On Testing Effectiveness of Metamorphic Relations

By

Mahmuda Asrafi

Faculty of Information and Communication Technologies
Swinburne University of Technology
Hawthorn, Victoria 3122, Australia

A Thesis Submitted for the Degree of Master of Science (by Research)

February, 2012
Abstract

Software testing is a very crucial approach for assuring the quality of the software applications. In this century, software is in use in every sphere of life. Hence, it is very essential to guarantee the correctness of the software functionality to a certain degree. Absence of proper methods to assure the correctness of the software can lead to disasters. Many testing methods have been proposed to select test inputs such that faults in the program can be detected effectively. However, several programs are non-testable due to the so-called “oracle problem”. The oracle problem means that either there does not exist a mechanism (called “oracle”) to verify the test output given any possible program input, or it is very expensive, if not impossible, to apply the oracle. Detecting faults in a program without the oracle is one of the major challenges of software testing. Metamorphic testing is an innovative approach to alleviating the oracle problem. In metamorphic testing, metamorphic relations are derived from the necessary characteristics of the system under test. These relations can help to generate test inputs and verify the correctness of the test outputs without the need of an oracle. Their effectiveness has a significant role in metamorphic testing. Hence, it is important to select effective metamorphic relations in metamorphic testing. Different metamorphic relations can generate test inputs with different code coverage. Thus, code coverage of generated test inputs can be considered as a characteristic of the corresponding metamorphic relation. This study aims to find out whether the fault-detection effectiveness of a metamorphic relation can
be determined by its code coverage. Some case studies have been conducted to analyze the relationship between the code coverage and the fault-detection effectiveness of metamorphic relations. The systems for these studies were selected from different domains that require diverse ranges of metamorphic relations being derived. Moreover, efforts have been given to conduct the research in a platform independent way by choosing the systems from different platforms. Automated mutant generator tools have been used to generate mutants (faulty versions of the system under test) for ensuring non-biased mutants in the experiments. Fault-detection effectiveness of metamorphic relations is calculated by applying metamorphic testing on these mutants. Two basic coverage criteria namely line and branch coverage is taken into considerations. Automated code coverage tools are used to collect coverage data achieved by different metamorphic relations. A certain degree of correlation between the code coverage and fault-detection effectiveness of a metamorphic relation was observed. Code coverage attained by metamorphic relations was found to be a good indicator of the fault-detection effectiveness of these relations. Thus, the selection of metamorphic relations with higher code coverage can help metamorphic testing to become more efficient.
ACKNOWLEDGEMENTS

I would like to express my appreciation and gratitude to my supervisor, Dr. Fei-Ching Kuo for her valuable guidance, advice and encouragement throughout the course of this thesis.

I would like to thank Dr. Huai Liu and Professor T Y Chen for their continuous support and suggestions to accomplish my research.

I would like to thank Dr. Robert Merkel for providing permission to use some of their published metamorphic relations in one case study to support this thesis.

I am thankful to Shengqiong Wang and Ling Chen for their contributions in the preliminary experiments. Appreciation goes to Peishi Yong for helping with data collection.

I wish to express my sincere gratitude to my husband, family members and friends for providing me the encouragement to carry out research for the last two years.

Last but by no means least; I would like to thank my dear parents, Md Abul Kasem and Moriam Begum. Nothing I can say to adequately express my gratitude for the adoration, support and encouragement they provided throughout my life. I am grateful to Omnipotent God for the gift of such caring parents. Special thanks to my mother, she was, is and will be beside me all the time.
DECLARATION

This thesis is a presentation of my original research work and it has not been submitted previously, in whole or in part, to qualify for any academic award. Wherever contributions of others are involved, every effort is made to indicate this clearly, with due reference to the literature, and acknowledgment of collaborative research and discussions.

Signature : __________________________
Name : Mahmuda Asrafi
Date : March 27, 2012
Contents

Abstract \hspace{1cm} i

Acknowledgments \hspace{1cm} iii

Declaration \hspace{1cm} iv

Contents \hspace{1cm} v

List of Tables \hspace{1cm} viii

List of Figures \hspace{1cm} x

1 Introduction 1

1.1 Introduction .......................... 1

1.2 Motivation .......................... 2

1.3 Research Questions .................. 3

1.4 Scope ................................ 3

1.5 Contribution .......................... 4

1.6 Outline ................................ 5

2 Preliminaries 6

2.1 Introduction .......................... 6
# List of Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Background Information of Subject Programs</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Number of MRs for Subject Programs</td>
<td>30</td>
</tr>
<tr>
<td>3.3</td>
<td>Fault Detection in the Original Program and Mutant Generation</td>
<td>33</td>
</tr>
<tr>
<td>3.4</td>
<td>Code Coverage Collecting Tools</td>
<td>35</td>
</tr>
<tr>
<td>4.1</td>
<td>Pearson Correlation Co-efficient Values Between Coverage and FDPs</td>
<td>55</td>
</tr>
<tr>
<td>B.1</td>
<td>Fault-Detection Effectiveness of Metamorphic Testing</td>
<td>89</td>
</tr>
<tr>
<td>B.2</td>
<td>Fault-Detection by Individual MRs of Trityp</td>
<td>89</td>
</tr>
<tr>
<td>B.3</td>
<td>Fault-Detection by Individual MRs of TCAS</td>
<td>90</td>
</tr>
<tr>
<td>B.4</td>
<td>Fault-Detection by Individual MRs of KNAPSACK</td>
<td>90</td>
</tr>
<tr>
<td>B.5</td>
<td>Fault-Detection by Individual MRs of QAP</td>
<td>90</td>
</tr>
<tr>
<td>B.6</td>
<td>Line Coverage Achieved by Individual MRs of Trityp</td>
<td>90</td>
</tr>
<tr>
<td>B.7</td>
<td>Branch Coverage Achieved by Individual MRs of Trityp</td>
<td>91</td>
</tr>
<tr>
<td>B.8</td>
<td>Line Coverage Achieved by Individual MRs of TCAS</td>
<td>91</td>
</tr>
<tr>
<td>B.9</td>
<td>Branch Coverage Achieved by Individual MRs of TCAS</td>
<td>91</td>
</tr>
<tr>
<td>B.10</td>
<td>Line Coverage Achieved by Individual MRs of KNAPSACK</td>
<td>92</td>
</tr>
<tr>
<td>B.11</td>
<td>Branch Coverage Achieved by Individual MRs of KNAPSACK</td>
<td>92</td>
</tr>
<tr>
<td>B.12</td>
<td>Line Coverage Achieved by Individual MRs of QAP</td>
<td>92</td>
</tr>
</tbody>
</table>
B.13 Branch Coverage Achieved by Individual MRs of QAP . . . . 92
List of Figures

