Software Reliability Enhancement through Error-Prone Path Identification using Genetic Algorithms

James Birt, BIT (Honours)

Research thesis submitted in fulfilment of the requirements for the degree of

Doctor of Philosophy

18th May 2006

Principal supervisor: Dr. Renate Sitte
Co-supervisor: Dr Ian Graham
ABSTRACT

This thesis presents a pre-processing stage for optimising software reliability by quantitatively identifying the most error prone regions in a software program. These error prone regions are identified using Genetic Algorithms on the source code’s graph representation, weighted with potential Sources of Error. By identifying these potentially error dense regions, the efficiency of the software quality activities can be increased. The information about quantitative error proneness can be used for more accurate effort and cost estimations of quality assurance.

Although various methods have been applied for detecting and reducing errors in software, little research has been done into partitioning a system into smaller, error prone domains for a more targeted Software Quality Assurance. To identify error proneness in software regions is important as these domains can be given priority in code inspections or testing.

Quality activities come at a high price, typically requiring more than half of the project resources to produce a working program. However, a working program does not necessarily mean a defect free program. Exhaustive software testing is rarely possible because it becomes intractable for even medium sized software. Inspections require experts; they can be subjective and expensive. Typically due to project budget constraints only parts of a program can be tested or inspected, but these parts are not necessarily the most error prone. A more effective approach is to focus inspection and testing efforts on those regions that are most likely to contain faults, that is, the most error prone regions.

The strategic approach presented in this thesis consists in parsing a software source code and attributing weights to software’s paths using a method for assessing quantitatively the error proneness of software modules. By representing these paths as a weighted connectivity matrix, a Genetic Algorithm is applied to the paths with a strategy of finding a selection of paths with maximum weights as potential error carriers. These maximum
error prone paths can then be selected for priority in testing and inspection. The approach does not deal with the actual inspection, testing or test cases per se, but it makes an informed choice on where to focus the main effort possible. This in turn aids project management by eliminating the guesswork of where to focus the effort and budget for quality assurance activities.

The technique presented in this thesis is supported by a set of experiments: (i) empirical analysis of Genetic Algorithm variables and their effect on performance; (ii) Pareto analysis using error seeding identification with best fit, random and clustered approaches; (iii) segmenting path strata and identifying error prone regions in the path (iv) comparison with traditional software inspection.

Results from the experiments conducted in the thesis support the proposed technique, through error identification rates greater than 85% from only 20% of the most error prone code. This is a strong result as it fits with the Pareto analysis or the 80/20 rule as a standard analysis technique.
I, James Birt, state that this work has not previously been submitted for a degree or diploma in any university. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made in the thesis itself.

James Birt

The framework described in Chapter 2 was presented at the Asia Pacific (IEEE) Conference on Quality Software APAQS'2000, Hong Kong, October 2000. For quick reference, this paper is attached in Appendix A.

James Birt
Candidate

Dr. Renate Sitte
Principal Supervisor
ACKNOWLEDGEMENTS

It is a pleasure to thank the many people who made this thesis possible.

It is difficult to overstate my gratitude to my PhD supervisor, Dr. Renate Site. Throughout my thesis, she provided encouragement, sound advice, good teaching, good company, and lots of great ideas. Her enthusiasm, inspiration and great efforts to explain things clearly helped me achieve this thesis.

I am indebted to my many colleagues for providing a stimulating and fun environment in which to learn and grow. I am especially grateful to Dr Michael Cowling, Dr Ken Udono and Dr Danny Powell.

I wish to thank my entire family for providing a loving and encouraging environment for me. My parents and sisters were all incredibly supportive throughout the entire journey.

To my family I dedicate this thesis.
# TABLE OF CONTENTS

1. Introduction .................................................................................................................. 1
   1.1 Background ........................................................................................................ 1
   1.2 Problem Definition .......................................................................................... 4
   1.3 Aims .................................................................................................................. 5
   1.4 Hypotheses ....................................................................................................... 7
   1.5 Domain and Scope .......................................................................................... 8
   1.6 Research Contribution .................................................................................. 10
   1.7 Thesis Structure ............................................................................................ 11
   1.8 Chapter Summary ......................................................................................... 13

2. Literature Review ....................................................................................................... 14
   2.1 Software Reliability ....................................................................................... 14
      2.1.1 Software Reliability Models .................................................................. 17
      2.1.2 Software Test Coverage ....................................................................... 34
      2.1.3 Software Metrics and Inspection ......................................................... 48
      2.1.4 Error Seeding and Clustering ............................................................... 53
      2.1.5 Quantifying Error Proneness ................................................................. 58
   2.2 Searching Algorithms ...................................................................................... 62
      2.2.1 Preparation of the Search Environment .............................................. 63
      2.2.2 Simple Search Techniques ................................................................... 65
      2.2.3 Local Search ......................................................................................... 68
      2.2.4 Genetic Algorithms ............................................................................... 71
      2.2.5 Problems with Search Methods ............................................................ 86
   2.3 Chapter Summary ............................................................................................. 87

3. Methodology and Experimentation ............................................................................. 89
   3.1 Tools and Development Environment ........................................................... 90
   3.2 Problem Solving Approach ......................................................................... 104
      3.2.1 Genetic Algorithm Strategies and Search Algorithm Comparison .... 106
      3.2.2 Error Seeding ....................................................................................... 112
      3.2.3 Path Strata ......................................................................................... 117
      3.2.4 Inspection ............................................................................................ 119
   3.3 Experiment Selection and Evaluation ............................................................ 120
      3.3.1 Search Experiment Selection and Evaluation .................................... 120
      3.3.2 Reliability Experiment Selection and Evaluation ......................... 122
   3.4 Chapter Summary ........................................................................................... 125

4. Results and Discussion .............................................................................................. 126
   4.1 Search Algorithm Comparison ....................................................................... 127
   4.2 GA Strategy Comparison .............................................................................. 132
   4.3 SOE Inspection ............................................................................................... 137
   4.4 Error Seeding ................................................................................................. 139
   4.5 Path Strata ..................................................................................................... 145
   4.6 Chapter Summary ........................................................................................... 150

5. Conclusions ............................................................................................................... 151
   5.1 Strengths of this Research ............................................................................ 153
5.2 Weakness of this Research ................................................................. 153
5.3 Future Work ..................................................................................... 155
5.4 Summary of the Thesis and Conclusion .............................................. 156
6. References .......................................................................................... R-1
7. Appendix A Paper ................................................................................ A-1
8. Appendix B SOE Software Inspection Exercise ...................................... B-1
9. Appendix C SOE Software Inspection Exercise Ethics Clearance Email .... C-1
LIST OF FIGURES

Figure 1: Remaining thesis chapter contents ............................................................ 12
Figure 2: Ariane 5 disaster takeoff and explosion ..................................................... 15
Figure 3: Software reliability engineering process .................................................... 16
Figure 4: Reliability growth feedback model ............................................................ 20
Figure 5: Example of decreasing failure intensity over time ..................................... 25
Figure 6: Example of increasing reliability growth over time .................................. 25
Figure 7: Reliability growth model checklist ............................................................ 27
Figure 8: The effect of pre-processing and recalibration on reliability models ......... 31
Figure 9: Increasing coverage growth with respect to increased testing effort .......... 32
Figure 10: Example of combinatorial explosion of paths due to loops ...................... 37
Figure 11: Software coverage techniques .................................................................. 38
Figure 12: Subsumption relationship of different testing coverage criteria ............... 43
Figure 13: Prioritisation of APFD ............................................................................ 47
Figure 14: Defect density class results with respect to module size ......................... 50
Figure 15: Project and inspection effort vs. project time ............................................ 52
Figure 16: SOE analysis of a simple loop .................................................................. 60
Figure 17: Sequence of steps for generating the search environment ......................... 64
Figure 18: Breadth first search strategy ..................................................................... 66
Figure 19: Depth first search strategy ....................................................................... 66
Figure 20: Local search tree showing potential for local minima .............................. 70
Figure 21: Search tree with zero/one string representation ......................................... 75
Figure 22: Roulette wheel selection ......................................................................... 76
Figure 23: Rank selection ....................................................................................... 78
Figure 24: Evolutionary testing ............................................................................... 85
Figure 25: Methodology structure ........................................................................... 91
Figure 26: Source list showing function name and attached LOC vector ................... 93
Figure 27: Expanded source list and LOC vectors ................................................... 94
Figure 28: SOE vector addition to the list and LOC vector structure ......................... 96
Figure 29: Expanded source list and SOE vector ..................................................... 97
Figure 30: Adding structure tree to source list ....................................................... 99
Figure 31: Variable length representation and crossover ........................................ 102
Figure 32: Search comparison of GA, and search algorithms in small environments ... 128
Figure 33: Time comparison of DFS, GA and RGLS in small search environments .... 129
Figure 34: Comparison of the effectiveness of different search algorithms .............. 130
Figure 35: Effect on SOE fitness by increasing crossover probability ...................... 133
Figure 36: Effect on SOE fitness by decreasing mutation probability ....................... 134
Figure 37: Effect on SOE fitness using roulette and tournament selection ............... 135
Figure 38: Effect on SOE fitness by examining three crossover types ..................... 136
Figure 39: Effect on SOE fitness with increasing population size ............................ 137
Figure 40: GA performance: effect of increasing lines of code and nesting depth ...... 141
Figure 41: GA performance: effect of increasing branching and nesting depth .......... 141
Figure 42: Comparison of GA and RP in identifying seeded errors ......................... 142
Figure 43: Comparison of GA and RP in identifying best fit seeded errors ............. 143
Figure 44: Comparison of GA and RP in identifying clustered seeded errors .......... 144
Figure 45: Comparison of GA and RP in identifying errors in top path layer .......... 146
Figure 46: Comparison of GA and RP in identifying errors in middle path layer ....... 147
Figure 47: Comparison of GA and RP in identifying errors in bottom path layer ....... 148
LIST OF EQUATIONS

Equation 1 .......................................................................................................................... 20
Equation 2 .......................................................................................................................... 21
Equation 3 .......................................................................................................................... 21
Equation 4 .......................................................................................................................... 22
Equation 5 .......................................................................................................................... 22
Equation 6 .......................................................................................................................... 23
Equation 7 .......................................................................................................................... 23
Equation 8 .......................................................................................................................... 98
Equation 9 .......................................................................................................................... 98
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANN</td>
<td>Artificial Neural Network</td>
</tr>
<tr>
<td>APFD</td>
<td>Average Percentage of Faults Detected</td>
</tr>
<tr>
<td>BFS</td>
<td>Breadth First Search</td>
</tr>
<tr>
<td>DFS</td>
<td>Depth First Search</td>
</tr>
<tr>
<td>GA</td>
<td>Genetic Algorithms</td>
</tr>
<tr>
<td>SOE</td>
<td>Sources of Error</td>
</tr>
<tr>
<td>LOC</td>
<td>Lines of Code</td>
</tr>
<tr>
<td>RP</td>
<td>Random Path Selection</td>
</tr>
<tr>
<td>RRGLS</td>
<td>Random Restart Greedy Local Search</td>
</tr>
<tr>
<td>SQA</td>
<td>Software Quality Assurance</td>
</tr>
<tr>
<td>SRGM</td>
<td>Software Reliability Growth Models</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

This thesis presents a strategic approach to increase software reliability by focusing on Sources of Errors (SOE). The purpose is to provide the conditions for an informed choice for inspection or testing priority. This is achieved in two stages: (a) by attributing weights of potential SOE to the paths in the software, and (b) by finding the most error prone regions in that software, by analysing its SOE weighted graph using Genetic Algorithms (GA). By identifying the most error prone regions one can gain certainty about finding errors and give these regions priority in inspection or testing. It is important to note that the strategic approach presented in this thesis does not deal with the testing, inspection or test cases per se, but that it provides information to systematically reduce wasted effort in inspecting or testing regions where errors are less likely to occur. This eliminates the guesswork of where to put the weight in testing and inspection, because it addresses directly the problem that exhaustive testing is rarely or never possible, and that inspections are costly.

1.1 Background

The past 30 years have seen a huge growth in the size, complexity and criticality of software code development. Consequently, software reliability and its associated costs for achieving better reliability have increased greatly.

Musa [MUS88, MUS04] defines software reliability as “the probability of failure free operation of a computer program for a specified time in a specified environment”. Therefore, applications such as real time software (given their environment and need for failure free operations) require higher software reliability than say a children’s computer game [COO91].
Several methods for measuring software reliability are commonly used. They include software reliability models [RAM82, YAM83, MUS88, BRO90, CAI91, LYU95, WOO96, WOO97, CHE01, MUS04, CHI05a], automated oracles [BIE92, PET98, BAR01] and fault detection [WON99, FEN00]. Malaiya et al. [MAL94, MAL02], proposed a link between the levels of testing coverage (that is, the expectation to detect as many errors as possible with a test case) and the reliability of software.

Testing coverage (also known as code coverage analysis) refers to the level of coverage that a set of test cases provides in a software system. Testing coverage techniques have the limitation that they potentially look at an entire program, but even with reduction methods a large program can still remain intractable for testing [MCC89]. More often than not it depends on the intuition or experience of the tester or inspector in deciding which parts will be examined first, rather than on quantitative information about the likelihood that those parts indeed contain yet undetected errors.

Quality comes at a high price and typically requires more than half of the project resources to produce a working program [BEI90]. However, a program that works, does not always guarantee a defect free program. Depending on the criticality of the program, multiple inspections and different percentages of testing coverage are required. The overriding issue is that budgets are tight and the quality activities are left to the end of the development phase. This implies that a more effective approach for inspection and testing in general is to focus on the regions most likely to reveal faults [BAL99] or the most error prone regions [SIT00]. This in turn allows prioritisation of those regions and budgets proportioned or reviewed based on analysis of those regions.

The intention of the research presented in this thesis is to focus on the regions more likely to reveal faults by attributing weights to the softwares paths using a method for assessing quantitatively the error proneness of software modules proposed by Sitte [SIT00]. This Source of Error (SOE) method works in the following way: By representing the paths in the software as a SOE-weighted connectivity matrix, a search algorithm can then be applied to the paths to assess their weight as potential error carriers. The goal is to
identify those parts that are most error prone, or contribute the most to the overall amount of potential errors. Success is achieved by finding a selection of the heaviest error prone paths. The advantage of using this technique is that it can be automated and optimised.

Finding a path in a large environment can quickly get out of hand even for medium sized software. Therefore, it is necessary to optimise the search algorithm. This is achieved with Genetic Algorithms (GA). GA were developed and formalised by Holland [HOL75]. They were further developed and shown to have wide applicability by Goldberg et al. [GOL89a, GOL89b]. Subsequently GA have been used for a number of software engineering and reliability applications [HUN95, JON98, LIN01, MIC01, DAI03, HAR04, KHO04].

The motivation for the GA approach is based on its heuristics for optimising large and complex search environments, indicative of software code. The different paths in a software program are not all the same length. This poses a problem because traditional GA require fixed length environments to search. To overcome this problem an extension on GA that is variable length GA [BIR04] was adopted using strategies from Goldberg et al. [GOL89b] Messy Genetic Algorithms (Messy GA).

Other methods based on exhaustive or local search such as depth first, greedy algorithm, or simulated annealing [KIR83] could possibly be applied. However, due to the lack of a global sampling capability, local search methods run greater risk of being trapped in local minima. Consequently exhaustive techniques cannot complete in polynomial time. In this case exhaustive techniques are unsuitable for finding the most error prone paths, given that the nature of the search space is rather large with possibly many local minima (non optimal solutions).

The implications and benefits of determining the most error prone regions are many. By identifying the most error prone regions the technique increases the efficiency of inspections and testing, because error prone parts can be given priority in Software Quality Assurance (SQA) activities. At the same time this technique contributes to more
accurate effort and cost estimations which in turn lead to improvements in software reliability.

The remainder of this chapter will discuss the motivation of the thesis, the problem definition, aim, hypothesis, domain and scope and contribution to the research community. It will conclude with an outline for the remaining chapters in the document.

1.2 Problem Definition

This thesis addresses the following problem:

**Problem:** Current SQA techniques rely on expert intuition and focus on the entire software program. However, it is usually infeasible to inspect or test a large software program and expert intuition is not always correct. With current practice there is no guarantee that the tested or inspected parts of software contain many errors. There is not much knowledge of where the errors are located. The highest density of errors could be in the untested or unreviewed parts. Therefore, it is necessary to develop a quantitative approach for pre-processing test and inspection selection that focuses on identifying the most error prone regions to give these regions priority.

**Proposed Problem Solution:** To identify the parts of code with highest likelihood of containing errors in an efficient and quantitative way. To study the performance of variable length Genetic Algorithms for the efficient identification of the cluster(s) of most error prone paths in a software program using an assessment of error proneness for modules based on programming constructs. To enable an informed choice of prioritising these regions for inspection and testing.

Considering the rapid increase in the size, complexity and criticality of software code development, it has become paramount to not only reduce errors, but also identify constructs or clusters of constructs that are more prone to error. By identifying these
constructs it is possible to test and inspect them more successfully or more importantly avoid them entirely in the future.

This can be achieved by developing a system to identify the most error prone parts in a quantitative way, thereby providing the necessary information for more efficient testing and inspection which in turn helps in lowering the costs of SQA. The important issue is to make an informed choice for priority of inspection and testing on regions of potential error proneness which may aid in identifying areas to inspect or test.

1.3 Aims

This section outlines the aims from the research conducted in the thesis, but before doing so, I will briefly summarise what is required.

Two key process steps have to occur to achieve the solution to the problem. Firstly, to apply the assessment of error proneness framework, the software construct must be converted to a sparse connectivity matrix that makes searching for different path weights possible. A sparse representation must be used to save on storage as this is an O(N^2) storage space problem. The search technique must be capable of utilising the SOE framework outlined in work by Sitte [SIT00] and working in large and complex search environments.

Secondly, error prone regions must be identified from industrial software using the SOE error framework and search algorithm. These selected parts must then be analysed to determine the percentage of defects detected. This gives a strong grounding to the research and shows its applicability to commercial applications.
**AIM OF THE RESEARCH:** To develop a pre-processing system to aid in prioritising software quality assurance activities by focusing on the most error prone software clusters by using Genetic Algorithms and the Sources of Error framework in a software module, whose structure has been mapped to a sparse connectivity matrix for the purpose of searching the weighted paths.

To facilitate this research aim the following subgoals must be met:

1. Investigate existing techniques for measuring reliability, testing coverage and inspection. This is to identify previously developed methods for reliability measurement and identify their positive and negative points. This gives a background of information and techniques currently in use and provides details for comparison.

2. Investigate existing techniques for converting a program construct or module into a domain for applying search techniques. This helps identify methods for converting a software module into an environment that allows the application of the GA search technique.

3. Investigate search techniques for identification of error prone paths in a sparse connectivity matrix environment. The aim is to identify search techniques capable of searching for the most error prone paths. Based on these algorithms a comparison can be made to determine which search technique is most successful, or which techniques are more suited to these environments and constraints.

4. Investigate methods to identify smaller error prone regions for testing and inspection. The aim is to improve the testing and inspection process by focusing on smaller regions which contain potentially the most errors. This allows for an informed decision to be made on more effective prioritisation of constructs for inspection and testing.
5. Investigate advanced techniques for possible algorithmic optimisation in finding the most potentially error prone paths (global optimum solutions). The reason is to improve the convergence speed and reduce the local minima syndrome (local optimal solutions).

6. Investigate methods of error seeding for verification of the Genetic Algorithm search technique. Error seeding in conjunction with Pareto Analysis is a strong indicator on the effectiveness of a given reliability technique. If the search technique can be supported with a Pareto Analysis, that is 80% of the errors are identified in 20% of the code, then the research can be deemed a success.

1.4 Hypotheses

Using the aims and problem definition outlined in the previous sections the following two hypotheses have been postulated for this thesis.

1. If suitable variable Length Genetic Algorithms can be designed such that they are able to identify the most error prone path clusters using potential sources of programming errors and error seeding methods, then quality assurance activities can be optimised by focusing on the most error prone regions first.

2. If Pareto analysis (80/20 rule) can be applied using the error seeding methods to the clusters of most error prone paths identified by using variable length Genetic Algorithms and potential sources of programming errors then the proposed technique is a suitable and valid pre-processing technique that provides information for efficient inspection and testing.

It is well known that testing all paths in a large software program becomes intractable for testing and may not be feasible in polynomial time [BIE89, MCC89, BAL96, PET96, WAL00]. Software inspections require all instructions to be addressed at least once.
Typically, due to project budget constraints, only certain parts of a program can be tested or inspected. The parts that are examined should take into consideration the most error-prone or fault-prone portions of the software.

Previous work has shown that it is possible to assess the most error-prone paths in a software program using the assessment of error proneness for modules based on programming constructs [SIT00].

Given the problem domain of a large search environment, GA are a suitable searching technique. The choice of the GA approach is justified by Goldberg et al. and Holland [HOL75, GOL89a, GOL89b] by the following key points:

- Finding optimal or near optimal solution in a large search environment;
- Focusing on a possible solution from a population of points, not a single point;
- Use of probabilistic not deterministic based rules.

Research by many authors [END75, ADA84, BAS84, HAT97, FEN00, CHO01, SHU02] shows that Pareto analysis (80% of defects arise from 20% of the code) is a strong guide to determine effective techniques in identifying errors and in this case of prioritising regions of error proneness for inspection and testing.

1.5 Domain and Scope

This section outlines the scope and constraints of the research and the research working domain.

The goal of this research is to develop a pre-processing system for informed software inspection and test selection. In part using GA that are capable of efficiently finding the clusters of error-prone paths in a software construct. This is achieved by developing GA
using the sources of errors framework developed by Sitte [SIT00] as its criteria for the fittest path (most error prone path) selection.

By using quantitative criteria the developed pre-processing system can identify regions of error proneness prior to testing and inspection. These regions are likely to contain more errors. By applying a Pareto analysis (or the 80/20 rule) the number of test cases or regions inspected should be lowered while achieving equal or greater error detection. This can be used in conjunction with other methods to achieve the desired quality goals of the project.

This domain limits the scope of this thesis to those SOE areas covered by Sitte’s research [SIT00]. This is desirable to comply with the time frame in the PhD candidature. Detecting other types of errors such as infeasible paths would take the research beyond this timeframe, while lesser errors (compilation errors) are meaningless because they would be picked up during compilation anyway.

For this research only procedural C code is examined. This is desirable to comply with the time in the PhD candidature. Examining other programming languages and programming paradigms such as Object Oriented Programming would take the research beyond the given timeframe. To this end 10 sample cases of rather large industrial open source C code have been sourced from www.sourceforge.net, a leading open source project community. The selection of these sample case software comprises a variety of different domains including networking, communication and application software.

Unfortunately due to a currently widespread unwillingness of cooperation with industry due to sensitivity of information I was unable to source industry inspection and test data cases, despite my offering of “Non Disclosure Agreement” implying anonymity and mathematical transformation (normalisation) of data. I have overcome this impasse by conducting an empirical study of the sourced programs by error seeding using a number of error seeding strategies. The goal is to determine how effective the SOE-GA technique is at identifying these seeded errors and how many errors can be identified from 20% of
the most error prone search environment. This allows validation of the effectiveness of the proposed technique by Pareto analysis.

By comparing with existing techniques and existing models and by using open source industry source code this allows the technique to be verified and validated in a commercial domain.

### 1.6 Research Contribution

This section describes the contributions made in the thesis and how they improve on existing knowledge in the research community.

The research in this thesis contributes to research into optimising quality assurance activities by enabling improved testing and inspection efficiency. It examines applications of artificial intelligence to SQA. It makes several contributions to the research community with an approach that has not been attempted previously.

1. This thesis outlines a pre-processing strategy for testing and inspection selection by partitioning a software system into smaller potentially error prone regions. While techniques exist for testing overall systems there is little research into partitioning a system into smaller error prone domains for testing and inspection selection.

2. The development of this partitioning system requires many methods for grouping paths into regions of error proneness and the identification of these groups with search techniques using GA. This thesis provides quantitative data to support different Genetic Algorithm variable strategies for varying sized and structured programs.
3. In the thesis a variable length Genetic Algorithm has been designed to solve the issues associated with fixed length GA chromosomes (paths) in this domain of variable sized code paths.

4. This thesis contributes with quantitative results to support Sitte’s [SIT00] framework for sources of error as a quantitative method of identifying error proneness in software constructs and its use as an optimising fitness criterion for GA searching in software constructs.

5. This research provides a base on which further research into refinement and efficiency of the proposed system and into applications of artificial intelligence in SQA can be built.

1.7 Thesis Structure

The thesis is divided into five parts: introduction, literature review, methodology and experiments, results and discussion and conclusions. Figure 1 provides a summary of the remaining chapters in the thesis.

Chapter 2 Literature Review comprises two main sections. Section one reviews the current techniques that can be used for achieving levels of software reliability. This includes software reliability modelling, test coverage techniques, inspection, metrics, error seeding and error proneness quantification. The second section investigates searching algorithms, methods for quantifying error prone regions and a technique for representing software structure for searching. It discusses three main types of search techniques for identifying the most error prone paths and outlines the issues associated with these techniques.
Figure 1: Remaining thesis chapter contents.

Chapter 3 Methodology and Experiments provides a detailed description of the proposed research methods, techniques, tools, models algorithms and experiments used to achieve the project’s quantitative results.

Chapter 4 Results and Discussion presents the quantitative results of the experiments preformed for the research and provides a detailed analysis and interpretation of the quantitative results.

Chapter 5 Conclusion summarises all the findings and draws the conclusions from the work completed in the thesis. This includes revisiting the aims and hypothesis of the thesis and relating this with the final results and previous research conducted. Strengths, limitations of the research and a plan for future work are included in this section.
1.8 Chapter Summary

This chapter has provided an overview for the research presented in this thesis; it identified the area of focus and summarised the methodology. The research problem was stated with research aims and two hypotheses derived. The domain and scope were outlined including the contributions to the research community. Finally the structure of the thesis was stated.

The following chapter presents the literature review and outlines the major problems and areas of research focus for the thesis.
2. LITERATURE REVIEW

This chapter presents the literature review conducted for the thesis. The literature review is divided into two main sections: software reliability and searching. Software reliability covers the areas of software reliability models, software test coverage, software metrics, software inspection, error seeding and error proneness quantification. Searching examines simple search techniques, local search, GA, messy GA, the search environment preparation, and searching issues in general.

2.1 Software Reliability

This section provides an overview to software reliability and highlights the reasons why software reliability is a necessary tool in modern software development.

Today, software reliability has become one of the most important measurements in the software design process. This emerges from the increased software complexity, software criticality, and famous failures like the Ariane 5 disaster [ESA96] shown in Figure 2.

Musa [MUS88, MUS04] defines reliability as, “the probability of failure-free operation of a computer program for a specified time in a specified environment” and is an important attribute in measuring overall software quality. This concept of reliability forms the back-bone of Software Reliability Engineering. This engineering process is illustrated in Figure 3.

