] indicated. Poor quality of test sets is encouraged by the goal of testing to find no defects. The primary goal of testing as [Myers 79] indicated is to show that a program contain defects. It is normally infeasible to test every possible path or input space through a software program, therefore absolute correctness cannot be achieved. Due to these limitations of testing, the objective of testing must shift away from '*absolute proof*' to a '*suitable convincing demonstration*' which implies a quantitative measure which in turns implies a statistical measure of software reliability [Beizer 90]. Therefore, the goal of testing is to demonstrate that the probability of failure due to hibernating defects is low enough to accept. The acceptance will depend on the type of system being developed; a video game will have a much higher level of acceptance than a safety critical system. Therefore, testing goals relate to relative correctness i.e. to capture specific classes of faults. How good the testing is, is dependent on the strengths and weaknesses of these goals or testing adequacy criteria.

Test adequacy criteria have generated much debate in the software testing literature. [Goodenough and Gerhart 75] raised the initial question 'what is a test criterion' to define an adequate test i.e. a test which if successful would imply no errors in the program under test. [Goodenough and Gerhart 75] define two requirements for a test criterion: Reliability and Validity. Reliability is that a test set always generates 'consistent' results to reveal defects. This is not saying the test set reveals all defects or the results are meaningful. Validity requires that every test always generate meaningful results, regardless of the ability of that test to generate consistent results. A test set which

incorporates these two requirements would successfully test a program. However, [Gardiner 99] indicated these two requirements were too stringent to enable practical adequacy criteria. [Zhu et al 97] noted that academic research questioned these two requirements. [Howden 76] indicated no computable criterion that satisfies the two requirements, and hence they are not practically applicable and [Weyuker and Ostrand 80] noted that the two requirements are not independent since a criterion is either reliable or valid for any given program.

Due to the issues surrounding [Goodenough and Gerhart 75] Reliability and Validity, the software testing literature has focused on the development of adequacy criteria as stopping rules and as measurements of test quality. Stopping testing after 100 percent statement coverage can be seen as a stop rule adequacy criterion. The level of structural coverage i.e. statement, branch etc. gained by the test set is an measurement criteria and moves away from declaring a test set as being good or bad. The '*stop testing*' rule can be seen as a continuous adequacy measure and code coverage is both commonly used as the adequacy stop rule and measure. Therefore, the adequacy stop criterion and adequacy measures are closely related as both [Zhu et al 97] and [Gardiner 99] note.

Testing goals can be applied to all phases in the development cycle and on different characteristics of testing i.e. functional, structural, dynamic and static. However, testing goals in the testing literature mainly focuses on the dynamic testing phases i.e. on the right side of the V-Model e.g. unit, integration, system and acceptance. Defining testing goal criterion early in the development life cycle limits the consequences of impacts on dynamic testing when delays occur in earlier phases. [Myers 79] notes one weak stop criterion is to stop when all tests executed pass without error. This commonly used criterion is dependent on how good the testing is. An even weaker criterion is to stop testing after a length of time or when resources are exhausted. While this type of stop criterion would appear '*laughable*' [Ng et al 04] discovered several organisations still using it. [Ng et al 04] also discovered that other stop criteria included testing after all critical or show stopping defects have been removed. However, as [Ng et al 04] noted the methodology used to determine this, was not formal or methodological.

[Myers 79] stop testing criteria preference is to estimate the number of errors in the software and then capture these defects during the testing phases. However, error estimation is not exact and as [Bezier 90] notes all defects are not equal. A more precise formal approach based upon defect counting is fault mutation. Testing would be stopped when all non-equivalent mutants are killed. Mutation testing illustrates a fault based adequacy criteria that examines the ability of a test set to find faults in a program. Other adequacy criteria include structure coverage to cover all or specific parts of the program. Three classes of structural measures include: control flow e.g. statement, branch; dataflow e.g. all definitions, all used; and program text based e.g. compound conditions, linear code sequence

and jumps as illustrated in Figure 7. The other adequacy criteria is error based, in that the techniques find specific types of faults i.e. domain based testing.

An alternative to program structure or specification criteria not defined in figure 7 is to use software reliability models to quantify when software testing should be stopped, as proposed by [Musa 89]. Reliability growth models were discussed in section 4. However, it should be noted that very few programmes used this approach, Sizewell B being one of the few exceptions. However, this is the only quantifiable approach currently available to determine when to stop testing. There has been little research in attempting to quantify how effective the currently applied software testing techniques are, in terms of finding defects and achieving overall software reliability.

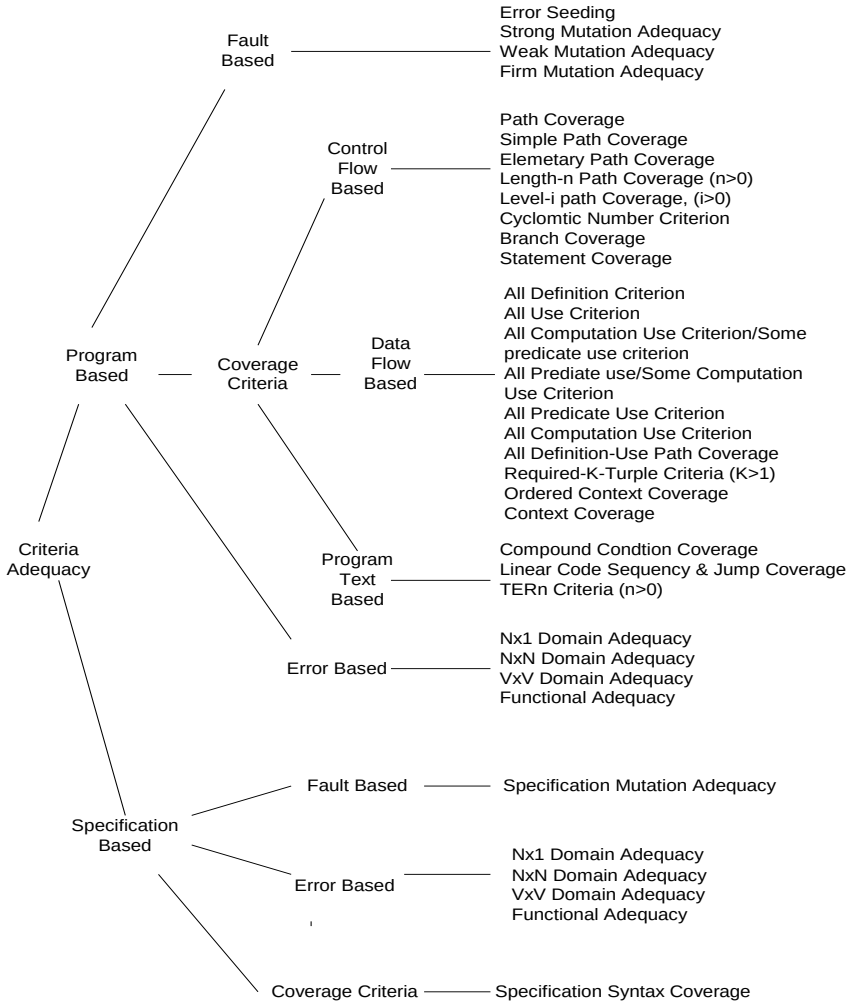

Figure 7: Testing Techniques Criteria Adequacy Taxonomies
(Quoted Directly from [Gardiner et al 00])

2.7 Test Set Sub-setting via Heuristic Search

A number of papers by [Offutt et al 95], [Harrold et al 93], [Harrold & Jones 08] and [Harrold & Jones 03] have investigated test set reduction. [Offutt et al 95] developed a heuristics based techniques called 'ping pong' that generated new test sets by either selecting tests starting from

beginning, middle or end of the test suite. The testing criterion was primarily based upon MS, with test cases being removed that do not improve the MS. A single empirical experiment based upon statement coverage was undertaken. [Offutt et al 95] focused more on reducing the test set size and the cost savings associated with requiring fewer tests to be conducted to meet the criterion than on the reliability of the criterion. The same is true with [Harrold et al 93], which applied a dataflow criterion to a number of small lines of code (LOC) procedures. This paper used a heuristics test reduction method based upon relationships to test cases and test requirements. By establishing the relationship of test cases to test requirements duplication can be observed and test case reduction can be performed. [Harrold & Jones 08] applied test-suite reduction based upon Statement reduction (code statements) and Statement Vectors – (set of statement executed by one test case) and examined the impact on fault localisation. [Harrold & Jones 03] re-ordered the execution of test cases to meet the MC/DC criterion to reduce the number of test cases required.

Our need of test set reduction is based upon meeting known criteria. Therefore, to examine how we could gain the optimum number of tests to achieve those coverage criteria from a large number of possible solutions we examined the following heuristic search based techniques:

- Greedy Algorithms
- Hill Climbing based algorithms
- Simulating Annealing
- Genetic Algorithms

A greedy algorithm uses a series of steps to find the solution, top down. It selects the choice with the greatest immediate improvement. However, such immediate 'greediness' may not achieve the global optimum. (Sometimes a better result might be obtained by forsaking immediate gain for longer-term gain. This is typical of so-called non-linear optimisation problems.) What makes greedy algorithm so popular according to [Corman et al 2009] and [Michalewize & Fogel 04] is it simplicity.

[Michalewize & Fogel 04] shows how greedy algorithms do not provide good solutions to complex problems with interacting parameters like SAT, TSP or NPL. [Corman et al 09] is more positive in the use of Greedy Algorithms and provides a number of examples were Greedy algorithms do produce an optimal solution i.e. minimum-spanning-tree algorithms, Dijkstra's algorithm for shortest path from a single source and Chvátals greedy set-covering heuristic. [Corman et al 09] unlike [Michalewize & Fogel 04] covers the '*Greedy –Choice Property'* – that you can assemble a globally optimal solution by making local optimal choices – greedy. Therefore you don't need to consider the subproblems. If you can prove that this is true then greedy algorithms can be used. To demonstrate

this point [Corman et al 09] uses the 0-1 knapsack problem and the fractional knapsack problem. In the latter a greedy algorithm can be used, while for the former it cannot, since it does not produce the optimum solution of the 0-1 knapsack problem.

A greedy algorithm design has the following steps according to [Corman et al 09]:

1. Cast the optimization problem as one in which we make a choice and are left with one sub problem to solve.

2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.

3. Demonstrate optimal substructure by showing that, having made the greedy choice what remains is a sub problem with the property that if we combine an optimal solution to the sub problem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

Hill Climbing is a local search method, which may find the local optimum, but may not find the global optimum in the search space. Hill Climbing generates a solution that meets the constraints of the problem. It then attempts to improve the solution by incrementally changing a single element of the solution, until no further improvement can be found. In short you start at the bottom of the hill and walk up to the top of the hill. [Corne et al 99] defines the steps in a simple Hill Climbing algorithm:

1. Begin – Generate and evaluate an initial 'current' solution s.
2. Operate – Change s, producing s' and evaluate s'.
3. Renew – If s' is better than s, then overwrite with s'.
4. Iterate – Unless a termination criterion is met, return to 2.

There exist a number of variants of Hill Climbing i.e. steepest ascent or stochastic hill climbing. However, have [Michalewize & Fogel 04] points out that Hill Climbing algorithms have a number of weaknesses:

- They usually terminate at solutions that are only locally optimal.

- There is no information as to the amount by which the discovered local optimum deviates from the global optimum or perhaps even other local optima.

- The optimum that's obtained depends on the initial configuration.

- In general it is not possible to provide an upper bound for the computation time.

Hill-climbing is fairly straightforward to implement. In addition, it often forms a component of more sophisticated techniques, e.g. to provide efficient fine grain optimisation at the end of a run of another optimisation technique. (Genetic algorithms for example may often produce solutions that are close to locally optimal solutions, but an aggressive hill-climb is used to climb the remaining part of the 'hill'.)

To overcome the local optimum problem with hill climbing based techniques a number of authors have recommended Simulated Annealing (SA) [Michalewize & Fogel 04]. An alternative to using SA is to use Genetic Algorithms (GA). GA is discussed in detail in section 3.1.5.1. Simulated annealing is based on a loose analogy with the annealing process of molten metals.

Figure 8 illustrates the structure of the SA algorithm. A starting solution is selected at random and evaluated. The search is guided by a parameter known as the temperature (T). The initial temperature is set to some 'high' value. The value of the temperature will be progressively lowered as the search proceeds. At each temperature level a number of candidate 'moves' are considered. In a move a new neighbouring solution to the current one is selected and evaluated. If the new solution is better than the current one, i.e. $eval(V_n)-eval(V_c) > 0$, it is replaced with the new solution ($V_c \leftarrow V_n$). Thus, this is the hill-climbing component of the algorithm: improving moves are always accepted. However, even if the new one is not as good, it may still replace the current one with probability $\exp(\Delta/kT)$ where $\Delta=eval(V_n)-eval(V_c)$, assuming we are maximising. Thus, the higher the temperature the more likely a non-improving move is to be accepted. Additionally, the worse a non-improving move is the less likely it is to be accepted. This allows the search to move downhill temporarily in order to escape a local optimum. After a set number of such moves have been considered the temperature is lowered and the process repeats until some stopping criterion is met (typically a maximum limit on moves considered or the number of consecutive non-accepted moves is reached)

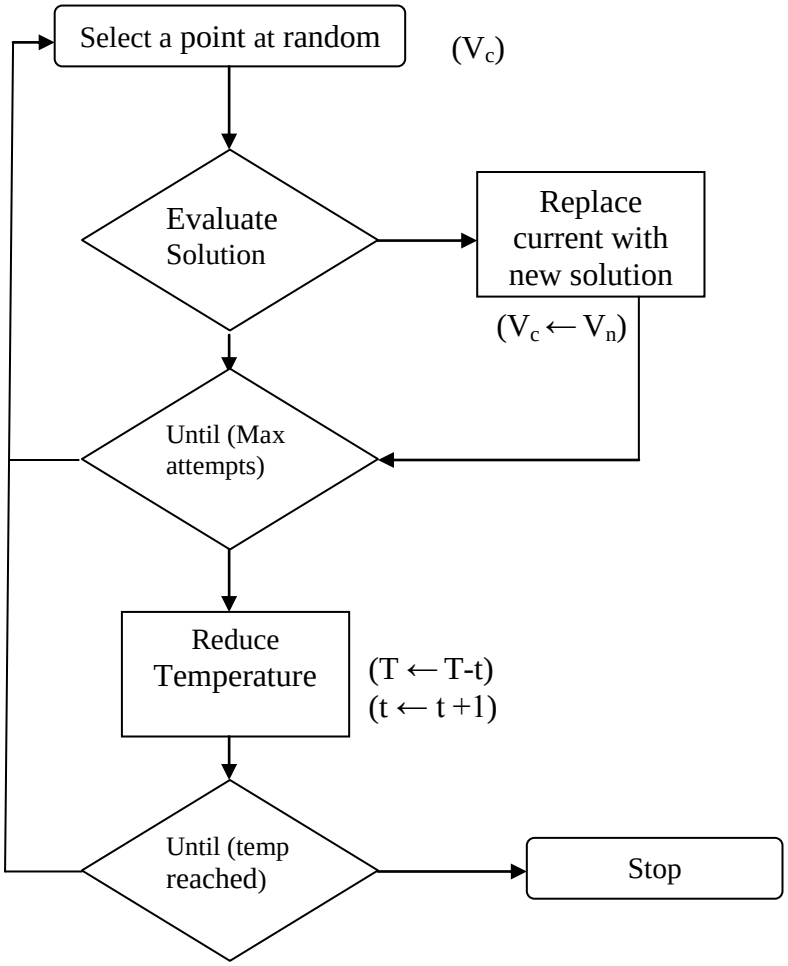

Figure 8: SA Structure

2.8 Summary

This chapter has provided a wide overview of the software testing literature – from fundamental concepts to characteristics of testing to when to stop testing. This literature review has shown by reviewing empirical studies in the effectiveness of software testing that there is no consensus on effectiveness of different testing techniques. However, none of these empirical studies has examined the effectiveness and reliability of statement, branch and MC/DC, reference test sets or use test automation. There is a clear research gap to examine the effectiveness and reliability of three widely used coverage criterions.

3. The Research Test Framework and its Components

This chapter describes the framework and the associated components that were used to conduct the experiments for this PhD thesis. The overarching requirement for the framework was as follows:

- To provide automated functions for:
    - Test Data Generation
    - Test Set Execution
    - Fault Injection and Mutation Score Generation
    - Code Coverage
    - Test Sub Set Extraction

The framework was generated to meet these requirements to allow rigorous determination of the effectiveness and reliability of coverage criteria. The framework covers:

- Overview of the framework (discussed in section 3.1.1)
- The automated instrumentation to capture source code coverage data (discussed in section – 3.1.2 Code Coverage).
- The automated generation of large samples of test data (discussed in section - 3.1.3 Test Set Generation).
- The use of JUnit to determine if each test has passed or failed (discussed in section 3.1.4)
- The automated generation of test sets satisfying an identified test objective, via the use of GAs. This enables us to generate a large sample of adequate test sets. Their fault-finding effectiveness can be determined by mutation methods (see below). The range of effectiveness scores across sets is an indication of the unreliability of the underlying criterion (discussed in section 3.1.5 Test Subset Extraction.
- To extract unique test sub-sets (no duplicated test cases) we developed a number of heuristic algorithms i.e. Simple Hill Climbing, Random-Hill Climbing, Greedy algorithm and Simulated Annealing all developed in C#. We also developed our own bit based Genetic Algorithm developed in C#. The basic design based upon [Coley 99] and [Goldberg 89]. We then compared our results with the results gained from using the ECJ GA framework based in Java. Since ECJ supported integer based GA and the parameter file inside ECJ enabled increased flexibility so that the GA could be applied to a range of programs under test rather than the developed bespoke algorithms developed. For these reasons it was decided to use ECJ to develop our GA to generate unique test subsets.
- 3.1.5.1 Genetic Algorithms.

- The automatic generation of mutants (discussed in section 3.1.6).
- The automatic generation of Mutation Scores (MS) for the test subsets (discussed in section 3.1.7).

3.1 Framework

3.1.1 Overview of the Framework

At a conceptual level the framework generates two tables. One table is the coverage table that records the coverage achievements of the tests. The other records the mutants killed by each test. The two tables are linked to each other via the test case numbers as illustrated in Figure 9.

The coverage table holds the coverage data for each test case. Each column entry indicates an executable element of code, while each row entry represents a different test case. A zero indicates that the element has not been exercised by that test; a one indicates that the element has been exercised by that test.

Likewise, the mutation table stores the effectiveness of each test case. Each column entry represents a mutant, while each row entry represents a test case. Similarly to the coverage table, a zero in the table indicates that that mutant has not been killed (i.e. remains undetected), while a one in the table indicates that mutant has been killed (detected). This table is used to calculate the 'mutation score' (MS).

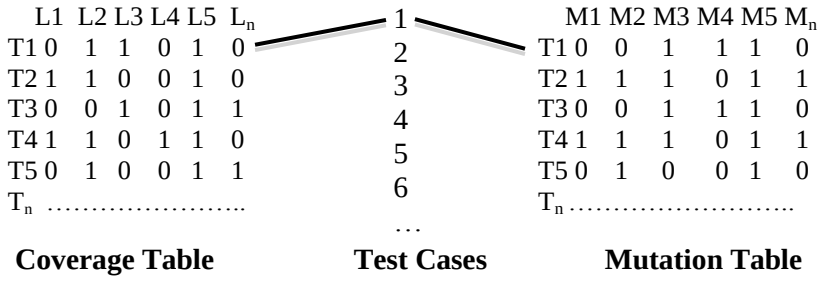

Figure 9: Conceptual View of the Framework

A coverage table (file) is generated for each testing objective (statement, branch and MC/DC).

The framework is made up of a number of developed and freeware components. The developed components have been developed incrementally as the framework evolved. The first component that was developed was the C# Wrapper. This component generates random test data by using internal random functions inside the C# libraries. JUnit is the test harness for executing the tests using the randomly generated test data. This data is read into JUnit by using the Read Test Input Data. To gain

code coverage for the programs under test a program called “Code-Cover” was used that instrumented the code with line counters that incremented every time that code statement was executed.

MuClipse was used for injecting faults into the programs under test. MuClipse applied a number of different mutation operators for injected faults. The Mutation Score Generator executed the JUnit tests over the mutated programs; while an additional component i.e. BitArray generated the overall Mutation Score. ECJ was used for test sub set generation for generating unique test sets to meet different the different coverage objectives. This therefore, enables us to measure the effectiveness and reliability of different test sets that meet the same coverage objective.

The components interact via the text file outputs of other components. Figure 10 shows the components and files generated by each of them. The framework consists of the following components as detailed in Table 9. A mixture of Java and C# was used in the development of the framework. C# was used since the author of this thesis had good domain knowledge of C#. However, while generating the framework, more freeware components supported Java i.e. test harness, coverage tools etc. then C#. Therefore, a mixture of the two languages was used. Since C# was developed in part from Java, there exist many similarities between the two languages. Since on the whole only the fundamental elements of the language constructs were used i.e. iteration, selection, arrays and file constructs. This made using the two different languages a relatively easy activity.

Component	Description of Use	Files Generated	Program Language
JUnit 3.8.1	JUnit provides the framework to run automated Java tests via the use of assert statements. The JUnit component in the framework also records the test case to run, test data and expected result.	JUnit_Test.Class Test_Inputs.txt	Java
Code Cover	Code-Cover is used to instrument the source code under test to capture coverage data.	*.clf (coverage log file)	Java
C# Wrapper	Randomly generates the test data for the JUnit test and call the JUnit test case to be executed.	Test_Output.txt	C#
Read Test Input Data	Reads the test data stored in the 'Test_Input.txt' file and passes this as a data string to the JUnit component.	-	C#
MuClipse	Java mutation[26] plug-in that generates mutants for the software under test.	*.java (Mutated java files)	Java
Mutation Score Generator (MSG)	Runs the test set over the mutated source code files via the Read Test Input Data component. Re-directs the JUnit test outputs to text files. Generates the summary results files for each mutant operator and the MS for each mutant operator applied. The MS summary for each mutant operator is also generated.	ResultX.txt (1..*) SummaryX.txt MS_Table_Full.txt	C#
Overall Coverage	Generates the coverage files for statement, branch and MC/DC in the form of '0' and '1' from extracting out the coverage data from each of the coverage log files.	Statement_Coverage.txt Branch_Coverage.txt MC/DC_Coverage.txt	C#
BitArray	Generates the mutant results. It stores the number of mutants killed by mutant type and overall MS for that test set.	Mutation_Table.txt	C#
GA – ECJ (Subset Extraction)	Java GA Eclipsed plug-in to generate unique test subsets based upon the required source code coverage.	Test_Cases.txt	Java

Table 9: The Framework Components

[26] Eclipse is an Integrated Develop Environment (IDE) that supports a range of different languages and plug-ins.

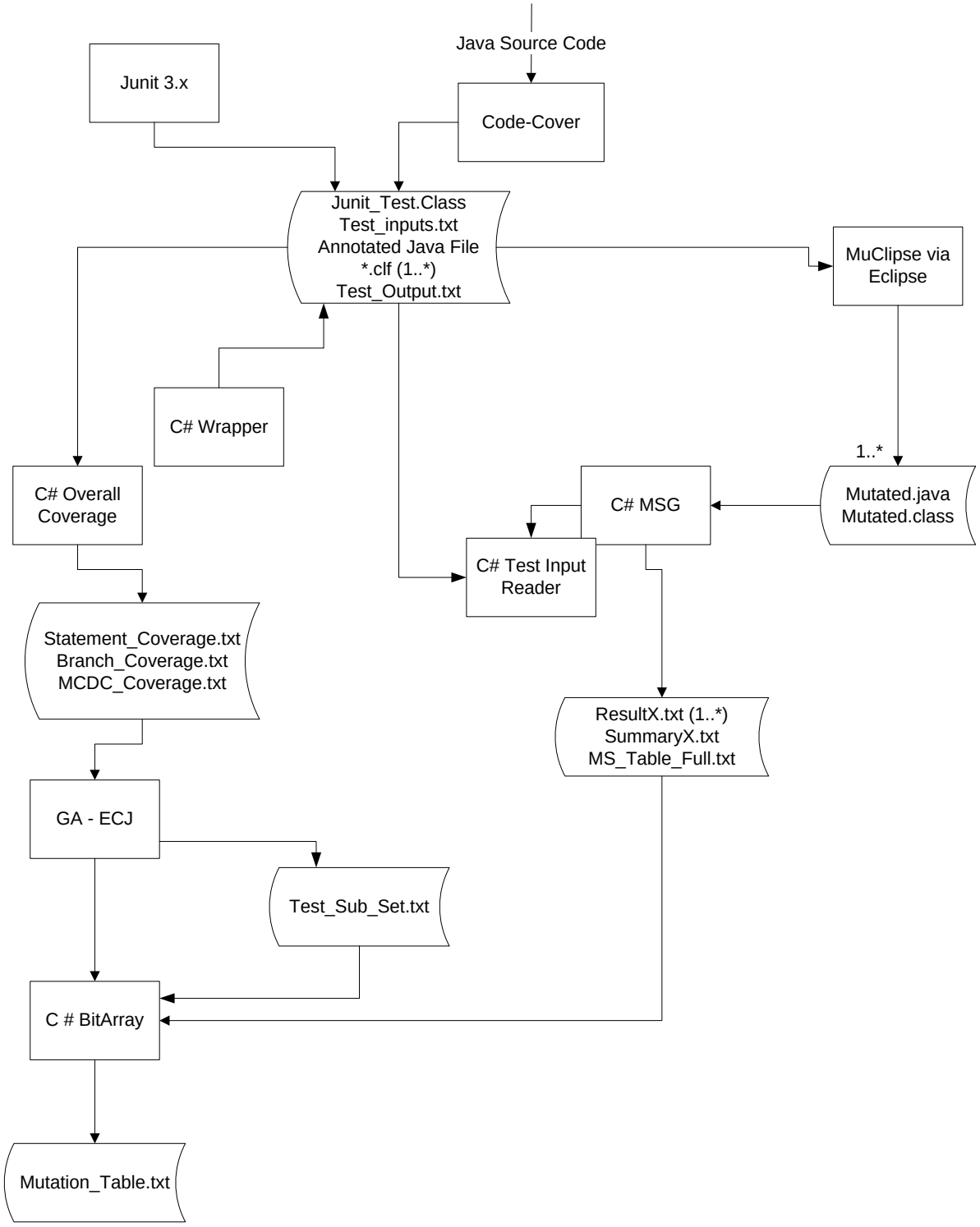

Figure 10: Framework Interactions

3.1.2 Code-Coverage

There are three different techniques to collect code coverage information, namely source code instrumentation, object code instrumentation and interpreter instrumentation. The majority of freeware Java coverage tools use either source code or object (byte) code instrumentation.[27]

The basic process for instrumentation is as follows:

1. Read the source file and parse it into a tree.

2. Annotate the parse tree with instrumentation that is subsequently used to count the number of times each statement is executed.

3. Write the annotated tree to a file. The file will contain Java source code with some additions.

4. Compile the annotated source.

5. Execute the program and collect instrumentation data.

6. Remove the annotations from the source code.

Java code coverage tools CoverLipse[28], Code-Cover[29], Ecl-Emma[30] and Cobertura[31] were all evaluated. Cobertura was executed via the Windows command line. All the other code coverage tools were Eclipse IDE plug-ins. These coverage tools are integrated into the Eclipse IDE framework.