2.1 Traditional Mutation Analysis \[38\] . . . . . . . . . . . . . . . 22

3.1 Workflow Diagram . . . . . . . . . . . . . . . . . . . . . 27

4.1 Number of Faults Detected for Each Subject Program by MT 40
4.2 Number of Faults Detected by Each MR in TriTyp . . . . . . 40
4.3 Number of Faults Detected by Each MR in TCAS . . . . . . 41
4.4 Number of Faults Detected by Each MR in KNAPSACK . . . 42
4.5 Number of Faults Detected by Each MR in QAP . . . . . . . 42
4.6 Code Coverage for TriTyp . . . . . . . . . . . . . . . . . . . . 44
4.7 Code Coverage for TCAS . . . . . . . . . . . . . . . . . . . . 45
4.8 Code Coverage for KNAPSACK . . . . . . . . . . . . . . . . . 46
4.9 Code Coverage for QAP . . . . . . . . . . . . . . . . . . . . . 47
4.10 Fault-Detection Probability of the MRs for TriTyp . . . . . . 50
4.11 Fault-Detection Probability of the MRs for TCAS . . . . . . 51
4.12 Fault-Detection Probability of the MRs for KNAPSACK . . . 52
4.13 Fault-Detection Probability of the MRs for QAP . . . . . . . 53
Chapter 1

Introduction

1.1 Introduction

There are many approaches of assuring software quality such as formal verification and software testing. The difficulties in proofs and automation limits the use of formal verification (i.e., program proving, model checking) in practice \[^{[33]}\]. Hence, software testing remains the most common method to assure software quality.

Many software testing methods have been proposed to select program test inputs such that faults in the program can be detected effectively. After executing the test inputs, the test outputs are checked against a test oracle (a mechanism which verifies the correctness of the test output of the system under test given any possible input). One challenge to use these conventional software testing methods is that they are not always capable of ensuring the quality of software applications in the absence of a test oracle \[^{[59]}\]. It is difficult to detect errors, faults, defects or anomalies in software applications
because there is no reliable test oracle to indicate what the expected test output should be for arbitrary test input. These kinds of software applications are sometimes referred to as “non-testable programs” [58]. The oracle problem is a fundamental problem in software testing referring to the situation where the output verification mechanism is unavailable or too expensive to apply.

1.2 Motivation

In order to alleviate the oracle problem, Chen et al. [13] proposed an innovative approach namely metamorphic testing. In metamorphic testing, metamorphic relations (MRs) are derived from the necessary characteristics of the system under test. These relations can help to generate test inputs and verify the correctness of the test outputs without the need of the test oracle. Given a list of metamorphic relations, it is crucial to evaluate these relations according to their fault-detection effectiveness (that is, the number of faults detected by the metamorphic relations). Metamorphic relations capable of detecting more faults should be considered as better metamorphic relations. Furthermore, selection of better metamorphic relations can help the testers to save their time and effort in future regression testing. According to Chen et al. [15], metamorphic relations that can cause the program under test to exhibit diverse execution behaviors should have high fault-detection effectiveness. Code coverage measures the percentage of code, exercised by the test inputs and thus reflects the software execution behavior [28]. For instance, test inputs having different execution paths should be able to cover
more parts of the code. On the other hand, two test inputs following similar execution paths will result in a small difference in their code coverage. Metamorphic relations can generate test inputs of different code coverage. Thus, code coverage of generated test inputs can be considered as a characteristic of the corresponding metamorphic relation. Hence, this thesis examines whether different code coverage reflects different fault-detection of an individual metamorphic relation.

1.3 Research Questions

After examining the existing literature [refer to Chapter2], one research question was formed in this study.

- Is there any relationship between code coverage of metamorphic relations and their fault-detection effectiveness?

Answering this question allows us to know whether coverage criteria can be useful criteria of selecting effective metamorphic relations.

1.4 Scope

The scopes of this thesis are stated as below:

The subject programs used in this study are primarily numerical in nature in the domains of geometrical classification, optimization and conflict detection. The input parameters of the numeric programs are numbers or Boolean values.
CHAPTER 1. INTRODUCTION

Code coverage criteria are used in this work as a measurement of execution behavior. Two basic coverage criteria, namely line coverage and branch coverage were used in this study.

Mutant generation is accomplished with automated mutant generator tools. Mujava [42] and MILU [37] were selected from the existing tools available for automated mutant generation.

Random test input generation is used for source test input generation, and follow up test inputs are generated according to their respective metamorphic relations.

1.5 Contribution

Regardless of the presence of an oracle, testing is expensive in most situations [33] because test case design, implementation, test output prediction and comparison, as well as documentation is labor intensive. Thus effective metamorphic relation selection and execution will set off the program verification in a more cost-effective way. The outcome of this research will lead to some regimes to judge the effectiveness of metamorphic relations in accordance with code coverage. The regimes found hereby can be applied to make the testing process more time effective and cost proficient.
1.6 Outline

This study is an attempt to find out some factors which may affect the fault-detection effectiveness of metamorphic relations. The research outcome is presented in this dissertation containing following chapters.

- Related literature review is summarized in Chapter 2.
- Methodology and experimental design are described in Chapter 3.
- Detail analyses of the experimental data, statistical significance and the validity of this work are discussed in Chapter 4.
- Finally, Chapter 5 gives the conclusions and discusses future research directions.
Chapter 2

Preliminaries

2.1 Introduction

This study falls into the field of software testing. This chapter covers the existing literatures that lead the path to perform this research. A detailed discussion is presented on software testing, oracle problem, metamorphic testing, selection of metamorphic relations and impact of code coverage on fault-detection.