In Figure 3 Musa defines six phases for improved software reliability. The first activity is to define the product. One must establish who the supplier is and who the customers and users are.
Figure 2: Ariane 5 disaster takeoff and explosion. The Ariane 5 software reused the specifications from the Ariana 4, but the Ariane 5's flight path was considerably different. Pre-flight tests had never been performed on the code under simulated Ariane 5 flight conditions, so the error was not discovered before launch [ESA96].

Then one lists all the systems associated with the product that must be tested independently. These are generally of two types: base product and variations. Variations are versions of the base product that one can design for different environments.

The second activity deals with quantifying how the software is used or the software operational profile (system logical task and occurrences). In this activity the development resources are divided among modules for code, code reviews, and unit tests.

The third activity defines the level of reliability for the product. To determine this objective, one analyses the needs and expectations of the software. One can improve the process by collecting metric information that is particular to that environment.
Figure 3: Software reliability engineering process, from defining the product to guiding the testing [MUS04].

The forth activity is test preparation. This uses the operational profiles from step two to prepare test cases and test procedures. One can allocate test cases in accordance with the operational profile.

The fifth activity is test execution. In this activity one can allocate test time among the associated systems and types of test (feature, load, and regression). Identify failures, along with when they occur. The “when” can be with respect to time.
The last activity involves guiding the product’s system test phase and release. For software that one develops, one can track reliability growth as faults are removed. This is then certified which simply involves accepting or rejecting the software in question.

Through the tasks outlined in Figure 3 one can define, prepare, execute and track the software’s reliability. The problem is that due to increased software complexity, huge costs are associated with achieving high levels of reliability. Various methods to measure, improve, and lower the costs of software reliability have been reported. The following sections provide an overview of these methods.

2.1.1 Software Reliability Models

This section describes the fundamentals of software reliability models including how they work, why they are used and problems associated with their use or potential use.

There are various methods used to measure software reliability. One technique introduced in the late 1960s was the software reliability model [RAM82, YAM83, MUS88, BRO90, CAI91, LYU95, WOO96, WOO97, CHE01, MUS04, CHI05a]. The past 30 years has seen a visible proliferation in the number and variations of software reliability models. This increase is due to attempts in understanding the characteristics of how and why software fails and the quantification of software reliability [MUS88, MUS04].

Both static and dynamic software reliability models exist to estimate the expected number of total failures or the number of remaining defects. Static defect estimation models include several types of models. In what follows I present three of the more common models, namely capture-recapture, curve-fitting and experience based.

The capture-recapture model [YAN95, BRI97, BRI98, RUN98] derives its techniques from biology. In biology, capture-recapture studies are used to estimate the size of an
animal population. In doing so, animals are captured, marked, and then released on several occasions. The number of marked animals that are recaptured allows the researchers to estimate the total population size based on the samples’ overlap. When many marked animals are recaptured, one can argue that the total population size is small.

The capture-recapture principle in biology can be transferred to software inspections: each inspector draws a sample from the population of defects in the inspected software artifact. In this way, an inspector is equivalent to a particular trapping occasion in biology. A defect discovered by one inspector and rediscovered by another is said to be recaptured.

The second type examined is curve-fitting methods [BRI98, WOH98]. Software engineers and test analysts often want to represent empirical data using a model based on mathematical equations. With the correct model and calculus, one can determine important characteristics of the data, such as the rate of change anywhere on the curve, the local minimum and maximum points of the function, and the area under the curve. The goal of data (or curve) fitting is to find the parameter values that most closely match the data. The models to which data are fitted depend on adjustable parameters.

There are general approaches for curve fitting that can be distinguished from each other on the basis of the amount of error associated with the data. First, where the data exhibits a significant degree of error or noise, the strategy consists in deriving a single curve that represents the general trend of the data. This is done because any individual data point may be incorrect and affected by noise. Second, where the data is known to be very precise, the basic approach is to fit a curve of a family of curves that pass directly through each of the points. Using this method an analyst can fit known data and obtain an understanding of the current error trend.

The final type examined is the experience based methods [YU88, BRI97, NEG04]. In this type of model historical information or experience is used to predict outcomes. This method is useful when historical information exists. The issue is that many companies do
not track previous projects with sufficient detail and the historical information is based more on expert-experience and not on quantitative detail.

All three static models examined use software metrics, like complexity metrics and inspection results to estimate the number of defects in a software program. In contrast, there are the dynamic models which model software failure with respect to time.

The most cited models shown in the literature are the time domain (dynamic) models known as Software Reliability Growth Models (SRGM) [LYU95, CHE96, CHE97, CHE01, MUS04, CHI05a]. SRGM represent the positive improvement of software reliability over time, accomplished through the systematic removal of software faults. The rate at which the reliability grows depends on how fast faults can be uncovered and removed.

The advantage of the SRGM over their static counterparts is that they provide early estimates of the software development duration and levels of reliability achieved dynamically. This gives project managers a tool for tracking the reliability progress of the software under development. This dynamic approach is based on software failure data collected early in the system test phase. These models can predict general trends of reliability improvements during that phase based on an operational profile [MUS88, LYU95].

SRGM work by using a repetitive cycle of testing and repairing a program. Inputs are presented to a program until it fails. When it fails that failure (or the time until it occurred) is recorded. The failure is then fixed and the process continues. The main assumption is that by removing an error, no further errors are introduced, albeit that this assumption is a weak one. This process is shown in Figure 4, which emphasises the iterative approach of this process.
Figure 4: Reliability growth feedback model: inputs are presented, problems are identified and failures and times between failures are produced.

The result from this process is the collection of data such as the sequence of failures and times between failures. The goal and challenge is to construct a model to predict the reliability growth based on this information. The results are often dependent on the data type and the application domain of the software. Therefore a range of models has been derived. The following examples illustrate the more popular models.

Trachtenberg [TRA90] presents a framework for general formulation of dynamic software reliability growth models with the relationship of his general model to major existing models. In the Trachtenberg model, failure rate is a function of software errors encountered. He considers $f = f(e)$ and $e = e(x)$ and differentiates $f(e(x(t)))$ with respect to time to obtain a form in terms of current number of failures per encountered error. The rate at which failures are experienced with the Trachtenberg model is defined in Equation 1. The failure intensity is defined in Equation 2.

\[
\frac{df}{dt} = \lambda = \frac{df}{de} \frac{de}{dx} \frac{dx}{dt} = s \cdot d \cdot w
\]
Equation 2

\[ \lambda = -\frac{dR}{dt} = SDW e^{SDW t/R} \]

Where,

- \( x \) is the number of executed instructions
- \( e \) is the number of encountered faults
- \( f \) is the number of failures
- \( S, s \) are the initial and current average size of remaining faults
- \( d \) is the apparent fault density
- \( D = dR/r \) is the actual fault density
- \( R, r \) are the initial and current remaining faults, respectively
- \( W, w \) are the initial and current workload

The classical models of software reliability can be derived from Trachtenberg’s General Model by modifying assumptions for the parameters, \( s, d, \) and \( w \).

Musa’s model [MUS88], for example, assumes that \( s \), the average size of remaining faults is constant, that \( d \) the apparent fault density is the same as the actual fault density \( D \), and that \( w \) the workload is constant. In this model, the failure intensity decreases by a constant amount each time a defect is removed. (The general assumption is that all defects are corrected when discovered.) The failure rate decreases at the same rate at which remaining errors decrease. Musa’s reliability growth is defined in Equation 3 with the failure intensity in Equation 4.

Equation 3

\[ R = e^{-\lambda t} \]

Where,

- \( \lambda_t \) failure intensity over time
Where,
\( t \) is the number CPU hours
\( \nu_0 \) is the total failures
\( \lambda_0 \) is the failure intensity (failures / CPU hour)

The Goel and Okumoto reliability growth model [GOE79] (G-O model) is similar to the Musa model and is concave. It uses the function outlined in Equation 5.

**Equation 5**

\[
\mu(t) = a(1 - e^{-bt}), \quad a \geq 0, \quad b > 0
\]

Where,
\( a \) is the expected total number of failures that would occur if testing was infinite. It is the upper limit that the reliability (or number of failures) approaches asymptotically as \( t \) approaches infinity.
\( b \) is the rate at which the failures detection rate decreases. It is a shape factor for the curve.

The delayed S-shaped model by Yamanda et al. [YAM83] is a modification of the G-O model to make it S-shaped. An S-shaped reliability growth curve describes a reliability growth trend with a lower rate of failures occurring during the early stages of development and a higher rate later. The higher rate eventually saturates as the time between failures increases because more and more errors are fixed. It is given by the formulae in Equation 6.
\begin{equation}
\mu(t) = a(1 - (1 + bt)e^{-bt}), \quad a \geq 0, \quad b > 0
\end{equation}

Where,

- \(a\) is the expected total number of failures that would occur if testing was infinite
- \(b\) is the failure detection rate during the steady-state, that is the value to which the rate converges as \(t\) approaches infinity. (The failure intensity rate initially increases from \(t = 0\) to \(t = 1/b\) and then gradually decreases, approaching zero.)

The models examined in this thesis have two parameters; other models may have more. An example of a model with more than two parameters is the Yamada exponential model. The Yamada exponential model, a concave model [YAM86], attempts to account for differences in testing effort. It does not assume that testing effort is constant over the testing period. It is given by Equation 7.

\begin{equation}
\mu(t) = a(1 - e^{-bc(1-e^{-dt})}), \quad a \geq 0, \quad bc > 0, \quad d > 0
\end{equation}

Where,

- \(a\) is the expected total number of failures that would occur if testing was infinite
- \(b\) is the failure detection rate per unit testing effort
- \(c, d\) are parameters in the testing effort function; \(c\) and \(d\) are based on assuming an exponential form for the testing effort function
The failure process is a non-homogeneous process that is the characteristics of the probability distribution vary over time. Each time a failure occurs, the error that caused it is immediately fixed, under the assumption that no other errors are introduced. To illustrate the reliability growth process I have included a worked example in the following paragraph. This example uses a basic Musa model [MUS88].

**WORKED EXAMPLE**

Using Equation 3 and Equation 4 consider a program with an initial failure intensity of 10-failures/CPU hour, a total of 100 failures and examined between 10 CPU hours and 100 CPU hours of execution.

**TABLE 1** outlines the results from this example problem. Figure 5 and Figure 6 show the increase in reliability vs. the decrease in failure intensity over time.

Although the models are useful in reliability estimation by measuring a reliability percentage at a given time, there are several problems associated with SRGM. One problem is the model selection for specific situations. **TABLE 2** outlines a proposed set of common models.

**TABLE 1: EXAMPLE DATA VALUES FOR BASIC MUSA MODEL**

<table>
<thead>
<tr>
<th>R</th>
<th>$\lambda(t)$</th>
<th>T</th>
<th>$v_0$</th>
<th>$\lambda_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.02525</td>
<td>3.6788</td>
<td>10</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>0.25836</td>
<td>1.3534</td>
<td>20</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>0.6078</td>
<td>0.4979</td>
<td>30</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>0.8326</td>
<td>0.1832</td>
<td>40</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>0.93482</td>
<td>0.0674</td>
<td>50</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>0.9755</td>
<td>0.0248</td>
<td>60</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>0.99094</td>
<td>0.0091</td>
<td>70</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>0.99661</td>
<td>0.0034</td>
<td>80</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>0.9988</td>
<td>0.0012</td>
<td>90</td>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>0.9995</td>
<td>0.0005</td>
<td>100</td>
<td>100</td>
<td>10</td>
</tr>
</tbody>
</table>
Figure 5: Example of decreasing failure intensity over time using the basic Musa model.

Figure 6: Example of increasing reliability growth over time using the basic Musa model.
<table>
<thead>
<tr>
<th>Model name</th>
<th>Formula for hazard function</th>
<th>Data and/or estimation required</th>
<th>Limitations and constraints</th>
</tr>
</thead>
</table>
| General Exponential (General form of the Shooman, Jelinski-Moranda and Keene-Cole exponential models) | $K(E_0 E_0^x)$ | - Number of corrected faults at some time $x$.  
- Estimate of $E_0$ | - Software must be operational  
- Assumes no new faults are introduced in correction  
- Assumes number of residual faults decreases linearly over time |
| Musa Basic | $\lambda_x = \lambda_0 e^{-\lambda_0/\nu_0 t}$ | - Number of detected faults at some time $x$ ($\mu$)  
- Estimate of $\lambda_0$ | - Software must be operational  
- Assumes no new faults are introduced in correction  
- Assumes number of residual faults decreases linearly over time |
| Musa Logarithmic | $\lambda_x \exp (-\Phi t)$ | - Number of detected faults at some time $x$ ($\mu$)  
- Estimate of $\lambda_0$  
- Relative change of failure rate over time ($\Phi$) | - Software must be operational  
- Assumes no new faults are introduced in correction  
- Assumes number of residual faults decreases exponentially over time |
| Littlewood/Verrall | $\frac{\mu}{(t + \Gamma(\mu))}$ | - Estimate of $\alpha$ (Number of failures)  
- Estimate of $\psi$ (Reliability growth)  
- Time between failures detected | - Software must be operational  
- Assumes uncertainty in correction process |
| Weibull model | $MTTF = \frac{\alpha}{\nu - 1}$ | - Total number faults found during each testing interval  
- The length of each testing interval  
- Parameter estimation of $a$ and $b$ | - Failure rate can be increasing, decreasing or constant |
| Geometric model | $D\Phi^{a+b}$ | - Either time between failure occurrences $X_i$ or the time of the failure occurrences  
- Estimation of constant $D$ which decreased in geometric progression ($0 < \Phi < 1$) as failures are detected | - Software is operational  
- Inherent number of faults assumed to be infinite  
- Faults are independent and unequal in probability of occurrence and severity |
| Thompson and Chelson’s Bayesian Model | $(\Lambda_i + \Lambda_0 + 1)\gamma$ | - Number of failures detected in each interval ($\Lambda_i$)  
- Length of testing time for each interval $i$ ($\Lambda_i$) | - Software is corrected at end of testing interval  
- Software is operational  
- Software is relatively fault free |
| Yamada, Ohba and Osaki’s S-Shaped model | $\Delta d \exp b$ | - Tie of each failure detection  
- Simultaneous solving of $a$ and $b$ | - Software is operational  
- Fault detection rate is S shaped over time |
Figure 7: Reliability growth model checklist. Showing a step by step approach to select a reliability growth model [LAK97].
Figure 7 shows a step-by-step approach to selecting when to use each model type. Using Lakey & Neufelder’s [LAK97] proposed reliability growth model checklist, one can select the type of model required based on current working domain knowledge. To illustrate the concept, let us consider the hypothetical case of software that is in a system test phase, no previous historical data are available, defects have been occurring throughout the testing process and a curved intensity vs. time graph has been plotted.

Table 3 provides an example of the identification steps to select the appropriate model for this hypothetical case. It also outlines in bold the models that can be used for this hypothetical example.

**TABLE 3: EXAMPLE OF STEPS TAKEN TO SELECT A RELIABILITY MODEL WITH LAKEY & NEUFELDER’S [LAK97] MODEL CHECKLIST**

<table>
<thead>
<tr>
<th>Steps</th>
<th>Known Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current phase?</td>
<td>System testing phase</td>
</tr>
<tr>
<td>Are failures increasing/decreasing or both?</td>
<td>Decreasing</td>
</tr>
<tr>
<td>What shape does the graph represent curved or straight line?</td>
<td>Curved</td>
</tr>
<tr>
<td>Any historical data?</td>
<td>No</td>
</tr>
<tr>
<td>Which model types can be used?</td>
<td>Schneidewind, S Shaped or Weibull</td>
</tr>
</tbody>
</table>

The problem is that collecting this information is time consuming and experienced personnel is required to correctly gather data and metrics from each of the software development phases. Consistency in the data collection is imperative, but not always practiced. This introduces errors into the prediction. As in any time series prediction,
early accuracy literally makes or brakes the accuracy, and hence the usefulness, of the prediction.

Another common problem with SRGM is that they tend to be too optimistic [CHE95a, CHE95b]. That is, the test time to remove defects is often much greater than can be achieved in polynomial time. For systems such as real-time systems, it can take exponential time to remove all the defects [LIT89, KEI91].

TABLE 4 shows results from an experiment by Nagel [NAG82], which looked at testing to high levels of reliability. This table shows that for this experiment to remove the last defect it would take an unacceptable amount of time. This can be seen in the final column where the time to remove the last defect is averaged at over 42 years.

<table>
<thead>
<tr>
<th>Program</th>
<th>Slope</th>
<th>y-intercept</th>
<th>Last defect</th>
<th>Test time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>-1.415</td>
<td>2.358</td>
<td>17</td>
<td>42 years</td>
</tr>
<tr>
<td>B1</td>
<td>-1.3358</td>
<td>1.1049</td>
<td>19</td>
<td>66 years</td>
</tr>
<tr>
<td>A2</td>
<td>-1.998</td>
<td>2.4572</td>
<td>13</td>
<td>31 years</td>
</tr>
<tr>
<td>B2</td>
<td>-3.623</td>
<td>2.3296</td>
<td>7</td>
<td>19 years</td>
</tr>
<tr>
<td>A3</td>
<td>-0.54526</td>
<td>-1.3735</td>
<td>42</td>
<td>66 years</td>
</tr>
<tr>
<td>B3</td>
<td>-1.3138</td>
<td>0.0912</td>
<td>19</td>
<td>32 years</td>
</tr>
</tbody>
</table>

Given these problems, various attempts have been reported to improve the estimation and predictive capability of the models.

Brocklehurst et al. [BRO90] investigated the application of recalibration techniques to improve the prediction of the software reliability growth models. This technique explored re-estimating the model parameters by recalculating the initial model parameters as new inputs became available. There was criticism of this technique by Sitte [SIT99] as it was
tedious, required additional effort such as re-calculation and techniques that can be obscure or difficult to understand in practical applications.

Rather than developing an additional technique, Lyu and Nikora [LYU92] proposed an extension of SRGM by simplifying the complicated software development and operational processes assumptions. However, when these models were applied to real world data, many discrepancies between actual and predicted reliability were observed.

Li and Malaiya [LI93] examined a different technique to improve prediction capabilities by pre-processing data using noise filtering as a recalibration technique. The work involved the four major reliability growth models: exponential, logarithmic, delayed s-shaped and power. A closer look at the different techniques of pre-processing, such as grouping with fixed group size, grouping failure intensity lumps, windowing, weighted least square and adaptive approach were examined. Figure 8 summarises the effect of pre-processing and recalibration techniques where, (Raw): is a value without using enhancing technique; (FG): is a fixed size grouping using 50 grouped points; (LG): is a Lump grouping twice; (Rec): is recalibration only; (F&R): uses fixed size grouping with recalibration and; (L&R): uses lump grouping with recalibration.

It was found through empirical testing that the different models have different characteristics with respect to the enhancing techniques.

Malaiya et al. [MAL94] proposed another direction for reliability modelling. This new direction explored the connection between software test coverage and reliability by developing a logarithmic model that related testing effort to testing coverage. The results show that software testing coverage techniques such as branch coverage (explained later in this chapter) achieve testing saturation at 84% and support a view that 80% branch coverage is often adequate to achieve reliable software. However, a problem arises when bringing this together with the required levels of reliability.
Figure 8: The effect of pre-processing and recalibration on reliability models in respect to the average number of errors. The largest gains are made with the Power based models [LI93]

For example, 100% branch coverage might not be enough and achieving 100% path coverage may not be attainable in polynomial time.

Chen, Lyu and Wong [CHE96] returned to the previous research proposed in Lyu and Nikora’s [LYU92] earlier work and applied a new technique of modelling failure rate with respect to both software test coverage and testing time in the estimation of software reliability. The new model revealed the efficiency of a testing profile so more effective testing could be conducted. The model was found useful in uncovering when a testing technique became ineffective for regression testing.
Chen, Lyu and Wong [CHE01] continued this work by studying the effect of code coverage on software reliability. It was proposed that usage profiles be provided for appropriate test case selection and that the test data be pre-processed before being used in the SRGM. The problem was relating test coverage to defect coverage.

Malaiya et al. [MAL02] also continue their work by relating testing effort to test coverage. This is summarised in Figure 9.

![Figure 9: Increasing coverage growth with respect to increased testing effort [MAL02]](image)

The figure shows that as the number of tests increases so does the overall test coverage. The authors claimed that developers can achieve reliability by evaluating reliability during the development process and using this to predict optimal allocations of resources. The problem is that reliability is affected by factors, such as strategies based on black or white box testing, relationships between execution of tests and real time
execution and testing of rarely executed modules. It was found that it is difficult to test these modules but they often contain error recovery routines and exception handing which makes them vital to the program and therefore they must be reliable.

Lyu continued his work with Chin [CHI05a], to study the impact of software testing effort and efficiency on the modelling of software reliability, including the cost for optimal release time.

The work presented two important issues in software reliability modelling and software reliability economics: testing effort, and efficiency. In this work the authors propose a generalised logistic testing-effort function that relates a work profile with software development. Furthermore, the authors incorporate the testing-effort function into software reliability modelling. The new model was evaluated with several numerical experiments. This research demonstrated that the model was successful in predicting when to stop testing in the examples that were examined by the authors.

Chin and Lyu [CHI05b] also considered two kinds of software testing-resource allocation problems. The first problem was to minimise the number of remaining faults given a fixed amount of testing-effort, and a reliability objective. The second problem was to minimise the amount of testing-effort given the number of remaining faults, and a reliability objective. In the research the authors proposed several strategies for module testing to help software project managers. Using a non-homogeneous Poisson process model, the authors allowed systematic allocation of a specified amount of testing-resource expenditures for each software module. Experimental results indicated the advantage of the approach was in guiding software engineers and project managers toward better testing allocation.

Finally, Karunanithi et al. [KAR92] demonstrated that it is possible to predict the Software Reliability time series for time between failures with Artificial Neural Networks (ANN). Sitte demonstrated [SIT99] that with a suitable ANN architecture equal or better prediction can be achieved in a simpler way than with the tedious curve fitting models.
The suitability of ANN architecture depends on the nature of the data. The software reliability data are good candidates for ANN training and predictions.

Although the methods discussed in this section aid in improving reliability modelling, none of them can fix the underlying limitations of SRGM in general. As with any time series, these limitations are that no universally accepted model can give accurate results in all circumstances, and that varying factors, in particular the type of data, strongly influence the predictive capability and accuracy of the model. The slightest estimation error can fan out into a large prediction error.

Consequently there is no single model at the present time that can be used in all situations, perhaps with the exception of Artificial Neural Networks [KAR92, SIT99]. Therefore, regardless of the many models that there are, and many more emerging, their predictive capability is limited, the amount and increasing complexity of software. Therefore, constraints and assumptions have to be made for quantifying the process of the models [LYU91, LYU92, LYU95, STR02 CHI05a, CHI05b].

2.1.2 Software Test Coverage

This section describes the purpose and methods for software test coverage and how testing coverage relates to software reliability.

As mentioned in the previous section, Malaiya et al. [MAL94, MAL02] suggested that software test coverage can be applied to reliability estimations. Software test coverage is based on a white box (structure) testing techniques and compares test program behaviour against source code. This contrasts with functional or black box testing, which compares program behaviour against a requirements specification. Table 5 gives a comprehensive comparison of the two techniques and is based on work reported in [MYE79, BEI95].
TABLE 5: COMPARISON OF WHITE AND BLACK BOX TESTING

<table>
<thead>
<tr>
<th>White box testing</th>
<th>Black box testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Potentially infinite number of paths have to be tested</td>
<td>Potential combinatorial explosion of test cases (valid &amp; invalid data)</td>
</tr>
<tr>
<td>White box testing often tests what is done, instead of what should be done</td>
<td>Often not clear whether the selected test cases uncover a particular error</td>
</tr>
<tr>
<td>Cannot detect missing use cases</td>
<td>Does not discover extraneous use cases (features)</td>
</tr>
<tr>
<td>Both types of testing are needed</td>
<td>Both are the extreme ends of a testing continuum</td>
</tr>
<tr>
<td>Both are the extreme ends of a testing continuum</td>
<td>Any choice of test cases lies in between and depends on:</td>
</tr>
<tr>
<td></td>
<td>• Number of possible logical paths</td>
</tr>
<tr>
<td></td>
<td>• Nature of input data</td>
</tr>
<tr>
<td></td>
<td>• Amount of computation</td>
</tr>
<tr>
<td></td>
<td>• Complexity of algorithms and data structures</td>
</tr>
</tbody>
</table>

Software test coverage works by examining the coverage of tests applied to a software program and involves the following steps:

- finding areas of a program not exercised by a set of test cases;
- creating additional test cases to increase coverage;
- determining a quantitative measure of software coverage, which provides a measure of software quality; and
- identifying redundancy where a possible test case would not contribute to overall test coverage

At first glance, white box testing seems unsafe. White box testing cannot find errors of omission. Unfortunately, requirements specifications sometimes do not exist, and are rarely complete. This is especially true towards the end of the product development timeline when the requirements specification is updated less frequently and the product itself begins to take over the role of the specification. It must be noted that both techniques should be used to better ensure more reliable software as mentioned by Beizer and Myers [MYE79, BEI95]. However, the major advantage of white box testing is to provide
mechanisms for selecting paths to test, because it is not possible to execute all paths through a program.

The diagram in Figure 10 outlines an example program structure to illustrate why one cannot execute all paths through a program. The problem is the combinatorial explosion of paths.

The concept of software test coverage drew its roots from hardware coverage. Hardware coverage is measured in terms of the number of possible faults covered [MAL94]. The problem with software test coverage is that the number of “faults” is not known. Therefore, a measure exists to overcome this in respect to structural or data-flow coverage criteria along with a percentage of total coverage.

There is a huge variety of structural or data-flow coverage criteria [WEY93], the most important and used methods are examined.

Structural coverage techniques include block coverage, branch coverage and path coverage. Block coverage is the most basic form of structural coverage. It reports the fraction of the total number of executable statement encountered and is also known as segment coverage [NTA88], and statement coverage. The flow graph in Figure 11 (a) represents a hypothetical statement coverage example.

The statement coverage corresponds to visiting the nodes of the flow graph. 100% block coverage is where every node is visited once. The path A-C-G-I-K if iterated once would give 5 in 10 node coverage or 50% block coverage. The main problem of block coverage is that it cannot test program control flow like if-statements or while-loops. Block coverage is essential but not sufficient as a measure of test completeness [HOR94].
Figure 10: Example of combinatorial explosion of paths due to loops illustrating why one cannot execute all paths through a software program.