To evaluate the coverage tools a number of small Java programs[32] were generated that included simple and complex conditional expressions inside Eclipse. The number of branches and modified condition decisions in the sample programs were worked out manually as a baseline. JUnit test cases were generated to exercise the sample programs. The coverage tools were then used to capture the code coverage and to detect coverage omissions. Additional JUnit test cases were generated manually to gain the required coverage.

[27] http://java-source.net/open-source/code-coverage
[28] http://coverlipse.sourceforge.net/
[29] http://codecover.org/index.html
[30] http://www.eclemma.org/
[31] http://cobertura.sourceforge.net/introduction.html
[32] The programs used were: Vector statistical package - http://math.nist.gov/javanumerics/jama/ (The Matrix Package), Calculator - http://klogd.users.sourceforge.net/ (It was the StringParser class that was used), Numeral roman year converter - "https://www.planet-source code.com/vb/ scripts/ShowCode.asp?txtCodeId=6305 &lngWId=2"

These tools have the capability to export the results of the code coverage achieved by the JUnit Test Set in HTML, XML or CSV files. However, none of these tools generates directly the data in the required format for sub-set extraction to be achieved.

A Java instrumentation package 'instr.instr.stmt'[33] that captures statement coverage was examined. We re-evaluated the use of Cobertura and Code-Cover. (Cobertura instruments directly to the bytecode and Code-Cover the source code.) Both tools were executed via the Windows commend line. From the Windows Command Prompt coberture instrumented the bytecode by using the following command line:

cobertura-instrument.bat [--basedir *dir*] [--datafile *file*] [--destination *dir*] [--ignore *regex*] *classes* [...]

Cobertura generates a metadata file (by default called coberture.ser) that records the method names, lines, branch conditions. When the instrumented PUT is then exercised the lines numbers and branch conditions are updated to indicate if that line or condition has been exercised. A separate command is required to generate reports:

cobertura-report.bat [--datafile file] [--destination dir] [--format (html|xml)] [--encoding encoding] source code directory [...] [--basedir dir file underneath basedir ...]

For the evaluation of cobertura we generated xml reports. One of these XML reports is shown in Figure 11.

```
- <packages>
  - <package complexity="2.0" branch-rate="1.0" line-rate="1.0" name="">
    - <classes>
      - <class complexity="2.0" branch-rate="0.5" line-rate="1.0" name="test" filename="test.java">
          <methods> </methods>
        - <lines>
            <line branch="false" hits="1" number="3"/>
          - <line branch="true" hits="1" number="4" condition-coverage="50% (1/2)">
            - <conditions>
                <condition number="0" coverage="50%" type="jump"/>
              </conditions>
            </line>
            <line branch="false" hits="1" number="6"/>
            <line branch="false" hits="1" number="12"/>
          </lines>
        </class>
      </classes>
    </package>
  </packages>
```

Figure 11: Cobertura XML Report

We then generated a C# program that manipulated the xml files and generated the coverage files in the required format. Since Cobertura only captures statement and branch coverage, we examined

[33] http://www.glenmccl.com/instr/instr.htm

Code-Cover.

You can run Code-cover in a number of ways. Two ways are via Eclipse or via the command prompt. CodeCcover uses four arrays to capture the different capture data i.e. Statement, Branch, Term Coverage and Loop coverage.[34] To ensure that Code-Cover captured Statement, Branch and MC/DC coverage, we manually assessed the code inside Eclipsed, by using JUnit tests on our sample programs. We deliberately generated tests not to gain complete coverage to ensure that this was detected by Code-Cover. We then incrementally added additional tests until we gained full coverage. Once we assured ourselves that the three coverage criterions were captured we ran Code-Cover via the command line.

To run Code-Cover via the command line we used the following command:

```
codecover instrument    --root-directory maths_module
                        --destination maths_module/instrumentedSrc
                        --container maths_module/test-session-container.xml
                        --language java
                        --charset UTF-8
```

The math_module is the source code directory of the source code file(s) to instrument i.e. the root directory. The destination is the directory to store the instrumented source code files in this case in a sub directory called instrumentedSrc under the root directory i.e. maths module. The container stores the test coverage results in xml files. This is not used in the framework. The language set is Java and charset is UTF-8.

The instrumented Java code files are then compiled. Running the compiled instrumented Java code files, generate a coverage log file (*.clf). The coverage log files report on the coverage achieved by the test(s) running during a single execution of the compiled program. Table 10 show the description of the coverage log types in the coverage log files. Since only one test is executed at any one time, a unique coverage log file was generated for each test execution. Therefore, 1000 separate tests would create 1000 unique coverage log files.

The overall C# Coverage component extracts out the coverage data from all of the Code-Cover log files and generates the statement, branch and MC/DC coverage files in terms of zeros and ones files i.e. Statement_Coverage.txt, Branch_Coverage.txt and MCDC_Coverage.txt. These coverage files contain the complete coverage for all the tests executed on the program under test.

[34] Term Coverage used Ludwig term coverage that is similar to MC/DC. However, it also measures part coverage, and for each individual term value set the term coverage. For short cut semantics (for example && and || operators) the Ludewig term coverage subsumes MC/DC.

Coverage Log file output	Description
S1 1 S2 10 S3 1	S indicate statement, the number indicate the statement line. The '1' after the whitespace indicate how many times that line was executed. For line S1 and S3 only once, for line S2 it been executed 10 times
B2 1 B4 1	B indicates statement coverage.
C1-1010100000 1 C8-11 8	C indicates conditional coverage (Term Coverage). C1 shows that condition one contains 10 conditions that needed to be exercised to ensure MC/DC coverage of that condition. One indicates that sub condition has been exercised while zero indicates that subcondition has not been exercised.
L2-2 1 L3-1 1	L indicates Loop coverage

Table 10: Coverage Log File Description

The C# Coverage component parses through the Code-Cover coverage log files and populate three internal arrays that store the three coverage types data. The arrays are initialised with zeros. When the Code-Cover coverage log files are parsed, the coverage arrays are updated to reflect coverage obtained from that code-cover coverage log file. On completion of parsing all the Code-Cover coverage log files for the PUT, the C# Coverage component writes the coverage gained for the three coverage types into coverage files i.e. Statement_Coverage.txt, Branch_Coverage.txt and MCDC_Coverage.txt.

In an attempt to gain dataflow coverage we considered generating our own dataflow coverage tool by using immediate language tools like SOOT (http://www.sable.mcgill.ca/soot/), InjectJ (http://insectj.sourceforge.net/). However, time limitations prevented such development. Since only Code-Cover provided us with MC/DC coverage, the log files could be easily manipulated and the tool was stable, supported and provided consistent results under our sample test programs we decided to use Code-Cover to capture the coverage data for our experiments.

3.1.3 Test Set Generation

JUnit provides a testing framework[35] for generating and running automated test cases for testing Java source code. JUnit tests through the public interfaces of the software under test i.e. through the return values of the software units. For example a Java method may return the sum for the total number of gallons in a garden pond[36]. The return value from this method can be compared against the expected value via use of the assert statements defined by the assert class inside JUnit. The expected value is generated from some form of oracle.

The Assert class is the largest class of the JUnit components[37] and consists of a number of Assert types. These can be seen as verification or checkpoints in the test cases. A typical AssertEquals statement is shown below[38]

AssertEquals (expected value, actual value, variance of value)

If the actual value was not inside the expected value variance, JUnit reports a failure. JUnit 3.x versions distinguish between JUnit 'failures' i.e. a violation of the JUnit assertions, and JUnit 'errors' i.e. an unexpected Java exception. Later versions of JUnit i.e. Version 4 and above do not. The JUnit version 3.8 is used in the framework.

One of the reasons for using JUnit 3.8 is that it provides a simple Test-Suite class. The Test-Suite class is the collection or individual test that makes up that test set. It is also used to manage and organise the tests or tests suites to be run. Tests or additional Test Suites can be added to the Test-Suite. This class therefore enables the user to dynamically bind a specific test at run time. This therefore provides the capability of selecting tests randomly at runtime.

The JUnit Results class generated the summary of the results of running one or more tests. Test failures are reported by '.F' followed by a description of the failure. Errors are reported by '.E' followed by the error description. A test that passed is reported by a single dot i.e. '.'. The time taken for each test is also reported. At the end of the test suite a summary of the test results are generated indicating the number of tests, failures and errors. In the framework case a JUnit test summary is

[35] We use testing framework to refer to the JUnit Testing framework, and only framework to refer to the overall framework, to distinguish between the two frameworks.

[36] The gallons in a garden pond can be determined by Length * Width * depth * 6.23. (UK gallons, multiply 7.48 for US gallons).

[37] JUnit include 5 basic classes: Assert, TestCase, TestSuite, Test and TestResults. [Beck 04] provide an overview of all these components.

[38] [Beck 04] provide a full listing of the JUnit Assert statements that can used in JUnit.

generated for every test since only one test is included in every JUnit test suite. The test_cases.txt data file holds the test set for the framework and not the JUnit Test-Suite class component.

3.1.4 JUnit and the Framework

The JUnit tests were primarily[39] developed in a simple text editor ConText[40] that provided basic text editing capability for a number of different programming languages. JUnit[41] was installed and an appropriate class path added to the Windows OS. Each JUnit test set developed would include the JUnit Framework Package, a Java Main Program method along with, Test-Suite and Test Case components. This therefore enabled the test set be compiled as a standalone Java Program via Javac[42]. The tool Javac reads class and interface definitions, written in the Java programming language, and compiles them into bytecode class files.

The JUnit component originally embodied oracles in each test case. This enabled parameterisation of the test cases and test cases to accept a wide range of test inputs randomly generated from the C# Wrapper. This was how the miscellaneous programs were tested. However, we only need to detect differences between the mutated source code and original source code. To achieve this, JUnit has been used in two ways: firstly as a data collector, in collecting expected results from the source code under test, with test data still generated from the C# Wrapper component; and secondly to actually test the program under test via the use of JUnit Assert statements. An internal flag inside the JUnit determine what mode the JUnit test is in. This was how all the other programs were tested apart from the miscellaneous programs.

Figure 12 illustrates how JUnit is used inside the framework, components that are shaded in light grey are JUnit components. The JUnit component received a String argument (String Arg()) generated by the C# Wrapper, that contains the randomised test data and the JUnit test case to run. This String Argument is converted into the appropriate data types in the main program of the JUnit component. Once converted the data type is used to populate an array[43], which in turn is used to define static variables (global variables) inside the JUnit component. The Test Mode flag value - False (No) or True (Yes) – determines whether the Expected Results are also stored in a static global variable.

[39] Some initial JUnit development was undertaken in Eclipse during the Vector Statistical Package test case generation. This was mainly undertaken when investigating the use of coverage tools.

[40] http://www.contexteditor.org/

[41] http://www.JUnit.org/

[42] Javac is the Sun command line Java compiler - http://java.sun.com/javase/downloads/index.jsp

[43] May only populate static variables, rather then an array, depending on the program under test.

The Main Program method also contains a timer that times out, 'kills' the Java thread after the specified length of time and records the failure[44]. This is to detect livelock/deadlock associated with mutant operators. The Main Program in the JUnit component calls the Test Suite class inside the JUnit framework and adds the selected test case to the test suite. This is achieved by the Test_Sel parameter that is passed to the Test_Suite method. The Test Suite is only ever associated with one test case and therefore one test case result. However, there may exist many test cases as shown by the multiplicity '1..*' joining Test-Suite to the Test-Case component as illustrated Figure 12.

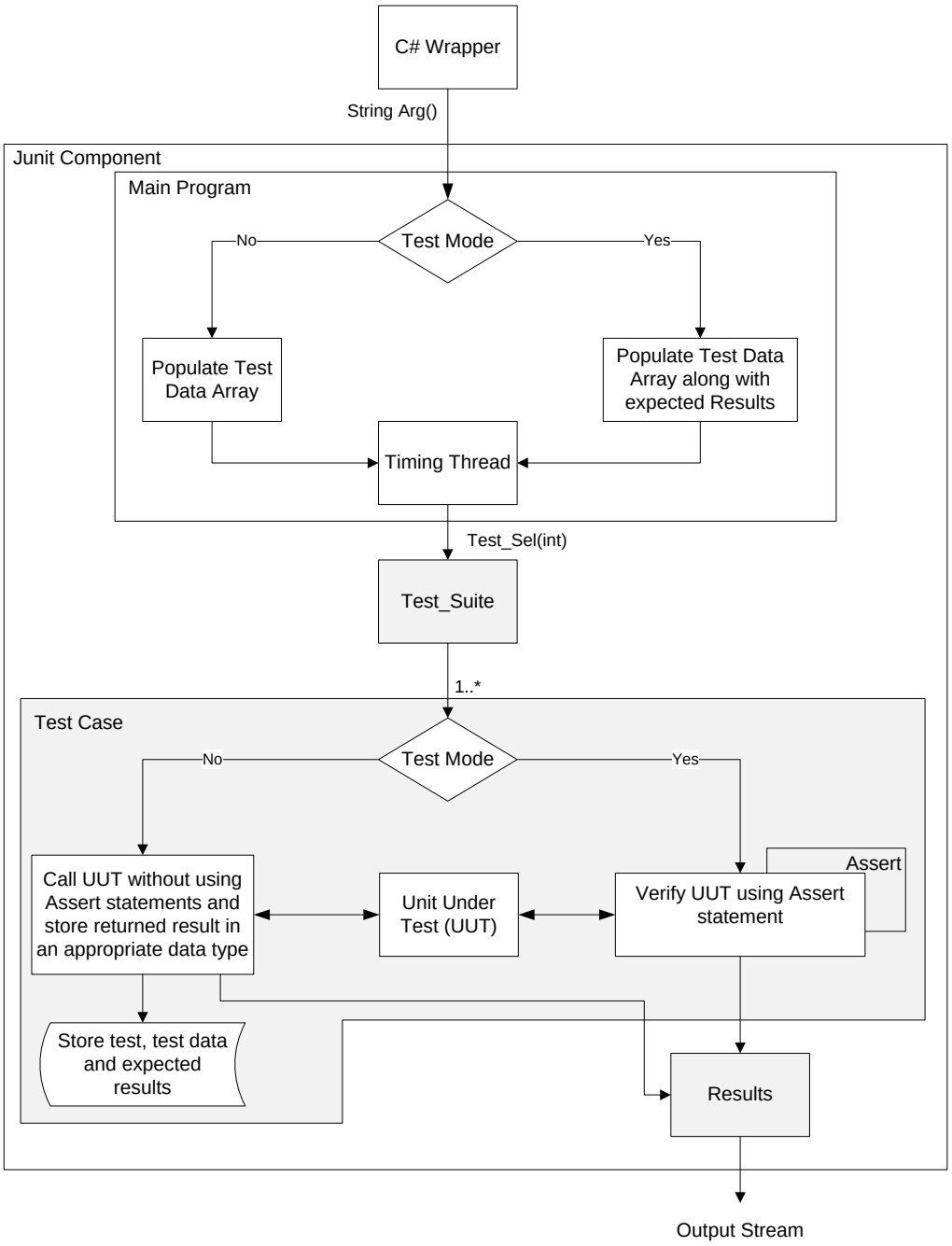

Figure 12: Test Case Structure

[44] JUnit 4 and above contain a timer for testing timing requirements. However, since JUnit 3.8 does not, therefore a timer has been developed.

The behaviour of the test case is dependent on the status of the Test Mode. If the test mode is set to false, the test case calls the PUT and stores the return value in an appropriate data type. The test data, along with the test case number and the expected result data is stored in a Test_Input data file. If the Test Mode is set to true, the test case uses the stored expected result value in the JUnit Assert Statement. It should be noted that the C# Wrapper is not used when the Test Mode is set to true, but the C# Test Input Reader component is used to read in the Test_Input data file. Both the C# Wrapper and C# Test Input Reader is described in 2.5.

This section has explained how we plan to generate automated random test cases. The next section explains how we plan to generate optimum test sets to meet the different coverage objectives via the use of GAs.

3.1.5 Test Subset Extraction

To extract unique test sub-sets (no duplicated test cases) we developed a number of heuristic algorithms i.e. Simple Hill Climbing, Random-Hill Climbing, Greedy algorithm and Simulated Annealing all developed in C#. We also developed our own bit based Genetic Algorithm developed in C#. The basic design based upon [Coley 99] and [Goldberg 89]. We then compared our results with the results gained from using the ECJ GA framework based in Java. Since ECJ supported integer based GA and the parameter file inside ECJ enabled increased flexibility so that the GA could be applied to a range of programs under test rather than the developed bespoke algorithms developed. For these reasons it was decided to use ECJ to develop our GA to generate unique test subsets.

3.1.5.1 Genetic Algorithms

GA use optimisation strategies based upon natural selection or '*Darwinian Evolution'.* Figure 13 below shows the steps involved in GA. The GA holds the initial population of individuals. Each individual (chromosome) represents a potential solution. Representation of individuals can be one of the hardest activities in developing GA's. Often bits are used; however, other representation can use integers and doubles. The selection process select individuals based upon their fitness i.e. by selecting fit individuals. Therefore, a new population is formed. However, none of these individuals are the ideal solution. Therefore, we alter this new population by mating – pairing individual chromosomes and using crossover and mutation to alter them. Crossover can be performed using a number of different strategies. In one point fixed crossover a certain point is randomly chosen and all the bits after that point are swapped as illustrated in Figure 13.

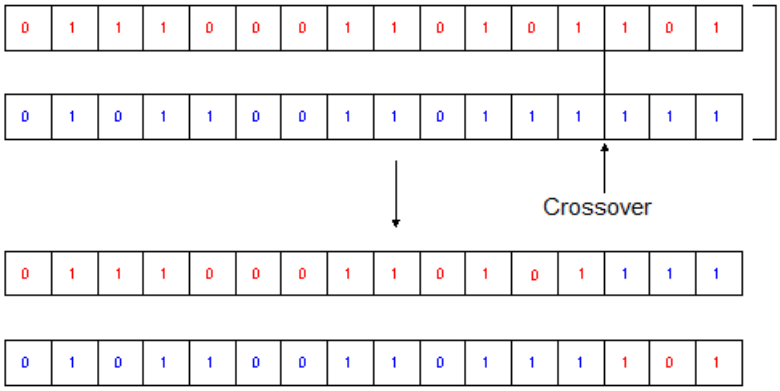

Figure 13: Bit Representation of a Chromosome

A variation on that is the two-point crossover, where two points are selected and everything between the two points are swapped. The one- and two-point crossovers work on specific segments of the chromosome. Therefore, new chromosomes will inherit segments from the parent chromosome. An alternative to using fixed crossover points is uniform crossover, which selects randomly the bits in the chromosome to swap. Each bit that makes up the chromosome has a probability (mixing ratio) of being swapped.

Using the uniform crossover allows the parent chromosome to be mixed across the whole gene level; with one or two point crossover mixing is done at segment level. However, this has the disadvantage of destroying the inheritance of the segment blocks. With mutation each bit that makes up the chromosome has a small probability of being flipped or inverted. Figure 14 shows that there is an evaluation function or '*fitness function*' that assessed the fitness of the solution. The GA is halted when a solution is found or when some number of evaluations has been reached.

Test subset extraction was achieved by developing GAs in ECJ[45]. ECJ is a Java Eclipse plug-in. ECJ contains a parameter file that defines the properties of the Genetic Algorithm. GAs generates solutions from the solution space i.e. the population. Solutions are represented by chromosomes. A chromosome is made up of sub-chromosomes. We use integers to represent the sub-chromosomes. Therefore, a chromosome is made up of integer values. We define the minimum and maximum sub-chromosome value range inside the ECJ parameter file. Chromosome is assessed by a fitness function. If the chromosome does not meet the fitness function, a new chromosome is generated by using cross over and mutation.

[45] http://cs.gmu.edu/~eclab/projects/ecj/

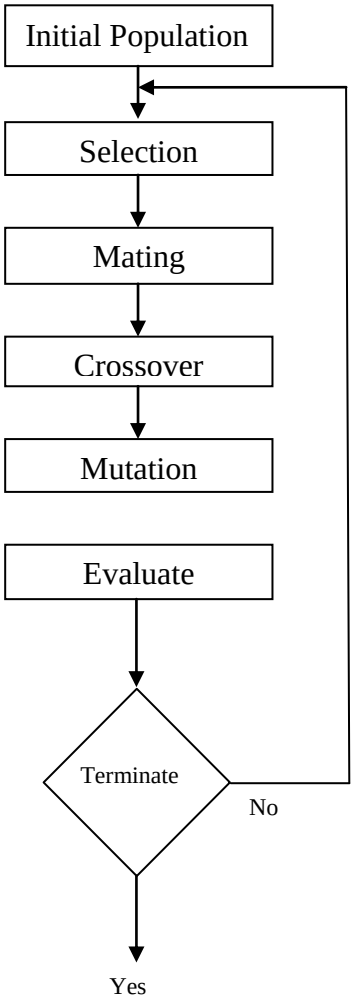

Figure 14: GA Structure

The GA uses the individual coverage tables generated for each coverage objective i.e. the statement, branch and MC/DC coverage tables to represent the set of solutions that the GA will select and gain the optimum number of tests to achieve that coverage objective. Each coverage file was turned into a Java bit array. Each test case represents a sub chromosome. Figure 15 illustrates this below:

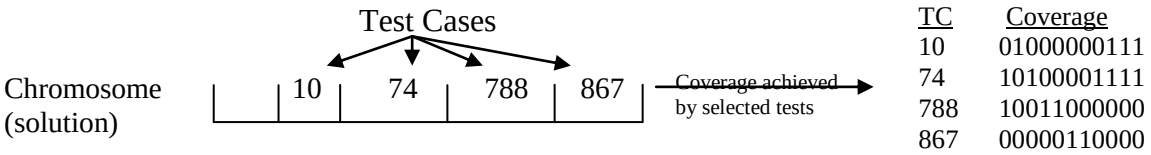

Figure 15: Chromosome Makeup

Table 11 describe the components that make up the GA for sub-set extraction and Figure 16 show the interaction between the components.

Components/Elements	Description
Program Parameters	Defines and initialises a number of variables that is used in the GA. These parameters include the size of the bit array that holds the coverage table, number of test cases, the number of lines, branches of MC/DC points in the PUT.
Evaluation Function	This function populates the bit array with the coverage data and evaluates the fitness of the selected solution. This use the coverage gained function to calculate the coverage gained from the selected solution.
Coverage Gained	Calculates the coverage gained from the selected solution.
Reset	This function deletes the selected test coverage from the coverage table. It achieves this by placing zeros in the coverage gained by that test. This therefore, enables unique optimum test sets to be created.
ECJ Parameter File	Define the parameters for the GA i.e. maximum number of solutions to generate, the minimum and maximum gene, genome size, crossover and mutation rate, don't accept duplicates etc.
Store Optimum Test Set	This saves the solution i.e. the test case numbers that achieve the solution.

Table 11: GA Components Description

The GA included a parameter file[46] that defined the GA properties. For the subset extraction, the parameter file defined the number of test cases generated i.e. the solution space for the GA to select test cases from. The parameter file also defines the number of test cases to select i.e. the ECJ genome size- this was heuristically selected. This was achieved by randomly selecting a small number, i.e. 5 tests. If the GA could not find a solution the number of test cases to select was increased by one, until the optimum solution was found. If the solution was found by the initial number of test cases to select, this number was reduced until a solution could not be found.

[46] A full definition of the parameter can be found at http://www.parabon.com/dev-center/origin/ecj/tutorials/tutorial1/

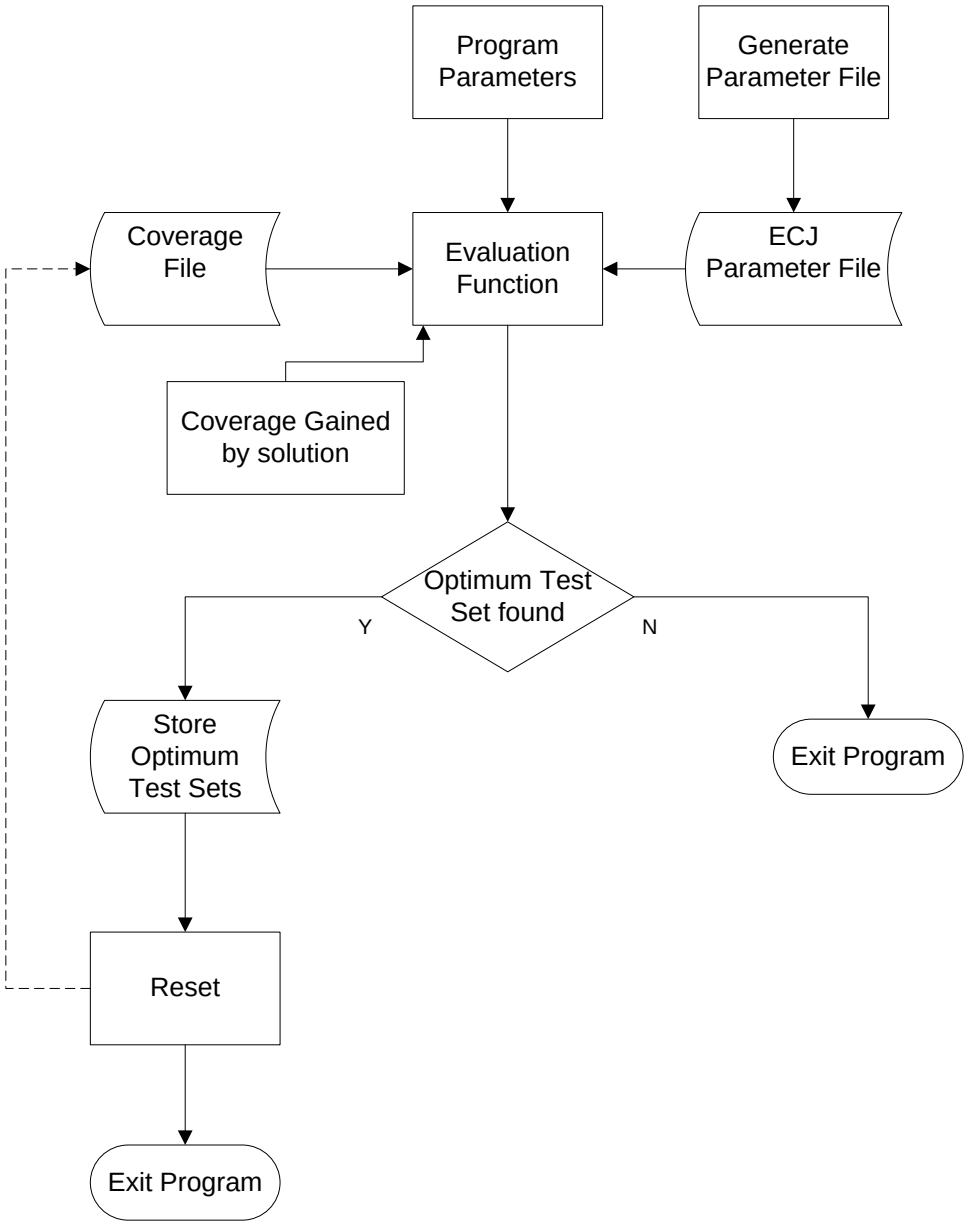

Figure 16 GA Subset Extraction

ECJ provides an evaluation method that evaluates the solution and compares it to the fitness function. The fitness function is based upon the required coverage i.e. every line of the PUT for statement coverage. The GA included a gain_coverage function that determined the coverage gained from the selected tests, if the solution could not be found, a new solution was generated by using GA crossover and mutation. If no solution could be found from the total number of attempts i.e. ECJ generations attribute in the ECJ parameter file, the bit array would not be up-dated. To ensure that unique test subsets were generated, the GA included a function to zeroise from the bit array the coverage achieved by the selected test cases when a solution has been found. This is shown by the dashed line in Figure 16. This was to ensure that the next test subset generated was unique i.e. no duplicated test cases.