2.2 Software Testing

2.2.1 Basics of Software Testing

In general, testing refers to a procedural investigation of the product under test to gather quality-related information about the product. This investigation involves experimentation, logical and mathematical analysis, and so on. According to Glenford Myers, software testing can be defined as the “process
of executing a program or system with the intent of finding errors”, which he
described in his classic book, The Art of Software Testing [50]. At seventies
when this book was written, the definition probably was the best available
that describes the concept of software testing. Software testing has evolved
and become a complicated and systematic process and involved in all the
stages of the software development life cycle, as stated by Myers et al. in
his second edition of The Art of Software Testing published in 2004 [51].
At present, the goal of software testing is not limited to revealing the faults
of the system under test, but also measuring and improving the quality of
software being tested. However, being not entirely extensive, software test-
ing can only advise the occurrence of faults but not their absence. Moreover,
because of the large input domain, various possible input paths and difficulty
of testing design and specification issues, it is impossible to exhaustively test
an application [9]. The popularity of software has kept increasing over the
decades. Software involvement is ever-increasing in critical situations where
single failure will cause unacceptable results. Hence the requirements for soft-
ware reliability, usability, stability, performance and security are an alarming
concern at all the time. All these facts lead to the rapid development of soft-
ware testing methods, processes and tools.

Testing techniques are divided into various categories and the ways to
divide these techniques are versatile. Based on the source of the test input
generation, it can be classified into black-box, white-box and gray-box [43].
Black-box testing usually generates test inputs from functional requirements
of the system. Its application requires no knowledge of code or the structure
of the program. However, in white-box testing, the test inputs are generated from the internal structure of the code. Testing techniques such as boundary value analysis, equivalence partition testing, random testing and ad hoc testing, all belong to black-box testing. Some examples of white-box testing include but not limited to coverage testing, data-flow testing and path testing. Gray-box testing is the combination of both black and white box testing techniques. According to the phase of SDLC (System Development Life Cycle) in which testing is applied, there are five levels of testing, namely as unit testing, integration testing, system testing, acceptance testing, and regression testing [43]. Almost every testing technique compares the test outputs against the expected output to justify the test outcome (pass/fail). In the absence of expected output for each test input, these techniques will not be able to detect faults.

2.2.2 The Oracle Problem and Non Testable programs

In software testing, “test oracle” is the mechanism to verify the correctness of software outputs. A test oracle should be accompanied with an originator, a comparator and the evaluator [39]. Originator offers the expected output for a given test input. Comparator checks the test output against the expected output. Finally, the evaluator verifies whether the system under test has passed or failed the testing. Without the presence of a test oracle, it is very difficult to verify the correctness of test outputs. Thus, the effectiveness of software testing is greatly hindered. This condition is known as the “oracle problem” in software testing.
According to Weyuker [58], program becomes “non-testable” when encountering the following two subclasses of the oracle problem: (1) when test oracle does not exist for a program; (2) when the oracle is potentially available, but the efforts required to get the oracle are unreasonable. Programs in the domains of numerical and scientific computing, optimization and machine learning [13, 31] are typical “non-testable” software systems.

Many researchers have tried to develop some possible approaches to deal with the oracle problem in software testing. Among them, the pseudo-oracle [23] is based on “N-version programming” [4]. Other approaches include the use of assertion checking [21, 41, 46, 20], extrinsic interface contracts [5], algebraic specifications [29, 52, 55], formal specification languages [22, 54, 55] and metamorphic testing [13].

2.3 Metamorphic Testing (MT)

Metamorphic testing proposed by Chen et al. [13] is an innovative technique to alleviate the oracle problem. The following sections will illustrate basic concepts of metamorphic testing, its application, as well as the derivation and selection of metamorphic relations.

2.3.1 Basics of Metamorphic Testing

Metamorphic testing is a property based testing. It tests the system by the necessary system properties, derived from the knowledge of the specification,
algorithm or implementation of the system under test. Instead of checking the correctness of individual test outputs, metamorphic testing checks the properties (called metamorphic relations) among multiple test outputs.

A metamorphic relation is used to convert a test input $x$ to another test input $x'$. It also defines the relation between corresponding test outputs $f(x)$ and $f(x')$. If the test outputs do not satisfy the metamorphic relation, then the system under test must have faults. The procedure of metamorphic testing consists of the following steps:

(i) *Construct metamorphic relations*: Construct metamorphic relations based on the knowledge of the algorithm or specification of the system under test.

(ii) *Create test inputs*: Two types of test inputs are involved in metamorphic testing. Source test inputs can be generated by applying some traditional testing techniques (such as random testing, fault-based testing and so on). Follow up test inputs are generated by converting source test inputs based on metamorphic relations.

(iii) *Execute test inputs*: Both the source and follow up test inputs are executed.

(iv) *Verify test outputs*: Verify the test outputs of the source and follow up tests against their corresponding metamorphic relations.

After the verification of test outputs, if a metamorphic relation is violated (that is not satisfied), a fault is said to be detected in the system. However, if a metamorphic relation is satisfied, it cannot ensure the correctness of
the whole system (alike the other testing techniques that cannot show the absence but presence of the faults).

To provide a better understanding of the steps described above, here is a simple example of metamorphic testing. Consider a sorting program, which sorts a set of integers in ascending order as the system to be tested.

(i) Suppose $x$ is a set of elements to be sorted. If the set $x$ is rearranged in reverse order to yield another input $x'$. The output of the sorting program for $x'$ will be the same as that for $x$ because the arrangement of the input set will not alter their value. This property can be considered as metamorphic relation for this application and this should be validated through multiple executions. Thus, the metamorphic relation can be constructed as, if $x' = \text{reverse}(x)$, then $\text{Sort}(x) = \text{Sort}(x')$.

(ii) Assume, the random generation of source test input provides $x=\{35, 15, 32, 25\}$. Reversing $x$ as a follow up test input will yield $x'=\{25, 32, 15, 35\}$.