Paths from A-X:
1. ABCX
2. ABDEGX
3. ABDEHX
4. ABDFIX
5. ABDFJX

- One iteration, i.e. not returning along the loop at X results in 5 paths.
- Two iterations of decision logic, i.e. return along loop at X the first time only results in 25 paths or $5^2$.
- Three iterations have 125 or $5^3$. Twenty has $5^{20}$.
- All combinations result in $5+5^2+5^3+\ldots+5^{20} = >10^{14}$ paths.
Figure 11: Software coverage techniques compare test program behaviour against the intention of the source code by examining structure and logic. Three common coverage techniques include (a) block, (b) branch and (c) path coverage.

Branch coverage is a more rigorous method of structural coverage and reports the fraction of the total number of branches that have been executed by the test data and is also known as decision coverage and all-edges coverage [ROP94]. The flow graph in Figure 11 (b) represents a hypothetical branch coverage example.

The branch coverage corresponds to visiting a node in a decision branch outlined by the red arrows. 100% branch coverage is where all decision branches are covered. The path A-C-G-I-K if iterated once would give 2 in 6 decision branch node coverage or 33% branch coverage. This measure has the advantage over statement coverage in its ability to cover testing of control flow instructions like if-statements and while loops.

Path coverage is the most disciplined technique in testing coverage and is determined by assessing the proportion of execution paths through a program. The flow graph in Figure 11 (c) represents a hypothetical path coverage example.
100% path coverage is where all paths in the program are traversed. The path A-C-G-I-K if iterated once would give 1 in 4 path coverage or 25% path coverage.

To overcome the problems of combinatorial path explosion illustrated in Figure 10 two common methods are widely used that is McCabe’s cyclomatic complexity [MCC76] and Beizer’s loop tests [BEI95].

It has been campaigned that cyclomatic complexity is recommended as the foundation of any software complexity problem [MCC94]. The cyclomatic complexity of a section of source code is the count of the number of linearly independent paths through the source code. For instance, if the source code contained no decision points such as IF statements or FOR loops, the complexity would be 1, since there is only a single path through the code. If the code had a single IF statement there would be two paths through the code, one path where the IF statement is evaluated as TRUE and one path where the IF statement is evaluated as FALSE.

Cyclomatic complexity is normally calculated by creating a graph of the source code with each line of source code being a node on the graph and arrows between the nodes showing the execution pathways. As some programming languages can be quite terse and compact, a source code statement when developing the graph may actually create several nodes in the graph.

In general, in order to fully test a module all execution paths through the module should be exercised. This implies a module with a high complexity number requires more testing effort than a module with a lower value since the higher complexity number indicates more pathways through the code. This also implies that a module with higher complexity is more difficult for a programmer to understand since the programmer must understand the different pathways and the results of those pathways.

One would also expect that a module with higher complexity would tend to have lower cohesion than a module with lower complexity. The possible correlation between higher
complexity measure with a lower level of cohesion is predicated on a module with more
decision points generally implementing more than a single well defined function.
However there are certain types of modules that one would expect to have a high
complexity number, such as user interface modules containing source code for data
validation and error recovery. The main problem with this method is the fact it examines
exhaustively all the possible paths in a software system [MCC76, MCC94].

Beizer’s loop tests [BEI95] use a similar frame work of basis path testing as adopted in
McCabe’s work. Beizer’s loop tests tried to resolve issues of loop termination and
switching between conditions used in different iterations of the loop by examining the
minimum number of tests required to test the functionality of the loop. Using the loop
tests can greatly reduce the combinations of tests required as it examines specific
iterations of the loop and the paths through the loop at that time. The tests used for this
are listed in TABLE 6.

However, like McCabe’s cyclomatic complexity as the software gains complexity, the
number of possible paths can grow exponentially and the main problem as with
McCabe’s work is that it examines exhaustively all the possible paths through the loop at
the set iteration.

Other methods examine the use of dataflow testing techniques such as P-use, C-use, All-
Defs and All-Paths. These methods are inherently similar to each other, using the same
basic approach [PRE97].

Dataflow testing techniques select tests paths of a program according to the locations of
definitions and uses of variables. The main difference among them is the level to which
each method can cover the software.

Given a set of test cases, let P be the set of complete paths exercised by a program
execution for these test cases. For each of the data flow based test coverage criterion, the
condition that P must meet for the test to be satisfied is given below.
**TABLE 6: BEIZER’S LOOP TESTS [BEI95]**

<table>
<thead>
<tr>
<th>Bypass</th>
<th>any value that causes loop to be exited immediately</th>
</tr>
</thead>
<tbody>
<tr>
<td>Once</td>
<td>values that cause the loop to be executed exactly once</td>
</tr>
<tr>
<td>Twice</td>
<td>values that cause the loop to be executed exactly twice</td>
</tr>
<tr>
<td>Typical</td>
<td>a typical number of iterations</td>
</tr>
<tr>
<td>Max</td>
<td>the maximum number of allowed iterations</td>
</tr>
<tr>
<td>Max + 1</td>
<td>one more than the maximum allowed</td>
</tr>
<tr>
<td>Max - 1</td>
<td>one less than the maximum allowed</td>
</tr>
<tr>
<td>Min</td>
<td>the minimum number iterations required</td>
</tr>
<tr>
<td>Min + 1</td>
<td>one more than the minimum required</td>
</tr>
<tr>
<td>Min - 1</td>
<td>one less than the minimum required</td>
</tr>
<tr>
<td>Null</td>
<td>one with a null or empty value for number of iterations</td>
</tr>
<tr>
<td>Negative</td>
<td>one with a negative value for number of iterations</td>
</tr>
</tbody>
</table>

All-Defs is satisfied if P includes a definition-clear path (that is the variable is not reassigned in the path) from every definition to some corresponding use.

All-C-uses is satisfied if P includes a definition-clear path from every definition to all of its corresponding c-uses.

All-C-uses/Some-P-uses is satisfied if P includes a definition-clear path from every definition to all of its corresponding C-uses. In addition, if a definition has no C-use, then P must include a definition-clear path to some P-use.

All-P-uses is satisfied if P includes a definition-clear path from every definition to all of its corresponding P-uses.
All-P-use Some-C-uses is satisfied if P includes a definition-clear path from every definition to all of its corresponding P-uses. In addition, if a definition has no P-use, then P must include a definition-clear path to Some-C-use.

All-Uses is satisfied if P includes a definition-clear path from every definition to each of its uses including both C-uses and P-uses.

All-Du-paths is satisfied if P includes All-Du-paths for each definition. Therefore if there are multiple paths between a given definition and a use, they must all be included.

Moreover each use is considered tested only if it is output influencing, that is, it directly or indirectly influences the computation of some program output during the program run.

From the above definitions of different data flow based criteria it is clear that each criterion identifies a minimal set of definition-use associations that must be exercised by a set of test cases to satisfy the criterion. However, this minimal set is not unique.

Various comparisons of data flow techniques and structural testing have been shown in the literature [BIE89, FRA93, WEY93, FRA97, FRA98]. Figure 12 outlines the subsumption relationship between the different techniques.

From this diagram it can be seen that the All-Paths technique is more discerning than All-DU-Paths which is more discerning than All-Uses etc. The other important point to be made is the upper bounds on the test complexity to apply each of these coverage methods and this is shown in TABLE 7.
**Figure 12:** Subsumption relationship of different testing coverage criteria [WEY93].

**TABLE 7: COVERAGE CRITERION AND COMPLEXITY [WEY93]**

<table>
<thead>
<tr>
<th>Coverage Criterion</th>
<th>Upper Bound</th>
<th>Observed</th>
</tr>
</thead>
<tbody>
<tr>
<td>All-Blocks</td>
<td>(d + 1)</td>
<td></td>
</tr>
<tr>
<td>All-Branches</td>
<td>(d + 1)</td>
<td></td>
</tr>
<tr>
<td>All-P-Uses</td>
<td>(\frac{1}{4} (d^2 + 4d + 3))</td>
<td>0.38d + 3.17</td>
</tr>
<tr>
<td>All-Defs</td>
<td>(m + (i \ast n))</td>
<td></td>
</tr>
<tr>
<td>All-P-Uses/Some-C-Uses</td>
<td>(\frac{1}{4} (d^2 + 4d + 3))</td>
<td></td>
</tr>
<tr>
<td>All-C-Uses/Some-P-Uses</td>
<td>(\frac{1}{4} (i^2 + 4d + 3))</td>
<td>0.36d + 2.82</td>
</tr>
<tr>
<td>All-Uses</td>
<td>(\frac{1}{4} (d^2 + 4d + 3))</td>
<td>0.39d + 3.76</td>
</tr>
<tr>
<td>All-DU-Paths</td>
<td>(2^d)</td>
<td>0.49d + 4.03</td>
</tr>
<tr>
<td>All-Paths</td>
<td>(\infty)</td>
<td></td>
</tr>
</tbody>
</table>

n: variables, m: assignments, i: input statements, d: binary decisions
The underlying concepts shown by Figure 12 and Table 7 is the All-paths (100% path coverage) software testing criterion is the most discriminating of the testing criteria, however its worst case has infinite complexity [WEY93] and can require exponential numbers of test cases and time to test [BIE89, FRA93].

The main problem with all these methods is the difficulty in achieving high coverage levels in a reasonable time frame [FRA98]. Various improvements to reduce this problem and prioritise testing coverage have been practiced over the years.

Poole [POO95] examined a depth first algorithm to determine the set of basis paths for potential use in software test automation. The method used McCabe’s cyclomatic complexity measure [MCC76] to evaluate the function under test. The algorithm was a good starting point determining a set of basis paths but failed to provide adequate test coverage measures.

Bieman et al. [BIE96] investigated the application of automated fault injection to improve testing coverage. It was nearly impossible to manually run all statements or branches in the test software and automated support was required to improve correctness. To achieve automation Bieman et al. experimented with C-Patrol, a code insertion tool to insert violations near boundary conditions to simulate common programming errors. The system was applied over four experiments shown in Table 8.

Table 8 shows the result in automated fault injection was effective in increasing coverage; however, it could not force all sections of code to be executed.

Petrenko, Bochmann and Yao [PET96] also investigated a fault injection technique using a method built on finite state models. Experiments were conducted to analyse the basic underlying techniques for fault coverage analysis. The authors found that branch coverage was insufficient however All-path testing was impractical. The outcome shows no solid results although it outlines the essential yet challenging problem for improving coverage techniques.
Briand and Pfahl [BRI99] used simulation for assessing the impact of test coverage on defect coverage. The research found that both testing and defect coverage were influenced by the amount and intensity of testing. As they suspected initially, there was a correlation between faults and the coverage of software.

Wong et al. [WON99] explored a reduction technique for reducing the testing set size in hopes of achieving better coverage results. By eliminating all ineffective test cases the size of the test set was reduced. The problem was, trying to keep block coverage of the code at a constant level. To achieve constant block coverage Wong et al. involved a two-step reduction method of the test set size and an exponential time algorithm. The results indicated that if a test set was generated using the reduction techniques, block coverage stayed constant and the resultant test set size was reduced.

Walton and Poore [WAL00] presented a new specification based on statistically typical paths. The system involved representing path coverage with respect to statistically typical paths or paths that are typical in a software system. The benefits were quantifying the size of a testing problem and demonstrating the need for mathematical assistance in using the test budget of time and money in an optimal way.

**Table 8: Evaluation of C-Patrol Experiments showing increases in code coverage [BIE96]**

<table>
<thead>
<tr>
<th>Evaluation Software</th>
<th>Original Coverage</th>
<th>Final Coverage</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 Function from 3 files</td>
<td>41.9%</td>
<td>94.6%</td>
<td>125.8%</td>
</tr>
<tr>
<td>6 Compiler files</td>
<td>77.7%</td>
<td>83.6%</td>
<td>7.6%</td>
</tr>
<tr>
<td>9 C-patrol files</td>
<td>84.8%</td>
<td>87.0%</td>
<td>2.6%</td>
</tr>
<tr>
<td>12 Autoland files</td>
<td>78.9%</td>
<td>87.3%</td>
<td>10.7%</td>
</tr>
</tbody>
</table>
Lin and Yeh [LIN01] presented an interesting technique using GA to automatically generate test cases to test a selected path. The algorithm used sequences of operators to converge on a path. The algorithms were based on extended hamming distance to evaluate fitness. The disadvantage is that it uses random path selection that is it could be equally the least or most error prone path selected for testing and subsequent coverage calculations.

Rothermel et al. [ROT01] and Jones and Harrold [JON03] investigated the application of testing on defect prediction by using prioritisation techniques. In [ROT01] tests were based on mutation testing and industry partner black box tests to generate the prioritisation of test cases based on how quickly faults could be detected. In [JON03] two addition algorithms were used to test based on modified condition/decision coverage. It was claimed by the authors that running a suite of test cases required seven weeks on a 20KLOC program supplied by an industry partner and this outlined their need for prioritisation.

S.Elbaum et al. [ELB01a, ELB01b, ELB02, ELB03, ELB04] examined test case prioritisation techniques to improve the coverage in limited time frames. The techniques that were investigated were based on various metrics, including code coverage, fault likelihood, and fault exposure potential. The authors claim that each technique could significantly improve the rate of fault detection during regression testing. In particular they focused on increasing the likelihood of revealing faults earlier in the testing process. That is, the goal was to find the average percentage of faults detected (APFD). Figure 13 shows how the authors prioritised these APFD.
Figure 13: Prioritisation of APFD with increased fault detection and test suite size [ELB01a].

The area inside the inscribed rectangles (dashed boxes) represents the weighted percentage of faults detected over the corresponding fraction of the test suite. The solid lines connecting the corners of the inscribed rectangles interpolate the gain in the percentage of detected faults. The area under the curve thus represents the weighted average of the percentage of faults detected over the life of the test suite. The results are that Figure 13 (c) represents a more effective test order than Figure 13 (b).

The experiments also show that there was significant statistical evidence of an association between rate of fault detection and the program under test. It was concluded that different programs offer varying rates of prioritisation potential. Also that different metrics can add value to variations in the prioritisation techniques.

The main issue is that to prioritise coverage cost-effectively, practitioners must be able to assess which techniques are likely to be most effective in their particular scenarios, that
is, given their particular programs, test cases, coverage percentages and modifications. With this in mind, the goal would be to seek an algorithm which, given various metrics about programs calculates and recommends regions to apply various techniques. However, the factors affecting prioritisation success are complex, and interact in complex ways [ELB01a].

Other approaches for increased reliability based on defect detection and prevention have also been examined; they include software metrics and inspection. The following section examines these techniques and their effect on developing reliable software.

### 2.1.3 Software Metrics and Inspection

This section describes the growing use of software inspections and software metrics¹ in improving overall software reliability.

Software metrics are a useful tool [BOE81, HUM96, FEN99b, KAN02, GIT04] and if done often and well add value to both management and technical decisions for both software and software intensive systems. However in most cases the gathering and using of metrics is poorly integrated into the education and practice of organisations. The biggest difficulties are establishing and using the metrics, management commitment and performance. Because of these benefits and issues many researchers have investigated software quality metrics.

Hochman et al. [HOC96] proposed a software metric approach using GA and Neural Networks for fault prone module detection. The metrics were used as inputs to the Neural Networks to classify fault based modules, however there were disadvantages using Neural Networks for this purpose. The Neural Networks were time-consuming and labour

---

¹ There is controversy among software practitioners that reaches into the ISO standards about the usage of the term “software metrics”. This term has been used for many years and is familiar to many readers, but in some ISO standards documents is being replaced by “measurement”. In this thesis I will use “measurement” as a single source quantity, and “metric” to express the relation of two measurements. This is in line with more recent trends.
intensive. A GA was proposed to simplify and automate the search by selecting the Neural Network with the greatest overall fitness. The resultant system was tested on a telecommunications program with 12 million lines of code and 9 software metric inputs. The outcome was improved testing effort by covering large search spaces that could not be covered manually in polynomial time.

Kumar, Rai, and Trahan [KUM98] also examined software metrics and Neural Networks for classifying fault prone modules. The classification was based on studying some basic problems associated with predicting errors using historical software metrics. Kumar, Rai, and Trahan found there were basic problems associated with predicting errors using software metrics however the authors claimed their results were useful in enhancing software quality by removing potential errors and reducing maintenance costs. Rosenberg et al. [ROS98] also backed the claim of Kumar, Rai, and Trahan by examining software metrics for software defect detection at early stages.

Fenton and Neil [FEN99a] outlined a critique on software defect prediction models. They claim that some programs contain defects that are hard to find as the code is not often executed by testing. Figure 14 examines the various results from the authors.

Fenton and Neil identified three main classes. Figure 14(a) shows the first class which illustrates how defect density falls as module size increases. The second class of models, shown in Figure 14(b), differ from the first because they show how defect density rise as modules gets bigger. The third class, shown in Figure 14(c), shows no discernible pattern. Here the relationship between defect density and module size appears random.

The key issues outlined by Fenton and Neil with respect to defect prediction are the unknown relationships between defects and failures, problems with statistical approaches, problems using size and complexity as sole predictors, statistical methodology and data quality and false claims about the decomposition. All these issues raise questions about the reliability and useability of the prediction models.
Figure 14: Defect density class results with respect to module size (a) falling density with module size; (b) rising density with module size; (c) random density with module size [FEN99a].

Another interesting point brought forth by the authors is that much of the defect prediction work has not encompassed testing or testability.

Li and Smidts [LI03] took a step back from the prediction models and presented research into the measures that most affect software engineering and what metrics are most important for detecting defects. The paper presented a ranking of 30 measures based on a panel of 10 experts. The goal was to rank the measures to identify which were the most important for software engineering. These measures were based on reliability, validation, repeatability, benefits and credibility. The ranks show that in terms of testing that code defect density, cyclomatic complexity, fault density and test coverage were all ranked around 70% or higher in terms of their applicability to the testing phase of software engineering.

Another method, which has increased in popularity and over the past 30 years, is the use of inspections and reviews for improved defect detection. Software inspections have proven to be a powerful disciplined engineering practice for detecting and correcting
defects early in software development [FAG76, BOE81, GIL93, STR93, POR96, FRE05]. When used properly they can be one of the best methods for early defect detection.

The main issues are the time investment needed to run the inspection process and the expertise required to achieve a level of inspection yield that warrants the process. Due to these issues there is not a widespread use of inspection in all organisations [ONE97].

Chatzigeorgiou and Antoniadis [CHA03] presented an empirical example of the issue with time investment from a telecommunications project. This is shown in Figure 15 which illustrates the project and inspection effort versus project time. Inspection effort presents spikes during the course of the project, although the overall project effort has a relatively smooth form.

Industry experience shows that through inspections, cycle time is reduced, costs are lowered, process visibility is increased and also programmers’ capability is improved [FAG76, GIL93]. Inspections form an integral part in all phases of software development and apart from code reviews, cover all software artefacts. Although inspection details vary according to the employed methodology and the target project, a set of common elements can be identified. These include a number of well-defined inspections steps (e.g. preparation, meeting, and rework), well defined inspection roles (e.g. moderator, producer, and inspector), formal collection of inspection data and a supporting infrastructure.

In various studies dealing with inspections [ACK89, BAR94, ONE97] planning of inspections is identified as the initial stage of the inspection process. However, the purpose of planning in [ACK89, BAR94, ONE97] is to define the goals, the objectives and the methodology of the inspection rather than identify regions to inspect. Even though the importance of scheduling inspections in time has been addressed in [JOH98] the planning of the inspections is not performed sufficiently ahead of time.
Approaching inspection activities only in a procedural manner and not by taking into consideration project management requirements like time and cost scheduling can give rise to problems, especially in large software development projects. Costs increase rapidly due to excess overtime, which have not been initially planned and due to unforeseen higher costs for fixing faults in later development stages.

Many organisations want to predict the number of defects for software development and many metrics and models have been developed. However, companies are still asking how they can predict, how they can improve prediction methods, what data should be gathered and most importantly where to start and what to focus on.

Another issue that is presented in this area is how we can quantitatively state that one inspection method or software metric is more effective than another. One technique that
can be applied is error seeding. The following section examines this technique and how error seeding can be used to show the effectiveness of a given reliability method.

2.1.4 Error Seeding and Clustering

In this section the method of error seeding is discussed with strong evidence to support the notion that a small number of modules contain the majority of errors that is an example of the Pareto principle or 80/20.

Error seeding is a method that can be used to determine software reliability by giving quantitative evidence to support testing and inspection tools. Error seeding estimates the number of errors in a program by using multistage sampling. Errors are divided into sets and introduced (seeded) to the software. The unknown number of errors is estimated from the number of introduced errors and a ratio of errors is obtained.

The question is how the errors are distributed and how can this aid in determining where to focus effort or how to improve the effectiveness of the given inspection or testing method. A number of researchers have worked on the area of error seeding, in particular how errors cluster.

Enders [END75] examined the distribution of errors in modules. It was found that a small number of modules contains the majority of errors and that errors affect a small number of modules in most cases. These results are shown in TABLE 9.

From TABLE 9 it is clear that the majority of errors are found in 1 module but also that errors are spread out through a number of modules. This is indicated by the number of models and errors per module. Where 1 module contains 28 errors and 112 modules contain 1 error per module.
Basili and Perricone [BAS84] also examined the relationships between the frequency, complexity and distribution of errors in modules. The authors conducted experiments on a 90KLOC program. The experiments again show that most errors are found in 1 module with a small number found in a few modules (see Table 10).

A trend was also observed in that more errors were detected in smaller sized modules even though the larger modules were more complex (cyclomatic complexity). The interesting point was that the average complexity of the error-prone modules was not greater than the average complexity of the full set of modules (see Table 11).
Adams [ADA84] revealed that a great proportion of latent software faults lead to very rare failures in practice, while the vast majority of observed failures are caused by a tiny proportion of the latent faults. Adams observed a remarkably similar distribution of such faults across nine different major commercial systems. One conclusion of the Adams' study is that removing large numbers of faults may have a negligible effect on reliability and only when the small proportion of large faults are removed will reliability improve significantly. It appears to contradict the very basic hypotheses that underpin the notions of structured and modular programming. However this is also shown in Basili and Perricone's work [BAS84] and work by Hatton [HAT97].

Hatton provided a review of similar studies and found compelling empirical evidence from disparate sources implying that in any software system larger components are more reliable than smaller.

Fenton and Ohlsson [FEN00] conducted similar experiments. It was found that in their research a small number of modules contain the most faults discovered in pre-release testing and that a very small number of modules contain faults after release.

<table>
<thead>
<tr>
<th>Number of Errors</th>
<th>Number of Modules Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>155 (89%)</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>
The interesting feature is that there was no correlation between software complexity and size with the modules containing the faults.

It was found that more faults were detected in pre-release and that a stable number of faults were discovered at each testing phase. It was also found that those modules that are tested before release have fewer errors than those modules found with errors after release. The issue is that testing effort and operation usage were not taken into consideration; however it does outline the need for error proneness detection. One could conclude that the modules that saw the majority of errors after release were more error prone than the ones before release, however due to testing methods these slipped through the testing process (see Table 12).

Chou et al. [CHO01] examined a number of questions relating to error distribution, including how are defects distributed? And do defects cluster? To answer these questions the authors examined 21 versions of various Linux kernels spanning several years. It was concluded that only a few files contain the majority of errors and that most errors are not independent. If a defect appears once another defect is more likely to appear close by.

### Table 11: Cyclomatic Complexity and Error Rate for Errored Modules

[BAS84]

<table>
<thead>
<tr>
<th>Modules Size</th>
<th>Average Cyclomatic Complexity</th>
<th>Errors/KLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>6.2</td>
<td>65</td>
</tr>
<tr>
<td>100</td>
<td>19.6</td>
<td>33</td>
</tr>
<tr>
<td>150</td>
<td>27.5</td>
<td>24.6</td>
</tr>
<tr>
<td>200</td>
<td>56.7</td>
<td>13.4</td>
</tr>
<tr>
<td>&gt;200</td>
<td>77.5</td>
<td>9.7</td>
</tr>
</tbody>
</table>
This can be explained with the argument that programmers become careless or can be incompetent and therefore defects may appear in close proximity of other defects. Another interesting point is that more defects occur in more complex code although the authors have not defined complexity.

In 2002 Shull et al. [SHU02], conducted a symposium on what researches and practitioners in Software Engineering have learned about fighting defects. It was concluded from the symposium,

“As a general rule of thumb, 80% of a system’s defects come from 20% of its modules. However, the relationship varies based on environment characteristics such as processes used and quality goals.”
This is an example of the Pareto principle [BAN89, JUR99]. From this research it is clear that small number of modules contain the majority of errors. It is also evident that cyclomatic complexity is not a great indicator of the number of defects in the modules. The main issue is to determine what quality goals a project has and how to find errors.

The following chapter outlines one such method of quantifying the errors using potential error proneness.

\section*{2.1.5 Quantifying Error Proneness}

In this section the current issues with software reliability are discussed and a quantitative method to aid in improving on these issues is outlined.

There are three main disadvantages with the methods discussed in this chapter; they are (i) testing covers only a portion of the software for the level of reliability required; (ii) exponential time for testing all paths and (iii) cost effectiveness, that is, the process might not justify the time taken to test. At first glance the first mentioned disadvantage does not appear to be a problem. However, the main issue is, has the best portion (most error prone) been tested?

If a coverage level of 80\% has been achieved, does this really mean the most error prone 80\% has been covered or is it just any 80\% overall (with the most error prone possibly remaining in the untested 20\%)?

The answer is uncertain unless quantitative information about the errors is examined. Some software is easier to write than others, and potentially less error prone, while more difficult and complex software is potentially more prone to error. A range of complexity measures have been designed, but they do not provide a quantitative answer that indicates to which parts of the software the testing and inspection effort should be concentrated.
To be able to find the most error prone portions in a quantitative way, it is necessary to assess the chances that a programmer can introduce an error in an instruction. This is consistent with the findings that complexity is not necessarily the source for more defects [BAS84, FEN00]

There has been very little research conducted into quantitatively detecting the most error prone portions of software constructs as shown by the literature to date. The findings in this literature review evoke the need for detecting error prone portions of software. As discussed in the section on error seeding and clustering many researches agree that both, error cluster and 80% of defects do come from 20% of the code [END75, ADA84, BAS84, HAT97, FEN00, CHO01].

One method that does allow identification of the most potentially error prone paths is presented in a framework developed by Sitte [SIT00] (see Appendix A). With the purpose of being able to find the most error prone portions in a quantitative way, Sitte developed a framework to assess the chances that a programmer can introduce an error in an instruction. This is done by applying a simple SOE analysis. An example of a SOE analysis for the case of a loop is shown in Figure 16.

The construct in Figure 16 has at least four SOE (if those were the only possible sources of error). All other programming constructs can be analysed in a similar way, yielding SOE weights (discussed later in this chapter) that are specific to an instruction [SIT00]. The method provides the framework for SOE quantification; it is simple, but powerful and can quickly be obtained from the code in an automated way.
Figure 16: SOE analysis of a simple loop. There are at least four SOE and all constructs can be analysed in a similar way [SIT00].