The GA included a parameter file[47] (part of the ECJ Parameter File is defined in Appendix A) that defined the GA properties. For the subset extraction, the parameter file defined the number of test cases generated i.e. the solution space for the GA to select test cases from. The parameter file also defines the number of test cases to select i.e. the ECJ genome size.

3.1.6 Mutation

This section examines mutation. It describe the different tools evaluated, the different mutation operators applied and how the mutates were executed.

Three different mutation tools were evaluated for mutating the source code under test. These were:

- Jumble[48] - byte level mutation tool.
- PIT[49] - byte level mutation tool.
- MuClipse[50] - source code level mutation tool.

While the Jumble and PIT tools were easily used via the Windows command line, they were both coarse-grained compared to the MuClipse tool. The MuClipse tool is an Eclipse IDE plug-in for generating mutants, executing JUnit test set over the generated mutants and generating a MS. MuClipse enables the user to generate both Class and Traditional Mutants, but we only used traditional mutants in our experiments.

MuClipse contained 15 different mutants types as shown in Table 12 based upon the following mutant operators:

- Arithmetic operators
- Relational Operators
- Conditional Operators
- Shift Operators
- Logical Operators
- Assignment Operators

[47] A full definition of the parameter can be found at http://www.parabon.com/dev-center/origin/ecj/tutorials/tutorial1/
[48] http://jumble.sourceforge.net/
[49] https://pitest.org/
[50] http://muclipse.sourceforge.net/

Type	Sub-Type	Description	Examples
Arithmetic Mutant Operators	AORB	Replace basic binary arithmetic operators with other binary arithmetic operators.	vals.length / m => vals.length * m b / a => b * a 1 + r * r => 1 * (r * r) r * r => r – r
	AORS	Replace short-cut arithmetic operators with other unary arithmetic operators.	i++ => +i
	AOIS	Insert short-cut arithmetic operators.	i => ++i
	AOIU	Insert basic unary arithmetic operators.	m => -m
	AODS	Delete short-cut arithmetic operators.	A++ => A --A => A
	AODU	Delete basic unary arithmetic operators.	+A => A -A => A
Relational Mutant Operators	ROR	Replace relational operators with other relational operators.	vals[i].length != n => vals[i].length > n b != 0 => b <= 0 a <= 0.0D => a != 0.0D
Conditional Mutant Operators	COR	Replace binary conditional operators with other binary conditional operators.	B.m != m || B.n != n => B.m != m && B.n != n
	COI	Insert unary conditional operator	i < m => !(i < m)
	COD	Delete unary conditional operators	a++ => a
Shift Mutant Operators	SOR	Shift Operator Replacement Replace shift operators with other shift operators.	valByte >> 7 => valByte << 7 valByte >> 7 => valByte >>> 7
Logical Mutant Operators	LOR	Replace binary logical operators with other binary logical operators.	A & B => A | B A<<2 => A<<<3 A^B => A|B
	LOI	Insert unary logical operator	A.length => -A.length
	LOD	Delete unary logical operator.	-A.length => A.length
Assignment Operators	ASRS	Replace short-cut assignment operators with other short-cut operators of the same kind.	s += Math.abs(A[i][j]) => s *= Math.abs(A[i][j])

Table 12: Method Mutant Operator Types

MuClipse generates a separate folder for each mutant generated. The folders names are incremented sequentially for each mutant operator. For example, if the AORB mutant operator was applied, the mutated source code subfolders would be named AORB_1, AORB_2, AORB_3 etc. If the mutant operator did not generate valid syntax code, no folder nor source code would be generated, however, the folder counter would be incremented. The mutant operator applied can generate livelock code e.g.

infinite loops. The only observed livelock cases occur where the AOIS mutant operator has been applied.

MuClipse generates a mutate log file for all the mutates generated. Each line represents a mutate. The mutate log file indicates the name of mutate applied, the line number of the code change and the transformation from the original to the mutated source code. This is illustrated below:

AORB_1: 29: void_Shell_Sort(int,int): k - 1 => k * 1[51]

AORB_1: - Indicates the mutant type and number.
29: - The line of number in the source code were the mutant have been injected.
void_Shell_Sort(int,int): - The function that the mutant has been injected into.
k - 1 => k * 1 – The source code transformation from the original to the mutated. The original source code on the left the mutated source code on the right.

When trying to run the JUnit tests over the mutated source code a ClassNotFound Exception was generated.[52] To overcome this issue and to generate the data in the required format a bespoke Mutation Score Generator (MSG) was developed as shown in Figure 17. The mutated source code files and Java Classes were exported from Eclipse into a single flatbed folder. The MSG has the following core functionality:

- Copies the JUnit test file and any supporting Java files that may be required to support that test to the required mutant directories. The File Directory field shown in Figure 17 points to the directory that includes all the sub folders that contain the mutated source files.

- Copies the C# Test Input Reader program test driver programs for executing the JUnit Tests.

- Generates a 'Bat' file for running each mutant type i.e. Arithmetic, logical etc. The 'Bat' file changes the directory to the appropriate mutant location, runs the selected C# test driver program and re-direct the JUnit results to appropriate text files. For example a bat file for the mutant operator AOIS may state the following:

[51] This example uses tabs to break-up the actual text logged in the mutation log file to aid readability.
[52] After numerous email correspondences with one of the authors of the MuClipsed, it was not possible to resolve the issue.

```
cd C:\...\AOIS_1
Read_data_full_all_tests.exe > C:\...\Results_AOIS1.txt
cd C:\...\AOIS_2
Read_data_full_all_tests.exe > C:\...\Results_AOIS2.txt
cd C:\...\AOIS_3
Read_data_full_all_tests.exe > C:\...\Results_AOIS3.txt
cd C:\...\AOIS_5
Read_data_full_all_tests.exe > C:\...\Results_AOIS5.txt
```

The above example indicates that no AOIS_4 folder existed, due to invalid syntax code, therefore no folder was generated by MuClipse. However, the folder names are incremented. Therefore, the MSG scans for the existence of each mutant folder; if no folder exists the next folder is searched and so on. If no folder exists after 20 attempts, the next mutant sub operator is selected.

- Analyses the result files and generates the MS for each mutant operator. The MSG walks through each of the mutate operator result files and generates an overall summary in terms test set passes and failures for each mutant operator. This file name uses the mutant operator subtypes followed by '_Results' e.g. for ASRS mutation operator, the file be called 'ASRS_Results'. An example taken from the ASRS_Results file is shown below:

```
Passes = 0
Failures/Errors = 2
Test Set 1 = F
Test Set 2 = F
Mutant Score = 1
```

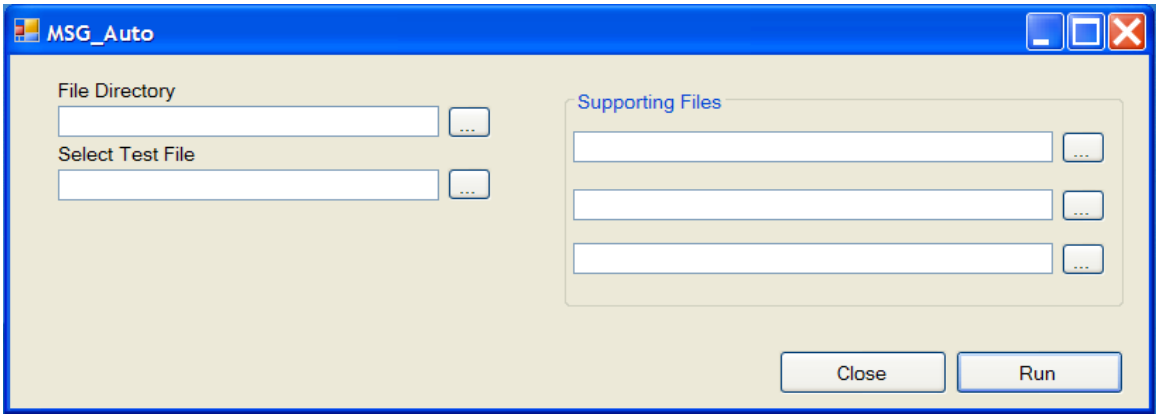

Figure 17: Mutation Score Generator User Screen

3.1.7 Mutation Score Generation

The Bit array component generates the mutation score by generating a two-dimensional bit array to store the results as shown in Figure 9 for the mutation table.

The Bit array program populates the bit array by reading the results files generated from running MSG. The bit array is initialised to zeros. It reads though all the results files and populates the bit array in terms of 1's to indicate a failure i.e. mutant detected or zero if the test passed i.e. mutant not detected. The bit array component then reads each row of tests listed in the test_sub_text.txt file that represent a test subset for that coverage type in turn and generates a mutation score for that test subset and the number of each mutation sub-type killed.

The bit array component also allows the Mutation Table to group mutants by mutation type as indicated in Figure 18. From this we can determine the mutation subtype killing ability of each test subset. The results are stored in comma separated variable (CSV) data files. The 'mutation_table.txt' stores the results for all the test subsets. The testX.txt (X being the unique test subset) stores the results just for that test subset. The Bit array component contains a check-sum that counts the number of mutants killed and the total number of mutants indexed by sub-type. Both these results are exported in CSV files. If the results to not match this indicates an anomaly.

		COR				COI			LOL	
		0..4				5..7			8..10	
	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}
T_1	0	1	1	1	1	1	0	1	1	1
T_2	1	0	0	0	1	1	0	0	0	1
...	1	1	0	0	0	0	0	0	0	0
T_n	1	1	0	1	0	0	0	0	1	0

Figure 18: Conceptual View of the Mutation Table

The data files are then exported into Microsoft Excel for data manipulation.

3.2 Testing Framework Refinement

3.2.1 Overview of the Framework Refinement

To improve efficiency (time to execute tests) and make the framework language independent the testing framework were modified. The modifications removed the need for the use of Junit and increase efficiency by comparing directly the results gained from the non-mutated and mutated programs.

The refined mutation framework is similar to the original mutation framework and uses some of the original mutation framework components. The components that make-up the refined mutation framework are detailed in Table 13. Some of the major differences between the two frameworks are as follows:

- Tests are automatically generated and executed via a test batch file. Each row (line) in the batch file represents a test. The test batch file is the test set to be executed on the PUT.

- Replaces the need for local verification i.e. JUnit. A comparison component compares the outputs of the non-mutated program with those of the mutated program. A result file is generated from this comparison for each MO sub-type applied.

- Replaces the MSG with a new MSG that analyse the results files to calculate the MS for each MO sub-type.

Component	Description of Component	Files Generated	Program Language
C# Test Data Generator	Automatically generates and executes a test batch file. The batch file re-directs the test results to unique test result files.	Batch file Test results Coverage data files	C#
C# Test Runner	Runs the test batch file over the mutated programs	Test Results (from mutated programs)	C#
C# Comparator	Generates a pass/failure log for each mutated program.	Results.txt	C#
C# MSG	Generates the MS for the PUT by analysing all the result files generated by the C# Comparator component.	Mutant Table	C#

Table 13: Mutation Framework Components

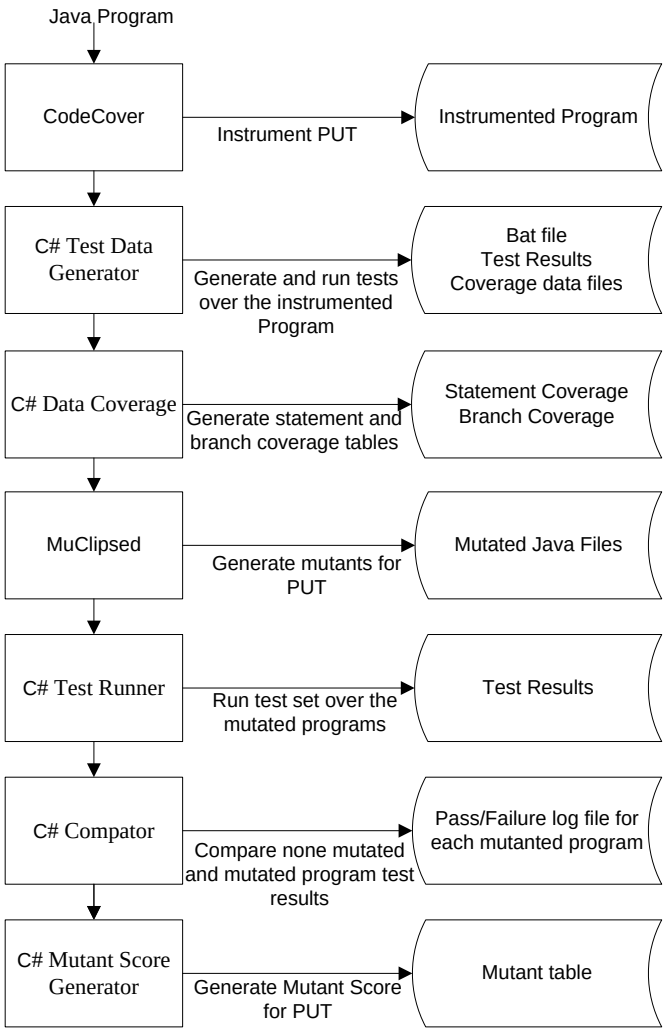

Figure 19: Components that Make up the Framework and their Outputs

3.2.2 Test Case Generation and Execution

Test generation and execution of the non-mutated programs was carried out by the C# Test Data Generator component. This component has three functions as shown in Figure 21 generates test data; test batch file and executes that test batch file, once the required number of tests has been generated. The test batch file includes the PUT execution command along with the associated test data. Later versions of test batch file directed tests to specific processor cores to examine test efficiency.

Test cases were generated by randomly selecting the appropriate type input data i.e. random generated integer, real number. Randomisation was achieved by using the random functions inside the C# programming language. Where specific string format test data was required this was either achieved by converting combinations of integer or real based data together to achieve the required string format. When this was not possible an enumeration array(s) was used that contained the foundation

elements of the required format. These elements were then randomly selected and combined to generate the required test data in the required format.

The test batch file is automatically generated by the C# Test Data generator component. The test batch file contains the PUT execution command, associated test data and redirects the program outputs to unique output files for post analysis. Figure 20 shows the elements of a test batch file.

```
Java FacadeDemo 159032 3447 155289 128058 -24613 46569 232 >output1.txt
Java FacadeDemo 141974 54752 86743 194835 57817 97295 326 >output2.txt
Java FacadeDemo 166127 116155 124210 104544 -3249 133095 353 >output3.txt
```

PUT Name | Test Data | Re-direct test output to unique files

Figure 20: Test Bat File Elements

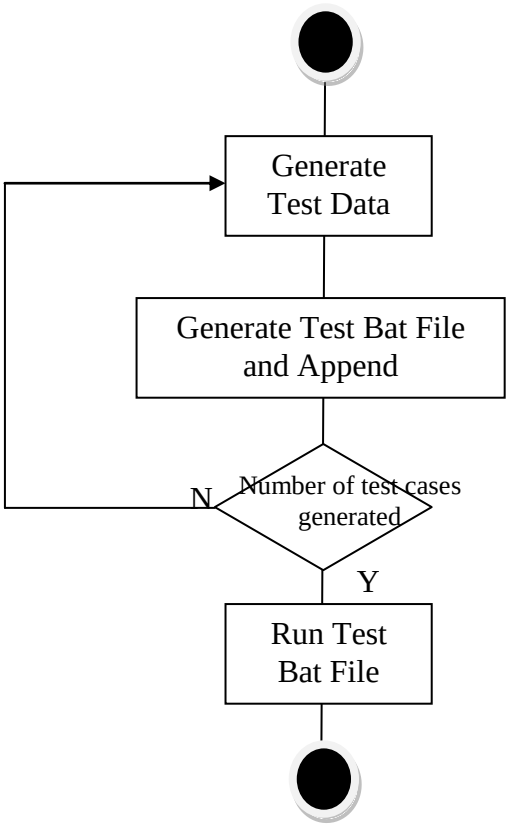

Figure 21: Test Generation and Execution for the Refined Mutation Framework

The main function for each of the PUT was amended to accept arguments on program execution. Once the required number of test cases has been generated and stored in the test bat file, the bat file was executed by the C# Test Data Generator component. The test bat file in the first instance was

executed over the non-mutated program that has been instrumented via code-cover to collect statement and branch coverage. Each test generated a data coverage file.

Later we refined the test bat file by using the Windows 7 'start' command to call the PUT. This enabled us to direct which processor core the test would be executed on via the start AFFINITY option command. For example:

```
start /AFFINITY 1 /b /wait Java FacadeDemo 204547 39668 143556 6112 15055 235569 351 >output1.txt
```

Figure 22: Test Bat File Elements using start command

The start command starts a new program or command. The command line option AFFINITY direct which processor core(s) to use. Affinity uses a hexadecimal affinity mask. Therefore, AFFINITY Ax00 specifies that the program should run using processors 1, 3, 5 and 7, but not 0, 2, 4 or 6. In Figure 22 AFFINITY 1 relates to processor core 0, AFFINITY 2 would relate to processor core 1 and so on. Option '/b' start the program without creating a new window and option '/wait', wait for the program to terminate.

3.2.3 Mutant Testing and Mutant Score Generation

The C# Test Runner runs the test batch file over the mutant programs. The mutant programs are generated via MuClipse and exported to unique file directories for each mutant generated as described in section 3.1.6.

The C# Test Runner component detects the mutant directories for the PUT and copies and runs the test bat file over the mutants. The C# Test Runner program contained a single thread. This therefore led to a number of instances of the test bat file running simultaneously over a number of different mutated programs i.e. 2500 instances. This caused the Java (TM) platform SE binary to stop work. Therefore, we introduced a timing delay in the process thread before copying and running the test bat file to the next mutated program. This still enabled simultaneously running of the test batch file but prevented more than 500 instances running simultaneously.

The running of the test batch file generated a large number of separate test result files. These result files were merged into master result files. The test result files generated from running the test batch file over each mutated program were merged into one results file for each mutant program generated. A master result file was also generated for the non-mutated program. Figure 23 illustrates this.

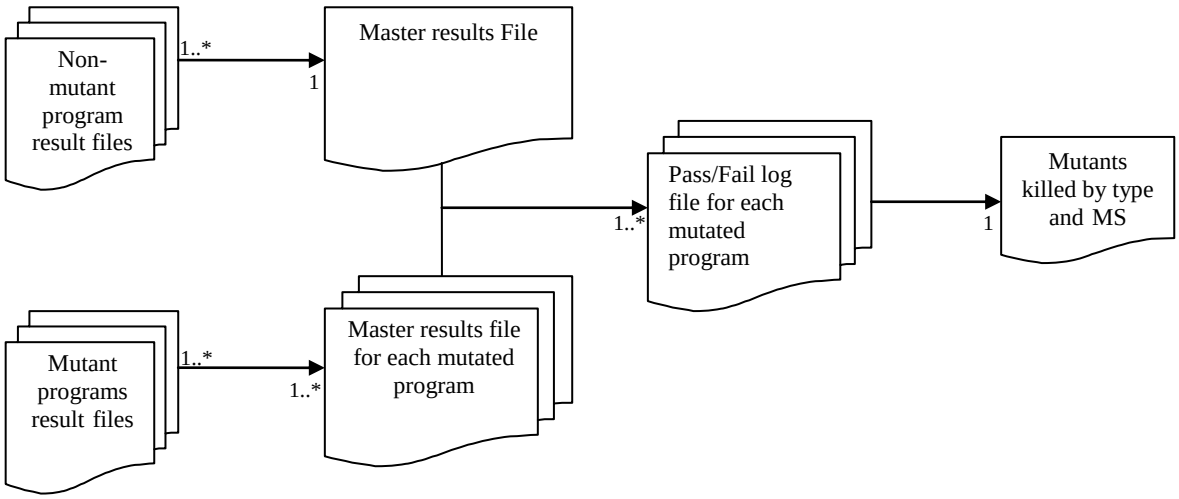

Figure 23: Mutation MS Generation

The master results file is then compared with the master mutant result files. There would be many master mutant results files. Since there would be a master mutant result file for each mutant program generated. A result log file would be generated for each master results file comparison with the master mutant results file. Each line in the two master files (master results file and the master mutant results file) was compared, any differences between the two would be logged has an '.F' indicating a difference or a '.' if no differences existed.

The C# Mutate Score Generator program generates the MS by analysing all of the log files. It generates a MS table that includes the number of mutants injected, killed, MS by mutant sub type and overall MS. The MS table for each of the PUT were exported into Microsoft Excel for data manipulation.

3.3 Framework Summary

This chapter has described a flexible testing framework exhibiting the following key functionality:

- The automated test data and test case generation.

- The automated source code coverage capture for statement, branch and MC/DC for each individual test. The individual coverage files were then automatically merged together to generate coverage tables for each coverage type – statement, branch and MC/DC.

- The automated generation of optimal test sets via a GA. This enables us to measure reliability of criteria.

- The automatic generation of mutants.
- The automatic generation of the MS to measure test effectiveness that in turn enables us to measure reliability of criteria.

We now proceed to apply the framework to software predominantly from two different domains.

4. Testing Code Coverage Criteria

Our first three research objectives, identified in section 1.2 and reproduced below, are concerned with the effectiveness and reliability of structural code coverage criteria and the effects of satisfying coverage criteria "optimally" (with least number of test cases) and also with some degree of redundancy. The first three research objectives are:

1. To measure the test effectiveness of the three coverage criteria (Statement, Branch and MC/DC) mandated by a widely used commercial airborne software standard for safety critical software D0-178B [178B] and its recent updated version D0-178C [178C].

2. To measure the reliability of those three coverage criteria by comparing the effectiveness of multiple minimal size tests sets meeting these criteria.

3. To measure the reliability of the three widely used coverage criteria used in the commercial airborne safety critical software e.g. [178B] and [178C] with test sets with a small degree of redundancy. To add redundancy we plan to combine the different optimum coverage test sets.

This chapter presents the research carried out in support of the above objectives. The primary software subject to testing is a library of fourteen numerical 'C' recipes (numerical algorithms) taken from [Press et al 92]. These programs are converted to Java and mutated. From our primary experiments we aim to determine the following:

- having achieved 100% statement (S) coverage with the optimum number of tests, what percentage of errors has been found;

- having achieved 100% branch (B) coverage, with the optimum number of tests, what percentage of errors has been found;

- having achieved 100% MC/DC (M), with the optimum number of tests, what percentage of errors has been found.

In addition to examining single coverage objectives, we generate test subsets to meet the following combined testing objectives:

- Statement and Branch coverage (SB)

- Statement and Branch coverage and MC/DC (SBM)

Since branch coverage subsumes statement coverage and likewise MC/DC subsumes branch coverage this adds redundancy to the test sets. We perform no pruning of these test sets; they are simply combined. We compare all the test subsets results against the MS obtained by the full test set; this therefore removes the issue of mutation equivalence.

In addition to conducting experiments on the numerical algorithms, we also carried out experiments on a number of sorting algorithms i.e. bubble, heap, shall, inserting, merge, quick and shell-sort and other miscellaneous algorithms. This was to provide a comparison between the results of numerical based algorithms and other computational algorithms.

The chapter covers the following:-

- Section 4.1 defines the numerical recipe programs under test (PUT) and their program properties.

- Section 4.2.1 shows the results from running all the tests over the mutated numerical recipe programs.

- Section 4.2.2 analyses the reason for the result gained for the numerical recipes.

- Section 4.2.3 shows the results for the optimum number of tests to achieve the required coverage objectives i.e. statement (S), branch (B) and MC/DC (M) for the numerical recipes. Section 4.2.3 also shows the results from combining these optimum test sets i.e. SB and SBM.

- Section 4.2.4 shows the results from the sorting and miscellaneous algorithms.

- Section 4.3 provides a summary of findings and conclusions.

4.1 Numerical Programs Under Test and Their Program Properties

Fourteen publicly available numerical 'C' recipes, taken from [Press et al 92] and presented in Table 14, were converted to Java and used as our primary programs for our experiments (i.e., the PUTs). The task to convert the original C programs were a relatively easy task due to the majority of constructs used in the original C programs was supported by Java. The only exception was the use of pointers. This was not an issue since use of variable referencing enabled the same implementation to occur. Since the algorithms were relatively small it was not time consuming to manually translate the code and to a degree by manually translating the code enabled a greater understanding of the code under test. To ensure correct translation of the programs a sample of the programs were compared against on-line algorithms using the same input. All the programs included sequential, selection and iteration constructs. Table 14 provides a short description of the numerical recipes applied has the experimental programs.

Recipe Name	Recipe Description
Bessj	Returns the Bessel function Jn(x) for any real x and n >=2.
Cosft2	Calculates the staggered cosine transform of a set of y [1..n] of real data points.
Dawson	Returns Dawson's integral $F(x) = \exp(-x^2) \int_0^x \exp(t^2)dt$ for any real x.
EI	Computes the exponential integral Ei(x) for x>0.
Expint	Evaluates the exponential integral En(x).
Gammp	Returns the incomplete gamma function P(a,x).
Gas	Returns a normal distribution (μ=0, σ=1) deviate, using random_1 as the source of uniform deviates.
Plgndr	Computes the associated Legendre polynomial $P_l^m(x)$.
Poisson	Returns as a floating point number an integer value that is a random value drawn from a Poisson distribution of mean xm, using Random_1 as the source of uniform random deviates.
Random_1	Returns a value uniformly at random between 0.0 and 1.0 using Park and Miller Bays-Durham shuffle.
Random_2	Returns a value uniformly at random between 0.0 and 1.0 using L'Ecuyer Bays-Durham shuffle.
Random_3	Returns a value uniformly at random between 0.0 and 1.0 using Knuth subtractive method.
RC	Computes Carlson degenerate elliptic integral, $R_c(x,y)$.
SVD	Given a matrix a[1..m][1..n], computes the singular value decomposition, $A = U.W.V^T$ The U replaced a on output. The diagonal matrix of singular values W is output as a vector w[1..n]. The matrix V (not the transpose V^T is output as v[1..n][1..n].

Table 14: PUT Numerical Recipes

To examine the impacts of program properties on testing effectiveness and reliability we use source-monitor to capture some simple metrics relating to the program under test. The definitions of these

metrics are defined in Appendix C. The PUTs have a wide range of program properties: the key metrics[53] are listed in Table 15.