(iii) Execute $x$ and $x'$ on the sorting program to obtain $\text{Sort}(x)$ and $\text{Sort}(x')$.

(iv) If $\text{Sort}(x)$ and $\text{Sort}(x')$ are equal, then no fault is revealed. On the contrary, if they are different, then a fault is said to be detected by the metamorphic relation.

Since the proposal of metamorphic testing, it has been used to detect faults in various application domains, such as, scientific programs \[13\], bioin-
formatics systems [14], machine learning systems [59], online service systems [10], network simulation [16] and computer graphics [32].

2.3.2 MT Based Frameworks and Methodologies

Test input generation based on identified metamorphic relations is an important task in metamorphic testing. Previous studies [31, 48] have shown that the processes of test input generation and test output verification in metamorphic testing can be automated. One framework was proposed in [31], where constraint logic programming techniques were used to automatically generate test inputs that violate a given metamorphic relation. An approach called heuristic metamorphic testing is presented in [48] to reduce false positives in test output verification and to address some cases of non-determinism (in cases of non-deterministic applications, such as situations where, it may not always be possible to predict the expected change to the test outputs for a given change in the test inputs) in test outputs. Although metamorphic testing can be applied to individual components, most of the work focuses on system testing by considering the properties of entire systems [13, 14, 48]. Metamorphic relations used in metamorphic testing are the necessary system properties; hence they are useful to verify the correctness of the system under test. A combination of metamorphic testing and symbolic execution namely semi-proving [18], was proposed to partially prove the correctness of a program based on metamorphic relations. Metamorphic testing was also integrated with fault-based testing to prove the absence of certain faults [19].
CHAPTER 2. PRELIMINARIES

2.4 Guidelines for the derivation of MRs

In any software testing approach, semantic knowledge of the program is required for creating test inputs [50]. Even the random testing approach demands that the tester should understand the input and output domains to generate test inputs [34].

Metamorphic testing, similar to other techniques requires that the tester will have enough knowledge of the system under test to identify the metamorphic relations. A metamorphic relation (MR) is a necessary property of the system under test which should be verified over a set of distinct inputs and their corresponding outputs for multiple executions.

Identification of metamorphic relations is a key task of metamorphic testing. Normally, the identification of metamorphic relations is guided by the system specifications and a general understanding of what the system is meant to do. Six types of metamorphic relations were classified in [47] and some specific metamorphic relations for machine learning applications were discussed in [59]. Properties, available in the literature [47, 59] for the identification of metamorphic relations used in metamorphic testing are summarized below. This can provide the testers some general guidelines in identifying metamorphic relations.

In general, numerical systems, which takes numerical inputs and/or produce numerical outputs, implement mathematical calculations. Certain numerical applications, such as sorting, euclidean distance calculation; calcula-
tion of standard deviation suffer from oracle problem (when the number of data to be processed is large). Thus, these types of applications are good contenders for metamorphic testing. Programs like numeric differentiation [19] also suffer for the oracle problem because the calculation takes time and is an error-prone process when it is done manually. The numerical applications share some common mathematical properties which can be taken into account while identifying metamorphic relations for these applications. For most of the numerical applications, the following mathematical properties [47] can be taken into consideration while converting the test inputs in a metamorphic relation.

- Additive: Increase (or decrease) the test input $x$ by a constant $c$ [Refer to MR3 of TriTyp in Appendix A.1].

- Multiplicative: Multiply the test input $x$ by a constant $c$ [Refer to MR1 of TriTyp in Appendix A.1].

- Permutative: Permute the order of values in a test input set [Refer to MR2 of TriTyp in Appendix A.1].

- Invertive: Operate an inversion of each value in a test input set [The relationship of the test outputs may vary on the basis of the particular application. For example, for the sorting program mentioned in Section 2.3.1, the test outputs must not be equal unless all the elements in the source test input is 1].

- Inclusive: Add a new value to a test input set [Refer to MR6 of KNAPSACK in Appendix A.3].
CHAPTER 2. PRELIMINARIES

- Exclusive: Remove a value from a test input set [Refer to MR7 of KNAPSACK in Appendix A.3].

- Merging: Combine some test input sets [Refer to MR9 of KNAPSACK in Appendix A.3]

After the conversion of source test inputs using the above properties to generate the follow up test inputs, change in the test outputs should be predictable depending on the particular application. The relation among the test outputs can be equal, non-equal, greater than, less than and so on.

The above described properties is generally only applicable to programs that deal with numerical inputs and outputs. However, other programs with non-numeric inputs like computational linguistics and discrete event simulation may not have mathematical properties. Metamorphic relations for these programs can be obtained from the specification, algorithm or the implementation. The specification may consist of the user requirements of the functionalities that a system intended to perform. An algorithm is a set of specific logics/instructions, used to achieve the functionalities mentioned in the specification. Again, an implementation is a realization of the technical specification or algorithm as a program, software module or computer system through programming and deployment.

The following properties can be taken into consideration while deriving metamorphic relations for both numeric and non-numeric systems.

- Metamorphic relations can also be derived from the specification of the system under test. Examples can be seen from MR1 to MR3 for TCAS
In order to derive metamorphic relations at the application level, properties of the software algorithm can be considered. Examples can be seen from MR4 and MR5 for TriTyp in Appendix A.1.

To derive metamorphic relations at system level, properties, which are specific to the implementation and the programming language should be taken into consideration. Distributional properties have been used in [45] as metamorphic relations to test image processing and analysis applications. Isotropic properties and checkpoints were used to derive metamorphic relations for testing context-sensitive applications [11, 12].

While deriving metamorphic relations in the domain level, properties shared by all applications operating in the same domain can be taken into consideration. The reason is that there must be some property specific to the nature of that domain. For example, optimization domain shares some common properties as shown in the metamorphic relations for KNAPSACK and QAP [Refer to Appendix A.3 and A.4].