Table 13 outlines theoretical values for the SOE calculation. This table is by no means exhaustive, but it serves as an indication on the process for quantifying error proneness.

While this method provides the minimum theoretical SOE [SIT00], additional information from test design strategies, risk analysis and historical data (if available) can be used to calibrate and fine tune the model by adding or subtracting weights to the theoretical SOE values.

By summing the SOE values (weighted or theoretical) for each instruction of a software module, it is possible to quantify the SOE for the different paths that can be stepped through in the software execution. The software structure can be represented as a sparse connectivity matrix whose heaviest paths can be identified. This can be achieved by applying a search algorithm to the environment.
TABLE 13: THEORETICAL VALUES FOR SOURCES OF PROGRAMMING ERRORS [SIT00]

<table>
<thead>
<tr>
<th>SOE opportunity</th>
<th>SOE value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Conditionals</strong></td>
<td></td>
</tr>
<tr>
<td>a) Boolean operator error</td>
<td>SOE = nvar + noperators</td>
</tr>
<tr>
<td>b) Boolean variable error</td>
<td>SOE = $2^{nvar}$</td>
</tr>
<tr>
<td>c) Boolean parenthesis error</td>
<td></td>
</tr>
<tr>
<td>d) Relational operator error</td>
<td></td>
</tr>
<tr>
<td>e) Arithmetic expression error</td>
<td></td>
</tr>
<tr>
<td><strong>Compound conditions</strong></td>
<td></td>
</tr>
<tr>
<td>Several conditions ANDed or ORed</td>
<td>SOE = nvar + $2^{nvar}$</td>
</tr>
<tr>
<td><strong>Loops</strong></td>
<td></td>
</tr>
<tr>
<td>a) Wrong initialisation</td>
<td>“while” loops: SOE = 5</td>
</tr>
<tr>
<td>b) Does not enter loop</td>
<td>“for” loops: SOE = 4</td>
</tr>
<tr>
<td>c) No progress in loop</td>
<td></td>
</tr>
<tr>
<td>d) Cannot reach end condition</td>
<td></td>
</tr>
<tr>
<td>e) Goes beyond loop end</td>
<td></td>
</tr>
<tr>
<td>f) Wrong exit condition</td>
<td></td>
</tr>
<tr>
<td><strong>Nested loops and conditionals</strong></td>
<td>SOE compounded for example: SOE = $5^{\text{nesting depth}}$</td>
</tr>
<tr>
<td><strong>Branching</strong></td>
<td>SOE = -2</td>
</tr>
<tr>
<td>Correct/incorrect branching alternatives for T / F</td>
<td></td>
</tr>
<tr>
<td><strong>Algebraic operations</strong></td>
<td>SOE = number of operators + number of operands</td>
</tr>
<tr>
<td>Wrong types such as an integer instead of a real number</td>
<td></td>
</tr>
<tr>
<td><strong>Function calls</strong></td>
<td>SOE = 3p + nret + 1</td>
</tr>
<tr>
<td>Missing parameters, Return values wrong</td>
<td></td>
</tr>
<tr>
<td><strong>Assignment</strong></td>
<td>SOE = 1 for constant and single variables else its considered algebraic</td>
</tr>
<tr>
<td>Incorrect/Missing/Misplaced/Typo assignment of a variable Eg A = temp when it should be A = temp + 1</td>
<td></td>
</tr>
<tr>
<td><strong>Input/Output errors</strong></td>
<td>For each I/O SOE = 2</td>
</tr>
<tr>
<td>Incorrect handling of input/output variables E.g. Output should be 5 and it outputs 6</td>
<td></td>
</tr>
</tbody>
</table>

Where,
- nvar = number of variables
- noperators = number of operators
- p = number of parameters
- nret = number of return values
With this information, the testing or inspection priority of paths can be chosen strategically starting with the most error prone paths to the less error prone ones for testing or inspection. This can be done to a desired extent e.g. as a percentage, and to the satisfaction of the testing coverage requirements of the software. It must be noted that the design of the test cases is not part of the error prone path identification. Testing and inspections are tools used after the pre-processing of potentially error prone regions.

The problem is to find a suitable searching method that can be applied to identify which paths are the most error prone by using the SOE framework as a fitness criteria.

The following sections describe search techniques and methods of applying the SOE fitness criteria to identify regions of potential error proneness.

### 2.2 Searching Algorithms

The following sections introduce the second key area for the research that is searching algorithms. In these sections current search algorithms used in software reliability and engineering are examined with both their positive and negative aspects in relation to this domain.

As discussed in the previous section, a searching method for applying the assessment of error proneness [SIT00] is required. There are many search algorithms used for different applications. The problem is to determine which is the most appropriate algorithm to use in the domain of fittest path selection. The other issue that needs addressing is how to convert the source code into an environment to apply the search algorithm.
2.2.1 Preparation of the Search Environment

This section identifies the domain of search (or the search environment) and the process of achieving this search environment.

To apply a search algorithm to find the most error prone paths the source code being examined must be transferred into an environment for searching. For simplicity a tree structure is used to represent the executing sequences in the software [BEI90]. The tree structure is used because it effectively shows the program paths and these program paths can be traversed with the search algorithm. For storage the tree is further refined to a sparse connectivity matrix because. This is because the graph poses a storage problem that increases with \( O(N^2) \) complexity.

The details of the parser implemented for this thesis can be found in the Methodology section. Figure 17 outlines the steps required to generate the search environment.

In this example the source code is parsed for structure and the directed graph is derived. An instruction (now a node) can have the SOE from more than one construct type. For example, in node 6 the SOE comes from a “while” and a “conditional”. Next, the tree building rules are applied to transform the graph into a binary search tree. Where \( A = 1,2,3; B = 4,10,11; C = \text{inserted node}; D = 5,10,11; E = 6; F = 8,9,10,11; G = 7,8,9,10,11; \) Next, the tree is represented as a connectivity matrix and finally the matrix is converted to a sparse representation to save on storage.

Once the matrix based search environment is established, the next task is finding a search algorithm that can apply the criteria of error proneness and find the most error prone paths. In doing this one has to consider the search domain presented. Given the number of paths in software can be extremely large this implies that the search environment is most likely extremely large. A search algorithm must be identified that can be used in larger search environments and find optimal solutions for the search space.
main(){
  1 int number = 0;
  2 cin >> number;
  3 switch(number){
  4 case 1:  break;
  5 case 2:  break;
  6 default: while(number != 0){
    7      cin >> number;
  8    }  
  9       break;
 10}  
11}

Figure 17: Sequence of steps for generating the search environment from source code to sparse connectivity matrix representation.
To this end simple code to matrix search environment steps were investigated, because the intention of this research is to optimise the search for error prone path identification. More sophisticated methods using graph searching and infeasible path identification may be examined in future work.

In the following sections techniques used to search the sparse matrix environment are examined.

### 2.2.2 Simple Search Techniques

In this section the focus is on exhaustive search techniques also known as simple search techniques.

Two simple (exhaustive) search technique commonly used are Depth First Search (DFS) and Breadth First Search (BFS). These techniques are very similar and work by exhaustively searching a domain for a solution. Both algorithms are illustrated in Figure 18 and Figure 19 respectively.

BFS works as follows: BFS starts at a given vertex, which is at level zero. In the first stage, the algorithm visits all vertices at level one. In the second stage, the algorithm visits all vertices at level two. These new vertices, which are adjacent to the level one vertices, are then examined and so on. The BFS traversal terminates when every vertex has been visited. BFS labels each vertex by the length of a shortest path (in terms of number of edges) from the start vertex.
In contrast, DFS starts at a specific vertex, which is at level one. Then the algorithm traverses the graph by any edge incident to the current vertex. If the edge leads to an already visited vertex, then the algorithm uses backtracking (return) to the current vertex. If, on the other hand, the edge leads to an unvisited vertex, then the algorithm will go to the new vertex and become the current vertex. The algorithm proceeds using this algorithm until the end vertex has been reached. At this point the algorithm starts the backtracking. The process terminates when backtracking leads back to the start vertex.

The pseudocode for Breadth First Search and Depth First Search algorithms is shown in TABLE 14 and TABLE 15 respectively.

Breadth first and depth first search are called exhaustive searches, because the search is guaranteed to generate all reachable states before it terminates. By examining the nodes at a particular level before proceeding to the next level this control regime guarantees that the space of possible moves will be systematically examined. Therefore the techniques, guarantee that a solution will be found. This is the key advantage of the exhaustive techniques.
TABLE 14: BREADTH FIRST SEARCH ALGORITHM [KRU97]

BREADTH FIRST SEARCH (G, S)
Input: A graph G and a vertex.
Output: Edges labeled as discovery and cross edges in the connected component.
Create a Queue Q.
ENQUEUE (Q, S)    // Insert S into Q.
While Q is not empty do
    for each vertex \( v \) in Q do
        for all edges \( e \) incident on \( v \) do
            if edge \( e \) is unexplored then
                let \( w \) be the other endpoint of \( e \).
                if vertex \( w \) is unexpected then
                    - mark \( e \) as a discovery edge
                    - insert \( w \) into Q
                else
                    mark \( e \) as a cross edge

TABLE 15: DEPTH FIRST SEARCH ALGORITHM [KRU97]

DEPTH FIRST SEARCH (G, \( v \))
Input: A graph G and a vertex \( v \).
Output: Edges labeled as discovery and back edges in the connected component.
For all edges \( e \) incident on \( v \) do
    If edge \( e \) is unexplored then
        \( w \leftarrow \) opposite \((v, e)\) // return the end point of \( e \) distant to \( v \)
        If vertex \( w \) is unexplained then
            - mark \( e \) as a discovery edge
            - Recursively call DSF (G, \( w \))
        else
            - mark \( e \) as a back edge

It is usually assumed that the space that will be searched is finite. Of course, it can be quite large and the solution may lay thousands of steps away from the start node. Consequently, in practice one would usually specify a maximum depth or moving distance from the starting node beyond which one would not search further. In this case it is unknown without using a SOE analysis where the errors are located or indeed at the node level how the errors reflect on the overall path error proneness.
In addition to these problems this method of search requires considerable memory resources for large software programs. When loading huge matrices, and having hundreds of recursive calls it may not be feasible to store them in conventional computer memory.

There are three inherent problems with these techniques. The first problem is associated with BFS. Due to the problem domain a BFS cannot be undertaken, as it does not follow the flow of software paths through a graph. This can be seen in Figure 18 where the direction shows unfeasible software path directions. Additional techniques to deal with these situations have the effect that they slow down the algorithm drastically. The second disadvantage associated with both these simple techniques as implied by their definitions is their exhaustive searching nature. When a large search space is to be traversed it may not be possible to search in polynomial time. Finally there are memory issues to overcome in large software programs when storing many nodes in recursive program calls.

To overcome the problems posed by exhaustive search algorithms, other methods have been investigated. In the next section local search methods are examined.

2.2.3 Local Search

In this section the potential of local search approaches for the domain of fittest path selection are examined.

There are many local search algorithms; they are all based on one commonality that is the optimisation of the search space using local neighbourhood improvements. These improvements include using greedy (or hill climbing) approaches, where the fittest solution at each decision branch drives the search. Methods, with probabilistic moves using values and parameters as used by simulated annealing [KIR83] or memory based approaches used in Tabu search [GLO89].
Hill-climbing is one such local search algorithm. The idea of hill-climbing is to start at a randomly generated state and move to the neighbour with the best evaluation value. This procedure repeats till the solution is found. Improvements on this technique include using random restarting, if a strict local-minimum is reached then the algorithm restarts at another randomly generated state. This allows the algorithm to escape local minima (non-optimal solutions). The issue with Hill-climbing algorithms is that they have to explore all neighbours of the current state before choosing the move. This can take substantial time.

The Tabu search [GLO89] is another method to avoid cycling and getting trapped in a local minimum. It is based on the notion of the Tabu list that is a special short term memory that maintains a selective history. This is composed of previously encountered configurations or more generally pertinent attributes of such configurations. A simple Tabu strategy prevents configurations of Tabu list from being recognised for the next k iterations. Such a strategy prevents Tabu from being trapped in short term cycling and allows the search process to go beyond local optima. The performance of a Tabu search is greatly influenced by the size of the Tabu list.

Simulated annealing [KIR83] is another searching model. It is applied in association with Monte Carlo [MET49] simulations. These techniques involve generating random numbers that follow a desired distribution used in simulation, and observing that fraction of the numbers obey some property or properties for minimising multivariate functions. The method is useful for obtaining numerical solutions to problems which are too complicated to solve analytically. The term simulated annealing derives from the analogous physical process of heating and then slowly cooling a substance to obtain a strong crystalline structure.

In simulation, a minima of the cost function corresponds to the ground state of the substance. The simulated annealing process lowers the temperature by slow stages until the system “freezes” and no further changes occur. At each temperature the simulation must proceed long enough for the system to reach steady state or thermal equilibrium.
Not all combinational optimisation problems can be annealed to give satisfactory solutions (e.g., the time taken to get a useful result may prove to be unreasonable). A configuration space that is mostly smooth, with gradually flowing hills and valleys is relatively easy to anneal. In contrast, consider a mostly flat landscape with numerous, densely packed holes, each of widely varying depth, but some of which lead to the very best solution. If movements keep falling into these local minima, and as the temperature cools, it will become impossible to climb out.

The advantages of the local search techniques are their speed and low computational overheads. However, there are trade-offs to these advantages. Many problems may be encountered by local search routines. Using the tree in Figure 20 some of these trade-offs can be examined.

![Figure 20: Local search tree showing potential for local minima.](image)

Highlighted in blue is the result of a greedy local search showing the worst solution in the search space. Highlighted in red is the result of a random restart greedy local search showing an improvement in searching.

If using a greedy local search technique which selects the best solution at each decision branch the path selected from this tree would be B-E-K = 100. However, this solution is
in fact the second worst solution in the search space. The reason for this is that the local
neighbourhood selected at the start was promising; however, as it turned out the rest of
the space was not.

Using other local search techniques such as random restart greedy local search can
improve the final solution. This solution works similar to a greedy local search but allows
restarts by flipping or swapping the path taken by the search algorithm. If the initial path
selected was B-E-K then a random flip can be employed to find new solutions. If the
number of iterations in the system was four then the following solution would occur. B-
E-K (flip position one) → A-D-I (flip position two) → A-C-H (flip position one) → B-E-
K (flip position two) → B-F-N = 125. Although an improvement has been made on the
greedy local search technique the final path is not close to the best solution.

This brings us to one of the major problems with local search techniques. Due to the lack
of global sampling capability, local search methods run the risk of being trapped in local
minima. Due to the requirements of this thesis research that is, the most error prone paths
and the way the possible search space can be weighted with many local minima this
method of local search can pose many problems.

Coming back to the error prone path identification, based on these problems of local
minima and the possibility of many local minima, we turn our attention to Genetic
Algorithms which is discussed in the next section.

2.2.4 Genetic Algorithms

This section describes the use of Genetic Algorithms for fittest path selection and outlines
modifications made to adopt GA to variable path length environments.

GA were developed and formalised by Holland [HOL75]. They were further developed
and shown to have wide applicability by Goldberg [GOL89a, GOL89b]. GA are adaptive

71
heuristic search algorithms; they are based on the evolutionary ideas of natural selection and genetics. As such they represent an intelligent exploitation of a random search used to solve optimisation problems.

Although using random strategies, GA are by no means random. Instead they direct the search into the region of higher fitness within the search space. The basic techniques of the GA are designed to optimise a function by simulating processes in natural systems necessary for evolution.

GA simulate the survival of the fittest among individuals over consecutive generations for solving a problem. Each generation consists of a population of character strings that are analogous to the chromosome in DNA. Each individual represents a point in a search space and a possible solution. The individuals in the population are then made to go through a process of evolution. GA are based on an analogy with the genetic structure and behaviour of chromosomes within a population of individuals.

Individuals in a population compete for resources and mates. Those individuals that are most successful in each ‘competition’ will produce more offspring than those individuals that perform poorly. Genes from ‘good’ individuals propagate throughout the population so that two good parents will sometimes produce offspring that are better than either parent. As a consequence each successive generation will become more suited to their environment.

A population of individuals is maintained each representing a possible solution to the given problem. Each individual is coded as a finite length vector of components, or variables, in terms of some alphabet, sometimes the binary alphabet \{0,1\}. To continue the genetic analogy these individuals are likened to chromosomes and the variables are analogous to genes. Thus a chromosome (solution) is composed of several genes (variables).
A fitness score is assigned to each solution representing the abilities of an individual to ‘compete’. The individual with the optimal (or generally near optimal) fitness score is sought. The GA aims to use selective ‘breeding’ of the solutions to produce ‘offsprings’ that are more fit than their parents by combining information from their chromosomes.

The GA maintains a population of $n$ chromosomes (solutions) with associated fitness values. Parents are selected to mate, on the basis of their fitness according to a fitness for a desired purpose criterion, producing offspring via a reproductive plan. Consequently highly fit solutions are given more opportunities to reproduce, so that offspring inherit characteristics from each parent.

As parents mate (share and select chromosomes) and produce offsprings, room must be made for the new arrivals since the population is kept at a static size. Individuals within the population die and are replaced by the newer potentially more fit solutions. Eventually a new generation is created once all mating opportunities in the old population have been exhausted. In this way it is expected that over successive generations better solutions will thrive while the least fit solutions die out.

New generations of solutions are produced containing, on average, fitter genes than a typical solution in a previous generation. Each successive generation will contain more fit ‘partial solutions’ than previous generations. After several iterations, when the population is not producing offspring noticeably different from those in previous generations, the algorithm is said to have converged to a solution to the problem at hand.

The GA cycle is implemented using, selection, which equates to survival of the fittest, crossover, which represents mating between individuals and mutation, which introduces random modifications.
2.2.4.1 Selection

This section describes the selection process of the GA and techniques used to achieve selection.

Selection gives preference to better individuals, allowing them to pass on their genes to the next generation. The goodness of each individual depends on its fitness. Fitness may be determined by an objective function or by a subjective judgement.

Using the weight of SOE as a fitness function, a GA can be employed to find the most error prone paths that is, the paths with the highest SOE. As discussed in the previous sections the advantage of the SOE framework is that the structure of the software, or its components, can be represented as a network, in the form of a connectivity matrix, whose path can be represented as a zero/one binary string.

Figure 21 shows an example of a graph search space, in its tree representation. The edges are labelled with either zero (right branch) or one (left branch). The nodes are arbitrarily identified by alphabetic characters. The numbers in the node are their SOE weights.

The genomes used for the GA are the paths constructed through the connectivity matrix by a string that is made of zeroes and ones. They represent a right (0) or left (1) direction shown on the edges of the tree structure. Using the example from Figure 21 this would give us chromosomes such as the paths A-C-G = 111 shown in red or B-E-K = 011 shown in blue. It is this sequence of zero-and-one genes that is used to develop a chromosome for application with the GA search approach.

There are several types of selection strategies. Perhaps the most popular are roulette wheel, steady state, tournament, elitism and rank selection.
Figure 21: Search tree with zero/one string representation. The genomes used for the GA are the paths constructed through the matrix by a string made of zeros and ones.

The theory behind roulette wheel selection is that each individual is given a chance to become a parent. The chance is proportional to its fitness. It is called roulette wheel selection as the chances of selecting a parent can be seen as spinning a roulette wheel with the size of the slot for each parent being proportional to its fitness. Obviously those with the largest fitness (slot sizes) have more chance of being chosen. Thus, it is possible for one member to dominate all the others and get selected a high proportion of the time.

It works by summing the fitness of all the population members $TF$ (total fitness). A random number $n$ is then generated between 0 and $TF$. The member whose fitness is greater than or equal to $n$ is then returned to the population. Figure 22 shows an example of the fitness proportions for roulette wheel selection.
Figure 22: Roulette wheel fitness proportions. Chromosome 1 has the highest chance of being selected for crossover and mutation with Chromosome 4 the least chance of being selected.

In Figure 22 Chromosome 1 would have the highest chance of being selected for crossover and mutation. The reason is that any number selected roughly between 0-45 would have Chromosome 1 selected. After this process two individuals are selected from the new population for crossover and mutation.

In **steady-state selection** one member of the population is changed at a time. To perform selection a member of the population is chosen according to its fitness. The difference between steady-state and roulette wheel is that for each member of the new population generated there is only one parent selected, but not two. Consequently the selection strength is twice that of the roulette wheel selection and appears twice as fast although it can lose out in the long term because it does not explore the landscape as well as the roulette wheel selection strategy.

**Tournament selection** works by selecting some number of individuals at random from the initial random population and copying the best individual from this group into an intermediate population. Usually this is a pair of individuals but it can be generalised and a group or tournament size can be used.
The theory works by generating a random number $R$, between 0 and 1. If $R < \text{threshold}$ use the first individual as a parent else if $R \geq \text{threshold}$ then use the second individual as the parent. This is repeated to select the population.

In **elitism selection**, the fittest chromosome (or a select few fit chromosomes) are copied to the population in the next generation. The rest are chosen using a roulette wheel approach. Elitism can very rapidly increase performance of GA, because it prevents losing the best found solution to date. The problem as with steady-state is the issue of premature convergence as the population is not kept in a diverse state.

**Rank selection** is an extension to roulette wheel selection. Roulette wheel can have problems when the fitness differs greatly among the population. For example, if the best chromosome fitness is 90% of the entire roulette wheel then the other chromosomes will have a slim chance of being selected, and get stuck in a local minimum. Rank selection first ranks the population and then every chromosome receives fitness from this ranking. The worst will have fitness 1, second worst 2 etc. The best will have fitness $N$ (number of chromosomes in population). Figure 23 illustrates an example of the rank selection strategy.

In this example a proportion is allotted to each of the chromosomes based on their fitness. While this allocates the highest fit solution to the largest portion, it gives a higher probability for other chromosomes to be selected diminishing the chances of staying trapped in a local minimum.
Figure 23: Rank selection example showing chromosome proportion before and after fitness calculation. Before, Chromosome 1 has 90% probability of selection. After ranking, the probability is more evenly proportioned.

2.2.4.2 Crossover and mutation

This section describes several methods used for the crossover and mutation processes in GA. For crossover, two individuals are chosen from the population using the selection operator. A crossover site along the bit strings is randomly chosen. The values of the two
strings are exchanged up to this point. There are many types of crossover, the most common are single point, two point, uniform and arithmetic.

**Single point crossover** works by randomly choosing a crossover point on the chromosome and the genes before the cross over point remain and the remaining genes are taken from the other parent after the selected crossover point. TABLE 16 gives an example of single point crossover using a crossover at gene 3.

<table>
<thead>
<tr>
<th>Chromosome 1</th>
<th>A B C D E F G H I J</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chromosome 2</td>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>Offspring 1</td>
<td>A B C 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>Offspring 2</td>
<td>0 1 2 D E F G H I J</td>
</tr>
</tbody>
</table>

**Two point crossover** works similar to single point however there are two crossover points. The genes remain till the first crossover point then genes are introduced from the other parents until the second point and then remaining genes are left after this point. TABLE 17 gives an example of two point crossover using crossovers at gene 2 and 9.

<table>
<thead>
<tr>
<th>Chromosome 1</th>
<th>A B C D E F G H I J</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chromosome 2</td>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>Offspring 1</td>
<td>A B 2 3 4 5 6 7 8 J</td>
</tr>
<tr>
<td>Offspring 2</td>
<td>0 1 C D E F G H I 9</td>
</tr>
</tbody>
</table>
**Uniform crossover** is slightly different from the first two methods described. It works by randomly selecting genes to be swapped. In this type the genes selected for crossover are swapped from parent to parent.

Table 18 gives an example of uniform crossover using a random crossover at genes 1, 5, 6 and 9.

### Table 18: Uniform Crossover

<table>
<thead>
<tr>
<th>Chromosome 1</th>
<th>A B C D E F G H I J</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chromosome 2</td>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>Offspring 1</td>
<td>A 1 2 3 4 F 6 7 8 J</td>
</tr>
<tr>
<td>Offspring 2</td>
<td>0 B C D E 5 G H I 9</td>
</tr>
</tbody>
</table>

**Arithmetic crossover** works by performing an arithmetic operation on the two selected chromosomes. Unlike the other crossover strategies listed this method only generates one offspring from the parents. It is feasible to perform two arithmetic operations to generate two children if necessary. Table 19 gives an example of arithmetic crossover using the AND operator.

### Table 19: Arithmetic Crossover

<table>
<thead>
<tr>
<th>Chromosome 1</th>
<th>1 0 1 1 0 1 1 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chromosome 2</td>
<td>1 1 0 0 1 0 0 1 1 1</td>
</tr>
<tr>
<td>Offspring 1</td>
<td>1 0 0 0 1 0 0 0 1 0 1</td>
</tr>
</tbody>
</table>
Using Figure 21 as an example, if S1=111 and S2=000 and a single point crossover is used at gene 2 then S1'=110 and S2'=001. The two new offsprings created from this mating are put into the next generation of the population. By recombining portions of good individuals, this process is likely to create even better individuals. This can be seen as S1 = 140 and S1' = 155.

With some low probability, a portion of the new individuals will have some of their bits flipped. This is called mutation. Its purpose is to maintain diversity within the population and inhibit premature convergence. Taking the child chromosome S2'=011, a percentage probably of flipping (mutating) the gene is set. If the chromosome is to undergo mutation a random gene is selected. In this case S2' = 001, where the second bit has undergone mutation.

Bearing in mind that the nodes are weighted with the SOE, a path is explored and its corresponding SOE values read and summed. If it is heavy in SOE (path is highly error prone) then the GA will "reward" it.

The problem is that this method only works in a fixed length environment. If chromosomes of different lengths are mated the result would be gaps in the gene structure.

To overcome this problem Goldberg et al. [GOL89b] examined the use of Messy Genetic Algorithms. The next sub-section describes the process of Messy Genetic Algorithms and how they can be used to resolve issues of variable chromosome lengths.

2.2.4.3 Messy Genetic Algorithms

In this section Messy Genetic Algorithms are briefly described to show how components of the algorithm can be used to solve problems of variable chromosome lengths.
Messy algorithms were not developed for variable length strings but I found that with some adaptations, I can use the technique for variable length GA. This is explained in detail in the methodology chapter.

Messy Genetic Algorithms were first proposed by Goldberg et al. [GOL89b] in the late eighties. These algorithms were developed for a better overall optimisation, by allowing better variability by using parts of the chromosome. This was done to overcome the deficiency of underspecified (missing) genes or overspecified (duplicate) genes with respect to the problem being solved. The process works by using messy chromosomes which are made up of messy genes.

These messy genes are a pair containing the GeneNumber and AlleleValue. The Gene number refers to the position in the chromosome and the allele value refers to the value of that position in the chromosome.

A Messy GA typically has three phases: (i) initialisation; (ii) primordial; (iii) juxtapositional.