File Name/Metric[54]	Lines	Statements	% Branch Statements	Method Call Statements	Methods per Class	Average Statements per Method	Maximum Complexity
Bessj.java	137	86	16.3	15	3	26.33	11
Cosft2.java	189	150	12.7	11	3	48.67	9
Dawson_fun.java	51	37	13.5	10	2	13.5	6
EI.java	63	38	26.3	4	1	32	11
Expint.java	82	54	27.8	7	1	48	20
Gammp.java	125	75	18.7	15	4	15.75	6
Gasdev_fun.java	89	58	15.5	6	2	20.5	9
Plgndr_fun.java	53	31	25.8	2	1	29	11
Poisson.java	122	89	18	15	3	24.33	11
Random_01.java	71	42	16.7	1	1	28	9
Random_02.java	84	54	16.7	1	1	34	10
Random_03.java	57	48	18.8	1	1	38	11
RC_fun.java	46	33	9.1	9	1	19	10
SVD_NR.java	296	245	21.6	21	4	60	55[55]

Table 15: Key Metrics for the Programs Under Test

The metrics defined above and listed in Table 15 tell us nothing about the use of complex and simple expressions in the decision structure of the source code i.e. IF statements[56]. We define a simple expression only containing one sub condition in the decision expression and a complex expression having more than one sub condition. For example a simple expression would be If(A>B) and a complex expression If(A>B || A>C). Table 16 indicates the type of expressions in the PUT. The number in brackets indicates the number of decisions in that complex expression. All the stated programs that had complex expressions also included simple expressions.

[53] The metrics were captures via using Source-Monitor V 2.6.3.104 - http://www.campwoodsw.com/sourcemonitor.html.
[54] The metrics are defined in Appendix B.
[55] The complexity is so high due to the high number of decision statements in the source code i.e. If statements.
[56] If a program only contained simple expressions and branch coverage was achieved, one could rightly argue that Multiply Conditional Coverage (MCC) that consumes MC/DC has been achieved.

Recipe Name	Simple or Complex Expressions used in the program under test
Bessj	Simple expressions
Cosft2	Include one complex expression (4)
Dawson	Simple expressions
EI	Simple expressions
Expint	Include one complex expression (8)
Gammp	Simple expressions
Gasdev	Include two complex expressions (4)
Plgndr	Include one complex expression (8)
Poisson	Simple expressions
Random_01	Include one complex expression (4)
Random_02	Simple expressions
Random_03	Include one complex expression (4)
RC	Include one complex expression (8)
SVD	Include one complex expression (4)

Table 16: Simple and Complex Expressions in the Numerical Programs Under Test

4.2 Mutations Injected and Results

4.2.1 Full Test Set Results

This section shows the results from running all the randomly generated tests. The results are presented in a number of tables. All the MS presented are raw MS and do not take into account mutation equivalence.

Table 17 shows the total number of tests generated for each numerical recipe program, the number of mutants injected, the number of mutants killed and the mutation score MS. We refer to the combined tests as the *full* test set. (We will subsequently extract subsets from it.) Programs Plgndr, Costf2 and Poisson achieved a MS of greater then 0.9, with Plgndr achieving the highest MS. The majority of PUT achieved a MS greater than 0.8. Three programs i.e. EI, Gas and Random_01 achieved MS lower then 0.8, with Random_01 achieving the lowest MS of 0.65502.

File Name	Number of Tests	Mutant Injected	Mutants Killed	Mutate Score
Bessj.java	5000	1020	848	0.83137
Cosft2.java	1000	1900	1720	0.90526
Dawson.java	1000	350	312	0.89142
EI.java	5000	237	189	0.79746
Expint.java	1500	460	388	0.84347
Gammp.java	5000	512	415	0.81054
Gasdev_fun.java[57]	500	424	318	0.75
Plgndr_fun.java	1150	289	269	0.93080
Possion.java	500	409	370	0.90441
Random_01.java	1000	229	150	0.65502
Random_02.java	1000	290	248	0.85517
Random_03.java	1000	268	236	0.88059
RC_fun.java	1000	347	298	0.85878
SVD_NR.java	1000	2445	1986	0.812269

Table 17: Full Test Set Mutant Injected, Mutants Found and MS

Table 18 shows the results gained by the full test sets by MO and MO subtype for each PUT. The square brackets after the recipe name indicate if that PUT contains simple or complex expressions denoted by S – Simplex Expressions and C – Complex Expressions. The number after this indicates the maximum complexity of that PUT. The number inside the parenthesis indicates the number of mutants killed, while the number outside the parentheses indicates the number of mutants still alive. By adding the two you gain the number of mutants injected. For example in Table 18 for the program Bessj, the mutant subtype COI, one mutant remained alive, 12 mutants were killed. Therefore, 13 mutants were injected. The dash '-' in Table 18 indicate that no mutants were injected of that mutant subtype.

[57] Could not obtain complete MC/DC, due to one condition.

Recipe Name	Conditional MO			Arithmetic MO						Logical MO			Relational MO	Shift MO	Assignment MO
MO Sub-Type	**COR**	**COI**	**COD**	**AODU**	**AODS**	**AOIU**	**AORB**	**AORS**	**AOIS**	**LOI**	**LOR**	**LOD**	**ROR**	**SOR**	**ASRS**
Bessj [S, 11]	0 (2)	1 (12)	-	1 (2)	-	14 (32)	18 (418)	0 (2)	113 (329)	2 (13)	0 (2)	-	14 (21)	-	9 (15)
Costf [C, 9]	0 (2)	0 (18)	-	0 (5)	-	7 (96)	50 (606)	0 (3)	110 (746)	1 (117)	-	-	5 (70)	5 (9)	2 (48)
Dawson [S, 6]	-	0 (5)	-	1 (0)	-	2 (18)	0 (128)	0 (2)	32 (116)	0 (10)	-	-	3 (17)	-	0 (16)
EI [S, 11]	1 (7)	-	-	1 (0)	-	0 (11)	3 (37)	0 (2)	31 (97)	2 (5)	-	-	10 (10)	-	0 (20)
Expint [C, 20]	2 (6)	0 (19)	-	1 (2)	-	1 (21)	6 (94)	0 (3)	45 (161)	3 (21)	-	-	14 (41)	-	0 (20)
Gammp [S, 6]	2 (0)	6 (9)	-	1 (4)	0 (1)	1 (28)	1 (131)	0 (7)	64 (188)	2 (8)	-	-	20 (15)	-	0 (24)
GAS [C, 9]	2 (2)	0 (13)	-	0 (1)	-	6 (18)	17 (87)	0 (1)	66 (132)	5 (31)	-	-	9 (26)	-	1 (7)
Plgndr [C, 11]	1 (3)	0 (10)	-	0 (1)	-	0 (15)	2 (74)	0 (2)	12 (106)	0 (20)	-	-	5 (30)	-	0 (8)
Poisson [S, 11]	0 (8)	-	-	0 (3)	0 (1)	2 (22)	1 (115)	1 (4)	32 (160)	0 (15)	-	-	3 (22)	-	0 (20)
Random_1 [C, 9]	0 (2)	1 (8)	-	0 (1)	-	10 (9)	-	0 (1)	53 (93)	7 (16)	-	-	4 (16)	-	4 (4)
Random_2 [S, 10]	-	0 (9)	-	-	-	5 (21)	-	0 (1)	33 (161)	2 (27)	-	-	2 (13)	-	0 (16)
Random_3 [C, 11]	0 (2)	0 (12)	-	-	0 (2)	3 (10)	0 (36)	0 (9)	20 (98)	2 (24)	-	-	7 (23)	-	0 (20)
RC [C, 10]	2 (10)	0 (15)	-	0 (2)	-	0 (15)	4 (104)	-	37 (133)	-	-	-	6 (19)	-	-
SVD [C, 55]	0 (2)	12 (44)	-	1 (5)	-	25 (155)	110 (318)	1 (35)	235 (1101)	22 (208)	-	-	31 (64)	-	22 (54)

Table 18: Number of Alive (Dead) Mutants by Mutation, Mutation Operator and Subtype

Mutation Operator Type	Number of Mutants Injected	Mutants Alive	Mutants Killed	M.S
Conditional	250	30	220	0.880
Arithmetic	7521	1179	6342	0.843
Logical	565	48	517	0.915
Relational	520	133	387	0.744
Shift	14	5	9	0.643
Assignment	310	38	272	0.877

Table 19: Number of Mutants Injected, Alive, Killed and Mutation Score by Mutation Operator

Table 19 shows the overall MS for the Mutation Operators (MOs) applied to *all* the numerical recipe programs. The Logical MOs had the highest overall MS of 0.915. The Arithmetic, Conditional and Assignment MOs all achieved an overall MS greater than 0.8. The relational MO was worst performing with a MS of 0.744. We ignore the Shift MO due to the very small sample size of mutants injected.

4.2.2 Rationale for Different MS in Numerical Recipes

To analyse some of the MS differences between the PUT we analyse the 3 random programs and specifically focused on the Random_01 program. We examine specifically the Relational and Arithmetic MOs.

The lowest achieved MS for the three Random based programs were achieved by Random_01 with a MS of 0.65502. To determine why this was the case we analysed three sub-mutation types AOIU, AOIS and LOI. The reason for this is that these three mutant sub types performed poorly compared with Random_02 and Random_03 for the same mutant sub types has shown in Table 20. Table 20 summarises the number of mutants live, killed and injected and MS for the three sub-mutants types to be analysed.

MO Sub-Type	Random_01	Random_02	Random_03	
AOIU	10	5	3	Live Mutants
	9	21	10	Killed Mutants
	19	26	13	Total number of mutants injected
	0.473684211	0.807692308	0.7692308	MS
LOI	7	2	2	Live Mutants
	16	27	24	Killed Mutants
	23	29	26	Total number of mutants injected
	0.695652174	0.931034483	0.9230769	MS
AOIS	53	33	20	Live Mutants
	93	161	98	Killed Mutants
	146	194	118	Total number of mutants injected
	0.636986301	0.829896907	0.8305085	MS

Table 20: AOIU, LOI, AOIS for Random_01, Random_02 and Raondom_03

Programs Random_01 and Random_02 differ in the method of generating the random number have explained in [Press et al 92]. Part of the code listing is shown in Table 21. The Random_01 program shuffles the array by selecting via the use of the iy variable of the random array elements stored in iv

in the array containing a random set of numbers. The Random_02 program uses two shuffles and combines the idum and idum2 variables to generate the output. This additional dependency on the outputs would appear to lead to increase observability of failures.

Random_01	Random_02

k=((long)idum)/(long)IQ;
idum = (long)IA*(idum-k*(long)IQ)-(long)IR*k;
if(idum<0)
{
idum+=IM;
}
j=(int)iy/(int)NDIV;
iy=iv[j];
iv[j]=(long)idum;
	k=(long)idum2/(long)IQ2;
	idum2=(long)IA2*((long)idum2-k*(long)IQ2) k*(long)IR2;
	if(idum2<0)
	{
	idum2 +=IM2;
	}
	j=(int)iy/(int)NDIV;
	iy=iv[j]-idum2;
	iv[j]=(long)idum;

Table 21: Random_01 and Random_02 Comparison

To analyse in detail why the Random_01 did not detect the mutants generated by the AOIU, LOI and AOIS mutants, we executed a single JUnit test using the same test data have applied originally before amending the test data to see if the mutant could be detected. The JUnit testing was performed in Eclipse. Table 22 lists all the mutants not killed for AOIU, LOI, AOIS and a rationale why this mutant was not detected.

Mutation Operator and Mutate	Reason
AOIU_9:54:double_ran1():idum => -idum	The idum variable is assigned to itself and post incremented inside the assignment had no side effect
AOIU_10:54:double_ran1():IQ => -IQ	The IQ variable is assigned to itself and post incremented inside the assignment had no side effect
AOIU_11:55:double_ran1():IA => -IA	The IA variable is assigned to itself and post incremented inside the assignment had no side effect
AOIU_12:55:double_ran1():idum => -idum	The idum variable is assigned to itself and post incremented inside the assignment had no side effect
AOIU_13:57:double_ran1():IM => -IM	The IM variable is assigned to itself and post incremented inside the assignment had no side effect
AOIU_14:59:double_ran1():iy => -iy	The iy variable is assigned to itself and post incremented inside the assignment had no side effect
AOIU_15:59:double_ran1():NDIV => -NDIV	The NDIV variable is assigned to itself and post incremented inside the assignment had no side effect
AOIU_16:60:double_ran1():j => -j	The j variable is assigned to itself and post incremented inside the assignment had no side effect
AOIU_17:61:double_ran1():idum => -idum	The idum variable is assigned to itself and post incremented inside the assignment had no side effect
AOIU_19:63:double_ran1():RNMX => -RNMX	The RNMX variable is assigned to itself and post incremented inside the assignment had no side effect
AOIS_25:36:double_ran1():iy => iy++	Post Incremental have no side effect on the IF expression
AOIS_26:36:double_ran1():iy => iy--	Post Incremental have no side effect on the IF expression
AOIS_49:44:double_ran1():idum => idum++	The idum variable is assigned to itself and post incremented inside the assignment had no side effect
AOIS_50:44:double_ran1():idum => idum--	The idum variable is assigned to itself and post incremented inside the assignment had no side effect
AOIS_65:44:double_ran1():k => k++	The k variable is post incremented inside the assignment and had no side effect
AOIS_66:44:double_ran1():k => k--	The k variable is post incremented inside the assignment and had no side effect
AOIS_87:54:double_ran1():idum => ++idum	The idum variable is post incremented inside the assignment and had no side effect
AOIS_88:54:double_ran1():idum => --idum	The idum variable is post incremented inside the assignment and had no side effect
AOIS_89:54:double_ran1():idum => idum++	The idum variable is post incremented inside the assignment and had no side effect
AOIS_90:54:double_ran1():idum => idum--	The idum variable is post incremented inside the assignment and had no side effect
AOIS_99:55:double_ran1():idum => ++idum	The idum variable is post incremented inside the assignment and had no side effect
AOIS_100:55:double_ran1():idum => --idum	The idum variable is post incremented inside the assignment and had no side effect
AOIS_101:55:double_ran1():idum => idum++	The idum variable is post incremented inside the assignment and had no side effect
AOIS_102:55:double_ran1():idum => idum--	The idum variable is post incremented inside the assignment and had no side effect
AOIS_103:55:double_ran1():k => ++k	The k variable is post incremented inside the assignment and had no side effect
AOIS_104:55:double_ran1():k => --k	The k variable is post incremented inside the assignment and had no side effect
AOIS_105:55:double_ran1():k => k++	The k variable is post incremented inside the assignment and had no side effect
AOIS_106:55:double_ran1():k => k--	The k variable is post incremented inside the assignment and had no side effect
AOIS_115:55:double_ran1():k => ++k	The k variable is post incremented inside the assignment and had no side effect
AOIS_116:55:double_ran1():k => --k	The k variable is post incremented inside the assignment and had no side effect
AOIS_117:55:double_ran1():k => k++	The k variable is post incremented inside the assignment and had no side effect
AOIS_118:55:double_ran1():k => k--	The k variable is post incremented inside the assignment and had no side effect
AOIS_119:56:double_ran1():idum => ++idum	The idum variable is post incremented inside the assignment and had no side effect
AOIS_120:56:double_ran1():idum => --idum	The idum variable is post incremented inside the assignment and had no side effect
AOIS_121:56:double_ran1():idum => idum++	The idum variable is post incremented inside the assignment and had no side effect
AOIS_122:56:double_ran1():idum => idum--	The idum variable is post incremented inside the assignment and had no side effect

AOIS_123:57:double_ran1():IM => ++IM	The IM variable is post incremented inside the assignment and had no side effect
AOIS_124:57:double_ran1():IM => --IM	The IM variable is post incremented inside the assignment and had no side effect
AOIS_125:57:double_ran1():IM => IM++	The IM variable is post incremented inside the assignment and had no side effect
AOIS_126:57:double_ran1():IM => IM--	The IM variable is post incremented inside the assignment and had no side effect
AOIS_127:59:double_ran1():iy => ++iy	The iy variable is post incremented inside the assignment and had no side effect
AOIS_128:59:double_ran1():iy => --iy	The iy variable is post incremented inside the assignment and had no side effect
AOIS_129:59:double_ran1():iy => iy++	The iy variable is post incremented inside the assignment and had no side effect
AOIS_130:59:double_ran1():iy => iy--	The iy variable is post incremented inside the assignment and had no side effect
AOIS_131:59:double_ran1():NDIV => ++NDIV	The NDIV variable is post incremented inside the assignment and had no side effect
AOIS_132:59:double_ran1():NDIV => --NDIV	The NDIV variable is post incremented inside the assignment and had no side effect
AOIS_133:59:double_ran1():NDIV => NDIV++	The NDIV variable is post incremented inside the assignment and had no side effect
AOIS_134:59:double_ran1():NDIV => NDIV--	The NDIV variable is post incremented inside the assignment and had no side effect
AOIS_137:60:double_ran1():j => j++	The j variable is post incremented inside the assignment and had no side effect
AOIS_139:61:double_ran1():idum => ++idum	The idum variable is post incremented inside the assignment and had no side effect
AOIS_140:61:double_ran1():idum => --idum	The idum variable is post incremented inside the assignment and had no side effect
AOIS_141:61:double_ran1():idum => idum++	The idum variable is post incremented inside the assignment and had no side effect
AOIS_142:61:double_ran1():idum => idum--	The idum variable is post incremented inside the assignment and had no side effect
AOIS_149:62:double_ran1():iy => iy++	The iy variable is post incremented inside the assignment and had no side effect
AOIS_150:62:double_ran1():iy => iy--	The iy variable is post incremented inside the assignment and had no side effect
AOIS_155:63:double_ran1():RNMX => ++RNMX	The RNMX variable is post incremented inside the assignment and had no side effect
AOIS_157:63:double_ran1():RNMX => RNMX++	The RNMX variable is post incremented inside the assignment and had no side effect
AOIS_159:63:double_ran1():RNMX => RNMX++	The RNMX variable is post incremented inside the assignment and had no side effect
AOIS_160:63:double_ran1():RNMX => RNMX--	The RNMX variable is post incremented inside the assignment and had no side effect
AOIS_161:63:double_ran1():temp => temp++	The temp variable is post incremented inside the assignment and had no side effect
AOIS_162:63:double_ran1():temp => temp--	The temp variable is post incremented inside the assignment and had no side effect
AOIS_163:64:double_ran1():temp => temp++	The temp variable is post incremented inside the assignment and had no side effect
AOIS_164:64:double_ran1():temp => temp--	The temp variable is post incremented inside the assignment and had no side effect
LOI_27:54:double_ran1():idum => ~idum	Does not lead to an observable difference
LOI_28:54:double_ran1():IQ => ~IQ	Does not lead to an observable difference
LOI_29:55:double_ran1():IA => ~IA	Does not lead to an observable difference
LOI_30:55:double_ran1():idum => ~idum	Does not lead to an observable difference
LOI_31:55:double_ran1():k => ~k	Does not lead to an observable difference
LOI_32:55:double_ran1():IQ => ~IQ	Does not lead to an observable difference
LOI_41:61:double_ran1():idum => ~idum	Does not lead to an observable difference

Table 22: Rational for Random_01 Low Mutation Score

By looking at Table 18, it can be seen that the Random_03 program had 30 ROR mutants injected, 23 were killed and 7 remained alive. Compared to the three random number generator programs this was the worst performing program relating to detecting ROR mutants. To evaluate why this was the case we reviewed all the RORs generated for the Random_03 program, that are listed below in Figure 24. The 7 mutant live mutants are highlighted in italics. The bold text to the right is the original code statement, is to provide a level of context to the mutation.

ROR_1:32:double_ran3(): iff == 0 => iff > 0
ROR_2:32:double_ran3(): iff == 0 => iff >= 0
ROR_3:32:double_ran3(): iff == 0 => iff < 0
ROR_4:32:double_ran3(): iff == 0 => iff <= 0 **// if(idum<0||iff==0)**
ROR_5:32:double_ran3(): iff == 0 => iff != 0
ROR_6:38:double_ran3(): i <= 54 => i > 54
ROR_7:38:double_ran3(): i <= 54 => i >= 54
ROR_8:38:double_ran3(): i <= 54 => i < 54 **// for(i=1;i<=54;i++)**
ROR_9:38:double_ran3(): i <= 54 => i == 54
ROR_10:38:double_ran3(): i <= 54 => i != 54 **// for(i=1;i<=54;i++)**
ROR_11:47:double_ran3(): k <= 4 => k > 4
ROR_12:47:double_ran3(): k <= 4 => k >= 4
ROR_13:47:double_ran3(): k <= 4 => k < 4
ROR_14:47:double_ran3(): k <= 4 => k == 4
ROR_15:47:double_ran3(): k <= 4 => k != 4
ROR_16:48:double_ran3(): i <= 55 => i > 55
ROR_17:48:double_ran3(): i <= 55 => i >= 55
ROR_18:48:double_ran3(): i <= 55 => i < 55 **//for(i=1;i<=55;i++)**
ROR_19:48:double_ran3(): i <= 55 => i == 55
ROR_20:48:double_ran3(): i <= 55 => i != 55 **//for(i=1;i<=55;i++)**
ROR_21:59:double_ran3(): ++inext == 56 => ++inext > 56
ROR_22:59:double_ran3(): ++inext == 56 => ++inext >= 56 **//if(++inext == 56){inext=1;}**
ROR_23:59:double_ran3(): ++inext == 56 => ++inext < 56
ROR_24:59:double_ran3(): ++inext == 56 => ++inext <= 56
ROR_25:59:double_ran3(): ++inext == 56 => ++inext != 56
ROR_26:65:double_ran3(): ++inexpt == 56 => ++inexpt > 56
ROR_27:65:double_ran3(): ++inexpt == 56 => ++inexpt >= 56 **//if(++inexpt == 56){inexpt = 1;}**
ROR_28:65:double_ran3(): ++inexpt == 56 => ++inexpt < 56
ROR_29:65:double_ran3(): ++inexpt == 56 => ++inexpt <= 56
ROR_30:65:double_ran3(): ++inexpt == 56 => ++inexpt != 56

Figure 24: Random_03 Relational Mutants

Table 23 provides an explanation of the impact of the mutants.

ROR	Description
ROR_4	Always set to 0 or 1 therefore, the mutant have no impact

ROR_8 & ROR_10	The original code is stated below before the mutation of the For loop condition:
	
	for(i=1;i<=54;i++)
	{
	ii=(21*i)%55;
	ma[ii]=mk;
	mk=mj-mk;
	if(mk<MZ)
	{
	mk+=Mbig;
	mj=ma[ii];
	}
	
	The mutation reduces the loop by one by removing the equality sign from the loop condition or by replacing the '<=" to '!='. This would mean that not all the array elements in the array would be populated. The array is indexed via the following line of code:
	
	ii=(21*i)%55;
	
	By removing the '=' or replacing the '<=' with '!=' operator from the for loop would result that the number 34 not being generated. Therefore, the array index 34 would not be populated. In the case of RoR_08 and RoR_10 this would mean the contents would be 0 – i.e. not being populated.

ROR_18 & ROR_20	The mutants reduce the indexing by one. The mutate lead to the one less array element at the end of the array.
ROR_22 & ROR_27	This wrap the indexer of the array back to once 56 is reached. Therefore>56 will have the same effect of '=56'.

Table 23: Rationale for the non-detection of the ROR Mutants in Random_03

4.2.3 Test Subset Results

From the full tests sets, numerous optimal (minimal) tests sets were developed for each of the Programs under Test (PUT) via GAs. The subsets were generated to satisfy each of the individual objectives i.e. S, B and M and then combined to make Statement and Branch Coverage (SB) and Statement, Branch and MC/DC (SBM) sets. No pruning of these unified test set combinations was undertaken. By construction these test sets clearly have redundant test cases.

Table 24 shows the number of unique optimum test sets generated to achieve each coverage objective. Table 24 indicate the number of tests in each test set. If we take the Gammp numerical recipe, 9 unique test sets were generated to achieve statement coverage, 10 for branch coverage and 8 for MC/DC. The 'No of Test Cases (S,B,M)' column in Table 24 shows the number of test cases in each unique test set based upon the different coverage objective i.e. S – Statement, B – Branch and M – MC/DC. In Gammp case this was 4 tests in each unique test set to achieve statement coverage, and 5 for branch and MC/DC. The combined test set of SB contained 9 tests and SBM contained 14.

Recipe	Statements	Branch	MC/DC	No of Test Cases (S,B,M)	No of test for SB	No of test for SBM
Bess	11	10	10	3,5,5	8	13
Cosft2	10	12	10	2,2,2	4	6
Dawson	15	15	15	2,2,2	4	6
EI	10	20	17	4,4,4	8	12
Expint	10	10	4	5,5,8	10	18
Gammp	9	10	8	4,5,5	9	14
Gas	16	16	16	2,2,2	4	6
Plgndr	13	8	5	2,4,6	6	12
Possion	13	6	20	2,4,4	6	10
Random_1	19	19	18	2,2,2	4	6
Random_2	9	11	9	2,2,2	4	6
Random_3	12	16	13	2,2,3	4	7
RC	15	14	2	3,3,5	6	11
SVD	15	11	11	2,3,3	5	8

Table 24: Number of Optimum Test Sets Generated and Number of Tests in each Test Set

To assess the results from each subset generated for S, B, MC/DC, SB and SBM for each PUT, statistical measures were spread to the subsets generated. All of these except for range were calculated by using Microsoft Excel Functions. The statistical measures applied were:

Average - Returns the average (arithmetic mean) of its argument, which can be numbers or names arrays or reference that contain numbers.

Variance - Estimates variance based on a simple

Standard Deviation - Estimates standard deviation based on a simple

Mode - Returns the most frequently occurring or repetitive

Median - Returns the number in the middle of the set of a given numbers

Min - Returns the smallest number in a set of values

Max - Returns the largest number in a set of values

Range - Max – Min

The results are presented in Appendix B.

Table 25 summarises the results for the average MS achieved by the test subsets for all PUT, in terms of a single testing objective and then any objective i.e. includes combinations. Table 25 also shows the highest MS achieved by subsets for a single coverage objective and then for any objective. Based upon the average MS for a single coverage objective, MC/DC achieved the highest average MS for 8 programs, Statement for 4 programs and Branch for 2 programs. SBM achieved the highest average MS for 12 PUT and jointly for 2 PUT with SB.

PUT	Best Single Obj (Avg MS)	Best Objective (Avg MS)	Highest MS by single Obj	Highest MS by any Objective
Bess	MC/DC	SBM	MC/DC	SBM
Cosft2	S	SBM	S=B=M	SBM
Dawson	S	SBM	S=B	SB = SBM
EI	S	SBM	S=B	SBM
Expint	MC/DC	SBM	S	SBM
Gammp	MC/DC	SBM	B=M	SBM
Gas	MC/DC	SBM	MC/DC	SBM
Plgndr	MC/DC	SBM	MC/DC	SBM
Possion	MC/DC	SBM	S	SBM
Random_1	S	SBM	S=B=M	SB = SBM
Random_2	B	SB = SBM	B	SB = SBM
Random_3	MC/DC	SBM	MC/DC	SB = SBM
RC	MC/DC	SB = SBM	MC/DC	SB = SBM
SVD	B	SBM	B	SBM

Table 25: Summary of Subsets Results Based Upon Average and Highest MS[58]

For the 8 programs that contained complex expressions, MC/DC gained the highest average MS for 6 programs. Where the complex expressions contained eight conditions, MC/DC gained the highest average MS for all of these programs i.e. Expint, Plgndr and RC.Based upon the highest MS achieved

[58] The equality sign in the table indicates that the testing objective achieved the same MS.

by the test subsets for each PUT, Table 25 shows that MC/DC achieved the highest for 5 PUT, statement, branch and MC/DC achieved equally the highest MS for 2 PUT. For 9 of the PUT SBM achieved the highest MS, while for 5 PUT SB achieved the same MS as SBM. For the eight programs that contained complex expressions, SBM achieved the highest MS for 4 programs i.e. Expint, Plgndr, Cosft2 and Gas. For the other 4 programs that contained complex expressions i.e. RC, Random_01, Random_02 and SVD, SB achieved the same MS as SBM. For the three programs that contained complex expressions that contained 8 conditions i.e. programs Expint, Plgndr and RC, SBM achieved the highest MS for Expint and Plgndr.