There may remain some properties those are applicable only to certain given inputs. It may be the case when some metamorphic relations of an application will only hold for certain inputs. MR4 to MR11 for TCAS in Appendix A.2 are examples of these kinds of metamorphic relations.
2.5 Guidelines for the Selection of Effective MRs

Metamorphic testing employs different metamorphic relations, which attempt to discover the faults in the system under test. The major task of metamorphic testing is to identify proper metamorphic relations. Mayer and Guderlei [44] conducted case studies to check the effectiveness of different metamorphic relations. They found that metamorphic relations with rich semantic properties are typically strong and proposed that testers should not select metamorphic relations that are too close to the implemented algorithm. Chen et al. [15] conducted case studies for selection of good metamorphic relations where metamorphic testing is applied to implementations of shortest path and critical path algorithms. It was suggested that metamorphic relations that can cause the system to execute through various paths are good metamorphic relations [15]. It was recommended to understand the algorithm of the system under test before selecting metamorphic relations.

2.6 Code Coverage

2.6.1 Code Coverage Basics

Code coverage measures the degree to which the source code of a program has been tested during the test. Software testing needs to define a standard for constituting an adequate test, which was pointed out by an early research conducted by Goodenough and Gerhart [30]. When a set of test inputs (called
test suite) $T$ is used to test a system $S$ against a selected criterion $C$, the test suite $T$ will be regarded as adequate when it satisfies criterion $C$; otherwise it will be considered as inadequate. Suppose that $C$ is the all-statement coverage criterion, then test suite $T$ must cover all the statements of system $S$ at least once during the execution to be considered as an adequate test suite.

Coverage criteria have played different roles in the field of software testing [62]. A coverage criterion can be treated as a quantitative measurement for evaluating different test suites. Furthermore, it can be used to guide the test input generation, that is, repeatedly generating test inputs for a particular program until coverage achieved by the test suite reaches a sufficient level.

In practice, test coverage criteria are enormously available and they can be categorized in several means. Zhu et al. [62] presented one possible framework to group basic coverage criteria in their research paper. The framework was focused on the underlying testing approach and the information source of the test coverage criteria. Test coverage criteria were divided into three groups: structural coverage criteria, fault-based coverage criteria, and error-based coverage criteria. Structural coverage criteria mainly cover the coverage of a certain set of structural elements in the program or the specification. Fault-based coverage criteria are mainly used to measure the fault-detecting ability of test suites. For example, mutation analysis is a fault-based testing technique which provides a testing criterion called the “mutation adequacy score”. The effectiveness of a test suite can be measured by the mutation adequacy score in terms of its ability to detect faults. Error-based cov-
CHAPTER 2. PRELIMINARIES

Coverage criteria use test inputs to check the program in error-prone points. Each group of test coverage criteria could be further divided into two subgroups, namely specification-based and program-based. For the specification-based criteria, test suites are constructed based on the functional features or the non-functional requirements of the software program; whereas for the program-based criteria, test suites are constructed based on the program structure. Information related to program-based structural coverage criteria and fault-based coverage criteria, is needed in this study. The following sections present this information.

2.6.1.1 Program Based Structural Coverage

Program-based structural coverage criteria can be subsequently sub categorized into two categories: control flow coverage criteria and data flow coverage criteria. These two sub-categories are based on the flow-graph model of program structure. Control flow coverage criteria focus on the structural elements of a program such as statements or branches, while data flow coverage criteria pay attention to different use of variables in the program execution.

The control flow coverage criteria include function coverage criterion, line/statement coverage criterion, branch/decision coverage criterion, condition coverage criterion, decision/condition coverage criterion, multiple condition coverage criterion and path coverage criterion [53].

All the interested functions must be covered at least once during the execution of the test suite to fulfil the function coverage criterion.

Again, all the executable statements of the program must be covered at least once during the execution of the test suite in the case of line/statement
coverage criterion.

In case of branch/decision coverage criterion, the branches from all decision points are required to be executed at least once during the execution of the test suite. For an instance, the statement like “if” or “while” has one decision point with two possible branches, namely, true or false. If both branches have been executed by some test input of the program under test it can be considered that this decision is covered.

Similar to branch/decision coverage, a condition is considered to be covered if all possible outcomes of the condition have been taken. All outcomes of conditions must be satisfied at least once during the execution of the test suite in the case of condition coverage criterion.

For decision/condition coverage criterion, it is required to satisfy both decision coverage and condition coverage criteria at the same time by the test suite.

Multiple condition coverage criterion requires all possible combinations of outcomes of conditions to be satisfied at least once during the execution of the test suite. For instance, K number of conditions will have $2^K$ number of possible outcome combinations. Test inputs are used to satisfy these $2^K$ outcome combinations.

Finally, path coverage criterion is the highest level coverage criterion. All possible routes of the code must be executed at least once during the execution of the test suite in case of path coverage criterion.

Some of the coverage criteria which have been discussed above are correlated. Such as, the adequacy of branch coverage criterion subsumes the adequacy of line coverage criterion. In other words, if a test suite satisfied
the branch coverage criterion it will also satisfy line coverage criterion.

The line coverage criterion is probably the most used coverage criterion because of its simplicity. However, it only gives the number of lines covered by the test suite in the source code. The branch coverage criterion would provide more precise coverage information of the code for the test suite.

### 2.6.1.2 Fault-Based Coverage

Fault-based coverage criterion requires that the test suite exercises a program with intent for detecting faults in the program. Mutation analysis is a fault-based coverage criterion where some faults are injected into the source code of the original program $P$ to generate some faulty versions, which are known as mutants. At the first step, test suite $T$ is needed to be executed on the original program $P$ and the test outputs must be verified to check the correctness of program $P$. If the program $P$ is found incorrect it should get fixed before executing the mutant $P'$, otherwise each mutant program $P'$ should be executed with the test suite $T$. The output of the original program $P$ is compared against that of a mutant $P'$ to detect any discrepancy between these outputs for the same test suite. If a discrepancy is found, then the mutant is said to be killed and thus fault to be detected by the test suite. In other words, the original program under test acts as the oracle to verify the correctness of the mutants in the traditional mutation analysis. However, there could still remain some mutants (called Equivalent Mutants) that are impossible to be killed, because they always produce the same test output as the original program. Detection of equivalent mutants is undecidable and equivalent to the halting problem in software engineering. This complex issue
CHAPTER 2. PRELIMINARIES

is kept outside of this study. Mutation analysis uses mutation operators to create fault(s) in the source code [25] of the original program. Several studies have been done on the effectiveness of mutation operators [57, 61]. The portion of mutant operator’s effectiveness is not discussed here as it is out of scope of this study. The overall procedure of mutation analysis can be visualized in Figure 2.1 [38].