During initialisation a population containing one copy of all substrings of length k is created. The expectation is that the recombination will find the proper building blocks and assemble them into optimal solutions.

In the Primordial phase chromosomes are sorted and a fraction of the best chromosome set forms the new population.

During juxtaposition, selection is used together with two operators, cut and splice. A cut splits the chromosome at random positions and a splice attaches two cut chromosomes together. These two operations are the equivalents of crossover in traditional GA. The difference is that the order in which genes appear in the chromosome affects that chromosome’s expression. In the messy GA, missing or duplicate bits require special repair or selection mechanisms to determine what is expressed. These mechanisms
increase the complexity of the system and can potentially introduce biases into the search space.

In Messy, recombination will construct enough of the chromosome for either a random selection of bits or a local search to quickly fill out the rest of the chromosome. To help drive the Messy GA progression to a solution, the least fit individual is eliminated from the population with each successive generation.

When we try to apply the messy algorithms to the paths in software (or its graph representation), we are faced with the problem that we do not know all the paths in the system. Therefore only portions of the Messy GA can be adopted in conjunction with the traditional approaches of the GA. The parts that are adopted are the notions of a cut and splicing technique (crossover) and the recombination process.

The following subsection details examples of GA in software engineering and reliability.

### 2.2.4.4 Applications of Genetic Algorithms in Software Engineering and Reliability

In this section previous research with GA and Software Engineering and Reliability is presented.

Numerous challenges have been tackled and solved by applying GA. More recently GA have been used for a number of software engineering and reliability applications.

Spears and Anand [SPE91] studied the effects of crossover operation in genetic programming. The most important point raised from the experimentation was that population size greatly affects the performance of GA. If the population size is small the solutions are found fast but suffer from inbreeding and subsequent local minima selection. If the population is large the solutions take longer to converge but result in
more optimal solutions. The authors indicate that a balance must be met with the population size.

Hunt [HUN95] presented research into test data input generation for an airbag control system. The approach of this research was the use of a significant portion of the chromosome rather than the whole chromosome. By using only a portion certain tests were given priority in the testing process. The problem was that the method of significance was qualified as significant not validates with quantitative results. In addition the search environments used were extremely small.

Clarke et al. [CLA00] explored the suitability of GA for software engineering problems. Several applications were explored including testing coverage. Although no experimentation was performed the software engineering domain was found suitable for GA only if the problem was first formulated as an optimisation problem.

Jones, Sthamer and Eyres [JON98] examined the optimisation of GA for automatically generating test sets to cover all branches and support fault based testing. To assess the effectiveness of GA five experiments to determine the best strategies for the GA were undertaken. The purpose of these experiments was to find specific GA requirements and characteristics for example, the types of crossover functions and percentage of mutations required.

The author’s results from the experiments found GA can be used to generate high quality test data. The claim was GA probe regions of a system under test that have expectedly high failure rates.

Sthamer, Baresel and Wegener [STH01] continued research conducted in [JON98] and adopted the optimisation techniques of GA to automate complex testing tasks for structural test cast design. The input to the GA was the “correct” test data that would be fulfilling the objectives of the test strategies. Tests revealed that using the GA the area in
which extreme execution time of the system was found could be closely defined and monitored. This process was described as cyclic and can be seen in Figure 24.

![Evolutionary Testing Diagram]

**Figure 24: Evolutionary testing. Shown is the iterative process of going from the initial population to test results by moving through evaluation, survival, selection, recombination and mutation [STH01].**

Figure 24 shows the cyclic structure that Evolutionary Testing goes through, using GA in this process. The process starts by providing an initial population of tests and moving through the process of selecting test data, crossover of testing and mutation, monitoring the results and evaluating the fitness.

Michael et al. [MIC01] also investigated GA for automatic software test data generation. Their work was an improvement on previous work which examined the use of local search techniques. It also examines the effect of program complexity on test data generation by examining generated programs of varying complexity. The authors claimed that previous work had been limited to simple programs and simple test adequacy criteria such as statement coverage. Tests were conducted on 10 (30 LOC programs) which were
based on benchmarks used in previous papers and randomly generated programs of 2000 LOC.

The problem as in several of the methods outlined in this section is that the size of the search environments is extremely small and no description of the randomly generated programs was offered. The next section describes the main problems with the search methods and the proposed search method for this research.

### 2.2.5 Problems with Search Methods

There are many types of search techniques to apply to the proposed search environment. These search techniques can be affected by two typical problems: (i) time to search and (ii) premature convergence.

Preliminary experimentation and comparison of a range of search techniques revealed that GA have a higher likelihood of more optimal solutions in the proposed search environment or suffer less from premature convergence. GA are more than likely to be faster than any of the exhaustive searches such as depth or breadth first. The consulted literature shows that GA have many applications in software engineering problems in particular prioritisation of test data selection and test cases. None of these methods have presented a technique for pre-processing the environments with the purpose to selecting error prone regions for testing and inspection. Applications into this direction are limited only to test data generation.

A typical problem is that GA can suffer from premature convergence and local minima problems [HOL75, GOL89a, GOL89b, SPE91]. Methods for overcoming these problems were discussed earlier in this chapter and include varying selection, crossover, population size and mutation methods.
One of the main problems for this research is that GA are suitable for fixed length representations. As discussed in this chapter a method proposed by Goldberg et al. [GOL89b] known as Messy Genetic Algorithms can be used to help resolve the problems of variable length environments. However the implementation may not be suitable in the current form presented by Goldberg et al. for the purpose of this research. Therefore components from the Messy Genetic Algorithm in combination with traditional GA will be adopted for the methodology and experimentation.

2.3 Chapter Summary

This chapter has summarised the literature survey conducted in the thesis. It investigated areas of software reliability and searching. In software reliability particular emphasis was made on areas of software reliability modelling, testing coverage, error seeding and error seeding strategies, metrics and inspection. For searching the main areas of focus were simple search, local search, GA, Messy GA and the search environment. From this literature survey, clear problems areas have been identified; they are (a) that conventional testing can cover a portion of the software based on the level of reliability required, but not necessarily where most of the errors are; (b) that exponential time is required for testing all paths; (c) that as a consequence cost effectiveness is compromised because the process might not justify the time taken to test; (d) the time to search and (e) premature convergence.

Based on the analysis from the literature review, and the problems identified this led to the problem definition as it was stated in Section 1.2.
**Problem:** Current SQA techniques rely on expert intuition and focus on the entire software program. However it is usually infeasible to inspect or test a large software program and expert intuition is not always correct. With current practice there is no guarantee that the tested or inspected parts of software contain many errors. There is not much knowledge of where the errors are located. The highest density of errors could be in the untested or unreviewed parts. Therefore, it is necessary to develop a quantitative approach for pre-processing test and inspection selection that focuses on identifying the most error prone regions to give these regions priority.

The next chapter presents the Methodology for the thesis and details how this problem will be resolved using the technique proposed.
From the literature review in Chapter 2, a clear problem statement has been derived and broken down into several parts.

In the case of reliability presently the methods suffer from the following problems:

(a) The testing covers only a portion of the software for the level of reliability required;
(b) Exponential time is required for testing all paths;
(c) As a consequence of exponential time the cost effectiveness is compromised because the process might not justify the time taken to test;
(d) There has been little or no research conducted into partitioning a program and prioritising quantitatively the most error prone areas.

In the case of search algorithms the methods suffer from: (a) premature convergence; (b) time taken to search and (c) problems of fixed length representations.

From this problem breakdown the following addressing of the problem has been formulated as was stated in Section 1.2:

**PROPOSED PROBLEM SOLUTION:** To identify the parts of code with highest likelihood of containing errors in an efficient and quantitative way. To study the performance of variable length Genetic Algorithms for the efficient identification of the cluster(s) of most error prone paths in a software program using an assessment of error proneness for modules based on programming constructs. The purpose, to enable an informed choice of prioritising these regions for inspection and testing.

The following chapter outlines the methodology used to address the problem stated for the research. This chapter is divided into 3 sub-sections. The first section details the tools and development environments used for the research. The second section details the
description of the experiments conducted for the research. Finally the experiments are evaluated against the aims of the thesis research.

3.1 Tools and Development Environment

In this section the tools and development environments used for the experiments in the thesis are explained.

This research is based on the philosophy and framework developed by Sitte [SIT00] which does examine the quantification of error prone regions to enable prioritisation in testing and inspection. My research extends this scheme with the application of efficient searching algorithms, that is with GA.

The goal of this research is to develop a GA capable of finding the most error prone regions in a matrix based search environment for optimising software inspection and testing regions. To achieve this goal a number of tools and development environments are required, these are shown in Figure 25. One of the important phases in this research is the data preparation, which is a transformation of source code into the zero/one string chromosomes, and the information that they carry in the form of weights, so that they are suitable to be processed by the GA.

Figure 25 shows the overall methodology for the research. This is structured into software forming two main components’ the parser and the evaluator.

The software must be converted from source code to a connectivity matrix for the purpose of automating the search for the most potentially error prone regions. To this end I have developed a parser to transform the source code into a sparse SOE weighted connectivity matrix that is suitable for searching algorithm computations. This parser has been developed in the C++ programming language.
Figure 25: Methodology structure. A parser translates the source code into a SOE weighted connectivity matrix. An evaluator generates zero/one gene paths and identifies and selects the most heavy weighted clusters with variable length GA using SOE weight as the fitness criterion

The parser operates in several phases or passes. Firstly, the parser determines language handling and execution. This involves parsing the header and source files for source language type that is C, C++, Java, etc. The current implementation of the parser is implemented for procedural C code. This distinction is necessary, as other programming
paradigms, for example object-oriented software require different decomposition approaches in the data preparation.

Next, the library (header) files are tokenised and white space is removed. This allows the identification of user defined functions, as shown in TABLE 20.

**TABLE 20: LIBRARY TOKEN SEPARATION AND WHITE SPACE REMOVAL**

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>#include &lt;string.h&gt;</td>
<td># include &lt;string.h&gt;</td>
</tr>
<tr>
<td>struct VNODE*</td>
<td>struct VNODE * { int name; }</td>
</tr>
<tr>
<td>{</td>
<td>; int test ( int , int ) ; VNODE * createNode ( VNODE * );</td>
</tr>
<tr>
<td>int name;</td>
<td></td>
</tr>
<tr>
<td>};</td>
<td></td>
</tr>
<tr>
<td>int test(int,int);</td>
<td></td>
</tr>
<tr>
<td>VNODE*</td>
<td></td>
</tr>
<tr>
<td>createNode(VNODE*);</td>
<td></td>
</tr>
</tbody>
</table>

Following this a function dictionary is generated. Each function name in the library file is stored in a vector named “dictionary”. A function is characterised by a word followed by a parenthesis. Only functions found in the dictionary will be evaluated. If a code inspector or tester wishes to examine portions of the software this dictionary maps back to those source code areas. Given that only procedural C code is being examined issues such as polymorphism are not a problem.

Next, the source code is tokenised with white space removed as shown in TABLE 21. From this, a list is generated named the source list; it contains the function names and segments the functions into their lines of code (LOC) vectors.
After separation a source list is generated. The source list is built using a linked list and vector structure. The linked list contains the function name segmenting the individual functions and the line of code (LOC) vector (see Figure 26).

![Figure 26: Source list showing function name and attached LOC vector.](image)

The source list is generated by first looking for a function name which is then added to the source list and a null LOC vector is generated. The underlying function code is then
added line by line to the LOC vector. This continues until the function is closed. If a function is read during a sub function state, no list entry will be generated.

Various tokens are used to denote an end of line, these are: [‘{’, ‘;’, ‘:’, ‘}’, ‘)’, ‘else’, ‘do’], depending on the current statement certain tokens can denote an end of line. E.g. if the current statement is a case then the ‘:’ can denote an end of line, if the current token is an ‘if’ then the ‘)’ denotes an end of line. Various stacks are used to determine when statements are ending e.g. bracketStack adds a bracket type (if, else, function, etc) to a stack. Other stacks include the parenStack which adds a left parenthesis to the stack and pops the stack when a right parenthesis is read.

Using Figure 27 it can be shown how the source code is converted into the source list and LOC vector structure.

```c
#include "test.h"
int main () { int a = 1 , b = 2 , c = 0 ;
    c = sum ( a , b ) ;
    if ( c == 3 ) printf ( "C is 3" ) ;
    else printf ( "C is not 3" ) ; } int sum ( int val1, int val2 ) { return val1 + val2 ; }
```

**Figure 27: Expanded source list and LOC vectors. The source code is converted into a vector structure.**

Next, the LOC vectors go through a process of bracket alignment and addition to solve the problem of nested conditions. This stage passes through the LOC vector and
determines if a start bracket should be inserted and when the end bracket should be inserted. An example of this can be seen in TABLE 22.

TABLE 22: ADDING BRACKETS TO LOC VECTOR FOR SOLVING PROBLEMS OF NESTED CONDITIONS

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ boolean a = true, b = true;</td>
<td>{ bool a = true, b = true;</td>
</tr>
<tr>
<td>if (a == true) printf(“A is true”) ;</td>
<td>if (a == true) {</td>
</tr>
<tr>
<td>else</td>
<td>} printf(“A is true”) ;</td>
</tr>
<tr>
<td>if (b == true) printf(“A is false, B is true”) ;</td>
<td>} else</td>
</tr>
<tr>
<td>else</td>
<td>{</td>
</tr>
<tr>
<td>printf(“A is false, B is false”) ;</td>
<td>if (b == true) {</td>
</tr>
<tr>
<td>}</td>
<td>} printf(“A is false, B is true”) ;</td>
</tr>
<tr>
<td></td>
<td>} else</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>printf(“A is false, B is false”) ;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

Following this the switch statements are expanded. This stage passes through the LOC vector and separates a switch statement. This separated switch statement is then broken down into its underlying case statements. At this stage the case statements are expanded to include all source code executed when that case statement is found to be true. At the end of the switch statement the expanded switch is reinserted back into the LOC vector. An example of this can be seen in TABLE 23.

From here SOE’s are calculated per LOC in the LOC vector. If historical information from the test design strategies or previous metrics is presented, weights can be adjusted in
this phase. This allows the technique to be tailored for individual organisations. Figure 28 shows the addition of the SOE vector to the current structure.

**Table 23: Expanding switch statements in LOC vector**

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>switch ( a )</td>
<td>switch ( a )</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>case ‘1’ :</td>
<td>case ‘1’ :</td>
</tr>
<tr>
<td>printf ( “case 1” ) ;</td>
<td>printf ( “case 1” ) ;</td>
</tr>
<tr>
<td>case ‘2’ :</td>
<td>printf ( “case 2 case 3” ) ;</td>
</tr>
<tr>
<td>case ‘3’ :</td>
<td>case ‘2’ :</td>
</tr>
<tr>
<td>printf ( “case 2 case 3” ) ;</td>
<td>printf ( “case 2 case 3” ) ;</td>
</tr>
<tr>
<td>break ;</td>
<td>break ;</td>
</tr>
<tr>
<td>case ‘4’ :</td>
<td>case ‘3’ :</td>
</tr>
<tr>
<td>printf ( “case4” ) ;</td>
<td>printf ( “case 2 case 3” ) ;</td>
</tr>
<tr>
<td>break ;</td>
<td>case ‘4’ :</td>
</tr>
<tr>
<td>default :</td>
<td>default :</td>
</tr>
<tr>
<td>printf ( “default” ) ;</td>
<td>printf ( “case4” ) ;</td>
</tr>
<tr>
<td>break ;</td>
<td>break ;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

**Figure 28: SOE vector addition to the list and LOC vector structure.**
The sources of error are calculated per LOC in a function. This is achieved by running through the LOC vector for each function. The SOE values are calculated using the information from [SIT00] and are based on theoretical potential SOE. Appendix A contains a copy of Sitte’s paper on the framework for SOE calculations. This has been done because the work in this thesis builds upon the SOE framework. The intention is that this thesis can be read and understood in a self-contained way. An example of the SOE calculation can be seen in Figure 29. At this time it is important to remind, that the weights as proposed in [SIT00] can be changed or compounded with probabilities based on actual occurrence statistics. However, the fine tuning of the SOE is not part or the purpose of this research.

```c
#include "test.h"
int main () { int a = 1 , b = 2 , c = 0 ; c = sum ( a , b ) ; if ( c == 3 ) printf ( "C is 3" ) ; else printf ( "C is not 3" ) ; } int sum ( int val1 , int val2 ) { return val1 + val2 ; }
```

Figure 29: Expanded source list and SOE vector. The source code is shown in the LOC vector with its corresponding potential SOE value.

The following equations from [SIT00] are used to calculate the SOE for weighting the potential errors present in the source code. Equation 8 shows the general equation to
calculate the total SOE per module and Equation 9 a hypothetical example showing how this is used to calculate SOE per LOC.

**Equation 8**

\[
SOE_m = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} \times (SOE_{inst})_{i,j}
\]

Where,

- \((SOE_{inst})_{ij}\) is the SOE from the \(j^{th}\) occurrence of the \(i^{th}\) instruction in that module for \(j \geq 0, i > 0\)
- \(a_{ij}\) is the number of times the \(ij^{th}\) instruction is repeated

**Equation 9**

\[
SOE_{LOC} = (SOE_{loop}) + (SOE_{cond}) + (SOE_{a_{lg, ebr}}) + (SOE_{funcaixs}) + (SOE_{assign}) + \cdots + (SOE_{I/O})
\]

The values from TABLE 13 are used on each occurrence of code to calculate their SOE values and thereby the SOE per LOC can be obtained.

From this a source tree is added to the function list. This tree gives path structure to the source code. The first step involves translating the function list and LOC vectors into a directed graph.

The most complicated step of the graph transformation is a function evaluation. Based on the SOE framework by Sitte [SIT00] functions are treated as another statement with their weights added to a node in the graph at the point of the function call. The nodes which make up the tree are structured and are shown in Figure 30.
Figure 30: Adding structure tree to source list in preparation for building the binary tree
This directed graph can then be further transformed into a binary tree which is obtained in three steps.

Firstly, the loops within the graph need to be collapsed both for reducing the combinations of paths and generating a binary tree structure. Loops refer to both looping source statements such as “while” and recursive program calls. This is achieved by applying rules from basis path testing and Beizer’s Loop Tests [BEI95]. In the case of removing loops two of the categories namely “Bypass” (zero times through the loop) and “Once” (single execution of the loop) are used. The reason for adopting “Once” and “Bypass” is that loop structures can be assigned values using the SOE analysis. As shown in the construct in Figure 16 the “for loop” has four SOE.

Next, all multiple outdegree nodes, that is, nodes from which greater than two edges emerge within the graph need to be transformed into sets of binary nodes. This is legitimate in graph theory. The reason for this step is to create a binary searching space for application of the searching algorithm. This is accomplished by adding a small weighted node to the graph.

Finally, the trivial paths (paths that have no branching) need to be collapsed. This step is used to reduce the size of the search space and reduce time spent analysing equivalent SOE path structures. It is achieved by collapsing the trivial paths down into one node. This node then contains a summed SOE of the previous structure for path selection with the GA. This process is achieved in a set of steps shown in Figure 17.

It must be noted that using the SOE as a basis for path calculation that loops and other branching statements are reduced to one statement that is either executed or not. In the case of a complex conditional branch the whole branch is equated as one branch which has the SOE for the whole complex condition. The structure is based on branching to build a binary tree.
The output from the parser is a sparse connectivity representation of the code and its corresponding SOE weights being examined. This matrix is then presented to the GA evaluator for evaluation.

The second stage is to evaluate the structure and SOE weights to identify regions of the most error prone paths. The development environment is based on using an efficient programming language for the GA and search environment development. To this end, Matlab™ is used due to its ease for matrix handling.

There are a number of GA toolboxes available to be used in conjunction with Matlab™; these were examined for their applicability for this research. One toolbox that appeared promising was the GAOT toolbox [HOU95]. However, through preliminary experimentation, the toolbox was found to be too specific and not general enough for this research. The GA was designed and coded from scratch, using variable length Genetic Algorithms.

There are two main problems facing a GA in this environment, (i) variable length chromosomes and (ii) local minima. Strategies for resolving local minima include using different selection, crossover and mutation types have been discussed in Literature review (chapter 2).

In terms of variable length chromosomes, with a standard fixed length GA and balanced tree structure, the evaluation of the chromosomes can occur without problem; this is because all gene values (directions in a tree) are valid. However source code by its very nature of structural path variety cannot be translated into a balanced tree whose paths map to fixed chromosome length. This can be seen clearly in Figure 17 binary tree where paths of different lengths are shown. Therefore, special strategies need to be applied.

In this section my extension and adaptation that was necessary on variable length GA techniques for processing variable paths in the software environment was discussed.
Variable length GA uses techniques to overcome the variable nature of the chromosomes and strategies for evaluation. When using variable length GA the first step is balancing the chromosome. This is achieved by padding the chromosome with a representation character. This effectively splits the chromosome into two parts, one part having the valid genes the other the invalid. This is similar to the cut and splicing techniques of Messy GA outlined by Goldberg et al. [GOL89b].

The second problem is one of chromosome evaluation during the representation and crossover phases. That is, invalid genes (paths) can occur during the crossover. Figure 31 outlines an example of the representation and crossover process.

Figure 31: Variable length representation and crossover. Existing paths are zero/one strings. Shorter paths are padded with a dummy character e.g. “2”, because crossover can create nonexistent paths that need to be recognised by the GA.

In this example two chromosomes of different lengths have been selected for crossover (1) and (011) however, the crossover works only on fixed length representations.
Therefore I insert a dummy character (in this case a “2”) to pad the remaining genes. For this example a single point crossover at gene 1 is used.

In my technique the crossover and mutation phases occur as in fixed length GA. After these stages an evaluation stage determines if a chromosome has undergone change. This is done via a random bit evaluation. Continuing the example, after crossover and mutation, two chromosomes (0 2 2) and (1 1 1) have evolved. The evaluation process determines the validity of these chromosomes. By following a set of rules:

Firstly, if a gene within the chromosome is invalid then a split occurs and the remainder of the chromosome is set to (2). Taking Offspring a (1 1 1) as an example. After the second gene has been examined it would be found to be an invalid gene (path) and the remaining chromosome (1 2 2) will be made invalid.

Secondly, if the gene is already an invalid character then a random (0, 1) gene is assigned. Using Offspring b (0 2 2) as an example the chromosome would be evaluated by checking the validity of the first gene, second etc. This would result in the first gene passing as valid the second gene being assigned a random (0, 1) value and the final gene being assigned a value based on the validity of the previous gene. If the second gene is assigned (0) the last gene would remain as 2 if however, the gene was set to 1 then another (0, 1) value would be assigned to the gene which would result in a chromosome such as (0 1 0). This is similar to the recombination strategy in Messy GA [GOL89b]. By using this process the variable path lengths are removed and the functionality of fixed length GA can be adopted.

In GA it is important to use strategies that are efficient for the case and the data involved. To do this a set of systematic experiments was required, to compare the efficiency of GA strategies, such as gene mutations, chromosome cross-over, selection algorithm (roulette wheel, tournament, etc), and evaluation. This experimentation was done on Griffith University supercomputer.
The reason for developing on the supercomputer was the memory usage and large number of variables and strategies involved. This experimentation was not used for error identification but for the large numbers of variable manipulations for identifying the GA strategies.

After the GA strategies identification, the experiments for error prone path identification with GA were done on a normal Pentium 4 desktop computer. It identifies error prone clusters from the software programs and prioritises them from most to least error prone for inspection and testing. The following pseudocode algorithm (see Table 24) shows the structure of the evaluator and GA used in the evaluation process.

The process works by loading the structure and source of error matrices from the output of the parser. While the overall coverage level of the software is not met, paths are selected with the GA and placed in a pool for evaluation. The content of this pool maps back to the original code using the structure matrix. This identifies the source code so that it can be used in the testing and inspection process if it were selected as highly error prone.

The GA works by a process of initialising the population to random values and then going through a process of selection, crossover, mutation and fitness evaluation. The following section details the experiments and presents the methodology to solving the problems.

### 3.2 Problem Solving Approach

In this section the problem solving approach is discussed with the details of the experiments for this research. The literature review has shown that testing all paths in a large software construct can become intractable for testing and that the inspection process can be subjective. It is a well known fact that in large software only a limited portion of a program can be tested and inspected. The parts that can be examined should be the most
error or fault prone. It has also been shown that it is possible to assess the most error
prone paths in a software program using the assessment of error proneness for modules
based on programming constructs [SIT00].

**TABLE 24: SOURCE OF ERROR EVALUATION USING VARIABLE LENGTH GA**

```
load structureMatrix, SOEMatrix, coveragePercentage, averagePathLength
while (coveragePercentage not met) do
    path ← selectPathWithGA (structureMatrix, SOEMatrix)
    areaForEvaluation ← addPathForEvaluation (path)
    structureMatrix, SOEMatrix ← trimPathFromEnvironments (path, structureMatrix, SOEMatrix)
end while

selectPathWithGA – Genetic Algorithm
set crossoverProbability, mutationProbability, selectionType, crossOverType,
populationSize
population ← initialiseRandomPopulation(populationSize)
path ← fittestChromosome(structureMatrix, SOEMatrix, population)
converge ← evaluatePopulationConvergence(population)

while (population not converged AND generation less than 10000) do
    increment generation
    parents ← selectParentsForMating(population)
    population ← crossover(parents, population, crossoverProbability, crossoverType)
    population ← mutate(parents, population, mutationProbability)
    path ← fittestChromosome(structure matrix, SOE matrix, population)
    converge ← evaluatePopulationConvergence(population)
end while
return path
```
This is followed by SOE calculations applied repeatedly on modules and groups of modules and then by examining the paths in the program with the highest error proneness. The purpose of this approach is to contribute to a more efficient inspection and testing.

To achieve the detection of error prone paths a searching algorithm must be developed using the framework presented in [SIT00] and then paths grouped into the most error prone regions.

The following sections present the proposed experiments and work towards the aims of the thesis and answer the research hypotheses. These experiments have been published or are improvements on our previous publications [SIT00, BIR04, BIR05a, BIR05b, BIR06].

### 3.2.1 Genetic Algorithm Strategies and Search Algorithm Comparison

This section describes the details of the GA experiments and comparisons with additional search algorithms.

As discussed in the literature review the flow of any program can be represented as a graph, e.g. a tree [BEI90] and this graph can be represented as a connectivity matrix which is a suitable form for applying search techniques (refer to Search Algorithms in the literature review).

I use vertical partitioning of paths, as opposed to dividing the program across the control flow. The latter would make little sense for identification of error prone regions. To achieve the vertical partitioning the most likely used or most error prone paths are drawn and used for the GA, using the SOE outlined in [SIT00] as fitness criteria. As discussed earlier, the GA provide the necessary advantages to be applied to this domain that being a
large search environment. To detect the most error prone paths the module (or clusters of modules) SOE are used as weights (costs) of an edge being executed.