Table 26 shows the differences between the MS achieved from the full test set (FTS) applied to the PUT and the highest (max) and lowest (min) MS achieved by the test subsets for any coverage objective. Also shown in Table 26 are the differences between the MS achieved by the full test set minus the highest average (HAVG) MS generated by the test subsets. By comparing the optimum test sets with the results gained from the full test sets mitigates the issue of mutation equivalence.

For the FTS – HAVG, all the test sets were taken from the SBM subset for the PUT. Three programs had a MS difference greater than 0.1 or more than 10%: Bessj (0.175), Expint (0.363) and Possion (0.189). For all other PUT the difference based upon the average are smaller. For the Random_01 program this is very small indeed (0.008). ***Thus for three of the fourteen PUT, an average day with a criterion that gives a good deal of redundancy can give rise to seriously poor fault-finding ability***. In the context of critical systems development this would be worrying.

By comparing the full test set MS minus the highest MS achieved by the test subsets, it is indicated that for two programs i.e. Random_01 and Random_02 there was no difference. For Costf2, Random_03, SVD, Plgndr and Dawson the difference between the full test set MS and the highest MS achieved was also small. Expint had the highest difference of 0.136 and the only program with a percentage difference of greater than 10%.

PUT	FTS – HAVG MS	Min MS	Max MS	FTS MS	FTS –Min	FTS -Max
Bessj	0.175	0.055	0.804	0.831	0.776	0.027
Cosft2	0.026	0.558	0.904	0.905	0.347	0.001
Dawson	0.059	0.369	0.886	0.891	0.522	0.005
EI	0.047	0.397	0.785	0.797	0.400	0.012
Expint	0.363	0.167	0.707	0.843	0.676	0.136
Gammp	0.046	0.270	0.773	0.811	0.541	0.038
Gasdev	0.052	0.665	0.710	0.750	0.085	0.040
Plgndr	0.018	0.249	0.924	0.931	0.682	0.007
Possion	0.189	0.232	0.863	0.904	0.672	0.041
Ran1	0.008	0.524	0.655	0.655	0.131	0.000
Ran2	0.032	0.603	0.855	0.855	0.252	0.000
Ran3	0.018	0.713	0.877	0.881	0.168	0.004
RC	0.022	0.032	0.809	0.859	0.827	0.050
SVD	0.020	0.756	0.809	0.812	0.056	0.003

Table 26: MS Differences Between the Full Test Set, Highest Average MS, Highest and Lowest Subset

Considering the MS obtained by the full test set minus the lowest MS achieved by any of the subsets, indicates a high degree in fault finding capability. All the lowest MSs were achieved by a single testing objective, i.e. in five cases it was statement coverage. For two PUT i.e. SVD and Gasdev the differences are small i.e. less than 10%. However, for all other programs this is not the case. For Bessj and RC the minimum MS achieved by them was below 0.1.

The main reason for the very low MS in some of the PUT was due to that these programs contained code that returned a default value i.e. zero if the input value was inside a given range. This lead to a number of test cases returning zero as the resulting output. In the case of Bessj, 4 test sets were generated for statement coverage that achieved a MS of less then 0.1.

Table 26 clearly shows that using a single coverage objective corresponds to a very significant - in all cases resulting in a reduction in fault-finding capability. While in two cases this was low, in all other cases the fault findings capability was significant. Thus, optimal tests sets (in terms of test set size) may come at a significant price. The developers may get “lucky” in that the test suite created happens to be at the top end of the fault finding range for sets of this size, or may be “unlucky”, or even very unlucky, with a seriously under-performing, albeit coverage criterion compliant test set.

4.2.4 Results from Miscellaneous and Sorting Algorithms.

The sorting algorithms were developed from using [Sedgewick 03] and [Knuth 98] algorithms. The miscellaneous algorithms were taken from three internet sites. The vector statistical package[59] (Matrix Package) is a matrix class that include a number of matrix manipulations methods i.e. sub setting of matrix, simple calculations i.e. multiplication, adding, subtraction, squaring a matrix, etc. The calculator[60] program based upon no preference order. The roman numerical[61] program converts Roman calendar to Gregorian calendar dates.

File Name	Lines	Statements	Percent Branch Statements	Maximum Complexity
Shell-Sort.java	273	33	12.1	3
Quick-Sort.java	299	28	21.4	7
MergeSort.java	58	44	18.2	7
Insertion-Sort.java	300	20	10	4
Heap-Sort.java	71	40	17.5	5
Bubble_Sort.java	291	23	13	4
Calc.java	246	104	39.4	32
Matrix.java	1,050	457	18.4	6
Year.java	102	66	19.7	14

Table 27: Non-numerical Program Properties

Unlike the numerical recipes, all the miscellaneous programs were tested by embedding an oracle inside each JUnit test case. For the sorting algorithms a simple oracle was used that checked the next element in the array was greater or equal to that element in the array. If this was not the case, a JUnit failure was flagged.

Table 28 shows the MS achieved for each miscellaneous program by mutant subtype. Since the six sorting algorithms generated a variety of MS, we decided to analyse the lowest MS sorting program i.e. Shell Sort. The unmutated shell sort program is shown in Figure 25. LN in Figure 25 refers to line number.

To analyse the Shell Sort program we used the same approach as defined in paragraph 4.2.2. We imported the Shell Sort program into Eclipse and generated a JUnit test to drive the shell sort program. We then manually injected the mutants into the program and compared the output from the

[59] Vector statistical package - http://math.nist.gov/javanumerics/jama/ (The Matrix Package),

[60] Calculator - http://klogd.users.sourceforge.net/ (It was the StringParser class that was used).

[61] Numeral roman year converter - "https://www.planet-source code.com/vb/ scripts/ShowCode.asp?txtCodeId=6305 &lngWId=2"

original and mutated programs. This was achieved by copying the output into two separate files and using the compare function in NotePad++.

We examined the test summary file for the two mutant sub types i.e. AORB and AOIS for the shell sort program. This indicated what test passed and failed. For the test passed, we extracted out from the mutation log those mutants for AORB and AOIS.

For all the mutants injected (apart from 5: 3 for AORB and 2 for AOIS) the mutation lead to no negative side effect or what [Beizer 90] refers to as coincidental correctness. Therefore, there were no detectable differences between the, none and mutated programs.

For five mutants, no differences in outputs were generated. However, in all of these cases the outputs still generated output that was in ascending order. However, the actual values were not the same as the original values. Since the injected mutants lead to a negative side effect on the actual value and did not lead to an incorrect ordering. Therefore, the test still passed, since the array was sorted in ascending order, but the values no longer matched the original values.

Due to this discover all other mutants for the Shell Sort program were examined, however, no other discrepancies was discovered. However, since we found this issue for the AORB and AOIS mutant sub types we examined the bubble sort program and discovered one discrepancy for the AOIU mutant sub-type.

PUT/No of Test [XXX]	COR	COI	COD	AODU	AODS	AOIU	AORB	AORS	LOI	LOR	LOD	ROR	SOR	ASRS	AOIS	Overall M.S
Bubble Sort [500]	-	1.000 (0,3)	-	-	-	1.000 (0,4)	0.813 (3,13)	1.000 (0,1)	0.909 (1,10)	-	-	0.933 (1,14)	-	-	0.813 (6,26)	0.865
Heap Sort [500]	-	1.000 (0,6)	-	-	-	1.000 (0,11)	1.000 (0,24)	1.000 (0,3)	1.000 (0,23)	-	-	0.800 (4,16)	-	-	0.833 (11,55)	0.902
Insertion Sort [500]	1.000 (0,2)	1.000 (0,4)	-	-	-	1.000 (0,4)	1.000 (0,8)	1.000 (0,2)	1.000 (0,11)	-	-	0.800 (2,8)	-	-	0.900 (3,27)	0.928
Merge Sort [500]	1.000 (0,2)	0.778 (2,7)	-	-	-	1.000 (0,8)	1.000 (0,8)	1.000 (0,10)	0.933 (2,28)	-	-	0.800 (1,4)	-	-	0.911 (5,51)	0.922
Quick Sort [500]	-	1.000 (0,6)	-	-	-	1.000 (0,10)	1.000 (0,8)	1.000 (0,4)	0.960 (1,24)	-	-	0.867 (2,13)	-	-	0.875 (10,70)	0.912
Shell Sort [500]	-	1.000 (0,4)	-	-	-	0.778 (2,7)	0.75 (10,18)	1.000 (0,1)	0.857 (3,18)	-	-	0.850 (3,17)	-	-	0.727 (22,46)	0.778
Avg	**1.000**	**0.963**	**-**	**-**	**-**	**0.921**	**0.909**	**1.000**	**0.943**	**-**	**-**	**0.842**	**-**	**-**	**0.835**	**0.8845**
Matrix [2500]	1.000 (0,2)	0.928 (6,77)	-	1.000 (0,1)	-	0.941 (9,144)	0.981 (3,153)	1.000 (0,74)	0.968 (13, 387)	-	-	0.825 (14, 66)	-	1.000 (0,16)	0.929 (71,927)	0.946
Year [500]	-	1.000 (0,13)	-	-	-	-	-	-	1.000 (0,13)	-	-	0.923 (5,60)	-	0.827 (9,43)	1.000 (0,52)	0.928
Calc [1000]	0.875 (3,21)	1.000 (0,48)	-	-	1.000 (0,4)	0.882 (2,15)	1.000 (0,60)	0.941 (1,16)	0.977 (1,42)	-	-	0.811 (33,142)	-	-	0.802 (48,194)	0.860
Avg	**0.938**	**0.976**	**-**	**1.000**	**1.000**	**0.912**	**0.990**	**0.971**	**0.981**	**-**	**-**	**0.853**	**-**	**0.913**	**0.910**	**0.911**

Table 28: Number of Mutants Alive, (Killed) by Mutation and Mutation Operator and Subtype

LN	Function Shell Sort
20	static public void Shell_Sort(int [] shell_array, int elements)
21	{
22	int k = 1;
23	int noswap;
24	do{
25	do{
26	noswap = 0;
27	noswap = scan (shell_array, elements, k);
28	}while(noswap >0);
29	k =(k - 1) / 2;
30	}while(k > = 1);
31	}//end of function Shell_Sort

LN	Function scan
33	static public int scan (int [] sub_shell_array, int sub_noswap, int sub_k)
34	{
35	int temp = 0;
36	int return_noswap = 0;
37	int ubound = (sub_noswap-1) - sub_k;
38	for(int i=0; i<=ubound; i++){
39	if(sub_shell_array[i] > sub_shell_array [i + sub_k]){
40	temp = sub_shell_array[i];
41	sub_shell_array[i] = sub_shell_array[i + sub_k];
42	sub_shell_array[i + sub_k] = temp;
43	sub_noswap = 1;
44	return_noswap = sub_noswap;
45	}
46	}
47	return return_noswap;
48	}//end of function scan

Figure 25: Shell Sort Java Source Code

Mutation Operator and Mutate	Reason
AORB_1:29:void_Shell_Sort(int,int):k - 1 => k * 1	The k variable refers to the nth element ^{in} dividing the array. No negative side effect.
AORB_2:29:void_Shell_Sort(int,int):k - 1 => k / 1	The k variable refers to the nth element ^{in} dividing the array. No negative side effect.
AORB_3:29:void_Shell_Sort(int,int):k - 1 => k % 1	.
AORB_5:29:void_Shell_Sort(int,int):(k - 1) / 2 => (k - 1) * 2	The k variable refers to the nth element ^{in} dividing the array. No side effect.
AORB_6:29:void_Shell_Sort(int,int):(k - 1) / 2 => (k - 1) % 2	The k variable refers to the nth element ^{in} dividing the array. No side effect.
AORB_7:29:void_Shell_Sort(int,int):(k - 1) / 2 => k - 1 + 2	The k variable refers to the nth element ^{in} dividing the array. No side effect.
AORB_8:29:void_Shell_Sort(int,int):(k - 1) / 2 => k - 1 – 2	The k variable refers to the nth element ^{in} dividing the array. No side effect.
AORB_21:41:int_scan(int,int,int):i + sub_k => i * sub_k	The value held in the array is manipulated. The values in the array is still sorted in ascending order, however, the actual values do not match the original values.
AORB_22:41:int_scan(int,int,int):i + sub_k => i / sub_k	The value held in the array is manipulated. The values in the array is still sorted in ascending order, however, the actual values do not match the original values.
AORB_23:41:int_scan(int,int,int):i + sub_k => i % sub_k	The value held in the array is manipulated. The values in the array is still sorted in ascending order, however, the actual values do not match the original values.
AOIS_4:27:void_Shell_Sort(int,int):elements => elements--	The array is still correctly sorted even when post decrement is applied. Since on the first sort the correct number of elements is sorted. No negative side effect.
AOIS_7:27:void_Shell_Sort(int,int):k => k++	The k variable refers to the nth element ^{in} dividing the array. No negative side effect.
AOIS_11:28:void_Shell_Sort(int,int):noswap => noswap++	The post increment operator lead to no negative side effect in calling the swap function the correct number of time.
AOIS_12:28:void_Shell_Sort(int,int):noswap => noswap—	The post decrement operator lead to no negative side effect in calling the swap function the correct number of time.
AOIS_13:29:void_Shell_Sort(int,int):k => ++k	The k variable refers to the nth element ^{in} dividing the array. No negative side effect.
AOIS_14:29:void_Shell_Sort(int,int):k => --k	The k variable refers to the nth element ^{in} dividing the array. No negative side effect.
AOIS_15:29:void_Shell_Sort(int,int):k => k++	The k variable refers to the nth element ^{in} dividing the array. No negative side effect.
AOIS_16:29:void_Shell_Sort(int,int):k => k--	The k variable refers to the nth element ^{in} dividing the array. No negative side effect.
AOIS_18:30:void_Shell_Sort(int,int):k => --k	The k variable refers to the nth element ^{in} dividing the array. No negative side effect.
AOIS_19:30:void_Shell_Sort(int,int):k => k++	The k variable refers to the nth element ^{in} dividing the array. No negative side effect.
AOIS_20:30:void_Shell_Sort(int,int):k => k--	The k variable refers to the nth element ^{in} dividing the array. No negative side effect.
AOIS_23:37:int_scan(int,int,int):sub_noswap => sub_noswap++	The post increment operator lead to no negative side effect in calling the swap function the correct number of time.
AOIS_24:37:int_scan(int,int,int):sub_noswap => sub_noswap--	The post decrement operator lead to no negative side effect in calling the swap function the correct number of time.
AOIS_59:42:int_scan(int,int,int):temp => ++temp	The value held in the array is manipulated. The values in the array is still sorted in ascending order, however, the actual values do not match the original values.

AOIS_60:42:int_scan(int,int,int):temp => --temp	The value held in the array is manipulated. The values in the array is still sorted in ascending order, however, the actual values do not match the original values.
AOIS_61:42:int_scan(int,int,int):temp => temp++	The post increment operator lead to no negative side effect.
AOIS_62:42:int_scan(int,int,int):temp => temp--	The post increment operator lead to no negative side effect.
AOIS_63:44:int_scan(int,int,int):sub_noswap => ++sub_noswap	The post increment operator lead to no negative side effect.
AOIS_65:44:int_scan(int,int,int):sub_noswap => sub_noswap++	The post increment operator lead to no negative side effect.
AOIS_66:44:int_scan(int,int,int):sub_noswap => sub_noswap--	The post increment operator lead to no negative side effect.
AOIS_67:47:int_scan(int,int,int):return_noswap => return_noswap++	The post increment operator lead to no negative side effect
AOIS_68:47:int_scan(int,int,int):return_noswap => return_noswap--	The post increment operator lead to no negative side effect

Figure 26: AORB and AOIS Defects Not Detected in the Shell Sort Program

4.3 Program Properties

We applied commonly used metrics to our PUTs, listed in Table 15. Our results indicate that there does not appear to be any correlation between commonly applied metrics and test effectiveness. It is noted that mutation equivalence has not been taken into account in our program properties.

Two of the programs that had the lowest complexity i.e. Random_01 and Gasdev had a complexity of 9, achieved the lowest MS of the 14 numerical recipes. Cost2 and SVD_NR programs included the largest number of statements, however, achieved a MS of 0.90 and 0.8122. The 4 programs that had the highest percentage of branches i.e. greater than 20%, i.e. EI, Expint, Plgndr and SVR_NR achieved a MS of 0.797, 0.843, 0.930 and 0.812. The three programs with the lowest branch percentage of branches i.e. RC_fun, Costft2 and Dawson achieved a MS of 0.85, 0.90 and 0.89.

The sorting algorithms results show that 4 out of the 6 achieved a MS greater than 0.90. The two sorting algorithms that achieved the lowest MS i.e. Shell-Sort and Bubble_Sort had a percentage branch of 12.1 and 13 and a maximum complexity of 3 and 4. The Calc program has highest percentage of branch statements of 39.4 and a maximum complexity of 32 and achieved a MS of 0.860.

4.4 Conclusions

This chapter has shown the results from measuring effectiveness and reliability from conducting automated random testing on programs that been fault injected. The programs used were primarily numerical recipes as our main experimental programs, but other programs i.e. sorting algorithms etc have been tested and the results reported in this chapter. For the numerical programs we examined effectiveness and reliability, while for the other programs we only examined effectiveness. For the numerical recipes we generated a number of optimum unique test cases that met each of the coverage criterions. We also combined these single optimum coverage based test sets that add redundancy to examine the effectiveness and reliability with limited redundant test sets.

The programs used in our experiments are relatively small, however, all these programs contained selection, iteration, and sequential program constructs. The majority of the programs under test used array constructs. It is noted that the programs are traditional programs and non-object oriented i.e. no use of classes and the programs were not real time. Therefore, in some instances the results gained here may not be scalable for some specific programs. However, it is believed that for a range of programs that used traditional programs and the basic foundation blocks of programs i.e. iterations and selection the results gained here are scalable. This is similar to the testing framework developed here. For any program that accepts an input and generates an output the framework is scalable. For

systems that does not generate external outputs but contains internal state data, this state data could be logged and used have the verification check points. Therefore, it is believed that the results gained here are scalable when comparing traditional developed programs and non real time. Also the framework is scalable to enable much larger programs to be tested in a similar fashion has achieved here or by small modifications to the program under test.

We have not taken into account mutation equivalence for the full test set i.e. adjusted the MS for the full test set, since we are primarily interested in examining the effectiveness and reliability of the optimum test sets that meet the three widely used coverage criterions. By comparing the optimum test set results against the full test sets results removes the issue of mutation equivalence. However, we have examined mutation equivalence in this chapter that would appear to count for the low MS for some programs. We have shown that for AOIS, AOIU, LOR and ROR that in the majority of cases the mutants were mutant equivalent. We have shown that program structure prevent injected mutants causing any differences at the local program level. While a small number of mutants did lead to localised changes, the global result was unaffected. This does lead to an interesting question about what counts as detection and where to place verification points. While here we are not directly interested on this issue, since we are performing strong mutation it is an issue relating to weak mutation and been raised by some authors i.e. [Woodward & Halewood 88].

Our results question the blind faith in the adequacy of three widely accepted coverage objectives. This is particularly relevant where MC/DC coverage is used as a stopping criterion, since this is the "*gold standard*" in many ways for civil avionics application testing. Our results show that for the programs tested coverage and criterion based testing may not be as reliable as people hope or expect. Developers may get "lucky" or "unlucky" in terms of fault finding ability of developed test sets. In particular, paring test suites down to be "optimal" with respect to an identified criterion may result in a very significant reduction in fault finding ability.

One aspect we have not addressed here is reduction of test set size whilst maintaining mutation score, i.e. what subsets maintain the full test set MS. This remains a credible objective but can clearly be addressed largely by the same search apparatus we have created. Indeed, there are many further notions of optimality that can and should be investigated. For example, if a means to evaluate the "cost" of an individual test (or indeed test set) is available, we could assess the statistical variability of mutation scores for test sets of a specified cost (budget constrained testing). Cost could be an estimate of manual effort to extract a test result, i.e. an oracle cost.

Reliability of criteria is an important topic of investigation, especially since important software safety standards mandate the use of specific coverage criteria. The standards are perceived as having

reached a view on the efficacy of testing resulting from applying those criteria. In practice, developers may be inclined to stop when they have satisfied the indicated criteria. However, unless the criterion is reliable this is no guarantee of fault finding effectiveness. Reliability of criteria needs to be established on a sound basis and is clearly under researched. This chapter contributes to our understanding of reliability of criteria.

The primary PUTs are taken from a specific domain (numerical algorithms) and undoubtedly have their own characteristics. This is also true for the other programs we assessed have part of this chapter. However, the framework described in chapter 3 has very significant potential for wider scale experimentation on other code repositories. The work presented here is proof of concept; its leveraging of automatic test data generation and subset extraction provides a promising exploitation avenue for other work in these areas.

Our repeated application of mutation analysis demonstrates very clearly that all criterion-satisfying **test subsets are not equal**. We have used mutation analysis repeatedly as a reference criterion for judging comparative effectiveness of test sets, i.e. we have used it as a scientific tool. But mutation analysis is widely recognized as a stringent criterion in its own right and perhaps the major implication of our work is that mutation adequacy might usefully be adopted more widely as the primary "coverage" criterion.

5. Testing Encryption Algorithms

Our final research objective, identified in section 1.2 and reproduced below, is concerned with how thoroughly high profile applications are tested by specific test suites crafted by experts. The final research objective is:

- To measure the effectiveness of three reference test sets developed to test a DES algorithm.

Reference test sets (i.e. a set of input and output pairs) for cryptographic algorithms have been developed by cryptographic standards bodies or other international groups of experts. The aim is to provide recognisable and widely adopted benchmark suites against which implementations can be tested. For some target algorithms there may be several such suites. This begs the question: why?
Do such suites actually test the target applications with the rigour expected? Are they different in what they achieve? Thus, we would wish know whether the developed test sets are actually effective and whether expert crafting of such test sets is a reliable approach. The former is the primary goal (though we may infer *some* evidence regarding the latter). The above questions are important: a faulty implementation that passes reference tests could be the subject of misplaced confidence and the source of unintended information leakage when exploited by a knowledgeable security expert.

The research reported in this chapter carries out mutation analysis to directly support the final research objective. DES is representative of a typical symmetric key block encryption algorithm (where the sender and receiver have the same key). Such algorithms typically have internationally supported reference test sets developed for them. However, our investigation also enables us to gain some insight into whether cryptographic algorithms are easy to test, i.e. whether gaining mutation adequacy is a trivial matter requiring only a few tests. Although attaining such insight is not the identified goal of the research, it is of interest to researchers in the field. Furthermore, since it emerges as a by-product of our primary research we report it here. DES is highly iterative and the internal state data is highly interdependent. We might hypothesise that this results in an extremely fragile problem, with likely propagation of the effects of inserted mutations, leading to significant levels of observability and detectability. We apply further mutation analysis to the Java BigInteger package. BigInteger forms a useful component for implementing many public key algorithms (e.g. modular exponentiation forms the basis of the RSA algorithm, where $C = P^d \bmod n$). We choose this extra target to ensure that any inferences with respect to testability are not specific to block ciphers such as DES.

As far as we are aware, there is no known formal assessment of the ability of available test sets to find flaws in the implementation. This chapter sets out to provide just that. The availability of the test sets makes it possible for others to repeat the tests conducted in this chapter and further the domain knowledge on the effectiveness of test reference systems for encryption-based systems.

For the DES java program we applied different publicly available test vectors to assess their effectiveness. We used the refined testing framework defined in chapter 3. As in the previous chapter we capture coverage metrics for each test. We originally planned to generate test subsets, however, during the execution of the test vectors it was shown that usually a single test sufficed to achieved statement, branch and MC/DC. Thus the mutation scores achieved by the set of test executions will simultaneously provide the reliability of the coverage criteria.

This chapter is divided into the following sections that cover:-

- Programs under test, program properties and test vectors (5.1).
- The results generated from running the test vectors (5.2).
- Conclusions (5.3)

5.1 Programs under Test, Program Properties and Test Vectors

The Java implementation of DES was taken from http://www.herongyang.com/. DES is a symmetric block cipher, operating on blocks of 64 bits of data and a key of 64 bits. 56 bits are working key bits, and the other 8 are parity bits. From the 56 working bits 16 subkeys are extracted, each of 48 bits. These subkeys are typically denoted by K0, K1, …, K15. The algorithm is essentially a concatenation of 16 identical mini-ciphers, called rounds and the jth round uses the subkey Kj. DES is often described as an *iterated block cipher*. DES has a particular structure, often referred to as a Feistel cipher. Feistel ciphers have the convenient property that encryption and decryption are essentially the same algorithm, but with the same subkeys as encryption, applied in reverse order.

In detail DES has the following steps:

1. A block of 64 bits is permuted by an initial permutation IP.

2. Resulting 64 bits are divided in two halves of 32 bits, left and right.

3. Right half goes through a function F (Feistel function).

4. Left half is XOR-ed with output from F function above.

5. Left and right are swapped (except last round).

6. If last round, apply an inverse permutation IP^{-1} on both halves and that's the output else, goto step 3

Figure 27 illustrates the above.

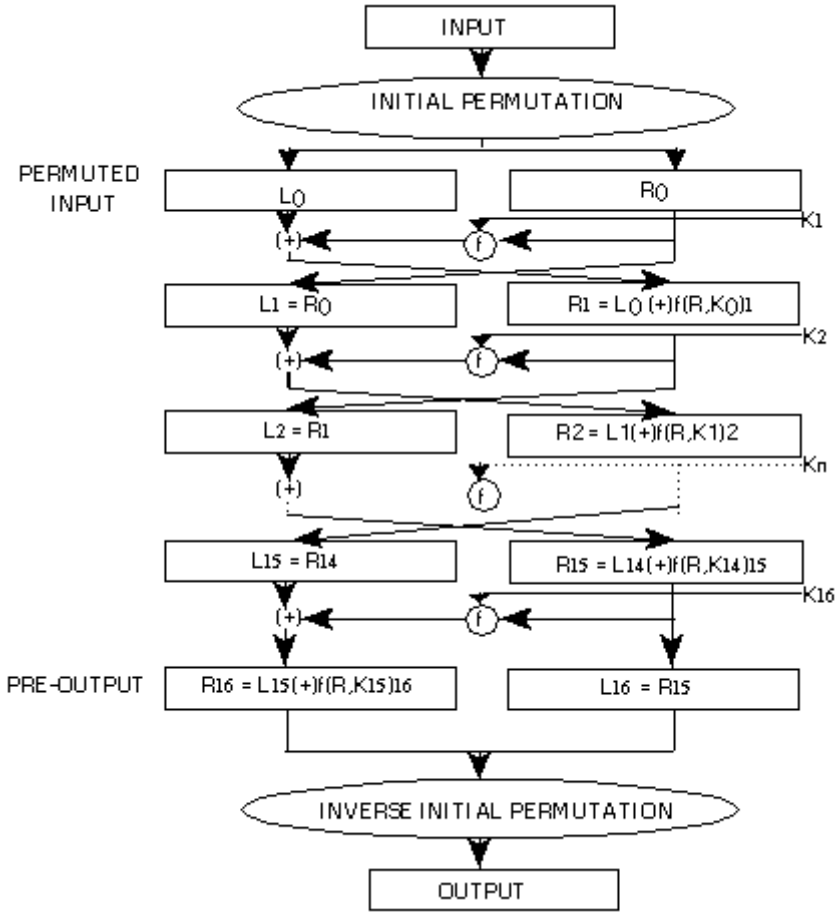

Figure 27: DES Implementation
(Quoted directly from [N32011])

DES can now be broken by raw computational power. However, it was for a long time the most widely used cryptographic algorithm in the world and was in many ways a remarkable intellectual achievement. It is used here since cipher source code is widely available and a number of test vectors are publicly available to test the cipher.