![Figure 2.1: Traditional Mutation Analysis](image)
2.6.2 Fault-Detection and Code Coverage

Frate et al. have observed that an increase in reliability comes with an increase in code coverage measures [24]. Similarly, a decrease in reliability is accompanied by a decrease in at least one code overage. In contrast to this finding, the experimental outcomes carried out by Lionel and Dietmar [7] do not support the conception of a causal dependency between test coverage and defect coverage. However, later on, mutation testing was employed by Xia and Michael to investigate the relationship between code coverage and fault-detection capability for test input selection and evaluation purposes [8]. They have conducted a large-scale study with real-world applications and different coverage criterion. They have concluded that code coverage is a moderate indicator for the capability of fault-detection on the whole test suite used in their study. The effect of code coverage on fault-detection varies under different testing techniques. The correlation between the code coverage and fault-detection is high in exceptional cases, while weak in normal cases [8].

2.7 Summary

Software testing performs a review that compares the test outputs of the application against test oracles. An oracle is a mechanism for determining whether a test has passed or failed by comparing the test outputs against the expected outputs of the application. However, in practice test oracles may not be available or may become too costly to apply for a number of applications, as such the applications suffer from the oracle problem and become non-testable. Metamorphic testing has been suggested by Chen et al. [19].
as a way of alleviating the oracle problem. Instead of oracle, metamorphic testing checks whether the system under test satisfy it’s expected “metamorphic relations”. If the relation is violated, then a fault must exist in the implementation. Different types of properties can be considered as metamorphic relations for an application. Deriving effective metamorphic relations is helpful to make the testing process more efficient. Researchers have argued that metamorphic relations with diverse software execution behavior can detect faults more effectively. Code coverage quantifies the scale to which the code of the application has been exercised throughout the testing process. Code coverage can be considered as the measurement of execution behavior of the test inputs. In this work, some case studies are performed in order to examine to what extent the execution behavior of a metamorphic relation is correlated to its fault-detection effectiveness.
Chapter 3

Methodology and Experimental Design

3.1 Introduction

The concept and procedure of metamorphic testing which helped to set the objectives of this research work have been explained in previous chapters. This chapter depicts the methodology and experimental design which have been carried out to achieve the research goal. Subject programs have been selected from different platforms and domains in this study. In order to evaluate metamorphic testing in terms of fault-detection, mutation analysis is used to generate mutants (faulty versions of the subject programs). Code coverage achieved by each metamorphic relation of the different programs is collected as well. The main focus of this study is to investigate the relation between fault-detection effectiveness and code coverage of metamorphic relations.
3.2 Methodology

The flow chart in Figure 3.1 provides an overview of the research methodology of this thesis. As a first step, some subject programs are chosen around different platforms and domains. The programs were specifically chosen from the fields of scientific computing, detection system and optimization domains. The selection of the programs is justified in Section 3.2.1. Then, metamorphic relations are derived for each subject program. All the possible properties mentioned in Section 2.4 applicable to these subject programs are taken into consideration while the metamorphic relations. In the following step, test inputs are generated for the applications meeting the requirements of their associated metamorphic relations. After that, mutants are generated for each subject program. On the next step, metamorphic testing is performed on each mutant version and coverage data are collected for each metamorphic relation. The data obtained from metamorphic testing is analyzed in two stages, (1) fault-detection effectiveness of metamorphic testing is evaluated for the applications; (2) fault-detection effectiveness of each metamorphic relation is calculated. In the final step, fault-detection and code coverage achieved by each metamorphic relation are merged and the relationship between these two independent variables is scrutinized. All the steps are explained in detail in the subsequent sections.

3.2.1 Subject Program Selection

Some subject programs are needed to initiate this study. The selection of subject programs has no hard and fast rule. However, attempts were made
to explore different domains in this study. Four numerical programs were identified as subject programs in this work. They are primarily geometrical classification, optimization and conflict detection systems. Among the four, two subject programs are from the domain of optimization. All of these subject programs have an input domain consist of numbers or Boolean values. Table 3.1 reports some background information such as programming language and platform of these programs. The functionalities and details of each subject program are explained below:
3.2.1.1 TriTyp

TriTyp [26] is a widely studied triangle classification program in the literature. The version used in this study is written in the object oriented language Java. TriTyp accepts three numbers as input parameters which represent the lengths of the sides of a triangle. The output can be any of the four integer numbers from \{1, 2, 3, 4\} indicating that the input triangle is scalene, isosceles, equilateral, or illegal, respectively. The input domain of this program is numbers. It consists of 56 lines of code (LOC).

3.2.1.2 TCAS

TCAS [36] is an implementation of an on-board aircraft conflict detection and resolution system written in the procedural language C. It accepts twelve input parameters comprising of numbers and Boolean values. TCAS judges whether there will be a conflict between the current aircraft and the intruder aircraft based on the inputs. Afterwards, this program suggests which kind of manoeuvre the current aircraft should take as an output. TCAS has three types of outputs: 0 represents UNRESOLVED that indicates no maneuver, while 1 and 2 represent UPWARD or DOWNWARD maneuvers, respectively. This program consists of 173 LOC.
3.2.1.3 KNAPSACK

The knapsack problem is one of the most studied problems in combinatorial optimization. The multiple knapsack problem is a generalization of the single knapsack problem. KNAPSACK \[35\] is an implementation of the multiple knapsack problem which accepts five input parameters. The first two integers \(n\) and \(m\) provide the numbers of items and knapsacks, respectively. Next two \(n\)-tuple sets \(P=\{p_1, p_2, ..., p_n\}\) and \(W=\{w_1, w_2, ..., w_n\}\) represent the profits and the weights of \(n\) items, respectively; and one \(m\)-tuple set \(C=\{c_1, c_2, ..., c_m\}\) denotes the capacities of \(m\) knapsacks. The outputs of KNAPSACK are one \(n\)-tuple set \(Y=\{y_1, y_2, ..., y_n\}\) and one positive integer \(TP\). If \(y_i=j\) (where \(i=1, 2, ..., n\) and \(j=1, ..., m\)), the program assigns the \(i\)th item to the \(j\)th knapsack. If \(y_i=0\), the program does not assign the \(i\)th item to any knapsack. \(TP\) represents the total profit related to the items inside these \(m\) knapsacks. The KNAPSACK program attempts to calculate the optimal solution and thus to maximize the total profit. KNAPSACK is written in Java and consists of 780 LOC.