The system works by taking an initial program for inspection and testing prioritisation and converting it into a sparse connectivity matrix (see Figure 17). Shown in this figure are the transformation steps to build the connectivity matrix environment by taking an initial graph, reducing, simplifying by removing loops, multiple lead nodes and trivial partial paths. The reduced and simplified matrix maintains the knowledge of its reduction nodes that being the source code lines within the reduced nodes. The reason for this step is to maintain the structure of the system to allow for the SOE weighting of the nodes.

For detecting the most error prone paths the SOE are calculated for the nodes in the connectivity matrix and used as weights. It should be noted that more than one type of SOE can be present. This is because a node can contain more than one instruction. For example each node that contains a nested while and a multiple condition. Because the multiple conditions (or nested loops) are in reality individual instructions, the SOE will be assigned for each instruction statement which in turn is summed for that node.

As discussed in the tools development there are two matrix representations of the system. One matrix is to remember the structure of the program and the other maintains the SOE weights of that structure. The purpose of the structure is for reversing the identified paths back to their source code. The SOE weighted matrix contains the weights for searching.

Consequently a path can be constructed through the structure network by a string that is made up of zeroes and ones. These represent a right or left direction using the structure of the system under examination. Using Figure 21 as an example this would give us a string such as the path A-C-G = 111 or B-F-N = 000. It is this sequence of zero and one genes that is used to develop a chromosome for application with the search approach.

A path is explored and its corresponding SOE values read and summed. If it is heavy in SOE that is if the path has high error proneness then the GA will "reward" it. The idea is
to generate random mutations of paths using crossover and mutation of more fit parents
to generate children that are more highly fit or contain more SOE.

To achieve a desired level of reliability, groups of paths (cluster) can be drawn from the
search environment using the GA search approach. When a path (cluster) is drawn it is
removed from the search space. This can be continued until a desired level of reliability
or coverage of the system has been achieved.

To overcome the problems of local minima, specific experiments have been conducted
using different GA strategies. Many different GA strategies were applied to reduce the
problems of local minima. To evaluate these strategies a top 10% error prone path is
selected within the search environment. This process is repeated fifty times to ensure
statistical validity.

The experiment was conducted on ten procedural C open source examples ranging from
2675 to 16718 uncommented lines of code. TABLE 25 outlines the complexity of the
sample cases used for SOE search.

It is important to note that complexity in this case does not mean cyclomatic complexity,
but the number of decision points relating to the length of the SOE paths. Branches refer
to the number of branching statements (if...else...switch) in the code. Maximum nesting
depth is important as this can greatly affect the number of SOE in a construct and is
therefore represented. Calls refer to user function calls in the software. Conditions refer to
conditional statements (e.g. AND…OR).

Loops are included under branch based on the conversion outlined in Section 3. All
samples are sourced from http://sourceforge.net and from different open source projects.
Many GA strategies were applied to reduce the problems of local minima; they are summarised in Table 26 where strategy refers to the strategy number and variable type the settings for those variables in that strategy.

A set of GA variable strategies were selected based on work by Jones, Sthamer and Eyres [JON98] and Spears and Anand [SPE91]. The strategies included increasing the crossover probability, increasing the mutation probability, using different selection methodologies and changing the size of the population being examined. Spears and Anand found that the size of the population is crucial to the performance of the GA. In general, small populations find good solutions quickly, but are often stuck on local optima. Larger populations are less likely to be caught by local optima, but generally take longer to find good solutions.
The experiments were performed using Matlab™ on one processing node of the Griffith University supercomputer.

To validate the GA strategy three alternative techniques were also tested in comparison to the GA approach these were depth first search, greedy local search and random restart greedy local search. These experiments improve on and relate to our earlier published work [BIR04]. In this earlier work the suitability of GA, depth first search and greedy local search was examined to find the most error prone paths, using the SOE weights in relatively small software examples (see Table 27).
These experiments are shown as they provide key points for additional experimentation on the environments outline in Table 25 and why certain algorithms were not examined in the larger search environments.

In the experiments the depth first search approach had a dual purpose: (i) results of the depth first search were used to determine all the possible paths in the environment, their coinciding fitness and exhaustive time to search; and (ii) to determine the effectiveness of the GA in outperforming an exhaustive search. In these experiments the effectiveness of the GA is measured in selecting a top 10% error prone path.

The second comparison used local search techniques, greedy local search and random restart greedy local search (a greedy search extension that can randomly restart at any point in the local neighbourhood to reduce problems with local minima) to determine the effectiveness of the GA in outperforming a local search technique. This comparison was achieved by taking the average GA search time and setting the local search techniques to run for this time or exiting if it outperforms the GA.

The main threat to validity for this experiment would be the sample set of programs used. Given that the software is open source implies that many different programmers of differing experience levels could work on the project. Selection of these programs was

<table>
<thead>
<tr>
<th>Environment</th>
<th>Branch</th>
<th>Nested Branches</th>
<th>Calls</th>
<th>Multiple Leads</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>9</td>
<td>21</td>
<td>9</td>
<td>977</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>17</td>
<td>17</td>
<td>0</td>
<td>882</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>23</td>
<td>5</td>
<td>0</td>
<td>764</td>
</tr>
<tr>
<td>4</td>
<td>38</td>
<td>26</td>
<td>40</td>
<td>9</td>
<td>1870</td>
</tr>
<tr>
<td>5</td>
<td>45</td>
<td>198</td>
<td>92</td>
<td>19</td>
<td>3151</td>
</tr>
</tbody>
</table>
based on their size (large size preferred) than on their applications purpose. However, this type of programming is becoming more common and there is a change management process that must be adhered to before changes are made. In addition the sample set of programs covers a wide selection of software from a number of different sources.

The next section details the error seeding experiments and proposes the Pareto evaluation process for the thesis.

### 3.2.2 Error Seeding

In this section the experiments with error seeding are described. These have been conducted in both small and large software environments. These experiments use refined GA strategies and results are compared with random path selection (RP) [BIR06]. Random path is often used as a benchmark for testing. This set of experiments involves a range of different settings to determine how many seeded errors would be detected (by testing or inspecting) by selecting 80% of the total SOE potential in a set of sample programs.

It is important to be aware that 80% of potential SOE errors does not equal 80% of the search environment. The question to answer follows closely to the Pareto principle: that is whether most times 80% of errors (seeded errors) would be found by testing or inspecting 20% of the most error prone paths in the software.

For the sake of clarity one must be aware of the following distinction, despite its redundancy: SOE errors are the potential sources of errors, that is a theoretic magnitude that is attributed according to a weighting scheme, that is [SIT00] (or any other). Seeded errors are those that are seeded into the code according to an error seeding strategy. Table 28 summarises the refined GA strategies adopted for the following experiments.
For the experiments a crossover probability of 70% using uniform crossover and a mutation probability of the reciprocal of the path length is used. The selection strategy adopted is Roulette wheel with a population size of 30. These values have been selected based on insight gained from the experiments described in the previous section and previous research [BIR05]. In this previously described set of experiments random error seeding was used in relatively small software programs. These experiments were based on functions (reducing the overall LOC) taken from the open source projects outlined in Table 25. They are shown because they provide key points for the additional experiments. The program structures are indicated in Table 29.

Using the strategies in Table 28 to initialise the GA, three experiments were conducted to compare the effectiveness of the GA vs. the random path selection techniques in larger search environments shown in Table 25. The experiments were done as a worst case scenario, as a best case scenario, and as a realistic case scenario.

In the first experiment (experiment 1) comparisons are made between the performance of GA and random path selection when errors are randomly seeded. The error seeding strategy uses a normalised random function from Matlab™.
The error seeding strategy works by counting the total number of LOC in the search environments and seeding the random number generator with the total LOC and randomly selecting a line of code to be error seeded using the random function. Based on the random nature of the seeding, multiple errors can be seeded to a line of code.

In the second experiment (experiment 2) comparisons are made between the performance of GA and random path selection when errors are seeded to the most potentially error prone functions of the software using the SOE calculated at parse time. In this experiment a modification of a roulette wheel is used for seeding. Each function from the source code has its corresponding SOE analysed and a total is generated. A simulated spin of a roulette wheel occurs and based on the number selected, that function which corresponds to that number is selected for seeding. This module is then seeded via the random error

### Table 29: Random error seeding environment structure

<table>
<thead>
<tr>
<th>Sample</th>
<th>Branches</th>
<th>Nesting Depth</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>28</td>
<td>3</td>
<td>162</td>
</tr>
<tr>
<td>B</td>
<td>24</td>
<td>2</td>
<td>166</td>
</tr>
<tr>
<td>C</td>
<td>42</td>
<td>3</td>
<td>174</td>
</tr>
<tr>
<td>D</td>
<td>52</td>
<td>4</td>
<td>184</td>
</tr>
<tr>
<td>E</td>
<td>42</td>
<td>3</td>
<td>242</td>
</tr>
<tr>
<td>F</td>
<td>32</td>
<td>4</td>
<td>267</td>
</tr>
<tr>
<td>G</td>
<td>62</td>
<td>3</td>
<td>291</td>
</tr>
<tr>
<td>H</td>
<td>81</td>
<td>4</td>
<td>315</td>
</tr>
<tr>
<td>I</td>
<td>89</td>
<td>5</td>
<td>334</td>
</tr>
<tr>
<td>J</td>
<td>98</td>
<td>5</td>
<td>374</td>
</tr>
</tbody>
</table>
seeding strategy outlined in experiment 1. In the final experiment (experiment 3) more realistic error occurrences are used by seeding in clusters following a range of results found from experiments by Fenton et al. [FEN00], Basili et al. [BAS84] and Endres [END75].

As discussed in the literature review (error seeding section) these authors conducted different experiments to determine the ratio of errors and modules, or conversely, the number of modules containing a given ratio of errors. Their experiments revealed that in most cases a small number of modules contain the majority of errors. Chapter 2 outlines the research conducted by these authors and from the averaged errors per module from Table 9, Table 10, Table 11 and Table 12 an error clustering model was derived. Table 30 shows the error distribution percentage for the error clustering model.

The samples shown in Table 30 refer to the experimental samples listed in Table 25. For example if sample A is examined, when the errors are seeded two modules in this case a module is represented by a software function (parser function) will contain (87.48%), four (8.37%), six (2.07%), eight (1.56%), ten (0.40%) and twelve (0.23%) of the total seeded errors. For a second example if sample I is examined, one module (function) will contain (87.48%), two (8.37%), two (2.07%), three (1.56%), four (0.40%) and five (0.23%) of the total seeded errors.

In this experiment with clustered error seeding a comparison is made between the amount of seeded errors that would be found by testing the SOE weighted paths from the most potentially error prone to the least error prone modules. The module error proneness is calculated during the search environment conversion. For this experiment the errors were seeded according to the distribution in Table 30. The lines of code for all modules to be seeded are placed in a pool for seeding. This was done to allow all lines of code in the selected modules an equal chance to be error seeded. It must be noted that the lines of code to be seeded are randomly selected in the chosen modules. An alternative in future work may examine prioritising these lines of code from most to least error prone using the SOE analysis.
All error seeding experiments were conducted on a standard Pentium 4 using Matlab™. All experiments were conducted using 40 seeded errors per KLOC. This number was chosen based on the research by Humphrey [HUM96], which states that non PSP trained people produce around 40 errors per KLOC. The experiments are conducted 50 times and results are averaged.

The main threat to validity for this experiment would be that error seeding is a simulation of errors in the software. In defence of the work the seeding has been done using seeding
distributions derived from real life error data collection, from several researchers working in the areas of error clustering and how these errors appear in source code.

The next section outlines the experiments with path strata and the distribution of errors per path layer.

### 3.2.3 Path Strata

In this section the experiments for examining the SOE-GA performance in different strata of paths and specific clustered error seeding strategies are described. These experiments are published in [BIR05b] and use refined GA strategies from Table 28. The results of these experiments are compared with random path selection.

It seems reasonable that as the number of the paths increases, so too the number of errors in those paths may potentially increase. Given the number of paths it is also reasonable that multiple constructs and specific types of constructs may be present in these paths. The issue is how the errors cluster: do they cluster randomly in the paths themselves or do certain layers of the paths lend themselves to more errors.

To resolve this problem the path strata or the horizontal layers in the paths are examined to identify clusters of errors as more error prone regions.

The question is how to inspect these paths layers based on their SOE. For the experiments the layering will examine a top third, middle third and bottom third split of the chromosome path. That is only a third of the chromosome at a time will be examined for identifying error prone regions. This is to see if more errors are found in different strata of the search paths. This is similar to the technique outlined in [HUN95] where the author examined significant genes. The difference is that unfortunately in that research no quantitative data was presented to assess statistical validity.
To achieve this strata (layer) splitting approach involves several sets of experiments. As in the seeding strategy the goal is to find out how many seeded errors or clusters would be found in selecting 80% of the total SOE potential in a set of sample programs (see Table 25). From this selection of paths containing 80% errors the potential error proneness of the proportioned upper third, middle third and lower third software paths identified by the GA and the number of seeded errors contained in these layers are examined.

By obtaining an average from the 80% selection the number of similar SOE paths present in the sample set can be identified. This is done by examining one standard deviation from the mean.

For the experiments a crossover probability of 70% using uniform crossover and a mutation probability of the reciprocal of the path length is used. The selection strategy adopted is roulette wheel with a population size of 30. These values have been selected based on insight gained from my previous research [BIR04, BIR05a, BIR06].

Using the strategies to initialise the GA, three comparison experiments were conducted identifying the effectiveness of the GA vs. the random path selection technique in identifying errors in the path strata.

In experiment one, the performance in the upper-third layer is examined. In experiment two, the performance in the middle-third layer is examined. In experiment three, the lower-third of the paths are examined. This is achieved by using an averaged path length in the program and making a three way split that is top third, middle third and bottom third.

This set of experiments was conducted on a standard Pentium 4 using Matlab™. All experiments were conducted using 40 seeded errors per KLOC. The errors were seeded as previously discussed using Humphrey’s PSP data [HUM96]. The process was conducted 50 times and results were averaged.
The main threat to validity for this experiment would be that different code has modules located in different path strata. This could influence certain software over others, depending on the applications, where the complicated modules could appear higher in the path strata for some software and lower in the path strata for others.

The next section describes the experiment performed to compare the SOE-GA with traditional inspection.

### 3.2.4 Inspection

In this section the details of the SOE inspection exercise are presented. Appendix B contains a copy of the instructions used by the people in conducting the study.

For the exercise software engineering students were given the instructions in Appendix B. The aim of the exercise was to develop a Quadratic Solver then inspect a pre compile version of the software and a post compile version. From these inspections the software was given to another person and details are recorded. The parameters recorded were the time for inspection and the number of defects detected (pre and post compile). From these details five samples of the software were taken and parsed through the SOE-GA parser.

The parser was set to run and identify paths constituting 80% of program total SOE (approximately 20% of the code). Based on the regions selected by the parser I conducted an inspection of the code at these error prone regions and detailed the errors. The aim was to compare the inspection time using a traditional approach used by the students to that using the SOE-GA. The total number of errors detected comparing both techniques was also of interest and recorded. The goal is to see if the result follows the Pareto rule. That is if 80% of the errors detected by the students found in 20% of the inspected code. The results were averaged for both the student and SOE experiments.

The main threat to validity for this experiment would be that students have been used to
complete the inspection exercise. However, leading researchers on empirical software engineering [KIT01] indicates that students are a valid source for inspections as they are the next generation of software engineers.

The next section evaluates the proposed experiments by linking with the research aims.

3.3 Experiment Selection and Evaluation

In this section the proposed experiments are validated by linking to the aims of the research. This is presented in two sections: search and reliability selection, and evaluation.

3.3.1 Search Experiment Selection and Evaluation

In what follows the aims are addressed in relation to the search requirements of this research and linked to the corresponding experiments for validation. My aims are:

1. Investigate existing techniques for converting a program construct or module into a domain for applying search techniques. This helps identify methods for converting a software module into an environment that allows the application of the GA search technique.

2. Investigate search techniques for identification of error prone paths in a sparse connectivity matrix environment. The aim is to identify search techniques capable of searching for the most error prone paths. Based on these algorithms a comparison can be made to determine which search technique is most successful, or which techniques are more suited to these environments and constraints.
3. Investigate advanced techniques for possible algorithmic optimisation in finding the most potentially error prone paths (global optimum solutions). The reason is to improve the convergence speed and reduce the local minima syndrome (local optimal solutions).

Based on the literature review presented in chapter 2, I have found that existing techniques suffer from premature convergence and time to search. Based on this finding several experiments were set up to determine the most suitable search algorithm for the required search domain. Further experiments were done to optimise the selected search approach.

The first set of experiments involves a comparison of GA and several search algorithms to determine the effectiveness of each algorithm in varying size and complex procedural C open source, after being converted to the matrix form search environments. In this experiment variable length GA is compared with depth first search, greedy local search and random restart greedy local search.

The purpose of the experiments is to gauge how each algorithm performs in the various environments. Two performance parameters were measured: the effectiveness of selected error prone paths, and the time to search (convergence to solutions). The questions to answer are:

- Which algorithm performs best over the varying environments?
- Do certain algorithms perform better in certain environments?
- Does complexity affect certain algorithms?
- Do overheads on certain algorithms impede on performance?

The second experiment involves examining variable length GA strategies to determine the most suitable GA variable combination to adopt for the experiments with SOE. This experiment requires many variable combinations including selection methods (roulette wheel and tournament), crossover strategies (single, two-point, uniform), probability rates of crossover and mutation and finally population size. The purpose of the experiment is to
improve the convergence of the GA algorithm and identify specific strategies or variable combinations that are more effective in varying constructs. The questions to answer are:

- Does complexity aid or hinder the GA variables?
- Do certain variable strategies lend themselves to certain environments?
- What major GA variables allow for successful identification of error prone paths?

3.3.2 Reliability Experiment Selection and Evaluation

In this section the aims that are related to the reliability component of the research are examined and linked to the corresponding experiments for evaluation. My aims are:

1. Investigate existing techniques for measuring reliability, testing coverage and inspection. This is to identify previously developed methods for reliability measurement and identify their positive and negative points. This gives a background of information and techniques currently in use and provides details for comparison.

2. Investigate methods to identify smaller error prone regions for testing and inspection. The aim is to improve the testing and inspection process by focusing on smaller regions which contain potentially the most errors. This allows for an informed decision to be made on more effective prioritisation of constructs for inspection and testing.

3. Investigate methods of error seeding for verification of the Genetic Algorithm search technique. Error seeding in conjunction with Pareto Analysis is a strong indicator on the effectiveness of a given reliability technique. If the search technique can be supported with a Pareto Analysis, that is 80% of the errors are identified in 20% of the code, then the research can be deemed a success.

Again, from the literature review presented in chapter 2, I have found that existing techniques suffer from time to test or inspect, cost effectiveness and effective coverage.
Another conclusion raised is that Pareto analysis and error seeding are a strong indicator on the effectiveness of a given technique at identifying potential errors.

From the literature survey a framework was identified that can aid in optimising the inspection and testing process. The framework selected was based on research by Sitte [SIT00], the SOE framework.

Several experiments were designed to determine the effectiveness of the SOE framework in this problem domain. These experiments were conducted on both small search environments and large open source industry projects.

To verify the potential of the SOE technique an inspection exercise is used. In this exercise thirty seven software engineering students\(^2\) were given the task of developing a quadratic equation solver from a given pseudo code algorithm. The task was split into development, self inspection, peer inspection, compilation and testing. Students were rotated on peer inspection, compilation and testing, such that they would get another student’s work each time. Both time and errors detected were recorded at each stage in the experiment. These results were averaged for all participants.

Five samples of the completed source code were passed through the SOE parser and evaluator. These results were compared with the averaged results by analysing how much time it takes to inspect and test by using areas identified by the SOE framework and the overall defect identification. The questions to answer are:

- Does the SOE method aid in prioritising regions for inspection and testing?
- Can the SOE method be used in a timely and efficient manner?

---

\(^2\) This experiment did not require specific ethical clearance. The result of the ethical clearance request is in Appendix C at the end of this thesis.
The second experiment involves using the SOE technique on commercial open source software programs. This software is based on ten open source procedural C projects ranging in size from approximate 2.5KLOC – 17KLOC.

This experiment set involves examining what types of structures occur at what levels in the search environment and evaluating performance of the SOE technique at different levels in the path strata. The purpose is to identify specific regions in the path layers to determine if different strategies need to be conducted at different levels. The questions to answer are:

- Does SOE perform at varying rates in different levels of the path strata, that is for example, in the top third, middle third or bottom third?
- What types of constructs make up different layers in the path strata and do these constructs affect the performance of the technique?

The third experiment involves error seeding to determine how effective the technique is at detecting seeded errors in three categories. Three sets of experiments were done: (a) with random error seeds; (b) errors seeding to the most error prone functions (best fit), and (c) clustered error seeding.

The purpose of this experiment is to seed known errors into an environment and determine how effective the technique is at identifying these seeded errors by analysing only a proportion of the search environment. The questions to answer are:

- Does the error seeding strategy affect the technique?
- Can consistent results be achieved over a number of different search environments?
- Can 80% of the errors be identified from 20% of the code?
3.4 Chapter Summary

This chapter has detailed the methodology for the thesis and how this methodology links in with the research aims and problem. From this methodology several key experiments have been formulated. The first experiment examines the comparison of search algorithms in terms of effectiveness in identifying error prone regions and also time. The second experiment compares different GA variable strategies for resolving issues of local minima. The third experiment examines how the SOE-GA performs when identifying seeded errors using a number of error seeding strategies. The fourth set of experiments examines the structure of the paths and how the different path strata affect the performance of the technique. The final experiment details a SOE inspection exercise where traditional inspection techniques are compared with regions selected from the SOE-GA.

From all these experiments a conclusion can be drawn answering whether or not the technique is successful in aiding prioritisation of regions for inspection and testing and indeed if GA are successful in identifying regions of error proneness. The next chapter presents the analysis of the results and discussion from these experiments.
4. RESULTS AND DISCUSSION

From the discussions in chapter 2 and designed methodology presented in chapter 3 several sets of experiments were conducted to develop a pre-processing system using variable length GA and sources of potential programming errors. This chapter covers the results and discussion of this research. The results are shown for the following experiments:

- Comparative experiment of GA, depth first search, greedy local search and random restart greedy local search showing both the effectiveness of each algorithm in small and larger software examples and averaged time to search these environments. This experiment outlines quantitatively how effective each technique is at fittest path selection from open source software programs.

- A comparison of different GA strategies for resolving issues of local minima. As indicated in the literature review, GA do suffer from local minima problems. These experiments are designed to identify key strategies to be adopted in varying software structures.

- SOE inspection exercise. This experiment determines the overall potential of the SOE framework and the impact on this domain.

- Path Strata. The results in this experiment allow for improvements in the SOE-GA technique by identifying potential areas for strategy change. It also identifies layers in the paths that contain more errors.

- Error seeding (random, most error prone (best fit), clustered). This set of experiments gives quantitative results and shows how effective the overall technique is at identifying errors using a set of error seeding strategies. From these results the hypothesis and overall problem statement can be answered in the positive or negative.
Discussions to interpret the results are provided in the next chapter. In this chapter the discussions are directed only on the results. The conclusion chapter relates these finding to previous research and identifies strengths and weaknesses from the results.

The following section presents and discusses the results of the search algorithm comparison experiment.

4.1 Search Algorithm Comparison

This section provides the results and discussions in relation to the search algorithm comparison.

In order to determine the most effective algorithm for searching the domain initial experiments were undertaken on small environments. These initial experiments looked at a comparison between GA, depth first search and Random Restart Greedy Local Search (RRGLS). The reason RRGLS is used instead of GLS is that the technique performs better in larger environments due to its restarting nature. This gives a better comparison to the GA technique. The results are based on an average from 50 trials of each search algorithm on each small environment outlined in TABLE 27. The figures show that the complexity of the search environment causes varying results with the GA and RRGLS technique. In particular the local search technique performs quite erratically over the different populations. This can be seen in Figure 32.

The result of the RRGLS search technique at search environment (1) is 100% probability of selecting the top 10% error prone paths compared with 21% probability at search environment (4). This is to be expected as the local search technique suffers from a lack of global sampling and quickly gets stuck in local minima. The GA technique does suffer from the local minima problem however; the results are more consistent across the environments and overall outperform the RRGLS technique.
Figure 32: Search comparison of GA, and search algorithms in small search environments. Local search performs erratically over the environments due to local minima.

The most successful technique is depth first search with a 100% selection in all cases. The reason for this is that depth first search will always select the most error prone path due to its exhaustive nature. Figure 33 shows the time comparison between the techniques.

In the smaller less complex environments (1 and 2) the GA performance is rather slow in comparison with the other algorithms with times of 49.78 seconds and 61.42 seconds. This compared with 2.13 seconds and 13.31 seconds for depth first search, and 1.75 seconds and 3.32 seconds for random restart greedy local search. It was concluded that these results were due to the size of the population used for the GA. A smaller population had the potential to improve convergence time but a higher probability for lower selection results.
Figure 33: Time comparison of DFS, GA and RGLS in small search environments. DFS the exhaustive approach has an exponential growth in time as the size and complexity of the environment increases.

The major point to notice is the average search time for the depth first search approach. The approach increased from a small time on the smaller search domains to greater than 350 times for the larger environments. These results are not unexpected, as depth first search is based on an exhaustive search approach, implying that if the size of the search space increases, so does the time to search it. For these reasons depth first search was dismissed as an alternative algorithm for addition comparison in larger environments.

The main question to answer is how the local search and GA perform in larger more complex examples.
Based on these initial experiments a more detailed analysis was undertaken using much larger search environments. These results are based on the details in Table 25. The results of these experiments are spanning several orders of magnitude. The typical approach of using a logarithmic scale would be meaningless in this case because we are not comparing different softwares from different applications, sizes and programming styles. The software used in these experiments is not an ordered set. The results were normalised between (1.2 and 0) for comparison as left in the original form the comparison would be hard to make due to the large differences in results.

Figure 34 outlines the results from the comparison of GA and the local search techniques in these larger open source procedural C search environments.

![Most Error Prone Path Algorithm Comparison](image)

**Figure 34:** Comparison of the effectiveness of different search algorithms on 10 sample test environments. The GA is more effective at finding the heaviest SOE path than the other searching methods.
In all cases examined the GA outperformed the local search techniques at selecting the most error prone path from the search environments. The length of the bars is related to the number of SOE found in each environment. Some environments such as (D, I and J) had more SOE than others. This is related to the complexity (number of decisions) of the statements and also the level of nesting depth.

The interesting point to note is the varying successes of the search techniques in the different environments. In seven of the ten environments (A, C, D, E, G, I and J) the GA performed far superior to the local search techniques.

From later examination of the search environments and paths selected it was found that some regions within the code contained more SOE than others.