To test the DES implementation we applied 3 different publically declared test vectors. These test vectors are listed below:

DES Test Vector One
Project NESSIE - New European Schemes for Signature, Integrity, and Encryption
https://www.cosic.esat.kuleuven.be/nessie/testvectors/bc/des/Des-64-64.test-vectors
Contained 772 Tests

DES Test Vector Two
NIST Modes of Operation Validation System (MOVS): Requirements and Procedures – Appendix B - Table 1
http://csrc.nist.gov/publications/nistpubs/800-17/800-17.pdf

Contained 64 Tests

DES Test Vector Three
NBS's validation for DES
http://www.skeptictank.org/files/faq/testdes.htm
Contained 34 Tests

The tests included three arguments to test the DES program, i.e. Key, Plaintext or message and the Ciphertext that should be returned from DES. The DES_01 test vector (TV) contained 8 different sub types that changed the key and plaintext. These are defined below:

TV1 - Change Key in systematically starting from 8000000000000000 to 0000000000000001 with identical plaintext of 0000000000000000. The key is changed from 8, 4, 2 then 1 for each bit.

TV2 - Same Key of 0000000000000000 but with the plaintext being amended in the systematic method as in TV1 for the changing the key.

TV3 - Different keys and plaintext that matched each other.

TV4 – Different key and different plaintext.

TV5 - Change Key as in TV1, cipher being 0000000000000000 and different plaintext.

TV6 – Same Key of 0000000000000000, but with Cipher changing has in TV1, different plain text.

TV7 – Key and Cipher that matched each other in a systematic method by incrementing the key. Different plaintext.

TV8 - Different Key and plaintext.

The DES_02 test vector used an odd parity set key i.e. 0101010101010101 and plain text amended in a systematic way as for DES_01 TV 2. The DES_03 test vectors are based upon different keys and plaintexts.

Testing the DES program required a separate program that passed the required arguments to the DES program. We used the test program provided in http://www.herongyang.com/. However, we amended it to accept arguments received from our test framework. Separate C# Test Data Generator components were used for each DES test vector. The test vectors were implemented via arrays inside the C# Test Date Generator. A VBA script was used to extract the test vectors into the required format. The C# Test Data Generator components then called the DES test programs with the three arguments in the test vectors.

All the test vectors were run in encrypt mode. We then ran all the test vectors again but only on the mutants that related directly to the decrypt mode source code. Nineteen mutants were executed specifically for the decrypt mode. We then combined the results from encrypt and decrypt to determine the final MS for the mutants injected in the DES program.

We also tested a Java Implementation of Big Integer. The Big Integer program was modified to accept two arguments from the C# Test Data Generator component. The two arguments were randomly generated by using C# Randomisation function. The Big Integer Program added, subtracted, divided and multiplied the two arguments and returned the results.

The program properties (metrics[62]) are listed in Table 29 below.

File Name/Metric	Lines	Statements	% Branch Statements	Method Call Statements	Methods per Class	Average Statements per Method	Maximum Complexity
Big_Integer	229	181	17.7	53	18	8.67	10
DES	270	142	11.3	41	11	11	5

Table 29: DES and Big Integer Program Properties

We instrumented the source code for DES and Big_Integer via Code-Cover to gain code coverage metrics for the PUT. For all the DES test vectors, 100% statement coverage was achieved when combining the encrypt and decrypt tests. For Branch and MC/DC coverage the test vectors gained 100% coverage except for one condition relating to error checking code that ensured the message length was the required length.

The six mutants relating to the error are shown below:

AOIS_1:21:byte_cipher(byte,byte,java.lang.String):theMsg.length => ++theMsg.length
AOIS_2:21:byte_cipher(byte,byte,java.lang.String):theMsg.length => --theMsg.length
AOIS_3:21:byte_cipher(byte,byte,java.lang.String):theMsg.length => theMsg.length++
AOIS_4:21:byte_cipher(byte,byte,java.lang.String):theMsg.length => theMsg.length--
COI_1:21:byte_cipher(byte,byte,java.lang.String): theMsg.length < 8 => !(theMsg.length < 8)
LOI_1:21:byte_cipher(byte,byte,java.lang.String):theMsg.length < 8 => -theMsg.length<8

MuClipse never generated source code for the four AOIS related mutants, since this code is not syntactically correct. For the COI_1 and LOI_1, the fault injected meant that the actual error trapping code was exercised by the mutants. Since in the COI_1 case the injection of the NOT inside the 'IF' statement executed the error trapping code. While in the case of the LOI_1 the '-', arithmetically negates the variable i.e. theMsg.length, therefore theMsg.length is then less than 8 and the error trapping code is executed.

[62] The metrics were captures via using Source-Monitor V 2.6.3.104 - http://www.campwoodsw.com/sourcemonitor.html.

5.2. Mutation Results

The results generated by the use of the refined framework are shown in Table 30. We show the results for each of the three different DES test vectors applied and the Big Integer The table also shows the number of test cases, mutants generated, mutants killed and MS for each PUT.

DES/PUT	No.of Test Cases	Mutants Generated	Mutants Killed	Raw Mutation Score
DES_01	772	774	681	0.87984
DES_02	63	774	657	0.84884
DES_03	34	774	676	0.87339
Big_Integer	500	860	712	0.82791

Table 30: Number of Tests, Mutants Generated, Killed and Raw MS

Table 31 shows the different types of MO sub-types injected into the DES implementation and the number of mutants killed by each DES test vector. Table 32 shows the MO Sub-types injected into the Big Integer implementation and the mutants killed by the Big Integer randomly generated test set. Table 33 shows the MS for each of the MO sub types for each of the DES test vectors and for the Big Integer.

DES/PUT	COI	AOIU	AORB	AORS	LOI	LOR	ROR	SOR	AOIS	Total
M.Injected	13	64	208	13	115	16	5	12	328	774
DES_01	13	64	190	13	115	15	4	7	260	681
DES_02	12	63	184	13	113	15	4	7	238	649
DES_03	13	64	190	13	115	15	4	7	255	676

Table 31: Mutant Operators Sub-Types Applied to the DES PUT.

PUT	COR	COI	COD	AODU	AOIU	AORB	AORS	LOI	ROR	AOIS	Total
M.Injected	12	41	1	2	43	96	13	113	145	394	860
Big_Int	12	39	1	1	33	87	13	105	117	303	712

Table 32: Mutant Operators Sub-Types Applied to the Big Integer PUT.

MS	COR	COI	COD	AODU	AOIU	AORB	AORS	LOI	LOR	ROR	SOR	AOIS	Total
DES_01	-	1.000	-	-	1.000	0.913	1.000	1.000	0.938	0.800	0.583	0.793	0.880
DES_02	-	0.923	-	-	0.984	0.885	1.000	0.983	0.938	0.800	0.583	0.726	0.839
DES_03	-	1.000	-	-	1.000	0.913	1.000	1.000	0.938	0.800	0.583	0.777	0.873
Big_Int	1.000	0.951	1.000	0.500	0.767	0.906	1.000	0.929	-	0.807	-	0.769	0.828

Table 33: MS for PUT

MS	COR	COI	COD	AODU	AOIU	AORB	AORS	LOI	LOR	ROR	SOR	AOIS	Total
DES_01	-	1.000	-	-	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
DES_02	-	0.923	-	-	0.984	0.885	1.000	0.983	1.000	1.000	1.000	93.29	0.849
DES_03	-	1.000	-	-	1.000	1.000	1.000	1.000	1.000	1.000	1.000	98.47	0.994

Table 34: Real MS taking into Mutation Equivalence

During the DES experiments we discovered the issue of livelock. With the original framework a timing thread was used inside JUnit that timed out and logged a failure. For the refined framework we terminate the processing thread that executes the test batch file that contained the test vectors. The consequence of this is that we cannot determine the effectiveness of tests that follow that test that caused live-lock.

To evaluate live-lock caused by the mutants for DES and Big Integer we amended the MSG component to only place a zero or one in the mutation table by examining the result files generating by running the test. If no result file existed for that test then this would leave an empty space in the mutation table, therefore, identifying test not executed. Our findings are summarised below:

DES_01 – 20 mutants generated live-lock. The test vectors generated failures for 15 of these mutants before live-lock occurred. For 5 of the test vectors no test failures occurred before livelock.

DES_02 – 7 mutants generated live-lock. The test vectors generated failures for all 7 mutants before live-lock occurred.

DES_03 – 3 mutants generated live-lock. The test vectors generated failures for all 3 mutants before live-lock occurred.

Besides the additional 5 AOIS mutants that DES_01 killed, the same results were discovered for DES_01 and DES_03 test vectors in terms of test effectiveness.

For the Big_Integer 357 of the mutants caused live-lock. For 19 of the mutants had no test failures occurred before livelock. 14 of these mutants related to LOI mutants and 4 related to ROR. Table 32 reflect the results and take into account livelock.

Due to the structure of the DES source code, it was discovered that 100% code coverage (statement, Branch and MC/DC) could be obtained by each individual test. (The only exception was the error trapping code that we discussed above). Big_Integer required only two tests to be executed to gain full coverage for statement, branch and MC/DC. The only exception was an If –Else condition that the randomly generated test cases could not exercise to achieve full branch and MC/DC coverage. It was impossible to generate a test case to fully exercise the If-Else condition.

Since each test case in each of the different DES test vectors gained the same coverage we could use each individual test case results has a measure of coverage reliability. We gained the mutation score for each test case in the different test vectors then applied the different dispersion based statistics to the results. The results are shown in Table 35. Samples of the individual test cases results i.e. mutants killed for each MO Sub-type are shown in Appendix A of this chapter across all the DES test vectors.

Since the DES_01 test vectors contained 8 different test vectors based upon changing the key or message, we show the results for all the tests and the individual sub-sets for the DES_01 test vectors. At the end of this chapter we show a small extraction of the Excel data sheets for some of the results that make up Table 35.

	Set	No of Tests	Mean	Var	Std	Min	Max	Range
	All	772	0.800942	0.002990752	0.054652	0.731266	0.865633	0.134367
	1	64	0.846212	0.000645024	0.025195	0.775194	0.865633	0.090439
	2	64	0.78991	1.33336E-05	0.003623	0.775194	0.795866	0.020672
	3	256	0.853291	6.93616E-05	0.008312	0.777778	0.860465	0.082687
	4	2	0.852713	0	0	0.852713	0.852713	0
	5	64	0.844387	0.000355643	0.018708	0.788114	0.855297	0.067183
	6	64	0.791081	1.3072E-05	0.003587	0.77261	0.794574	0.021964
	7	256	0.731266	5.45698E-30	2.33E-15	0.731266	0.731266	0
DES_01	**8**	2	0.852713	3.33847E-06	0.001292	0.851421	0.854005	0.002584
DES_02	**All**	63	0.829109	9.59576E-06	0.003073	0.824289	0.834625	0.010336
DES_03	**All**	34	0.861453	0.000298	0.017005	0.803618	0.869509	0.065891

Table 35: Statistics for the DES Test Vectors[63]

Table 35 shows the low level of dispersion based upon the statistical measures applied between the different tests in the test vectors. The overall dispersion for DES_01 is impacted by the relatively poor results for subset 7. However, the dispersion of subset is extremely small. For DES_02 this may be due to the fact that the same key is used for the entire test in DES_02. The dispersion for DES_03 is also relatively small but the keys and plaintext applied are different in each test case.

To examine the possibility of mutation equivalence, we used Eclipsed, part of the original and mutated DES source code. We developed our own test/debugging code to assess if the mutants could be detected by testing or whether failure to kill was due to mutation equivalence. By looking at the MS achieved in Table 33 it can be seen that the SOR MO were worst performing mutants for all the

[6363] Figures updated based upon when the program existed from the test program.

PUT. (We ignore AODU since only 2 mutants were inserted). By reviewing the SOR mutates injected we discover that all the mutants not detected related to the replacement of the shift operator i.e. the signed operator - '>>' with the unsigned operator '>>>'. Since all the mutants were using positives values, replacing the signed with the unsigned operator led to equivalence. To confirm this we used Eclipsed and mutated the original code manually with the mutants injected by MuClipse. We observed that no observable difference occurred. Our simple test program used both the signed and unsigned shift operators. We tested the whole range of standard integers and converted them into bytes before applying the shift operators.

The one LOR mutant not detected related to replacing the '|' bitwise inclusive OR operator with the '^' bitwise exclusive OR operator. We used a For Loop to feed a range of different integers from negative to positive values to the original source code and the mutated source code. We then compared the results, in all instances the results were the same. Deeper inspection of the code revealed that the mutant was in fact equivalent.

The one ROR mutants not detected related to replacing equality comparison in an if statement with '<=' e.g.'if (b%2==0)' with if (b%2<=0). The b variable is set to zero in a For Loop condition. Therefore, since b could never be less than zero, the code is equivalent. When all positive integers used, no differences were detected, however, this was not true when negative integers were applied.

Since AOIS mutants form the largest number of mutants injected for the DES program the mutants description and our assessment of them are detailed in Table 36. To evaluate the AOIS mutants we used the same approach has for LOR and ROR, i.e. used Eclipsed and some simple test programs, this at times used the same legacy source code to determine if mutants could be detected.

Mutate Explanation	Result
AOIS_7:26:byte_cipher(byte,byte,java.lang.String):blockSize => ++blockSize	No side effect
AOIS_9:26:byte_cipher(byte,byte,java.lang.String):blockSize => blockSize++	No side effect
AOIS_11:27:byte_cipher(byte,byte,java.lang.String):blockSize => ++blockSize	No side effect
AOIS_13:27:byte_cipher(byte,byte,java.lang.String):blockSize => blockSize++	No side effect
AOIS_15:27:byte_cipher(byte,byte,java.lang.String):blockSize => ++blockSize	No side effect
AOIS_17:27:byte_cipher(byte,byte,java.lang.String):blockSize => blockSize++	No side effect
AOIS_41:42:byte_cipher(byte,byte,java.lang.String):blockSize => ++blockSize	No side effect
AOIS_43:42:byte_cipher(byte,byte,java.lang.String):blockSize => blockSize++	No side effect
AOIS_45:42:byte_cipher(byte,byte,java.lang.String):blockSize => ++blockSize	No side effect
AOIS_47:42:byte_cipher(byte,byte,java.lang.String):blockSize => blockSize++	No side effect
AOIS_48:42:byte_cipher(byte,byte,java.lang.String):blockSize => blockSize— **All the blocksize mutates are used in expressions.**	No side effect
AOIS_81:65:byte_substitution6x4(byte):valByte => ++valByte	No side effect

AOIS_83:65:byte_substitution6x4(byte):valByte => valByte++	No side effect
AOIS_85:65:byte_substitution6x4(byte):valByte => ++valByte	No side effect
AOIS_87:65:byte_substitution6x4(byte):valByte => valByte++	No side effect
AOIS_89:66:byte_substitution6x4(byte):valByte => ++valByte	No side effect
AOIS_91:66:byte_substitution6x4(byte):valByte => valByte++	No side effect
AOIS_92:66:byte_substitution6x4(byte):valByte => valByte--	No side effect
AOIS_99:67:byte_substitution6x4(byte):r => r++	No side effect
AOIS_100:67:byte_substitution6x4(byte):r => r--	No side effect
AOIS_101:67:byte_substitution6x4(byte):c => c++	No side effect
AOIS_102:67:byte_substitution6x4(byte):c => c--	No side effect
AOIS_109:69:byte_substitution6x4(byte):hByte => hByte++	No side effect
AOIS_110:69:byte_substitution6x4(byte):hByte => hByte--	No side effect
AOIS_113:71:byte_substitution6x4(byte):lhByte => lhByte++	No side effect
AOIS_114:71:byte_substitution6x4(byte):lhByte => lhByte--	No side effect
AOIS_115:71:byte_substitution6x4(byte):hByte => hByte++	No side effect
AOIS_116:71:byte_substitution6x4(byte):hByte => hByte--	No side effect
AOIS_207:118:byte_rotateLeft(byte,int,int):numOfBytes => numOfBytes++	No side effect
AOIS_208:118:byte_rotateLeft(byte,int,int):numOfBytes => numOfBytes--	No side effect
AOIS_229:121:byte_rotateLeft(byte,int,int):val => val++	No side effect
AOIS_230:121:byte_rotateLeft(byte,int,int):val => val--	No side effect
AOIS_239:129:byte_concatenateBits(byte,int,byte,int):numOfBytes => numOfBytes++	No side effect
AOIS_240:129:byte_concatenateBits(byte,int,byte,int):numOfBytes => numOfBytes--	No side effect
AOIS_253:133:byte_concatenateBits(byte,int,byte,int):val => val++	No side effect
AOIS_254:133:byte_concatenateBits(byte,int,byte,int):val => val--	No side effect
AOIS_267:138:byte_concatenateBits(byte,int,byte,int):val => val++	No side effect
AOIS_268:138:byte_concatenateBits(byte,int,byte,int):val => val--	No side effect
AOIS_273:147:byte_selectBits(byte,int,int):numOfBytes => numOfBytes++	No side effect
AOIS_274:147:byte_selectBits(byte,int,int):numOfBytes => numOfBytes--	No side effect
AOIS_293:150:byte_selectBits(byte,int,int):val => val++	No side effect
AOIS_294:150:byte_selectBits(byte,int,int):val => val--	No side effect
AOIS_299:158:byte_selectBits(byte,int):numOfBytes => numOfBytes++	No side effect
AOIS_300:158:byte_selectBits(byte,int):numOfBytes => numOfBytes--	No side effect
AOIS_315:161:byte_selectBits(byte,int):val => val++	No side effect
AOIS_316:161:byte_selectBits(byte,int):val => val--	No side effect
AOIS_323:169:int_getBit(byte,int):pos => pos++	No side effect
AOIS_324:169:int_getBit(byte,int):pos => pos--	No side effect
AOIS_325:170:int_getBit(byte,int):posByte => posByte++	No side effect
AOIS_326:170:int_getBit(byte,int):posByte => posByte--	No side effect
AOIS_329:171:int_getBit(byte,int):valByte => valByte++	No side effect
AOIS_330:171:int_getBit(byte,int):valByte => valByte--	No side effect
AOIS_333:171:int_getBit(byte,int):posBit => posBit++	No side effect
AOIS_334:171:int_getBit(byte,int):posBit => posBit--	No side effect
AOIS_335:172:int_getBit(byte,int):valInt => valInt++	No side effect
AOIS_336:172:int_getBit(byte,int):valInt => valInt--	No side effect
AOIS_343:178:void_setBit(byte,int,int):pos => pos++	No side effect

AOIS_344:178:void_setBit(byte,int,int):pos => pos--	No side effect
AOIS_351:180:void_setBit(byte,int,int):oldByte => oldByte++	No side effect
AOIS_352:180:void_setBit(byte,int,int):oldByte => oldByte--	No side effect
AOIS_355:181:void_setBit(byte,int,int):val => val++	No side effect
AOIS_356:181:void_setBit(byte,int,int):val => val--	No side effect
AOIS_359:181:void_setBit(byte,int,int):posBit => posBit++	No side effect
AOIS_360:181:void_setBit(byte,int,int):posBit => posBit--	No side effect
AOIS_361:181:void_setBit(byte,int,int):oldByte => oldByte++	No side effect
AOIS_362:181:void_setBit(byte,int,int):oldByte => oldByte--	No side effect
AOIS_365:182:void_setBit(byte,int,int):newByte => newByte++	No side effect
AOIS_366:182:void_setBit(byte,int,int):newByte => newByte--	No side effect

Table 36: AOIS Summary for DES Program

The above mutants in Table 36 relate to the insertion of the post or pre incrementer/decrementer. Our detailed evaluation of these mutants show that these mutants are equivalent. The use of decrementers/incrementers on local variables that are not used again in a function makes no effect. Similarly variables mutated at the end of a function that are never used, again lead to no effect.

	Mutation Explanation
29	AORB_29:61:byte_substitution6x4(byte):in.length / 2 => in.length * 2
31	AORB_31:61:byte_substitution6x4(byte):in.length / 2 => in.length + 2
32	AORB_32:61:byte_substitution6x4(byte):in.length / 2 => in.length – 2
109	AORB_109:117:byte_rotateLeft(byte,int,int):len - 1 => len * 1
110	AORB_110:117:byte_rotateLeft(byte,int,int):len - 1 => len / 1
112	AORB_112:117:byte_rotateLeft(byte,int,int):len - 1 => len + 1
113	AORB_113:117:byte_rotateLeft(byte,int,int):(len - 1) / 8 => (len - 1) * 8
114	AORB_114:117:byte_rotateLeft(byte,int,int):(len - 1) / 8 => (len - 1) % 8
115	AORB_115:117:byte_rotateLeft(byte,int,int):(len - 1) / 8 => len - 1 + 8
116	AORB_116:117:byte_rotateLeft(byte,int,int):(len - 1) / 8 => len - 1 – 8
129	AORB_129:128:byte_concatenateBits(byte,int,byte,int):aLen + bLen => aLen * bLen
133	AORB_133:128:byte_concatenateBits(byte,int,byte,int):aLen + bLen - 1 => (aLen + bLen) * 1
134	AORB_134:128:byte_concatenateBits(byte,int,byte,int):aLen + bLen - 1 => (aLen + bLen) / 1
136	AORB_136:128:byte_concatenateBits(byte,int,byte,int):aLen + bLen - 1 => aLen + bLen + 1
137	AORB_137:128:byte_concatenateBits(byte,int,byte,int):(aLen + bLen - 1) / 8 => (aLen + bLen - 1) * 8
138	AORB_138:128:byte_concatenateBits(byte,int,byte,int):(aLen + bLen - 1) / 8 => (aLen + bLen - 1) % 8
139	AORB_139:128:byte_concatenateBits(byte,int,byte,int):(aLen + bLen - 1) / 8 => aLen + bLen - 1 + 8

Table 37: AORB Equivalent Mutants

Analysis of all the mutants evaluated in Table 37 indicated that they were equivalent. For example the three mutants 29, 31 and 32 all related to assigning the size of an array to a local variable. By replacing the divide by the other 3 operators make the sizes of this local variable larger. This gets ignored later since the actual array length is assigned again the variable b that is used inside the condition for the For statement. The original source code is shown in Figure 28 below.

```
private static byte[] substitution6x4(byte[] in) {
  in = splitBytes(in,6); // Splitting byte[] into 6-bit blocks
  byte[] out = new byte[in.length/2];
  int lhByte = 0;
  for (int b=0; b<in.length; b++) { // Should be sub-blocks
    byte valByte = in[b];
    int r = 2*(valByte>>7&0x0001)+(valByte>>2&0x0001); // 1 and 6
    int c = valByte>>3&0x000F; // Middle 4 bits
    int hByte = S[64*b+16*r+c]; // 4 bits (half byte) output
    if (b%2==0) lhByte = hByte; // Left half byte
    else out[b/2] = (byte) (16*lhByte + hByte);
  }
  return out;
```

Figure 28: Substitution Source Code

We have reasoned about mutation via informal testing. We also applied static code inspection to provide confirmatory evidence. However, it is clear that the equivalence of a number of these mutants could have been determined by using Static Analysis, specifically Data-Flow Analysis.

5.4 Conclusions

The key conclusions from this chapter are as follows:

1. The test vectors applied were actually very effective. In one case i.e. for DES_01 a real MS of 1 was achieved, for DES_03 a real MS of 0.994 was achieved. However, DES_01 contained 772 tests while DES_03 contained 34 tests.

2. For the DES Algorithm the level of mutant equivalence is higher than is typical in other software [Offutt et al. 1993]. However, the level of mutation equivalence would appear to be similar to the numerical recipes documented in chapter 4 of this PhD Thesis.

3. Equivalent mutants exhibit typical characteristics: they show patterns. We examined all un-killed mutants for the DES implementation. The equivalent mutants included the following typical patterns:

- Last reference using a pre or post condition or a function that is never returned i.e. AOIS mutants.

- Mutations using incrementers often got masked if the variable incremented was subsequently a numerator in a division, e.g. 32/8 is equivalent to 33/8 in integer arithmetic. (Both evaluate to 4.)

We primarily used informal testing and manual code inspection to argue about mutation equivalence. Many of these equivalent mutants could have been identified by static source code analysis or manual code inspection.

4. The applied international standard test sets were not uniformly effective. Some of the mutants were killed by the test vectors but missed by another test vector. This shows some implementation flaws could slip though the testing.

5. Mutation adequacy is a reasonable and effective criterion for assessing international standards test vectors.

Algorithms such as those we tested are somewhat unusual. Our work above demonstrates that for the most part cryptographic algorithms are not good at hiding implementation faults. In a sense they are very testable.

However, even though individual tests suffice to achieve the three traditional coverage measures, the results indicate that there is variation in achieved mutation scores between tests. In a rather extreme way, this shows some degree of unreliability of the three criteria.

However, the main observation is that available test sets are actually quite good! Future work should seek to apply the approach to other cryptographic algorithms to determine whether international test sets for more contemporary algorithms are as good. No doubt the developers of the public test sets were confident they had produced something useful for developers. The work in this chapter largely provides a more formal assessment that this is indeed the case.

6. Conclusions and Summary

This chapter sets down the conclusions and findings of this research. We recall that the identified objectives of this thesis were as follows:

1. To measure the test effectiveness of the three coverage criteria (Statement, Branch and MC/DC) mandated by a widely used commercial airborne software standard for safety critical software D0-178B [178B] and its recent updated version D0-178C [178C].

2. To measure the reliability of those three coverage criteria by comparing the effectiveness of multiple minimal size tests sets meeting these criteria.

3. To measure the reliability of the three widely used coverage criteria used in the commercial airborne safety critical software e.g. [178B] and [178C] with test sets with a small degree of redundancy. To add redundancy we plan to combine the different optimum coverage test sets.

4. To measure the test effectiveness of three reference test sets developed to test a DES algorithm.

We now summarise the major implications of our research for the research objectives, presented in the following sections:

- Conclusions (section 6.1)
- Contribution of the Thesis (section 6.2)
- Discussion (section 6.3)
- Threats to validity (section 6.4)
- Possible future work (section 6.5)

6.1. Conclusions

From the experiments conducted we can conclude the following:

1. Our findings from the numerical recipes have shown that optimum coverage criteria (in the sense of smallest size test suites satisfying the identified coverage criteria) are neither effective nor reliable. Two separate test sets that meet the same coverage criterion cannot be guaranteed to achieve the same level of test effectiveness. (This much was known.) Our research shows that they frequently do not do so and the range of effectiveness may be worrying. Our research questions the '*blind faith*' in three coverage criteria applied for commercial airborne software and used by many in the software domain as the '*stop testing criterion*'. Developers may get "lucky" or "unlucky" in terms of fault-finding ability of

developed coverage-adequate test sets. Paring test suites down to be the "optimal" with respect to an identified criterion may result in a very significant reduction in fault-finding ability. The above summarises our contribution to research objectives 1 and 2.