3.2.1.4 QAP

The Quadratic Assignment Problem (QAP) \[17\] is also a combinatorial optimization problem. It deals with the situation, where each facility produces items needed by another facility and shifting items between facilities is required. As the shifting cost depends on the weight of items being shifted and distance between the locations of the facilities, the problem is to locate facilities in a suitable location such that, total shifting cost, that is, the sum of
the distances multiplied by the weights is minimized. The QAP implementation accepts three input parameters. One integer \( n \) provides the numbers of facilities and locations, while the other two \( n \times n \) sets \( W \) and \( D \) represent the weights of items shifted from different facilities and the distances between the facility’s locations, respectively. The weight \( w_{ij} \in W \) represents the weight of item shifted from facility \( i \) to facility \( j \). The distance \( d_{ij} \) represents the distance taken to shift items from the location of facility \( i \) to the location of facility \( j \). The outputs of QAP are one \( n \)-tuple set \( Y=\{y_1, y_2, ..., y_n\} \) and one positive integer \( TC \). If \( y_i=j \) (where \( i=1, 2, ..., n \) and \( j=1, ..., n \)), the program assigns the \( i \)th facility to the \( j \)th location. \( TC \) represents the total shifting cost. This implementation of QAP is written in Java and consists of 900 LOC.

### 3.2.2 Derivation of MRs

For each subject program described in Section 3.2.1, a set of metamorphic relations was derived. All the available properties (described in Section 2.4) have taken into consideration for the derivation of metamorphic relations. Table 3.2 summarizes the number of metamorphic relations which were derived for the four subject programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Number of MRs</th>
</tr>
</thead>
<tbody>
<tr>
<td>TriTyp</td>
<td>5</td>
</tr>
<tr>
<td>TCAS</td>
<td>14</td>
</tr>
<tr>
<td>KNAPSACK</td>
<td>10</td>
</tr>
<tr>
<td>QAP</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 3.2: Number of MRs for Subject Programs

Detailed elaboration of all the metamorphic relations along with suitable
3.2.3 Test Input Generation

As mentioned in Section 2.3.1, source test inputs are generated using some traditional source input generation techniques (such as random testing, fault-based testing, etc.) in metamorphic testing. In this study, the source test inputs are generated randomly that is, values of all the input parameters of the applications are taken randomly from the applicable input domains. The random generation of source test inputs will alleviate bias in test input generation. In the subsequent phase, these source test inputs are modified in order to generate their associated follow up test inputs. This conversion is done on the basis of the change required by their corresponding metamorphic relations. For each of the metamorphic relations of each program, 10,000 source test inputs were generated randomly from their input domains. A corresponding set of follow up test inputs is generated by altering the source test inputs according to the requirements of their associated metamorphic relations. As the test inputs are generated randomly and no test adequacy measure is taken, generation of these large test inputs in our experiment was a precaution to avoid low test adequacy.

3.2.4 Mutant Generation

Mutation analysis has been applied in this study to evaluate the fault-detection effectiveness. Experiments have suggested that mutants are useful alternatives for real life faults while conducting comparisons of testing.
In this work, mutation analysis is used to systematically insert faults into the source code and then determine whether or not the mutants can be killed (i.e., whether the faults can be detected) using metamorphic testing. Standard automated mutant generator tools (MuJava and MILU) have been used for all the subject programs to generate mutants randomly. These two tools are frequently used in the literature of mutation analysis. All functions in the subject programs were candidates for the insertion of mutations and all the mutant operators available within the tool were taken into account to generate the mutants. All of the generated mutants have injected a single fault. However, mutants which have the same test output as the original program’s output for every test input in the test suite are discarded. These mutants run with a risk of being equivalent. Additionally, mutants yielding a fatal error (crash) or an infinite loop are also discarded.

Mutation tool, number of available mutant operators and number of mutants considered for each application in this study are summarized in Table 3.3. The variation in the number of mutants for each application is due to the difference in the numbers of lines in the source code, number of operators available in the source code and number of mutants that led to fatal errors or an infinite loop.

MuJava is a well known mutation tool for Java programs. The tool uses 15 traditional statement level and 28 class-level mutation operators to generate mutants automatically. Traditional statement level deals with statements and operators (mathematical, Boolean and logical). While, class level deals with the features of object oriented language such as inheritance, poly-
CHAPTER 3. METHODOLOGY AND EXPERIMENTAL DESIGN

<table>
<thead>
<tr>
<th>Program</th>
<th>Fault Detected in the Original Program?</th>
<th>Mutant Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>TriTyp</td>
<td>No</td>
<td>MuJava 15</td>
</tr>
<tr>
<td>TCAS</td>
<td>No</td>
<td>MILU 45</td>
</tr>
<tr>
<td>KNAPSACK</td>
<td>Yes</td>
<td>MuJava 43</td>
</tr>
<tr>
<td>QAP</td>
<td>No</td>
<td>MuJava 43</td>
</tr>
</tbody>
</table>

Table 3.3: Fault Detection in the Original Program and Mutant Generation

mutant operators from MuJava because the program does not have any classes which result 320 distinct mutants for TriTyp. KNAPSACK uses all the 43 mutant operators from MuJava. From the pool of generated mutants, 150 distinct mutants were taken for the experiment in this thesis. A total of 900 distinct mutants were considered for QAP using all the 43 operators from MuJava.