The main weakness of local search techniques as found in the smaller environments is that they get trapped in local minima. This explains their lower performance results. In the other three environments (B, F, H) the results were more similar. Subsequent examination of these search environments revealed that the majority of the paths contained similar SOE and therefore many potential local minima. It was concluded that the GA outperformed the local search in these environments given their ability to find a better solution from a population of points. From these results the variable length GA approach was selected.

The major problem encountered with the GA is local minima or sub optimal solutions, which is a well-known as reported by Holland [HOL75] and Goldberg [GOL89a, GOL89b]. Experiments were conducted to study different strategies to slow convergence or maintain diversity in the GA. These are outlined in the next section.
4.2 GA Strategy Comparison

This section describes the results from the comparison of GA Strategies to resolve issues of local minima.

To achieve more optimal results experiments were conducted with thirty different GA strategies to slow convergence or maintain diversity in the GA. These strategies are listed in TABLE 26.

Some strategies were selected based on work by Jones, Sthamer and Eyres [JON98] and Spears and Anand [SPE91]. The strategy changes included increasing the crossover probability, increasing the mutation probability, using different selection methodologies and changing the size of the population being examined.

The following results are calculated by averaging the results for each strategy over 50 trials and normalising the data between 1 and 0. A number of interesting results emerged from these experiments. In Figure 35 it can be seen that in the current sample domains using higher levels of crossover probability result in more effective selection strategies.

It was found that the average effectiveness of the GA increased by 9% when using the increased crossover rate. This is expected, because an increase in crossover rate increased the probability of getting the fittest solution.

From the experiments it was found that decreasing the mutation probability from the reciprocal of the path length (chromosome length) to the reciprocal of the path length + 6 increased overall effectiveness of the GA by 12%. This is shown in Figure 36.
Increased crossover probability leads to increases in SOE fitness.

While this might seem a paradox at first; it is consistent with the results found by Jones, Sthamer and Eyres [JON98]. If there is too much random diversity in the population gene pool it can affect negatively the population fitness.
In Figure 37 I found that the average fitness of the SOE is drastically affected by the selection strategy. When using the binary tournament selection strategy the result of the selection is 45% less than that of the roulette wheel. The conclusion drawn from these experiments is that the population diversity is not maintained with the binary tournament selection with the consequence that convergence occurred too quickly.

---

3 Binary tournament selects two parents from the mating pool and retains the parent with the highest fitness for reproduction.

4 Roulette wheel selects parents based on a simulated spin of a roulette wheel setup with slots biased by the chromosomes fitness.
Figure 37: Effect on SOE fitness using roulette and tournament selection. Roulette wheel selection is more successful at selecting higher SOE fit chromosome populations.

In Figure 38 I found that uniform crossover resulted in the highest average SOE fitness with a 25% increase over single point and 26% increase over two point crossover. This difference in crossover strategy can be explained by the increase in diversity of the population when using the uniform, over the other two crossover techniques.
Finally, based on work by Spears and Anand [SPE91], experiments with different population sizes was conducted. Spears and Anand found that the size of the population is crucial to the performance of the GA. In general, small populations find good solutions quickly, but are often stuck on local optima. Larger populations are less likely to be caught by local optima, but generally take longer to find good solutions. Our findings shown in Figure 38 corroborated this theory that while the increase in population increased the effectiveness of the selection the time to find the more fit solution also increased. This is reflected in the results where an increase in population has increased the average SOE fitness by 56% and time by 92% from population size 10 to 30.

Figure 38: Effect on SOE fitness by examining three crossover types single, two point and uniform. Uniform crossover results in overall higher SOE chromosome fitness.
Figure 39: Effect on SOE fitness with increasing population size. SOE fitness is greatly affected by the population size increase.

From examining all graphs there is a clear trend towards the complexity of the environment or number of decisions and nesting depth and its relationship on both the SOE fitness and resulting GA selection. The points raised from these results are that the GA SOE search approach is more effective in systems with higher complexity.

The next section outlines the results and discussions from the SOE inspection experiments.

### 4.3 SOE Inspection

This section details the results from the SOE inspection exercise outlined in the Methodology using the instructions from Appendix B.
For the experiment thirty-seven samples were taken from the software engineering students. The results from the students (shown in Table 31) were not surprising.

**Table 31: Average Inspection Results (Students)**

<table>
<thead>
<tr>
<th></th>
<th>Assignment</th>
<th>I/O</th>
<th>Function</th>
<th>Branch</th>
<th>Loop</th>
<th>Algebraic</th>
<th>Conditional</th>
<th>Total Errors</th>
<th>Inspection Time (mins)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre Compile</td>
<td>1.9</td>
<td>1.9</td>
<td>0.9</td>
<td>0.6</td>
<td>0.6</td>
<td>1.0</td>
<td>0.6</td>
<td>5.7</td>
<td>14.3</td>
</tr>
<tr>
<td>Compiled</td>
<td>1.7</td>
<td>1.8</td>
<td>1.1</td>
<td>0.3</td>
<td>0.4</td>
<td>1.0</td>
<td>0.5</td>
<td>5.3</td>
<td>19.1</td>
</tr>
</tbody>
</table>

The students averaged 14.3 minutes for inspecting the pre-compiled code and 19.1 minutes for inspecting the compiled code. The inspecting of the compiled code included running the software and getting the errors in compilation (from the compiler) as well as the manual inspection of the code after changes were made.

The average number of errors found in this time was 5.7 (pre-compile) and 5.3 (compiled).

In comparison when I inspected five samples of the code using the SOE-GA selecting 80% of the total SOE the results were very encouraging.

The average time for inspecting the pre-compiled code dropped from 19.1 minutes with the students to 4.6 minutes using the SOE-GA.

The average number of errors found in this time was 5.4 (pre-compile) and 4.9 (compiled). This is within the 80/20 rule with the SOE-Parser averaging 94% of the errors found.
TABLE 32: AVERAGE INSPECTION RESULTS (SOE-GA)

<table>
<thead>
<tr>
<th></th>
<th>Assignment</th>
<th>I/O</th>
<th>Function</th>
<th>Branch</th>
<th>Loop</th>
<th>Algebraic</th>
<th>Conditional</th>
<th>Total Errors</th>
<th>Inspection Time (mins)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre Compile</td>
<td>1.4</td>
<td>1.4</td>
<td>0.6</td>
<td>0.8</td>
<td>1.0</td>
<td>0.4</td>
<td>0.6</td>
<td>5.4</td>
<td>4.6</td>
</tr>
<tr>
<td>Compiled</td>
<td>0.6</td>
<td>1.4</td>
<td>0.4</td>
<td>0.4</td>
<td>0.8</td>
<td>0.6</td>
<td>0.7</td>
<td>4.9</td>
<td>3.6</td>
</tr>
</tbody>
</table>

This experiment supports that the SOE-GA is successful in identifying regions containing errors. It also lends itself to the inspection process by prioritising regions for inspection as shown in the reduction in time to inspect.

The next section details the error seeding experiments performed.

4.4 Error Seeding

As mentioned in the scope of the project (see Chapter 1) industrial results were not available for the thesis and to overcome this an empirical study was conducted to determine how effective the GA were at selecting error prone regions. As stated error seeding approaches were utilised to gauge how effective the GA was at selecting regions containing errors. Initial experiments were conducted with error seeding on small environments to determine if the GA is capable of identifying errors. These initial experiments were conducted by randomly seeding errors in the lines of code in the small examples identified in Table 29. After seeding the most error prone paths were identified and by hand the selected paths were inspected and tested to determine if a seeded error can be found.

Figure 40 and Figure 41 illustrate the results of the experiment with randomly seeded errors in small environments. It was found that on average greater than 65% of all randomly seeded errors would be detected by testing the SOE heavy paths found with the
GA. This is a good result within 15% of the aimed 80%. It was concluded that nesting depth plays a significant factor in the performance of the GA in selecting the seeded errors. If the nesting depth is too large as shown in Figure 40 and Figure 41 (nesting depth 5) then the number of paths selected from the environment is rather low because a small number of paths contain the majority of SOE. This is good, because once those paths are identified they can be pruned from the search space. This can be continued until a percentage of desired coverage for expected testing is reached, but this time the most error prone paths within the testing coverage is known.

Another feature is that the SOE GA technique is more effective at selecting error prone paths when the environment is more complex or contains more branching. This can be seen in Figure 40 (LOC 162 and 166) and Figure 41 (Branching 24 and 28) where less complexity resulted in lower than expected detection rates. This can be explained by the nature of the GA. In addition a comparison is made between random path selection and GA.

It was found that the GA took on average 148 times longer (71.9 GA vs. 0.49 random seconds) to select the paths but was 4.5 times more effective at selecting SOE heavier paths (1606 GA vs. 357 Random SOE) and 1.6 times more effective at finding paths with randomly seeded errors (65% GA vs. 40% random). It was concluded from these results that even better performance in larger, more complex environments, typical of modern software would be expected.

Based on these conclusions more rigorous experiments were conducted on larger industry software programs (see Table 25). These experiments once again looked at a comparison of GA and random path selection. Results from experiment 1 were interesting. Both techniques performed similar at identifying random error seeds (see Figure 42).
Figure 40: GA performance: effect of increasing lines of code and nesting depth on seeded error detection. The genetic algorithm is more effective at selecting paths containing the seeded errors when the environment is more complex but the nesting depth is kept low.

Figure 41: GA performance: effect of increasing branching and nesting depth on seeded error detection. The genetic algorithm is more effective at selecting paths containing the seeded errors when the environment is larger but the nesting depth is kept low.
Figure 42: Comparison of GA and random path selection in identifying seeded random errors in 10 samples with increasing lines of code. Both techniques perform similar.

The GA averaged 22.5% selection of errors in comparison with random path selection at 23.6%. In our previous work on smaller search environments [BIR05a] GA averaged 65% identification of randomly seeded errors. It was concluded that this rather low performance is due to the GA being too focused (in-bred) in the error prone regions. This lowered the random sampling identification rate.

This is not necessarily a negative outcome as the goal of the exercise is to focus the GA on the most error prone regions. The random path performed slightly better due to its random selection nature.
Results from experiment 2 were more positive. When selecting as a seeding algorithm the most error prone regions (see Figure 43) the GA obtained good results. The average number of seeded errors that would be contained in those paths identified by the GA is greater than 87%. This compared to 16% with random path selection.

![Figure 43: Comparison of GA and random path selection in identifying most error prone seeded errors in 10 samples with increasing lines of code. GA is more effective over random path selection.](image)

In experiment 3, where an error clustering method was used (see Figure 44) the average GA identified paths would contain greater than 81% of seeded errors. In comparison random path selection would contain only 17%. This is a positive result with the GA identifying more than 80% of the clustered errors.
Figure 44: Comparison of GA and random path selection in identifying clustered seeded errors in 10 samples with increasing lines of code. GA is more effective over random path selection.

This is consistent with the work by Enders [END75], Basili and Perricone [BAS84], Adams [ADA84], Hatton [HAT97], Fenton and Ohlsson [FEN00], Chou et al. [CHO01] and Shull et al. [SHU02], that a small number of modules contain the majority of defects and as a general rule of thumb, 80% of a system’s defects come from 20% of its modules.

In both, experiment two and experiment three, the random path performance is significantly lower than the GA performance. This is expected as the random path selection does just that: it samples randomly from the search environment, without any strategy. When errors are seeded to a smaller subset of the search space, the random path selection strategy failed to contain a single error in more than 20% of the runs. In comparison the GA performed 70% better on both experiments, for both the SOE and the clustered error seeding strategies.
Although errors have been weighted using our SOE scheme this method only provides the minimum theoretical SOE, additional information from test design strategies, risk analysis and historical data (if available) can be used to calibrate and fine tune the model by adding or subtracting weights to the theoretical SOE values. Overall the results from the experiments in error seeding in large environments strongly supported the combination of combining the SOE and GA method. The GA obtained an average detection rate over 85% in the clustered and error prone seeding. This answered positively the question whether 80% of seeded errors can be detected by analysing approximately 20% of the code.

By summing the SOE values (weighted or theoretical) for each instruction of a software module, it is possible to quantify the SOE for the different paths that the software can step through in its execution. With this information, the testing or inspection priority of paths can be chosen strategically from the most error prone paths first to the less error prone ones to whatever required extent e.g. as a percentage to the satisfaction of the coverage requirements of the software.

In the following section the results of experiments with path strata layers and error distributions are presented.

### 4.5 Path Strata

This section examines the results from the experiments with partitioned path layers. In the first experiment, the focus is on the top level stratum, that is, the upper third of the paths. Results from experiment one were interesting (see Figure 45).

It was found that 70% of the selected paths have similar potential SOE values. In this proportion the GA selected an average of 27.1% of the seeded errors. From analysing the code related to this strata, the paths on average are in the high level (decision modules) functions with typical code constructs that are less error prone.
In a speculative manner this may be attributed to decision modules and function calls being less complex than multiply nested loops that would appear more in the lower algorithm strata. This can be studied in the future. The conclusions raised from this experiment is that (a) having similar SOE values makes it harder for the GA to select the more error prone regions and (b) the distribution of seeded errors is low due to the code statements in this portion.

Figure 45: Comparison of GA and random path selection performance in identifying seeded errors in the top path layer. Ten samples are examined with the GA selecting 27.1% and random path selection 5% errors

In experiment two the middle stratum is examined. Results from this experiment were surprising (see Figure 46:)

146
It was found that 50% of the paths have similar SOE values. This results in a slightly better distinction between the individual paths containing the SOE. In this portion the GA selected an average of 44.1% of the seeded errors although only 17% are new errors with 27.1% found in the first-third portion. From analysing the code related to these paths, these regions focus more on function calls to lower level algorithms resulting in more distinction in the potential SOE but again low weighted potential for errors. The conclusions raised about this lower than expected error seeding identification compared with the first-third experiment is that the upper stratum contains more shorter paths with higher SOE weighted code instructions.

Figure 46: Comparison of GA and random path selection performance in identifying seeded errors in the middle + top path layers. Ten samples are examined with the GA selecting an additional 17% and random path selection 4.2% errors in this layer.
In experiment three the lower third stratum is examined. Results from experiment three were very positive (see Figure 47).

![Whole Strata](image)

**Figure 47: Comparison of GA and random path selection performance in identifying seeded errors in the whole path strata.** Ten samples are examined with the GA selecting an additional 41.3% and random path selection 7.6% errors in this layer.

It was found that the similar SOE path values lowered to 35%. The GA selected an average of 85.4% of the seeded errors with 41.3% of these errors identified in this portion. From analysing the code related to this portion it is clear why this occurs. On average the majority of algorithms and nested structures occur in this portion. These have higher potential for SOE.

The conclusion from this is that although there is 35% similar weighted SOE paths the majority of the SOE weight are present in this strata layer. This results in higher error
seeds and GA identification. In all experiments the random path performance is significantly lower than the GA as shown in TABLE 33.

**TABLE 33: COMPARISON OF GA PERFORMANCE VS. RANDOM PATH SELECTION PERFORMANCE**

<table>
<thead>
<tr>
<th>Stratum</th>
<th>GA performance %</th>
<th>RP performance %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top 1/3</td>
<td>27.1</td>
<td>5</td>
</tr>
<tr>
<td>Middle 1/3</td>
<td>17</td>
<td>4.2</td>
</tr>
<tr>
<td>Bottom 1/3</td>
<td>41.3</td>
<td>7.6</td>
</tr>
<tr>
<td>TOTAL</td>
<td>85.4</td>
<td>16.8</td>
</tr>
</tbody>
</table>

While the GA is able to identify regions with 85.4% of the seeded error clusters, the random path selection can only find 16.8%. This is expected as the random path selection does just that: it samples randomly from the search environment, without any strategy. Random Path Selection is used as a baseline to the GA searching approach.

This quantitative information from the experiments on horizontal layers can then be used to refine the approach proposed in this thesis by determining where potential search algorithm strategy changes can be made or which regions need adjusted SOE weights.

Last but not least, a word about the SOE weighting [SIT00]. These weights are theoretical values, based on an analysis of what can go wrong in the code instruction. This is not necessary the unique scheme or solution to the identification of error prone regions in software but the SOE scheme can be enhanced by adding probabilities of occurrence, or whatever error classification based on statistical occurrence. This would be different values for different languages or programming environments. What is important
and essential is that there is a weighting scheme based on code instructions, and the use of GA to identify error prone regions, as I have demonstrated in this thesis.

### 4.6 Chapter Summary

This chapter has detailed the results from experiments conducted in the thesis. The goal of these experiments was to develop genetic algorithms using sources of program errors as its fitness function to detect the most error prone regions within a software construct. The primary goal was to find what proportion of these seeded errors would be contained in a selection of the 20% most SOE-heavy paths. The results revealed that an average of 85% of the errors seeded to the most potentially error prone regions are within the 20% most SOE heaviest paths. It was found that the GA was 70% more effective at selecting paths containing seeded errors when compared with the selection capability of random path selection.

The secondary goal was to investigate whether the approach presented outperforms other search techniques both in time and effectiveness. This was achieved with the GA outperforming the local search techniques in the larger environments and the issue of exponential time to search shown in the depth first search algorithm. This addresses the initially stated research problem and gives a strong foundation for the research.

The next chapter concludes the research by linking the results to the aims of the research and to the previous work in this area. Discussions are conducted outlining the strengths, limitations and future direction of the work.
5. Conclusions

In this thesis I have demonstrated that it is possible to use Genetic Algorithms to identify error prone paths in software. This was done by presenting the structure of the software as a directed graph, that is, as a matrix representation, whose nodes are weighted with quantities of potential sources of errors. Specifically designed GA were then used to identify the heaviest paths as the potentially most error prone. They should be given priority attention in SQA activities.

I have demonstrated that the use of GA is consistently superior to other searching algorithms including local search, exhaustive and random path. I have also demonstrated that the results are consistent in randomly occurring errors, as well as clustered errors that were seeded into the software using a range of clustering paradigms. GA was 70% more effective at selecting paths containing seeded errors than random path selection.

I have successfully adapted Messy GA techniques to achieve variable length GA to solve the problem posed to the GA by uneven, unbalanced tree structures that are typical in any graph representation of real life software.

I have also demonstrated that GA are able to identify an average of 85% of the errors seeded to the 20% most SOE heaviest paths or the most potentially error prone. With the GA, I have examined whether the upper strata of software behave differently from the lower strata and found out that lower strata contain more potential sources of errors due to the nature of the code in those modules and the division into more branches at lower strata. I found that in the 85% of error prone paths, the lower third of the paths contained 41.3% of the errors. This clearly identifies that the bottom stratum in the paths is the major contributor to error proneness. This is consistent with the findings by other software engineering authors, that a small number of modules contain the majority of defects. This follows the Pareto Principle.
In this way, I have positively presented support to my initial hypothesis. I have done this by demonstrating that the most important contributor in this research is the strategic identification of error prone regions using variable length Genetic Algorithms, and a Pareto analysis to provide quantitative information for decisions regarding inspection and testing.

With a small experiment I have confirmed that the SOE-GA technique is likely to increase the inspection efficiency with the potential to reduce the inspection time of pre-compiled code by almost 75%.

By using the SOE-GA as a pre-processor one can identify the most potentially error prone regions in the software and give these priority in the testing. This addresses the problems posed by McCabe and Butler, Walton and Poore, Ball and Larus, Petrenko and Bochmann, and Bieman and Schultz, that is testing all paths in a large software program becomes intractable for testing and may not be feasible in polynomial time. An informed selection is necessary. Through my work in this thesis I can provide that necessary information.

By using the selected regions from the SOE-GA one can gain indicative metrics about the software and provide these to the inspectors for aiding the quality activities by allowing focus on the most error prone regions first. This aids in the problem posed by Sitte, Ball and Larus, Fagan, Gilb and Graham, Strauss and Ebenau, Boehm and Elbaum et al., that more often than not, due to project time and budget constraints only certain parts of a program can be tested or inspected. The parts that are examined should be the most error or fault prone.

By using my proposed GA weighted graph search testing and inspection efficiency can be improved. This in turn allows more accurate effort and cost estimations for the software quality assurance activities.
5.1 Strengths of this Research

The strength of this thesis is the method that provides quantitative information for an educated decision of which parts in large software should be given priority in testing and inspection, as opposed to guesswork or experience. This is important because it allows the efficient distribution of effort required in software quality assurance activities. In this way I have brought a solution to a long standing problem, even when my work is the bare beginning of the solution, it implies a potential for the expansion and optimisation of this problem. It can be easily automated and has the potential to be brought to a commercial CASE tool in its own right.

I have presented a pre-processing strategy for transforming the structure of a software’s source code to a means that is suitable to be processed by an AI method, that is, the Genetic Algorithms.

I have found a way for modifying GA-chromosomes to map to variable length situations. This method can be used in other applications where variable lengths of chromosomes are required. In this way, my work contributes to the world of AI because it widens the applicability of this class of GA to variable length.

I contribute with quantitative results to support Sitte’s framework for sources of error which in turn can be expanded to other programming paradigms, or alternative ways of weighting potential errors. It implies a definition and usage of software quality metrics.

5.2 Weakness of this Research

In the current domain and scope of my research I have only used procedural C code. Software development is not all procedural. With many applications in Object Oriented program development, questions arise whether this technique is suitable in that environment. In theory there should be no hindrance in analysing Object Oriented
structures in the same way as the Procedural. Sources of Programming Errors still apply and a search environment can still be constructed. The questions relate to how complex the code modules are in terms of nesting depth, number of branches, etc and if errors are still distributed towards a small subset of modules.

The other area of limitation is, as explained in the scope of the thesis, that only those areas covered by Sitte’s research have been examined. My research has not investigated non-feasible path regions, or additional error types as potential sources of errors. This would improve performance of the system and allow more metrics to be used in identifying the most error prone regions.

Finally my experimentation in this research is limited to procedural C open source projects. It is well known that many open source projects are developed by professional developers and all changes made go through an inspection and testing process. Unfortunately software developers are not only very secretive about their code and error data, but also somewhat reluctant to cooperate with researchers outside their own company on unfamiliar projects whose success is not visible at the beginning. The technique would benefit from an analysis of the full Software Development Lifecycle. Where errors detected during inspecting using the SOE technique can be related back and again during the testing phases. This would give quantitative results in relation to the time and cost savings from using this technique over current techniques.

Overall and in defence, the technique does show undisputable results and success in identifying over 80% of the error prone regions based on a Pareto analysis and from the literature this is a key factor in improving software reliability.
5.3 Future Work

The discussion in the weaknesses of the research and findings from the results opens challenges for future work.

The immediate opportunity for future research would be translating this technique into an Object Oriented analysis and on multiple languages. This would benefit the research community as there is currently very limited research into error proneness identification. Using the SOE-GA technique in this environment would provide much needed metrics and quantitative results to be used by other researchers.

Another opportunity for future research would be the application of the SOE-GA approach to a project in the scope of the full development lifecycle. This would give benefit to the technique and lay the foundations for the types of metrics to gather (as they relate to the SOE) given that metrics gathering is a common issue in industry. It would also give companies another option in developing reliable software by improving their estimation techniques with historical data and regions of error proneness. From this a cost benefit analysis can be undertaken, where the costs of existing techniques are compared with the costs of the SOE-GA and the benefits in terms of errors identified and overall reliability can be measured.

Yet another opportunity for future research would be developing the software into an integrated quality management package where source code can be presented with historical metric data to the system. From this inspection forms with the areas of high error proneness can be delivered along with test cases for those most error prone regions. Using this information testers or inspectors could focus on the most potentially error prone regions in an automated fashion.

Finally, the current literature suggests that more metrics and case studies are required to identify regions for prioritisation. This theory and research presented lends itself to
software quality metrics and case studies in industrial settings and sets a strong foundation for further research in this area.

### 5.4 Summary of the Thesis and Conclusion

In conclusion, I have presented a thesis on the research for error proneness identification using Sources of Programmer Errors and Genetic Algorithms.

Testing all paths in a large software program becomes intractable for testing and may not be feasible in polynomial time. Software inspections require all instructions to be addressed at least once. The reality is that typically, due to project time and budget constraints only certain parts of a program can be tested or inspected. The parts that are examined should be the most error or fault prone. Therefore, the goal would be to seek an algorithm which, given indicative metrics about programs, is able to identify error prone regions and provide this information to the inspectors and testers for informed decisions and enhance the efficiency of their software quality assurance activities.

The core of my methodology is a search based system utilising variable length Genetic Algorithms for identifying error prone regions. A range of experiments were conducted to study and demonstrate the applicability of the technique to a set of industry based open source C software programs. These results have been published in [SIT00, BIR04, BIR05a, BIR05b, BIR06].

The experiments were conducted on ten Open Source C programs using previously developed and refined GA strategies to investigate the effectiveness of the SOE-GA technique. The goal of the experiments was to find (a) how many seeded errors would be contained in a selection of the 20% most SOE-heavy paths and (b) how many errors is in each stratum of the selected paths, and (c) whether performance in finding error prone regions of the GA is different in higher or lower strata of the paths of a program.
The results revealed that an average of greater than 85% of the errors seeded to the most potentially error prone regions are within the 20% most potentially error prone paths, and that the bottom stratum in the paths are the major contributor to error proneness. The GA was consistently more effective than other commonly used searching algorithms.

Overall this thesis makes a contribution to the efficiency of inspection and testing in my endeavour for better software reliability.
6. REFERENCES


7. APPENDIX A PAPER

A Framework for Quantifying Error Proneness [SIT00]

A Framework for Quantifying Error Proneness in Software

Renate Sitte
Griffith University, Gold Coast Campus
School of Information Technology
PMB 50 Gold Coast Mail Ctr.
Gold Coast Qld 9726
Australia
r.sitte@mailbox.gu.edu.au

Abstract

This paper proposes a framework for assessing quantitatively the error-proneness of computer program modules. The model uses an information theory approach to derive an Error Proneness Index, that can be used in a practical way. Debugging and testing take at least 40% of a software project's effort, but do not uncover all defects. While current research looks at identifying problem-modules in a program, no attempt is made for a quantitative error-proneness evaluation. By quantitatively assessing a module's susceptibility to error, we are able to identify error prone paths in a program and enhance testing efficiency. The goal is to identify error prone paths in a program using Genetic Algorithms. This increases software reliability, aids in testing design, and reduces software development cost.

Acronym: SOE Source of error  
EPI Error Proneness Index

1. Introduction

In many industries product maturity and perfection is largely achieved through automation. Increasingly more software activities can nowadays be automated, but not all. The development of software is not an automated process, it still has to be done by humans and is therefore subject to defects. This leads to reliability problems ranging from nuisance bugs to software crashes with all its consequences. The industry strives for early defect detection practices and better testing strategies. Pareto analysis (known as 80/20 rule) [1] has repeatedly shown that only few modules are causing most of the problems. Techniques to recognize such modules are being researched [2,3]. A variety of quality indicators for design were developed, but they are aimed at software architectures and module interaction at a higher level. While such quality indicators give a general perception of the software design quality, they do not provide any quantitative information about the likelihood of possible errors in a module.