2. It was observed that combining optimum test sets, therefore adding redundancy of tests, increases both reliability and effectiveness. This summarises our contribution to research objective 3.

3. Our findings from applying test vectors to the DES implementation show that effectiveness and reliability is significant higher when compared with the numerical recipes. Our work demonstrates that for the most part cryptographic algorithms are not good at hiding implementation faults. In a sense they are extremely testable. This contributes to research objective 4.

4. When comparing three different test vectors for testing the DES implementation it was observed that the test vectors are not equal in their mutant killing ability. Therefore implementation flaws could be accepted in released software if passing all tests in those test vectors was the sole correctness criterion. This contributes to research objective 4.

5. Widely used program metrics e.g. LOC, percentage of Branches, Complexity etc do not impact on the effectiveness of finding errors. However, program properties in terms of iteration would appear to increase the likelihood of detecting failures.

6. We have used mutation analysis repeatedly as a reference criterion for judging comparative effectiveness of test sets, i.e. we have used it as a scientific tool. But mutation analysis is widely recognized as a stringent criterion in its own right and perhaps the major implication of our work is that mutation adequacy might usefully be adopted more widely as the primary "coverage" criterion. This conclusion is a by-product of reflection on our methodology. We note however that no widely used standard mandates the use of mutation testing. In addition we note also that the actual implementation of mutation is somewhat subjective – the use of different mutation operator set might affect results, and so the reliability of mutation itself could be an issue. However, for practical purposes we assume that some sincerely motivated set is used.

7. Mutation equivalence is far greater in the programs used here compared with earlier findings from [Offutt et al 93]. This will be discussed further in Section 6.2. We primarily used

informal testing and manual code inspection to argue about mutation equivalence. Many of these equivalent mutants could have been identified by static source code analysis or manual code inspection. This is a reflection on the methodology of our research, observation on the consequences of using particular program suites, and the efficiency of the tools infrastructure used to support our research. We acknowledge that there may be implications for the use of mutation as a primary test criterion.

6.2. Contribution of the Thesis

The central focus of this thesis was to examine the effectiveness and reliability of '*adequacy criteria'*. We evaluated by empirical experiments the *effectiveness* and *reliability* of two different types of adequacy criteria: test sets that meet specific structural code coverage and pre-defined domain-specific test sets. To enable us to undertake our research we developed a framework that performed automated random testing, mutation injection, subset extraction and captured code coverage for the PUT.

The primary achievements are:

- The development of a flexible framework to automatically generate test sets satisfying coverage criteria.

- The demonstration that for our extensive set of test programs, the targeted criteria were not particularly robust, i.e. test sets exhibited considerable variation in their fault-finding efficacy.

- The demonstration that extant test vectors for cryptographic algorithms are actually very thorough, although not all test vectors were fully mutation adequate.

- The first demonstration of repeated automatic generation of coverage adequate test sets to determine the reliability of test coverage criteria.

The first three contributions are directly related to our research objectives stated earlier. The final achievement is a methodological one. Automatic test case and test data generation has been a hot topic for many years, but the observation that repeated automated generation of test suites satisfying identified coverage criteria is a means of investigating the reliability of such criteria is an important one. As automatic test data generation improves, so may our understanding of the reliability of criteria.

6.3 Discussion

There is a lack of research in evaluating effectiveness and reliability of adequacy criteria that are used to determine when to stop testing. For imperative languages testing techniques have not fundamentally changed much since the 1970s. Many techniques defined in [Myers 79] as state of practice in the late 1970s/1980s are still state of practice today. However, how adequate are these 40 year-old techniques? This PhD thesis has evaluated the *effectiveness* and *reliability* of two different types of adequacy criteria: test sets that meet specific high-profile structural code coverage criteria and also pre-defined test sets in a specific domain. This research has provided reasons to question currently held views.

Currently, to certify safety critical airborne software we rely on a qualitative process based assessment as defined in [178B]. Furthermore, considerable faith is placed in the coverage criteria used, e.g. MC/DC. We have shown that such faith may well be misplaced. The empirical studies e.g. [Myers 78], [Basili & Selby 87] etc, show no consensus on the most effective coverage criteria, but they all agree that combining testing techniques is the most effective approach in defect detection. However, none of these empirical studies examine reliability of criteria.

This PhD is the only research we are aware of that assesses the reliability of structural testing criteria and the effects of meeting that testing criterion with minimum numbers of test cases. There exist a small number of papers that apply different subset extraction techniques to test sets to reduce test set size. However, none would appear to evaluate the reliability of test criteria. Papers looking at the future software testing research by [Harrold 00], [Bertolino 07] have indicated the need for empirical studies to examine the effectiveness of software testing techniques. However, effectiveness of techniques must be combined with reliability of technique. We also believe that we are the first in using mutation to assess the effectiveness of internationally applied test vectors for acceptance of cryptographic algorithm implementations.

[Zhu et al 97] indicate that a major cost of mutation testing is mutation equivalence. Authors like [Offutt et al 93] indicated that mutation equivalence only plays a small role and accounts for under 9% of mutants. However, our experiments have indicated this not to be the case. Table 38 clearly indicates the percentage of equivalence mutants is much higher.

Program	Mutant Type	Number of Equivalent Mutants	Percentage
DES_01	AOIS	68	20.73
	SOR	5	41.66
	LOR	1	6.25
	ROR	1	20
Random_01	AOIU	10	52.63
	AOIS	53	36.30
	LOI	7	30.43
	ROR	7	23.33
Shell_Sort	AORB	6	25
	AOIS	20	30.30

Table 38: Mutation Equivalence

6.4 Threats to Validity

We translated numeric library software into Java largely for convenience sake; we were familiar with the original libraries and they had scientific application that was of general interest to us. It is possible that the means of production of code might affect reliability of coverage based testing. In this respect the fact that we have chosen a *specific means* might affect the validity of our results; direct extrapolation without further evidence would be unwise. Thus, had we chosen auto-generated Java from, say, Statecharts or similar, our results might have differed. However, the fact that we have chosen *one means of code* generation and revealed interesting aspects of its coverage-based testing is the real point of our paper. The specific minimisation means of test set selection could also introduce a bias. However, much the same considerations apply as immediately above. Other means of minimisation should be investigated. Specific non-linear optimisation algorithms will always exhibit biases, however, it will likely be impossible a priori to identify precisely what those biases are.

Our studies are based on meaningful but quite small sets of software. It is possible that coverage based testing of larger scale or less domain specific software, software from another domain, or software developed using a different process might show more reliability. **However, we have little a priori reason to rule out the possibility that such testing would be less reliable than reported here**.

Our whole approach uses mutation testing as the reference standard for judging effectiveness. Mutation testing itself is generally regarded as a stringent criterion, but is not without its controversies. The relationship between mutation score and rigorous testing against *a notion of practical risk* would benefit from research in its own right.

We believe our work offers insights and highlights potential problems with aspects of current approaches to software testing and the confidence we have in its reliability. We tested what we did with the methods described in this paper and obtained the results reported here. We do not say that the same results would be obtained if aspects of our investigation were changed (in ways including those identified above). But we can reasonably conclude that **even stringent commonly used criteria are not *unquestionably reliable* and testers cannot simply "tick the box" – satisfy a particular mandated criterion or criteria – and believe that absolves them entirely from a need to justify *why* their testing is appropriately rigorous for their specific application.**

6.5 Possible Future Research

The following may be beneficially subject to further research:

- The application of the evaluation approach to large-scale software systems. This thesis has established only proof of concept. Larger systems inevitably raise significant issues.

- Further application of mutation analysis to further crypto-algorithms and types of algorithm. Thus, for example, hash functions might pose interesting challenges. But applying the evaluation approach to high-profile modern ciphers such as the Advanced Encryption Standard (AES) would seem appropriate.

- The implementation of the approach in a massively parallel fashion. Scientific determination of reliability of criteria requires a large amount of compute power. We now have 'the cloud': as far as we are aware, no-one has harnessed this resource in the name of investigating reliability of criteria.

- The work revealed that although the cryptographic algorithms under test had large numbers of equivalent mutants these mutants conformed to particular patterns. Furthermore these patterns should often be detectable automatically. Work in the automated detection of equivalent patterned mutants would seem beneficial. Note we: were somewhat fortunate that our primary target (reliability determination) did not require equivalent mutants to be detected. All we were interested in was ***variability*** in the mutation scores achieved by test sets. All tests sets would fail to kill equivalent mutants.

- Propagation of errors leading to observable failures is an important topic. We have seen that for DES implementation the properties of this implementation lead to high observability of errors, however, this is not true for all programs has we observed during some of the

numerical based programs. Further research is required to determine why this may be the case.

- The framework could be amended to use API calls and amend and/or add additional software components to remove the use of BAT files. By using API calls would remove the need to run number separate software components and support further modifications.

We believe the approach and results outlined in this thesis have considerable potential to shed light on the very old, but still highly current, topic of "How good are test sets?" We recommend this area to the research community.

7. Appendices

Appendix A – ECJ Parameter File

Part of the ECJ Parameter File is defined below and annotated with a brief explanation.

```
//number of generations to run
generations = 1000
//If a solution is found quit
quit-on-run-complete     = true
// stores the results of the evolutionary process at, to roll back to that state.  We set this to false, since our execution is short
checkpoint= false
prefix = ec
checkpoint-modulo = 1
// We only create one sub-population
pop.subpops = 1
// we use the default sub-population for sub-population 0
pop.subpop.0 = ec.Subpopulation
// the sub population will contain 100 individuals
pop.subpop.0.size = 100
//since we don't went duplicated, it try 100 times to try and generate a unique
pop.subpop.0.duplicate-retries = 100
//we use integers to represent the species.
pop.subpop.0.species                = ec.vector.IntegerVectorSpecies
//minimum gene value
pop.subpop.0.species.min-gene  = 0
//maximum gene value
pop.subpop.0.species.max-gene  = 4999
//the number of species to select for the solution
pop.subpop.0.species.genome-size         = 5
pop.subpop.0.species.crossover-type      = one
pop.subpop.0.species.crossover-prob      = 1.0
//=> "default" mutation, then each bit will have a 1% probability of getting bit-flipped, independent of other bits. we'll mutate with a gene-independent probability
pop.subpop.0.species.mutation-prob       = 0.01
```

This stipulates that 5 individuals will be generated, that their "default" crossover will be one-point crossover, that if we use the default crossover we will use it 100% of the time to breed individuals (as opposed to 0% direct copying), and finally that if we use the "default" mutation, then each bit will have a 1% probability of getting bit-flipped, independent of other bits.

Appendix B – Numerical Statistical Result Tables

Recipe	Coverage	Av	Var	Std	Mode	Median	Min	Max	Range
Ran1	Statement	0.62009	0.00071	0.02656	0.61572	0.61572	0.52402	0.65066	0.12664
	Branch	0.61894	0.00066	0.02562	0.61572	0.61572	0.52402	0.65066	0.12664
	MC/DC	0.61408	0.00099	0.03148	0.62882	0.62227	0.55895	0.65066	0.09170
	SB	0.63586	0.00025	0.01579	0.65066	0.63319	0.61572	0.65502	0.03930
	SBM	0.64656	0.00014	0.01198	0.65066	0.65066	0.61572	0.65502	0.03930
Ran2	Statement	0.73142	0.00277	0.05262	0.70345	0.70345	0.70345	0.82414	0.12069
	Branch	0.75207	0.00627	0.07921	0.70345	0.75345	0.60345	0.84828	0.24483
	MC/DC	0.73142	0.00277	0.05262	0.70345	0.70345	0.70345	0.82414	0.12069
	SB	0.82261	0.00405	0.06366	0.85517	0.85517	0.71034	0.85517	0.14483
	SBM	0.82261	0.00405	0.06366	0.85517	0.85517	0.71034	0.85517	0.14483
Ran3	Statement	0.78638	0.00192	0.04384	0.74627	0.78545	0.71269	0.85075	0.13806
	Branch	0.78965	0.00019	0.01375	0.79104	0.79104	0.77239	0.82836	0.05597
	MC/DC	0.81315	0.00124	0.03528	0.79851	0.80597	0.74627	0.86940	0.12313
	SB	0.83427	0.00131	0.03623	0.87687	0.83396	0.78358	0.87687	0.09328
	SBM	0.86256	0.00032	0.01799	0.87687	0.86940	0.81716	0.87687	0.05970
Bessj	Statement	0.32112	0.05901	0.24291	#N/A	0.47157	0.05490	0.70490	0.65000
	Branch	0.43961	0.01606	0.12675	#N/A	0.47206	0.08725	0.51961	0.43235
	MC/DC	0.55692	0.01417	0.11904	0.51373	0.50196	0.46275	0.78333	0.32059
	SB	0.55735	0.01378	0.11740	0.47941	0.51029	0.47941	0.78137	0.30196
	SBM	0.65608	0.02213	0.14878	0.80000	0.65441	0.50196	0.80392	0.30196
Dawson	Statement	0.65619	0.02674	0.16352	0.68571	0.68286	0.36857	0.87714	0.50857
	Branch	0.65048	0.01580	0.12572	0.68286	0.68286	0.36857	0.87714	0.50857
	MC/DC	0.64324	0.02312	0.15206	0.68571	0.68286	0.36857	0.87143	0.50286
	SB	0.78800	0.01028	0.10140	0.87714	0.87714	0.65429	0.88571	0.23143
	SBM	0.83162	0.00766	0.08754	0.88286	0.88000	0.68857	0.88571	0.19714
EI	Statement	0.70970	0.01240	0.11135	0.73418	0.73418	0.39662	0.77637	0.37975
	Branch	0.62321	0.02571	0.16033	0.73418	0.73418	0.39662	0.77637	0.37975
	MC/DC	0.65624	0.02151	0.14666	0.73418	0.73418	0.39662	0.74262	0.34599
	SB	0.72822	0.00765	0.08748	0.73418	0.73418	0.39662	0.77637	0.37975
	SBM	0.74957	0.00037	0.01933	0.73418	0.74262	0.73418	0.78481	0.05063
Gasdev	Statement	0.67025	0.00002	0.00457	0.66509	0.66981	0.66509	0.67925	0.01415
	Branch	0.67025	0.00002	0.00466	0.66509	0.66981	0.66509	0.67925	0.01415
	MC/DC	0.68234	0.00018	0.01339	0.69575	0.68278	0.66509	0.70283	0.03774
	SB	0.67762	0.00011	0.01036	0.67925	0.67689	0.66509	0.70047	0.03538
	SBM	0.69752	0.00008	0.00885	0.70047	0.69929	0.67925	0.70991	0.03066
Poidev	Statement	0.40248	0.03336	0.18264	0.32274	0.33007	0.23227	0.82152	0.58924
	Branch	0.51793	0.01986	0.14092	#N/A	0.59169	0.32763	0.63325	0.30562
	MC/DC	0.58252	0.02930	0.17118	#N/A	0.59535	0.36186	0.77751	0.41565
	SB	0.58811	0.02588	0.16088	0.59413	0.61614	0.32763	0.82641	0.49878
	SBM	0.71488	0.01234	0.11107	0.66015	0.66015	0.57457	0.86308	0.28851
SVD	Statement	0.77061	0.00015	0.01214	0.76442	0.76687	0.75583	0.79918	0.04335
	Branch	0.77921	0.00032	0.01794	0.76196	0.77198	0.75828	0.80695	0.04867
	MC/DC	0.77747	0.00022	0.01488	0.76851	0.76892	0.76196	0.80082	0.03885
	SB	0.77747	0.00022	0.01488	0.76851	0.76892	0.76196	0.80082	0.03885

Recipe	Coverage	Av	Var	Std	Mode	Median	Min	Max	Range
	SBM	0.79192	0.00021	0.01438	0.77546	0.79918	0.76933	0.80900	0.03967
RC	Statement	0.74448	0.00155	0.03939	0.79539	0.74352	0.68588	0.79539	0.10951
	Branch	0.65130	0.06977	0.26414	0.79539	0.74640	0.03170	0.79539	0.76369
	MC/DC	0.81844	0.00027	0.01630	#N/A	0.81844	0.80692	0.82997	0.02305
	SB	0.83718	0.00005	0.00722	0.84150	0.84006	0.81556	0.84150	0.02594
	SBM	0.83718	0.00005	0.00722	0.84150	0.84006	0.81556	0.84150	0.02594
Plgndr	Statement	0.61618	0.08307	0.28821	0.24913	0.81661	0.24913	0.86851	0.61938
	Branch	0.82353	0.01809	0.13448	0.89965	0.87543	0.50519	0.90311	0.39792
	MC/DC	0.89343	0.00031	0.01754	#N/A	0.88927	0.87543	0.92042	0.04498
	SB	0.88538	0.00136	0.03692	0.91696	0.89792	0.80969	0.91696	0.10727
	SBM	0.91280	0.00016	0.01257	0.92388	0.91696	0.89619	0.92388	0.02768
Expint	Statement	0.31543	0.01708	0.13067	0.18261	0.30978	0.18261	0.60217	0.41957
	Branch	0.29261	0.01731	0.13157	#N/A	0.23478	0.16739	0.59783	0.43043
	MC/DC	0.34674	0.00085	0.02914	0.36957	0.35435	0.30870	0.36957	0.06087
	SB	0.39957	0.02252	0.15005	#N/A	0.39130	0.23043	0.66522	0.43478
	SBM	0.48000	0.01399	0.11829	0.45000	0.44457	0.37826	0.70652	0.32826
Cosft2	Statement	0.79947	0.01174	0.10834	0.90105	0.81579	0.66474	0.90211	0.23737
	Branch	0.77697	0.01236	0.11120	0.90105	0.71474	0.66316	0.90211	0.23895
	MC/DC	0.78226	0.01764	0.13283	0.90158	0.81605	0.55789	0.90211	0.34421
	SB	0.79947	0.01174	0.10834	0.90105	0.81579	0.66474	0.90211	0.23737
	SBM	0.87863	0.00562	0.07498	0.90158	0.90211	0.66526	0.90368	0.23842
Gammp	Statement	0.59766	0.03577	0.18913	0.74023	0.72070	0.26953	0.76367	0.49414
	Branch	0.70723	0.00727	0.08524	0.76367	0.74609	0.54492	0.76367	0.21875
	MC/DC	0.70723	0.00727	0.08524	0.76367	0.74609	0.54492	0.76367	0.21875
	SB	0.70728	0.00837	0.09146	0.76563	0.75195	0.56055	0.76953	0.20898
	SBM	0.76504	0.00007	0.00824	0.76563	0.76660	0.74609	0.77344	0.02734

Table A1: Statistical summary results of the test subsets

Appendix C – Source Monitor Metrics

Source Monitor metrics definitions are defined below:

Lines - is the number of physical lines in a source file.

Statements - computational statements that are terminated with a semicolon character, Branches i.e. if, for and while. All attributes are counted as statements as well, though calls inside attributes are ignored. The exception controls i.e. try, catch, and finally are also counted as statements.

Percent Branch Statements - Statements that cause a break in the sequential execution of statements are counted separately. These are the following: if, else, for, do, while, break, continue, switch, case and default. The exception block statements try, catch and finally are also counted as branch statements, as are throw statements.

Methods Call Statements – All method calls are counted, in statements as well as in logical expressions.

Methods per Class - Once inside a class or interface, the "<method name>(...) {" construction identifies methods. This metric is a global average: the total method count divided by the total class count. Since most files contain only a single class, this is usually the number of methods in one class. For a Source-Monitor checkpoint, it is an average across all classes and their methods.

Average Statements per Method - The total number of statements found inside of methods found in a file or checkpoint divided by the number of methods found in the file or checkpoint.

Maximum Complexity - The complexity metric is counted approximately as defined by [McConnell 1993]. The complexity metric is calculated by the number of execution paths through a function or method. Each function or method has a complexity of one plus one for each branch statement such as if, else, for, foreach, or while. Arithmetic if statements (MyBoolean ? ValueIfTrue : ValueIfFalse) each add one count to the complexity total. A complexity count is added for each '&&' and '||' in the logic within if, for, while or similar logic statements.

Switch statements add complexity counts for each exit from a case (due to a break, goto, return, throw, continue, or similar statement), and one count is added for a default case even if one is not present. (Note: when a project's Modified Complexity option is selected, switch statements add a count of one to the complexity and the internal case statements do not contribute to the complexity metric.) Each catch or except statement in a try block (but not the try or finally statements) each add one count to the complexity as well.

Appendix D – DES Results Tables

	Test	COI	AOIU	AORB	AORS	LOI	LOR	ROR	SOR	AOIS	MS
Set 1	1	13	64	189	13	115	15	4	7	244	0.857881
	2	13	64	189	13	114	15	4	7	250	0.864341
	3	13	64	189	13	114	15	4	7	250	0.864341
	4	13	64	189	13	114	15	4	7	251	0.865633
	5	13	64	188	13	114	15	4	7	245	0.856589
	6	13	64	187	13	114	15	4	7	245	0.855297
	7	13	64	187	13	114	15	4	7	247	0.857881
	8	11	61	174	13	108	15	4	7	210	0.77907
	9	13	64	187	13	114	15	4	7	245	0.855297
	10	13	64	187	13	114	15	4	7	251	0.863049
Set 2	100	12	62	172	13	109	15	4	7	218	0.790698
	101	12	62	172	13	109	15	4	7	220	0.793282
	102	12	62	172	13	109	15	4	7	220	0.793282
	103	12	62	172	13	109	15	4	7	216	0.788114
	104	12	62	172	13	109	15	4	7	216	0.788114
	105	12	62	172	13	109	15	4	7	216	0.788114
	106	12	62	172	13	109	15	4	7	220	0.793282
	107	12	62	172	13	109	15	4	7	216	0.788114
	108	12	62	172	13	109	15	4	7	220	0.793282
	109	12	62	172	13	109	15	4	7	222	0.795866
	110	12	62	172	13	109	15	4	7	222	0.795866
Set 3	200	13	63	185	13	114	15	4	7	248	0.855297
	201	13	63	185	13	114	15	4	7	248	0.855297
	202	13	63	185	13	114	15	4	7	250	0.857881
	203	13	63	185	13	114	15	4	7	248	0.855297
	204	13	63	185	13	114	15	4	7	248	0.855297
	205	13	63	185	13	114	15	4	7	248	0.855297
	206	13	63	185	13	114	15	4	7	250	0.857881
	207	13	63	185	13	114	15	4	7	250	0.857881
	208	13	63	185	13	114	15	4	7	248	0.855297
	209	13	63	185	13	114	15	4	7	248	0.855297
	210	13	63	185	13	114	15	4	7	250	0.857881
Set 4	384	13	61	170	13	111	15	4	7	234	0.81137
	385	13	64	184	13	114	15	4	7	246	0.852713
Set 5	400	13	64	184	13	114	15	4	7	248	0.855297
	401	13	64	184	13	114	15	4	7	248	0.855297
	402	12	61	170	13	109	15	4	7	219	0.788114
	403	13	64	184	13	114	15	4	7	241	0.846253
	404	13	64	184	13	114	15	4	7	244	0.850129
	405	13	64	184	13	114	15	4	7	248	0.855297
	406	13	64	184	13	114	15	4	7	244	0.850129
	407	13	64	184	13	114	15	4	7	246	0.852713
	408	13	64	184	13	114	15	4	7	246	0.852713

	Test	COI	AOIU	AORB	AORS	LOI	LOR	ROR	SOR	AOIS	MS
	409	13	64	184	13	114	15	4	7	248	0.855297
	410	12	61	171	13	109	15	4	7	223	0.794574
Set 6	500	12	61	171	13	109	15	4	7	223	0.794574
	501	12	61	171	13	109	15	4	7	219	0.789406
	502	12	61	170	13	109	15	4	7	223	0.793282
	503	12	61	171	13	109	15	4	7	217	0.786822
	504	12	61	171	13	109	15	4	7	219	0.789406
	505	12	61	171	13	109	15	4	7	219	0.789406
	506	12	61	171	13	109	15	4	7	223	0.794574
	507	12	61	171	13	109	15	4	7	217	0.786822
	508	12	61	171	13	109	15	4	7	223	0.794574
	509	12	61	171	13	109	15	4	7	223	0.794574
	510	12	61	171	13	109	15	4	7	219	0.789406
Set 7	600	8	59	166	12	102	14	4	4	197	0.731266
	601	8	59	166	12	102	14	4	4	197	0.731266
	602	8	59	166	12	102	14	4	4	197	0.731266
	603	8	59	166	12	102	14	4	4	197	0.731266
	604	8	59	166	12	102	14	4	4	197	0.731266
	605	8	59	166	12	102	14	4	4	197	0.731266
	606	8	59	166	12	102	14	4	4	197	0.731266
	607	8	59	166	12	102	14	4	4	197	0.731266
	608	8	59	166	12	102	14	4	4	197	0.731266
	609	8	59	166	12	102	14	4	4	197	0.731266
	610	8	59	166	12	102	14	4	4	197	0.731266
Set 8	771	13	64	184	13	114	15	4	7	245	0.851421
	772	13	64	184	13	114	15	4	7	247	0.854005

Table D1: Sample of DES_01 Results

Test	COI	AOIU	AORB	AORS	LOI	LOR	ROR	SOR	AOIS	MS
1	12	63	191	13	113	15	4	7	227	0.833333
2	12	63	188	13	113	15	4	7	226	0.828165
3	12	63	185	13	113	15	4	7	230	0.829457
4	12	63	185	13	113	15	4	7	230	0.829457
5	12	63	185	13	113	15	4	7	228	0.826873
6	12	63	185	13	113	15	4	7	228	0.826873
7	12	63	185	13	113	15	4	7	230	0.829457
8	12	63	185	13	113	15	4	7	230	0.829457
9	12	63	185	13	113	15	4	7	228	0.826873
10	12	63	185	13	113	15	4	7	232	0.832041
11	12	63	185	13	113	15	4	7	228	0.826873
12	12	63	185	13	113	15	4	7	228	0.826873
13	12	63	185	13	113	15	4	7	230	0.829457
14	12	63	185	13	113	15	4	7	234	0.834625
15	12	63	185	13	113	15	4	7	230	0.829457
16	12	63	185	13	113	15	4	7	234	0.834625

Test	COI	AOIU	AORB	AORS	LOI	LOR	ROR	SOR	AOIS	MS
17	12	63	185	13	113	15	4	7	230	0.829457
18	12	63	185	13	113	15	4	7	226	0.824289
19	12	63	185	13	113	15	4	7	226	0.824289
20	12	63	185	13	113	15	4	7	230	0.829457
21	12	63	185	13	113	15	4	7	232	0.832041
22	12	63	185	13	113	15	4	7	232	0.832041
23	12	63	185	13	113	15	4	7	228	0.826873
24	12	63	185	13	113	15	4	7	232	0.832041
25	12	63	185	13	113	15	4	7	228	0.826873
26	12	63	185	13	113	15	4	7	228	0.826873
27	12	63	185	13	113	15	4	7	228	0.826873
28	12	63	185	13	113	15	4	7	228	0.826873
29	12	63	185	13	113	15	4	7	230	0.829457
30	12	63	185	13	113	15	4	7	230	0.829457
31	12	63	185	13	113	15	4	7	230	0.829457
32	12	63	185	13	113	15	4	7	234	0.834625
33	12	63	185	13	113	15	4	7	226	0.824289
34	12	63	185	13	113	15	4	7	230	0.829457
35	12	63	185	13	113	15	4	7	226	0.824289
36	12	63	185	13	113	15	4	7	230	0.829457
37	12	63	185	13	113	15	4	7	232	0.832041
38	12	63	185	13	113	15	4	7	232	0.832041
39	12	63	185	13	113	15	4	7	228	0.826873
40	12	63	185	13	113	15	4	7	228	0.826873
41	12	63	185	13	113	15	4	7	228	0.826873
42	12	63	185	13	113	15	4	7	232	0.832041
43	12	63	185	13	113	15	4	7	228	0.826873
44	12	63	185	13	113	15	4	7	232	0.832041
45	12	63	185	13	113	15	4	7	234	0.834625
46	12	63	185	13	113	15	4	7	234	0.834625
47	12	63	185	13	113	15	4	7	234	0.834625
48	12	63	185	13	113	15	4	7	230	0.829457
49	12	63	185	13	113	15	4	7	226	0.824289
50	12	63	185	13	113	15	4	7	226	0.824289
51	12	63	185	13	113	15	4	7	226	0.824289
52	12	63	185	13	113	15	4	7	230	0.829457
53	12	63	185	13	113	15	4	7	228	0.826873
54	12	63	185	13	113	15	4	7	232	0.832041
55	12	63	185	13	113	15	4	7	228	0.826873
56	12	63	185	13	113	15	4	7	228	0.826873
57	12	62	185	13	113	15	4	7	232	0.830749
58	12	62	185	13	113	15	4	7	228	0.825581
59	12	62	185	13	113	15	4	7	228	0.825581
60	12	62	185	13	113	15	4	7	232	0.830749
61	12	62	185	13	113	15	4	7	230	0.828165

Test	COI	AOIU	AORB	AORS	LOI	LOR	ROR	SOR	AOIS	MS
62	12	62	185	13	113	15	4	7	234	0.833333
63	12	62	185	13	113	15	4	7	234	0.833333

Table D2: All the DES_02 Test Vector results

Test	COI	AOIU	AORB	AORS	LOI	LOR	ROR	SOR	AOIS	MS
1	11	62	183	13	111	15	4	7	219	0.807494
2	13	62	182	13	113	15	4	7	249	0.850129
3	13	64	189	13	114	15	4	7	252	0.866925
4	13	64	188	13	114	15	4	7	253	0.866925
5	13	64	188	13	114	15	4	7	253	0.866925
6	13	64	188	13	114	15	4	7	255	0.869509
7	11	62	181	13	110	15	4	7	219	0.803618
8	13	64	188	13	114	15	4	7	253	0.866925
9	13	64	188	13	114	15	4	7	253	0.866925
10	13	64	188	13	114	15	4	7	255	0.869509
11	13	64	188	13	114	15	4	7	255	0.869509
12	13	64	188	13	114	15	4	7	253	0.866925
13	13	64	188	13	114	15	4	7	255	0.869509
14	13	64	188	13	114	15	4	7	255	0.869509
15	13	64	188	13	114	15	4	7	253	0.866925
16	13	64	188	13	114	15	4	7	255	0.869509
17	13	64	188	13	114	15	4	7	255	0.869509
18	13	64	188	13	114	15	4	7	253	0.866925
19	13	64	188	13	114	15	4	7	255	0.869509
20	13	64	188	13	114	15	4	7	255	0.869509
21	13	64	188	13	114	15	4	7	253	0.866925
22	13	64	188	13	114	15	4	7	251	0.864341
23	13	64	188	13	114	15	4	7	255	0.869509
24	13	64	188	13	114	15	4	7	255	0.869509
25	13	64	188	13	114	15	4	7	255	0.869509
26	13	64	188	13	114	15	4	7	255	0.869509
27	13	64	188	13	114	15	4	7	255	0.869509
28	13	64	188	13	114	15	4	7	255	0.869509
29	13	63	182	13	113	15	4	7	251	0.854005
30	13	64	188	13	114	15	4	7	253	0.866925
31	12	62	181	13	111	15	4	7	228	0.817829
32	13	62	181	13	113	15	4	7	249	0.848837
33	13	64	187	13	114	15	4	7	251	0.863049
34	13	64	187	13	114	15	4	7	255	0.868217

Table D3: All the DES_03 Test Vector Results

8. References

Reference	Full Reference Title
[Adrion et al 82]	Adrion, W, et al, Validation, Verification and Testing of Computer Software, Computer Surveys, Vol 14, No 2, June 1982.
[Al-Khanjari et al 02]	Al-Khanjari, Z, Woodward, M, Ramadhan, H, Critical Analysis of the PIE Testability Technique, Software Quality Journey, 10, p. 331-354, 2002.
[Andersson et al 03]	Andersson, C, Theline, T, Runeson, P, Dzamashvilli, N, An Experimental Evaluation of Inspection and Testing for Detection of Design Faults, Proc IEEE/ACM International Symposium Empirical Software Engineering, IEEE CS Press 2003.
[Appropriate Accounts 1989-89 Vol 1:Close 1- MoD]	Appropriation Accounts 1998-99, Volume 1: Class I, Ministry of Defence, National Audit Office, Report of the Comptroller and Auditor General. http://www.nao.gov.uk/publications/nao_reports/990011i.pdf
[Araki et al 91]	Araki, K, Furukawa, Z, Cheng, J, A General Framework for Debugging, IEEE Software, vol. 8, no. 3, May 1991.
[Arora 95]	Arora, V, Kalaichelvan, K, Goel, N, Munikoti, R, Measuring High-Level Design Complexity of Real-Time Object-Oriented Systems, Proc Annual Oregon Workshop on Software Metrics, Silver Falls, OR, June 1995.
[Bache & Bazzana 94]	Bache, R, Bazzana, G, Software Metrics for Product Assessment, McGraw-Hill, 1994,
[Baker & Habli 13]	Baker, R, Habli, I, An Empirical Evaluation of Mutation Testing for Improving the Test Quality of Safety Critical Software, IEEE Transactions on Software Engineering, Vol 39, No 6, June 2013.
[Basili & Selby 87]	Basili, V, Selby, R, Comparing the Effectiveness of Software Testing Strategies, IEEE Trans on Software Engineering, Vol SE-13, No 12, December 1987.
[Beizer 84]	Beizer, B, Software System Testing and Quality Assurance, Van Nostrand Reinhold, 1984.
[Beizer 90]	Beizer, B, eds, Software Testing Techniques, Thomson Computer Press, 1990.
[Beizer 95]	Beizer, B, Black Box Testing: Techniques for Functional Testing of Software and Systems, Wiley, 1995.

[Berling & Thelin 03] Berling, T, Thelin, T, An Industrial Case Study of the Verification and Validation Activities, IEEE Symposium on Software Metrics, Sept 2003.

[Berry & Boudol 98] Berry, G, Boudol, G, The Chemical Abstract Machine, INRIA, 26 October, 1998.

www-sop.inria.fr/meije/personnel/Gerard.berry/cham.ps.

[Bertolino 07] Bertolino, A, Software Testing Research: Achievements, Challenges, Dreams, Future of Software Engineering (FOSE 07), IEEE, 2007.

[Bertolino 96] Bertolino, A, Strigini, L, On the Use of Testability Measures for Dependability Assessment. IEEE Trans. Software Eng. 22, 1996.

[Bertolino et al 13] Bertolino, A, Daoudagh, S, Lonetti, F and Marchetti, E, XACMUT: XACML 2.0 Mutants Generator, ICST 2013.

[Binder 00] Binder, R, Testing Object-Oriented Systems: Models, Patterns and Tools, Addison-Wesley, 2000.

[Boehm 88] Boehm, B, A Spiral Model of Software Development and Enhancement, IEEE Computer May 1998.

[Boehm et al 78] Boehm, B, et al, Characteristics of Software Quality, TRW, North-Holland Publishing Company, 1978.

[Bosch 00] Bosch, J, Design and Use of Software Architecture, Addison-Wesley, 2000.

[Brooks 87] Brooks, F, No Silver Bullet: Essence and Accidents of Software Engineering, IEEE Computer, April 1987.

[Budd 80] Budd, T. A, Mutation analysis of program test data. Ph.D. thesis, Yale Univ., New Haven, Corm, 1980.

[Butler & Finelli 96] Butler, R, Finelli, G, The infeasibility of Quantifying the reliability of life Critical Real time software, 1996.

[Coley 99] Coley, D, An introduction to Genetic Algorithms for Scientists and Engineers, World Scientific, 1999.

[Collins 01] Concise Dictionary & Thesaurus, Collins, 2001.

[Corman et al 2009] Corman, T, Leiserson, C, Rivest, R, Stein, C, Introduction to Algorithms, Massachusetts Institute of Technology, 2009.

[Corne et al 99] Corne, D, Dorigo, M, Glover, F, New Ideas in Optimization, McGraw-Hill, 1999.

[Cullyer, et al 91] Cullyer, W, Goodenough, S, Wichmann, B, The choice of computer languages for use in safety critical systems, Software Engineering Journal, March 1991.

[Davies 93] Davis, A, Software Requirements, Prentice Hall, 1993.

[Def Stan 00-55 Issue 2] Defence Standard 00-55, Requirements for Safety Related Software in Defence Equipment, Part 1: Requirements, Issue 2, 1 August 1997.

[Def Stan 00-56 Issue 4] Defence Standard 00-56, Safety Management Requirements for Defence Systems, Part 1: Requirements, Issue 4, 1 June 2007.

[Def Stan 05-95] Defence Standard 05-95, Quality System Requirements For The Design, Development, Supply and Maintenance of Software, Issue 3, 23 June 1995.

[Delamaro 01] Delamaro, M, Interface Mutation: An Approach for Integration Testing, IEEE Software Engineering, Vol 27, No 3, March 2001.

[Demillo et al 87] Demillo, et al, Software Testing and Evaluation, The Benjamin/Cummings Publishing Company, 1987.

[Demillo et al 78] Demillo, R. A, Lipton, R.J, and Sayward, F.G, Hints on test data selection: Help for the practising programmer. *Computer 11*, (April), 34–41, 1978.

[Deming 86] Deming, W.E., "Out of the Crisis", MIT Center for Advanced Engineering Study, Cambridge, Mass. 1986.

[Deng et al 13] Deng, L, Offutt, J, Li, N, Empirical Evaluation of the Statement Deletion Mutation Operator, ICST 2013.

[178B] Software Consideration in Airborne Systems and Equipment Certification, RTCA DO-178B, 1 December 1992.

[178C] Software Considerations in Airborne Systems and Equipment Certification, RTCA DO-178C, 13 December, 2011.

[DOD 2167A] DOD – STD – 2167A, Defense System Software Development, 29 Feb 1988.

[Dunn & Ullman 82] Dunn, R, Ullman, R, Quality Assurance for Computer Software, McGraw – Hill, 1982.

[Dustin et al 1999] Dustin, E, Rashka, J, Paul, J, Automated Software Testing: Introduction, Management and Performance, 1999.

[Ellims et al 06] Ellims, M, Bridges, J and Ince, D, The Economics of Unit Testing, Springer Science + Business Media, Inc. 2006.

[EN-61508] Functional Safety of Electrical/Electronic Programmable Electronic Safety-Related Systems-Part 3: Software Requirements, BS EN 61508-3:2002.

[Fenton & Ohlsson 00] Fenton, N, Ohlsson, N, Quantitative Analysis of Faults and Failures in a Complex Software System, IEEE Transactions on Software Engineering, Vol 26, No 8, August 2000.

[Fenton & Pfleeger 97] Fenton, N, Pfleeger, Software Metrics, PWS Publishing Company, 1996.

[Frankl et al 98] Frankl, et al, Evaluating Testing Methods by Delivered Reliability, 1998.

[Friedman 95] Friedman, M, Voas, J, Software Assessment, John Wiley & Sons, Inc, 1995.

[Gacek & Boehm 98] Gacek, C, Boehm, B, Composing Components: How does one detect potential architecture mismatches, Proceeding of the OMG-DARPA-MCC Workshop on Compositional Software Architectures, Los Angles, January 1998.

[GAO 83] Greater Emphases On Testing Needed To Make Computer Software More Reliable And Less Costly, GAO/IMTEC-84-2, Government Accounting Office, Washington 1983.

[Gardiner et al 99] Gradiner, S, et al, Testing Safety-Related Software: A Practical Handbook, Springer, 1999.

[Gelperin & Hetzel 88] Gelperin, D, Hetzel, B, The Growth of Software Testing, Vol 3, No 6, Communication of the ACM, June 1988.

[Gilb 77] Gilb, T, Software Metrics, Winthrop, 1977.

[Goldberg 89] Goldberg, D, Genetic Algorithms, Addison-Wesley, 1989.

[Gourley 83] Gourlay, J, A Mathematical Framework for the Investigation of Testing, IEEE Transactions on Software Engineering, Vol SE-9, No 6, November 1983.

[Hadley & Clark 13] Hadley, M, Clark, J, Good Days and Bad Days: Investigating Effectiveness and Reliability of “Optimum” Test Sets, Proceedings of the ICST 2013.

[Hadley 08] Hadley, M, Software Testing Literature Review, York University, 2008.

[Hamlet & Voas 93] Hamlet, D, Voas, J, Faults on Its Sleeve: Amplifying Software Reliability Testing, ACM 1993.

[Hamlet 90] Hamlet, D, Taylor, R, Partition Testing Does Not Inspire Confidence, IEEE Transactions on Software Engineering, Vol 16, No 12, December 1990.

[Harrold 00] Harrold, M, Testing: A Roadmap, 22nd International Conference on Software Engineering, June 2000.

[Harrold et al 08] Harrold, M, Jones, J, Yu, Yanbing, An Empirical Study of the effects of Test-Suites Reduction on Fault Localization, ICSE 2008, Leipzip, Germany.

[Harrold et al 93] Harrold, M, Gupta, R, Softa, M, A Methodology for controlling the size of a Test Suite, ACM Transactions on Software Engineering and Methodology, Vol 2, No 3, July 1993.

[Hatzel 88] Hetzel, B, 2nd edition, The Complete Guide to Software Testing, Wiley-QED, 1988.

[Helmet 77] Hamlet, R, Testing Programs with the Aid of a Compiler, IEEE Transactions on Software Engineering, Vol.SE-3, No 4, July 1977.

[Herbsleb 94] Herbsleb, J, Carleton, A, Rozum, J, Siegel, J, Zubrow, D, Benefits of CMM-Based Software Process Improvement: Executive Summary of Initial Results, SPECIAL REPORT CMU/SEI-94-SR-013, September 1994.

www.sei.cmu.edu/pub/documents/94.reports/pdf/sr13.94.pdf

[Herzner et al 05] Herzner, W, et al, Comparing Software Measures with Fault Counts Derived from Unit Testing of Safety Critical Software, SAFECOMP 2005, LNCS 3688, PP 81-95, 2005.

[Howden 76] Howden, W. E. 1976. Reliability of the path analysis testing strategy. IEEE Trans. Software Eng. SE-2, (Sept.), 208–215.

[Howden 78] Howden, W, Theoretical and Empirical Studies of Program Testing, IEEE Transactions on Software Engineering, Vol.SE-4, No 4, July 1978.

[Howden 81] Howden, W, Errors in data processing and the refinement of current program test methodologies, NBS, Washington D.C, July 1981.

[Howden 81b] Howden, W, Errors in data processing programs and the refinement of current program test methodologies, National Bureau of Standards, Washington DC, July 1981.

[Howden 82] Howden, W, Weak Mutation Testing and Completeness of Test Sets, IEEE 1982.

[IEEE 1044] IEEE Guide to Classification for Software Anomalies, IEEE Std 1044.1-1995.

[IEEE Std 610.12-1990] IEEE Standard Glossary of Software Engineering Terminology, ANSI/IEEE Std 610.12-1990.

[IEEE Std 829-1983] IEEE Standard for Software Test Documentation, 1983.

[IEEE Std 982.1 1988] IEEE std 982.1-1988, IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE 1989.

[IEEE Std 982.1 2005] IEEE Std 982.1 - 2005 IEEE Standard Dictionary of Measures of the Software Aspects of Dependability, IEEE 2006.

[Jalote & Murphy 04] Jalote, P, Murphy, B, Relaibility Growth in Software Products, ISSRE, IEEE, 2004.

[Jin & Offutt 01] Jin, Z, Offutt, J, Deriving Tests From Software Architectures, 12 IEEE International Symposium On Software Reliability Engineering, November 2001.

[Jin & Offutt 98] Jin, Z, Offutt, J, Coupling-Based Criteria for Integration Testing, Department of Information and Software Engineering, July 1998.

[Jones & Harrold 03] Jones, J, Harrold, M, Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. IEEE Transactions on Software Engineering, Volume 29, Number 3, March 2003.

[Offutt et al 93] Offutt, A. J, Rothermel, G, Zapf, C. 1993. An experimental evaluation of selective mutation. In *Proceedings of 15th ICSE* (May), 100–107.

[Jorgensen & Erickson 94] Jorgensen, P, Erickson, C, object-Oriented Integration Testing, Communication of the ACM, Vol 37, No. 9, September, 1994.

[Juran 88] Juran, J.M., "Juran on Planning for Quality" The Free Press, New York, 1988.

[Jursito & Vegas 03] Juristo, N, Vegas, S, 2003, Functional Testing, Structural Testing and Code Reading: What Fault Types Do They Each Detect?

[Kamsties & Lott 95] Kamsties, E, Lott, C, An Empirical Evaluation of Three defect detection techniques, 1995.

[Kaner 00] Kaner, C, Measurement of the Extent of Testing, Pacific Northwest Software Quality Conference, October 1991.

[Kaner 93] Kaner, C, Falk, J, Nguyen, H, 2nd edition, Testing Computer Software, Van Nostrand Reinhold, 1993.

[Kazman 00] Kazman, Klein, Designing and Analysing Software Architectures Using ABASs, ICSE 2000.

[Killer & Miller 91] Keiller, P, Miller, D, On the use and the performance of software reliability growth models, Reliability Engineering and System Safety, pp 95-117, 1991.

[Laitenberger 98] Laitenberger, O, Studying the Effects of Code Inspection and Structural Testing on Software Quality, 9th IEEE Software Reliability Engineering International Symposium, Nov 1998.

[Lauterbach & Randall 89] Lauterbach, L, Randall, Experimental Evaluating of Six Test Techniques, Computer Assurance, 1989. COMPASS '89, 'Systems Integrity, Software Safety and Process Security', Proceedings of the Fourth Annual Conference on 19-23 June 1989 Page(s):36 – 41.

[Lipton 78] Lipton, R, Sayward, F, The status of research on program mutation, Digest for the Workshop on Software Testing and Test Documentation, Fort Lauderdale, Florida, pp. 355-372, Dec. 1978.

[Littlewood & Strigini 92] Littlewood, B, Strigini, L, Validation of Ultra-High Dependability for Software –based system, ACM 1992.

[Littlewood 2011] Littlewood, B. & Strigini, L. (2011). "Validation of ultra-high dependability…" – 20 years on. Safety Systems, the Newsletter of the Safety-Critical Systems Club, 2011.

[Lutz 92] Lutz, R, Analysing Software Requirement Errors in Safety-Critical Embedded Systems, IEEE Software 1992.

[Marick 00] Marick, B, 1990, Two Experiments in Software Testing, Technical Report UIUCDCS-R-90–1644, University of Illinois.

[Marick 91] Marick, B, The Weak Mutation Hypothesis, ACM, 1991.

[Marick 95] Marick, B, The Craft of Software Testing, Prentice Hall, 1995.

[Mat 91] Mathur, A, Performance, effectiveness, and reliability issues in software testing. In Proceedings of the Fifteenth Annual International Computer Software and Applications Conference, pages 604-605, Tokyo, Japan, September

1991.

[Mazza et al 96] Mazza, Fairclough, Melton et al, Software Engineering Guides, Prentice Hall, 1996.

[McCabe & Butler 89] McCabe, T, Butler, C, Design Complexity Measurement and Testing, Communications of the ACM, December 1989, Vol 12, No 12.

[Melton 96] Melton, A, Software Measurement, International Thomson Computer Press, 1996.

[Michalewicz & Fogel 04] Michalewicz, Z, Fogel, D, How to Solve It: Modern Heuristics, Springer, 2004.

[Mills 72] Mills, H, On the statistical validation of computer programs. Technical report FSC 72-6015, IBM, 1972.

[MIL-STD 498] Software Development and Documentation, MIL STD 498, 5 December 1994.

[Misherghi & Su 06] Misherghi, G and Su, Z. HDD: hierarchical delta debugging, In Proc. 28th Int. Conf. on Software Engineering (ICSE 2006), pages 142–151, Shanghai, China, 2006. ACM Press.

[MISRA C] MISRA, Guidelines for the use of the C language in vehicle based software, April 1998.

[Musa 87] Musa, J et al, Software Reliability, McGraw-Hill, 1987.

[Musa 89] Musa, J, Ackerman, A, Quantifying Software Validation: When to Stop Testing?, IEEE 1989.

[Musa 99] Musa, J, Software Reliability Engineering: More Reliable Software: Faster Development and Testing, McGraw-Hill, 1999.

[Myers 78] Myers, G, A controlled Experiment in Program Testing and Code Walkthroughs/Inspection, Communications of ACM, 1978.

[Myers 79] Myers, G, The Art of Software Testing, Wiley & Sons, 1979.

[Nair et al 98] Niar, V, James, D, Ehrlich, W, Zevallos, J, A statistical assessment of some software testing strategies and application of experimental design techniques, Statistica Sinica 8, 1998.

[Nakajo & Kume 91] Nakajo, T, Kume, H, A Case History Analysis of Software Error Cause-Effect Relationship, IEEE Transactions on Software Engineering, Vol 17, No 8,

August 1991.

[Ng, et al 04] Ng, S.P, et al, A Preliminary Survey on Software Testing Practices in Australia, Proceedings of the 2004 Australian Software Engineering Conference, 2004.

[Offutt 89] Offutt, J, The Coupling Effect: Fact or Fiction?, ACM, 1989.

[Offutt 92] Offutt, J, Investigation of the Software Testing Coupling Effect, ACM Transactions on Software Engineering and Methodology, Vol 1, No 1, Jan 92, P.5-20.

[Offutt et al 93] Offutt, J, Rothermel, G, Zapf, C, An Experimental Evaluation of Selective Mutation, IEEE, 1993.

[Offutt et al 93] Offutt, A. J., RothermeL, G., and Zapf, C. 1993. An experimental evaluation of selective mutation. In *Proceedings of 15th ICSE* (May), 100–107.

[Offutt et al 95] Offutt, A, Pan, J, Voas, J, Procedures for reducing the size of coverage based test sets, Twelfth International Conference on Testing Computer Science, June 1995, Washington DC.

[Parliamentary Memorandum Appendix 8] Appendix 8, Memorandum from the Ministry of Defence on Major Procurement Project Survey (March 2002)

www.publications.parliament.uk/pa/cm200102/cmselect/cmdfence/779/779ap01.htm.

[Perry 00] Perry, W, Effective Methods for Software Testing, Wiley, 2000.

[Perry 87] Perry, D, Evangelist, W, An Empirical study of software interface faults, AT & T Bell Laboratories, 1987.

[Poulding et al 13] Poulding, S, Clark, J, Alexender, R, Hadley, M, The Optimisation of Stochastic Grammars to Enable Cost-Effective Probabilistic Structural Testing, GECCO 2013.

[Press et al 92] Press, W, Teukosky, S, Vetterling, W, Flannery, B, eds, Numerical Recipes, Numerical Recipes in C, The Art of Scientific Computing, Cambridge, 1992.

[Richardson & Wolf 96] Richardson D, Wolf, A, Software Testing at the architecture Level, Proceeding of the second international software architecture workshop, San Francisco, October 1996.

[Runeson & Andrew 03] Runeson, P, Andrews, A, Detection or Isolation of Defects? An Experimental Comparison of Unit Testing

and Code Inspection, IEEE 2003.

[Runesson et al 06] Runeson, P, Andersson, C, Thelin, T, Andrews A, Berling, T, What Do We Know about Defect Detection Methods?, IEEE SOFTWARE, 2006.

[Selby & Basili 87] Basili, V, Selby, R, Comparing the effectiveness of software testing strategies, IEEE Transactions on Software Engineering, v.13 n.12, p.1278-1296, December 1, 1987.

[Selby 86] Selby, R, Combining software testing strategies: An empirical evaluation. In Proc. Workshop on Software Testing, pages 82-91. IEEE Computer Society Press, July 1986.

[Selby 90] Selby, 1990, Empirically Based Analysis of Failures in Software Systems.

[Smith & Wood 87] Smith, Dm Wood, K, Engineering Quality Software, Elsevier Applied Science Publishers, 1987.

[So et al 02] So, S, et al, An Empirical Evaluation of Six Methods to Detect Faults in Software, Software Testing, Verification, and Reliability, Vol 12, No 3, 2002.

[Sommerville 96] Sommerville, I, 5 edition, Software Engineering, Addison-Wesley, 1996.

[Storey 96] Storey, N, Safety-Critical Computer Systems, Addison-Wesley, 1996.

[Telles and Hsieh 01] Telles, M, Hsieh, Y, The Science of Debugging, Coriolis Group Books, 2001.

[Testing Glossary 04] Glossary of terms used in Software Testing, Version 1.0 (dd. December, 8th 2004), Produced by the 'Glossary Working Party' International Software Testing Qualification Board.

[Thelin & Berling 03] Berling, T, Thelin, T, An Industrial Case Study of the Verification and Validation Activities. IEEE METRICS 2003.

[UK Parliament, Defence Offet Obligations] The UK Parliament, Defence Offet Obligations – Ref 69841.

http://www.parliament.uk/index.cfm.

[Voas & McGraw 98] Voas, M, McCraw, G, Software Fault Injection, John Wiley & Sons, 1998.

[Voas 03] https://www.softwaretechnews.com/stn_view.php?stn_id=

8&article_id=62

[Voas 91 et al] Voas, J, Morell, L, Miller, K, Predicting Where Faults Can Hide from Testing, IEEE Software, March, 1991.

[Voas 92] Voas, J, PIE: A Dynamic Failure Based Techqniue, IEEE Trans on Software Engineering, Vol 18, No 8, Aug 1992.

[Walsh 79] Walsh, T. J. A reliability study using a complexity measure. In AFIPS Conference Proceedings (New York, NY, 1979), AFIPS Press.

[Weyuker & Ostrand 80] Weyuker, E, Ostrand, T, Theories of program testing and the application of revealing sub-domains, IEEE Trans. Software Eng, SE-6, 3 May 1980.

[Woods et al 97] Wood, M, Roper, M, Brooks, A, Miller, J, Comparing and Combining Software Defect Detection Techniques: A Replicated Empirical Study, 1997.

[Woodward & Halewood 88] Woodward, M, Halewood, K, From weak to strong, dead or alive? An analysis of some mutation testing issues, In Proceedings of the Second Workshop on Software Testing, Verification,and Analysis, pages 152- 158, Banff Alberta, July 1988. IEEE Computer Society Press.

[Woodcock et al 06] Woodcock, J, Larsen, P, Bicarregui, J, Fitzgerald, J. ACM Computing Surveys, Vol. 41, No. 4, October 2009.

[Zeller & Hildebrandt 02] Zeller, A and Hildebrandt, R, Simplifying and isolating failure inducing input. Software Engineering, 28(2):183–200, 2002.

[Zhu et al 97] Zhu, H, et al, Software Unit Test Coverage and Adequacy, ACM Computing Surveys, Vol.29, No.4, December 1997.