Testing comes at a high price: debugging and testing take about 40% of effort in a software project. Safety critical software - where human lifes depend on such as aircraft navigation or medical equipment - requires additional quality assurance. If we can replace error prone programming constructs with equivalent, less error prone constructs, we are able to reduce the cost of incurring in safety critical errors.

Testing is not to show how good the program works, but to uncover defects. More testing means more defect-free software. But testing does not uncover all defects. Software released for use containing only 1 defect per 100 lines of code is considered as very high quality. A study at Hitachi Software found that to produce such software, requires one test case for every 10 - 15 lines of source code [4]. But the sequence of program instructions matters too. For example, a nested "while loop" requires more testing than two consecutive loops. Depending on the program it may not be necessary to have the loops nested.

In the early seventies Dijkstra [5] first recommended theorem proving to verify the correctness of algorithms leading to techniques known as formal methods [6]. In the eighties clean-room strategies [7], analogous to those used in integrated circuit fabrication, were introduced. While both methods are successful in reducing the amount of defects, they are labor intensive and require specialized training in techniques that are not defect-free themselves. It is time-wise impossible to fully test each instruction even in medium sized software, and consequently testing strategies have been designed. McCabe's cyclomatic complexity [8] calculation helps to ensure completeness in finding all possible paths or test cases in individual software modules (white box testing), although some of the paths may never be tested. All the above methods have in common that they examine an algorithm for correctness after its design. In other words,
defects are "weeded out" a posteriori. In this process, new defects are introduced. Current software engineering techniques focus on re-use and object oriented programming to reduce the number of defects, but they cannot eliminate the need for algorithmic design. While re-use is based on proven and supposedly defect free software, it has not been assessed whether for example object oriented programming produces less defects than procedural programming.

Definitions: A Software defect is an improper or unacceptable (static) existence in software and may lead to one or multiple faults or failures [9]. A fault (or bug) is an accidental condition that causes a functional unit to fail to perform its required function. A fault if encountered, may cause a failure. An error is a human action with results in software containing a fault [10]. A defect refers to the software, while the fault refers to the consequence of the defect(s), and the error to actually doing the mishap.

The set of steps (the algorithm) to perform a certain task in a computer program module can usually be implemented in more than one way. Essential algorithms are well documented since the early seventies [11] and their choice is often a matter of program efficiency. Such well known algorithms would introduce very few - if any - defects to a program, but they are not the focus of interest in this paper. We focus on the programming constructs in general, as a choice and sequence of instructions in a program.

No method has been developed to measure error-proneness of original (as opposed to changed) source code constructs. There are rules of thumb such as expected defect rates per lines, or multiples of lines. It is not known which programming constructs are more prone to error than others, not even if at all some constructs are more error prone than others. Our suspicion is based on experience, but not on quantitative assessment. To make a clear assessment we require a quantitative way of measuring error proneness at a low level.

This paper proposes a framework to assess the error-proneness for a module, based on programming constructs. It does not intend to give "another method of measuring complexity", rather to provide the information of error proneness a priori, giving the opportunity of choice for alternatives at the time of design or coding. The SOE assessment is suitable for automation. The goal is to use module error proneness for detecting the most error prone paths in a program using Genetic Algorithms (not presented in this paper).

The implications and benefits of a quantitative of error-proneness of program constructs at module level are manifold. Here are the most important ones:

- With module error-proneness information we can identify error prone paths in software and increase the testing efficiency.
- We can refine software reliability prediction and the amount (effort, cost) of testing that will be required.
- This in turn improves cost calculations and cost predictions.
- The (theoretical) SOE value can be compared with actual defect records, providing a metric for process quality.

2. The framework for quantifying error proneness

In this section we explain the framework for calculating error proneness in a software module. By error proneness we understand the number of possibilities of introducing an error at the time of coding. We call these sources of error (SOE). The model does not intend to quantify the probability of software failure (software reliability) or possible consequences of SOE. At this stage, no assessment is made on human tendencies to incur in some errors more than others. They depend on individual cases and thus need to be empirically determined before they can possibly be incorporated as weights in the error-proneness calculations.

In this paper we use the term algorithm to indicate a sequence of programming constructs to accomplish a given task in a programming module.

The fundamental question is: how can we distinguish between more or less error prone algorithms? In other words, how can we find ways to quantify the error-proneness and establish the superiority of one algorithm over another? We propose to do this by analyzing the potential source(s) of error(s) in each sequence of instructions that are used to implement an algorithm in a software module. The values are then linked to calculate a figure (number) of error-proneness for that algorithm. This paper builds upon works by McCabe and Zuse [8, 12].

The basic procedure is to find the SOE for the different instructions in a module, then add these together to a module-SOE. This information is then used to calculate the error proneness of modules and the whole program. From the executable sequences of modules we can calculate the error proneness of a path.
2.2 SOE Analysis

To begin, we focus only on errors within one module. We intend to quantify the SOE by counting the instances in an instruction where a programmer can introduce an error. We do not count those errors that would be detected by a compiler. To investigate the "proneness" to error, we analyze each programming construct. As a simple example we look at a loop. It can have

1. no precondition to enter the loop
2. no condition to advance in the loop
3. no end condition
4. bound exceeded

This construct has four SOE (if those were the only possible sources of error). For simplicity we give equal chance for each error to happen.

All other programming constructs have to be analyzed in a similar way. We do this with information from test design strategies, in particular from control structure testing. For some constructs it is possible to find a general expression to quantify their SOE, but others are case dependent, in particular control structures. Table I gives a sample of basic programming constructs SOE and how they are calculated. This table is by no means exhaustive.

From the examples in this table, we can see that some of these SOE opportunities can belong to more than one category of programming constructs; also that some SOE calculations could be the source for lengthy disputes. For the purpose of the paper it is not too relevant whether one or another category has one SOE too many, or one missing. What is important is to see the principle behind it. For the purpose of application, consistency is always the wise approach in the case of ambiguity. The general equation to calculate the total SOE per module (SOEm) for code with n types of instructions where each of them can appear in p variations is as shown in equation 1:

$$SOE_m = \text{SOE} = \text{SOE}$$

<table>
<thead>
<tr>
<th>SOE opportunity</th>
<th>SOE value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Conditionals</strong></td>
<td></td>
</tr>
<tr>
<td>a) Boolean operator error (incorrect/missing/extra)</td>
<td>SOE = nvar + noperators</td>
</tr>
<tr>
<td>b) Boolean variable error</td>
<td>SOE = 2nvar (approximation)</td>
</tr>
<tr>
<td>c) Boolean parenthesis error</td>
<td>nvar = number of variables</td>
</tr>
<tr>
<td>d) Relational operator error</td>
<td>noperators = number of operators</td>
</tr>
<tr>
<td>e) Arithmetic expression error (can be considered as a special case of (a) and (b))</td>
<td>(Boolean or numeric)</td>
</tr>
<tr>
<td><strong>Compound conditions</strong></td>
<td></td>
</tr>
<tr>
<td>several conditions ANDed or ORed</td>
<td>SOE = nvar + 2nvar</td>
</tr>
<tr>
<td><strong>Loops</strong></td>
<td></td>
</tr>
<tr>
<td>a) Wrong initialization</td>
<td>&quot;while&quot; loops: SOE = 5</td>
</tr>
<tr>
<td>b) Does not enter loop</td>
<td>&quot;for&quot; loops: SOE = 4</td>
</tr>
<tr>
<td>c) No progress in loop</td>
<td></td>
</tr>
<tr>
<td>d) Cannot reach end cond.</td>
<td></td>
</tr>
<tr>
<td>e) Goes beyond loop end</td>
<td></td>
</tr>
<tr>
<td>f) Wrong exit condition (can be considered as a conditional or a loop error)</td>
<td></td>
</tr>
<tr>
<td><strong>Nested loops and conditionals</strong></td>
<td></td>
</tr>
<tr>
<td>SOE compounded for example: SOE = 5nesting_depth</td>
<td></td>
</tr>
<tr>
<td><strong>Branching</strong></td>
<td></td>
</tr>
<tr>
<td>correct/incorrect branching alternatives for T, F</td>
<td>SOE = 2</td>
</tr>
<tr>
<td><strong>Algebraic operations</strong></td>
<td></td>
</tr>
<tr>
<td>e.g. a wrong type such as an integer instead a real number is an operand error</td>
<td>SOE = number of operators + number of operands</td>
</tr>
<tr>
<td><strong>Function calls</strong></td>
<td></td>
</tr>
<tr>
<td>A parameter can be wrong, missing, or extra. Return value and Function name can be wrong.</td>
<td>SOE = 3p + nret + 1</td>
</tr>
<tr>
<td>p = number of parameters</td>
<td></td>
</tr>
<tr>
<td>nret = number of return values</td>
<td></td>
</tr>
<tr>
<td><strong>Assignment</strong></td>
<td></td>
</tr>
<tr>
<td>Assignment by returned value is considered as function call</td>
<td>SOE=1 for constant or single variable, else considered it as algebraic operation</td>
</tr>
<tr>
<td><strong>Input/output errors</strong></td>
<td></td>
</tr>
<tr>
<td>They do not really affect an algorithm, but contribute to wrong results. Any input value (variable or field) can be wrong or missing (unassigned).</td>
<td>For each I/O item SOE=2</td>
</tr>
<tr>
<td>This could be expanded to include wrong filenames, etc.</td>
<td></td>
</tr>
</tbody>
</table>
\[ SOE_m = \sum_{i=1}^{n} \sum_{j=1}^{p} a_{ij} \times (SOE_{\text{inst}})_{ij} \]  

(1)

where \((SOE_{\text{inst}})_{ij}\) is the SOE from the \(j^{th}\) occurrence of the \(i^{th}\) instruction in that module for \(j \geq 0, i > 0\). The \(a_{ij}\) is the number of times the \(ij^{th}\) instruction is repeated. Note that the \(a_{ij}\) occurrences of an instruction do not necessarily accumulate the same number of SOE for different \(j\)s. For example if the \(i\) stands for loops, and the program contains three non nested loops, one while-loop and two for-loops, the SOE contribution from the loops would be \((1 \times 5 + 2 \times 4)\), with \(a_{ij}\) being the 1 and 2 respectively.

For example for a simple module, where each instruction appears only once, we would calculate

\[ SOE_m = (SOE_{\text{loops}}) + (SOE_{\text{conds}}) + 
(SOE_{\text{assign}}) + \cdots + (SOE_{I/O}) \]  

(2)

By applying this and the values from table I to a hypothetical example of a module that has one nested for-loop, one conditional, an algebraic function, one function call, two assignments, and inputs and outputs, we would obtain

\[ SOE_m = (4^{\text{ndepth}}) + (2^{\text{nvar}}) + 
(\text{operands + operators}) + 
(\text{funparams + 2}) + (2) + 
(\text{inputs + outputs}) \]  

(3)

In this case \(\text{ndepth}\) has to be replaced with whatever the nesting depth would be, and \(\text{nvar}\) with the number of boolean variables in its expression, the number of \(\text{operands}\) and \(\text{operators}\) in its algebraic function, and so on. It should be noted that at this point for simplicity we are assuming that the sequence of instructions does not affect the SOE. This is not necessarily the case.

**3. Analytical considerations of module-SOE**

In what follows we will distinguish between the module-SOE and the instruction-SOE when-ever it is necessary.

Analyzing the instruction-SOE shows that nesting loops or complex conditionals contribute more than simple ones. While this is nothing new, we are now able to express quantitatively not only the influence of nesting loops or conditionals, but also to see how much they weigh relative to other instructions. Equation (3) calculates the SOE for a simple program constructed with just seven (or less) different basic instructions, as per equation (2). In this example loops - if more than one - are nested, and conditionals are compound, but not nested. One can see that the dominant terms are the nested loops and conditionals, while the remaining instructions are less dominant.

Although the set of instructions that are used in low level programming is small and common to most languages, their possible combinations are vast, even for small algorithms, and we cannot analyze each of them. As an example, combining two loops, three conditions, and four assignments produces 1260 different algorithms, but some of them are meaningless. Some of them are equivalent to each other in performance, but not in SOE. Also, there is an upper limit to the complexity of an algorithm above which the program becomes un-testable [8].

To illustrate this visually figure 1 shows the SOE distribution for simple hypothetical programs constructed with seven basic instructions, and increasingly nested loops and conditionals.

Figure 1 The SOE distribution for simple hypothetical programs constructed with seven basic instructions, and increasingly nested loops and conditionals.
"combinatorial curse", about 1210104 combinations that would arise from calculating all possible combinations of variable values between 0 and 7 (0 for the case where a particular instruction does not appear in the program, e.g. no function call). The values of those combinations would be between the upper and lower limit symbols.

In this example one can see that the dominating nested loops increasing dramatically over the slower rising compound conditionals. From the analytical model in equation (3) one can see that if we were to include just one nesting depth into the conditional, the sharp rise of SOE would go in both directions.

4. Determining an index for error proneness

SOE focuses only on errors within one module. The natural expansion is then to include the SOE links and interactions with other modules [13]. We wish to find an indicator (a number) that maps error-proneness in a useful, manageable way. To do this we lean on information theory [14], but first we try the probability approach and see why it is not practical as an error proneness indicator in software.

We calculate the probability of introducing an error in any module $P_m$ as the complement of the probability of having a correct module:

$$ P_m = 1 - P_{correct} $$

where $2^{SOE}$ is the probability of having a correct module. An example helps to explain the probability of having a correct module: If we have an instruction with for instance 4 SOE, each of these errors can either occur, or not occur. We have then $2^4$ or 16 possibilities of error for this instruction, in general $2^n$ or $2^{SOE}$ in our considerations. The probability that this instruction is correct is $1/2^{SOE}$, because one of these 16 possibilities is that none of the errors is present.

Errors are introduced into modules independently. We can calculate the total probability of errors for the program $P_p$ as

$$ P_p = P_{m1} \times P_{m2} \times \cdots \times P_{mn} $$

where $P_{m1}$, $P_{m2}$, ..., $P_{mn}$ are the probabilities of errors in each module.

The problem that remains now is that the expressions in (5) provides little perceivable difference for high or low probabilities of error. It is merely a number that even for small modules is close to 1.

We look for a better alternative and define the error proneness index (EPI) as

$$ EPI = \log(\text{variety}_{SOE}) $$

where the variety is the number of discernible states; hence the SOE variety $(\text{variety}_{SOE})$ maps to the SOE, i.e. the error configurations of a module.

We will show that

$$ EPI = \log(2^{SOE}) $$

From the previous discussion we found that the probability for a module to be correct also happens to be the inverse of its SOE variety

$$ P_{m\text{correct}} = \frac{1}{\text{variety}_{SOE}} = \frac{1}{2^{SOE}} $$

we take logarithms

$$ \log(P_{m\text{correct}}) = \log\left(\frac{1}{\text{variety}_{SOE}}\right) $$

$$ = \log\left(\frac{1}{2^{SOE}}\right) $$

and decompose

$$ \log(P_{m\text{correct}}) = \log 1 - \log(\text{variety}_{SOE}) $$

$$ = -\log(2^{SOE}) $$

from this we can see that the error proneness index (EPI) we are looking for, is indeed

$$ EPI = -\log(P_{m\text{correct}}) = \log(\text{variety}_{SOE}) $$

as stated at the beginning.

This form is consistent with the definition of information. We can define the error proneness of a program or module as the negative logarithm of the module's probability of being correct, with information
based on its sources of error. This form also penalizes high SOE, and "rewards" low SOE per module, which was not possible in the probabilistic form.

The table below gives a trivial but illustrative example with simple numbers for 4, 8, 12 and 16 SOE. While using $P_m=1 - \frac{1}{2^{SOE}}$ shows little difference (all values are close to 1 in the middle column as it would come from simple probability calculation), the information theoretic approach applying EPI (the negative logarithm of its SOE variety), enhances their difference (right column) and becomes a meaningful quantity, at least for modules.

Using the EPI, it is easy to see why modular programming is less error prone than "spaghetti code" as illustrated by the following example: A program with two modules of four SOE gives an error proneness = 1.2, while a program with one large module with 16 SOE gives an EPI = 4.8, which is in accord with our intuition.

The calculation of error proneness proposed here is independent of the lines-of-code count. However, by dividing the EPI by its LOC one can derive interesting results. For example that very compact C code would be more "dense" or error prone than simple code. The concept and quantification of error proneness is suitable to derive software metrics, and in due time to set standards for error-proneness limits.

The discussion throughout this section supports that the SOE are a meaningful and practical for modeling error proneness, because SOE are proportional to the logarithm of the variety as defined in information theory. The proportional constant is the base of the logarithm we chose. By using the logarithmic form in EPI, it is easier to perceive and distinguish error proneness than in the simple, probabilistic approach. In practical applications, one would not calculate the variety, unless wanted for comparisons. Once the SOE for each programming construct are determined, the SOE for a program can easily be calculated in an automated parse. Programmers can use the SOE information for choosing programming constructs efficiently.

5. Conclusions

This paper proposes a framework for quantitative assessment of error-proneness of modules and programs using an information theory approach. The model is simple but powerful. By quantifying the error proneness we can find error prone paths in a program, increase the testing efficiency and improve software reliability predictions. This in turn improves cost calculations and cost predictions. By reducing the amount of testing required, it contributes to a cheaper, more robust software product. The calculation process can be automated and is intended to be used in conjunction with Genetic Algorithms for test coverage. The (theoretical) SOE value can be compared with actual error records, providing a metric for process quality. Experimental SOE data gathering is currently in progress.

6. References

8. APPENDIX B SOE SOFTWARE INSPECTION EXERCISE

This is an individual exercise and it’s important that you follow each step correctly. The goal of the exercise is to develop a program for solving quadratic equations by converting pseudocode to source code and collecting metrics for both errors and time.

<table>
<thead>
<tr>
<th>Start [mins]</th>
<th>Activity</th>
<th>End [mins]</th>
<th>Total [mins]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Step 1 Reading:</strong> As an individual read over the attached page of pseudocode for solving quadratic equations. If there are any problems in understanding the pseudocode please consult your tutor.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Step 2 Coding:</strong> Write source code for the quadratic solver using the attached pseudocode. You may code in (C). This MUST be done in WORD or NOTEPAD. The goal of the exercise is to find errors in converting from pseudocode to source code.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Step 3 Code Swap:</strong> Save a copy of this document with your student number and _PRE appended, eg. 1158773_PRE.doc. Print a copy of your source code and write your student number on top and number the lines of code down the left hand side. Hand this numbered source code to your tutor. This will be randomly handed out and you will receive a page of source code for inspecting.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Step 4 Buddy Review:</strong> Inspect the source code for errors and record these in the error-recording table (pre-compile section). Remember to check for errors in assignments, conditions, loops and logic.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Step 5 Compile and fixing errors:</strong> Compile the source code with changes found in the inspection and record the errors found by the compiler (post-compile section). Fix the errors detected by the compiler and run the program. Save a copy of this document with the student number on the top of the inspected page and PST appended, eg. 1158773_PST.doc.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Step 6 Hand in:</strong> Hand all inspected pages and saved documents to your tutor before leaving the tutorial. This will be used in an upcoming exercise.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total Time
Pseudocode for Quadratic solver

main() : void
A ← 0.0
B ← 0.0
C ← 0.0
X1 ← 0.0
X2 ← 0.0
continue ← N
Put “Do you wish to solve quadratic (Y,N)”, continue
while continue is Y do
A ← 0.0
B ← 0.0
C ← 0.0
X1 ← 0.0
X2 ← 0.0
clearscreen
put “Quadratic Formula Solver”
put “Enter values for A, B, C”, A, B, C
if(A equals 0) then
put “Cannot divide by 0”
else if(B^2 – 4AC is negative) then
put “Negative root”
else
X1 = (-B + sqrt(B^2 – 4AC)) / 2A
X2 = (-B - sqrt(B^2 – 4AC)) / 2A
endif
put “X = X1, X=X2”, X1, X2
put “Do you wish to continue (Y,N)”, continue
endwhile

Additional Information

← = assign
put = print

Quadratic Formula

\[ x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]
Produc No. ______________ (the producer number on the top of the printed page NOT your student number)

ERROR RECORDING TABLE
(See attached page for error types)

<table>
<thead>
<tr>
<th>Pre-compile</th>
<th>Post-compile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line No.</td>
<td>Error Type</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Error Types**
- Assignment (As)
- Input/Output (IO)
- Function (Fu)
- Branching (Br)
- Loops (Lo)
- Algebraic (Al)

Example: You found a Conditional error in source code line 5 (Line No. 5 | Error Type: Co)

ERROR RECORDING SUMMARY (*Total errors*)

<table>
<thead>
<tr>
<th></th>
<th>Assignment</th>
<th>Input/output</th>
<th>Function</th>
<th>Branching</th>
<th>Loops</th>
<th>Algebraic</th>
<th>Conditional</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-compile</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Post-compile</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Error Descriptions

### Assignment:
Incorrect assignment of variable (only for single variable assignments)

<table>
<thead>
<tr>
<th>Want</th>
<th>Correct</th>
<th>Incorrect</th>
</tr>
</thead>
<tbody>
<tr>
<td>A equals 1</td>
<td>A = 1</td>
<td>A = 2</td>
</tr>
<tr>
<td>A equals B</td>
<td>A = B</td>
<td>A = C</td>
</tr>
</tbody>
</table>

### Input/output:
Incorrect/Missing variable assignment or wrong variables displayed (use when error occurs in an I/O statement)

<table>
<thead>
<tr>
<th>Want</th>
<th>Correct</th>
<th>Incorrect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read A from keyboard</td>
<td>printf(&quot;Enter A&quot;); scanf(&quot;%d&quot;, &amp;A);</td>
<td>printf(&quot;Enter A&quot;); scanf(&quot;%d&quot;, A);</td>
</tr>
<tr>
<td>Output A and B</td>
<td>printf(&quot;A=&quot; A &quot;B=&quot; B);</td>
<td>printf(&quot;A=&quot; B &quot;B=&quot; A);</td>
</tr>
</tbody>
</table>

### Function calls:
Incorrect/missingextra parameters or return values or incorrect function name in a function call. (use when error occurs in a function call)

<table>
<thead>
<tr>
<th>Want</th>
<th>Correct</th>
<th>Incorrect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call function with 2 parameters</td>
<td>A = swap(A,B);</td>
<td>A = swap(A);</td>
</tr>
<tr>
<td>Call function with A and B</td>
<td>A = swap(A,B);</td>
<td>A = swap(A,C);</td>
</tr>
</tbody>
</table>

### Branching:
Incorrect branching alternatives for T, F or logic error in branch statement (use when error occurs in if…else…switch statements)

<table>
<thead>
<tr>
<th>Want</th>
<th>Correct</th>
<th>Incorrect</th>
</tr>
</thead>
<tbody>
<tr>
<td>If A less than B then C</td>
<td>If(A &lt; B) C</td>
<td>If(A &gt; B) C</td>
</tr>
<tr>
<td>If end of file then exit</td>
<td>If(EOF) exit</td>
<td>If(!EOF) exit</td>
</tr>
</tbody>
</table>

### Loops:
Wrong initialization, doesn’t enter, no progress, no end condition, goes beyond end, wrong exit condition (use when error causes problem with a loop)

<table>
<thead>
<tr>
<th>Want</th>
<th>Correct</th>
<th>Incorrect</th>
</tr>
</thead>
<tbody>
<tr>
<td>for 1 to 10 do</td>
<td>for(i=1; i&lt;10;i++)</td>
<td>for(i=0; i&lt;10;i++)</td>
</tr>
<tr>
<td>do while i &lt; 10</td>
<td>while(i&lt;10) i++;</td>
<td>while(i&lt;10) i--;</td>
</tr>
</tbody>
</table>

### Algebraic:
Incorrect/missingextra operators or operands (use when error occurs in an algebraic)

<table>
<thead>
<tr>
<th>Want</th>
<th>Correct</th>
<th>Incorrect</th>
</tr>
</thead>
<tbody>
<tr>
<td>X equals sum of A and B</td>
<td>X = A + B</td>
<td>X = A – B</td>
</tr>
<tr>
<td>X equals sum of A and B</td>
<td>X = A + B</td>
<td>X = A + C</td>
</tr>
</tbody>
</table>

### Conditional:
Incorrect/missingextra Boolean operators (AND, OR, parenthesis etc).

<table>
<thead>
<tr>
<th>Want</th>
<th>Correct</th>
<th>Incorrect</th>
</tr>
</thead>
<tbody>
<tr>
<td>D equals B multiplied by the sum of A and C</td>
<td>D = B*(A + C)</td>
<td>D = B*A+C</td>
</tr>
<tr>
<td>If X and B then C else D</td>
<td>If(X AND B) C Else D</td>
<td>If(X OR B) C Else D</td>
</tr>
</tbody>
</table>
Dear James

I think the reviewers understood that the students are not RAs in the strict sense of the word, but equally it was not clear whether they should be considered participants in human research. It sounds from the below that the students component in the activity was actually part of their teaching & learning activities and so probably meets the scope test discussed in s2 of Booklet 17 of the Manual. This Booklet also provides advice on the subsequent research use of data collected for a purpose that is exempt from ethical clearance. As long as this situation can meet the tests in Booklet 17, the work might be outside of the scope of the University's human research ethics arrangements and so the application should be withdrawn.

Please refer to Booklet 17, and on that basis clarify for me whether or not there is a human research component to this activity. I suggest that you contact the Research Ethics Advisor for your area if you're not sure.

However, as noted in the provisional approval, based upon the application you submitted, if this work is human research there are national standards which must be met in terms of informed consent from participants.

I hope this clarifies matters.

Gary Allen
Manager, Research Ethics
Office for Research
Griffith University
ph: 3735 5585
fax: 3735 7994
email: g.allen@griffith.edu.au
url: www.gu.edu.au/or/ethics

Hi Allen,

Just to answer your question. The students are not research assistants. The exercise undertaken is a tutorial on software inspection for Software Test and Management (2015INT) and has been used for many years in other subjects for the same purpose of software inspections (Software Quality Management, Software Engineering and Advanced Software Engineering).
We used the anonymous summaries of errors and times found by the students to complement simulated data for my research. The participants cannot be identified as we conducted a double blind test with the students choosing a random alphabetic letter and depositing their own results on a floppy disk.

The students were informed before the tutorial that the averaged data will be used in an upcoming exercise and at the end of the tutorial the averaged results were discussed so that all students could get an idea of the class averaged results (which is done at the end of the tutorial regardless).

This same exercise can be done on any people that can program and is not a representation of demographic or any other class or group. The results are only an averaged table indicating the number of errors found and the time to find those errors.

If you have any other queries please feel free in contacting me.

Regards,

James Birt