It is to be noted that metamorphic testing was initially applied to the original source code of each subject program. Some failures were detected in the original source code of KNAPSACK [refer to Table 3.3]. Hence this study considers the original program of KNAPSACK in the experiment, data collection and analysis stages. This fact leads to count 151 mutants for KNAPSACK in the following parts of this thesis. This principle is only being applied to KNAPSACK.

MILU [37] is a customizable mutation testing tool implementing 77 mutant operators (covered within 45 defined sets) for C programs. Using these 45 sets of mutant operators available in this tool, 422 distinct mutants were
CHAPTER 3. METHODOLOGY AND EXPERIMENTAL DESIGN

generated for TCAS.

3.2.5 Fault-Detection by Metamorphic Testing

The fault-detection effectiveness of each metamorphic relation is calculated using mutation analysis. Generated test inputs are executed on the mutant programs. After the execution of the source and follow up test inputs, the test outputs are checked against their associate metamorphic relations. If one or more test inputs violate the metamorphic relation, the metamorphic relation is said to detect the fault inside that mutant. Thus, the fault-detection effectiveness of each metamorphic relation is calculated as the number of mutants that can be killed by the test inputs associated with that particular metamorphic relation. After the execution of all the test inputs for all of the metamorphic relations, the numbers of faults detected by each metamorphic relation are calculated and analyzed in Chapter 4.

3.2.6 Coverage Data Collection

Code coverage is used to measure the degree of execution of the source code of a program. The coverage percentages achieved by the test inputs of a metamorphic relation are considered as representative of its execution behaviors. In order to collect the coverage information, extra effort is required. One simple way to achieve this is that testers insert extra code into the source code of the program manually and use the extra code to collect the coverage information. This inclusion of extra code is known as instrumentation. This manual approach runs with the risk of human error and is not cost-effective.
A large number of automatic coverage measurement tools have been developed in recent years. Coverage data were collected for all the subject programs using automated coverage data collection tools. The data are collected specifically for two basic coverage criterion namely line and branch coverage. As mentioned in Section 2.6.1, line coverage calculates the percentage of lines/statements of the code covered throughout the execution of test suite. Similarly, branch coverage calculates the percentage of the branches in the code that are covered by test suite. The tools used for code coverage collection are summarized in Table 3.4.

<table>
<thead>
<tr>
<th>Program</th>
<th>Coverage Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>TriTyp</td>
<td>CodeCover</td>
</tr>
<tr>
<td>TCAS</td>
<td>gcov, LCOV</td>
</tr>
<tr>
<td>KNAPSACK</td>
<td>CodeCover</td>
</tr>
<tr>
<td>QAP</td>
<td>CodeCover</td>
</tr>
</tbody>
</table>

Table 3.4: Code Coverage Collecting Tools

- CodeCover [56] is a free white-box testing tool for java programs, that measures line, branch, loop, MC/DC operator and sync-coverage.

- The test coverage program gcov [1] can be used in conjunction with GCC to analyze the programs. This analysis can help create more efficient, faster running code and discover untested parts of the program [1]. LCOV [2] is a graphical interface for gcov. By gathering data from gcov on multiple source files, it generates HTML pages with coverage information associated with the source code. LCOV also provides overview pages for easy navigation inside the file structure. LCOV supports line/statement, function and branch coverage measurement.
The whole set of 10,000 source test inputs and their corresponding 10,000 follow up test inputs for each metamorphic relation were executed on all the mutant versions of the subject programs. The accumulative coverage percentages achieved by these two sets of 10,000 test inputs were considered as the coverage achieved by a particular metamorphic relation on a particular mutant program. For example, TriTyp has 320 mutants, so each metamorphic relation of this program has a set of 320 coverage percentages for each coverage criterion.

3.3 Summary

Along the way of investigation to find out the answers of the research question, four numerical programs were selected from different domains such as geometrical classification, optimization and conflict detection as subject programs. Metamorphic relations were derived for each subject program. Mutants, which are representative of single fault in programs, were generated using automated mutant generator tools. Afterwards, test inputs were generated using random test input generation technique and metamorphic relations. Finally, these test inputs were executed on the mutant versions of each subject program in order to calculate the accumulated coverage information. Apart from these, test inputs and metamorphic relations were used to detect failures in mutants. As the test inputs were generated randomly and consist of a large number (20,000 in total for each metamorphic relation), it is expected the test inputs have not influenced the achievement of accumulative coverage and failure occurrence for any particular metamorphic relation in
a biased way. Chapter 4 illustrates the details of the collected data with analysis and discussions.
Chapter 4

Data Analysis and Results

4.1 Introduction

Based on the review of the literatures presented in Chapter 2, code coverage was attributed as the measurement of software execution behavior. Following the methodology explained in Chapter 3, test inputs were exercised on the mutants (faulty versions of the subject programs). The test outputs and coverage information achieved by these test inputs were collected and analyzed. The analyses of the collected data are presented in this chapter. The relationship between code coverage and fault-detection effectiveness of metamorphic relations is investigated by this analysis.
CHAPTER 4. DATA ANALYSIS AND RESULTS

4.2 Fault-Detection Effectiveness of Metamorphic Testing

Metamorphic testing was applied on every mutant of the subject programs. The test outputs of each source and follow up test input are verified against their corresponding metamorphic relation. If the source and follow up tests violate a metamorphic relation, then the mutant is considered to be killed by that particular metamorphic relation. The percentage of mutants being killed over the mutants being tested is calculated for each subject program. This percentage will illustrate the fault-detection effectiveness of metamorphic testing as a testing technique. Mutants are representative of faults within a particular program and the number of killed mutants is a good indicator of fault-detection. Thus, for the ease of understanding, each killed mutant is going to be termed as a detected fault from onwards for this research work.

The fault-detection effectiveness of metamorphic testing for different subject programs are depicted in Figure 4.1. Raw data of fault-detection effectiveness are available in Table B.1 in the Appendix B.

From Figure 4.1, it can be seen that metamorphic testing performs well in most of the cases in terms of being able to detect faults. For all the subject programs except TriTyp, metamorphic testing was able to detect all the faults. Thus, the fault-detection effectiveness is 100% for the other three programs, but 98.43% for TriTyp.
CHAPTER 4. DATA ANALYSIS AND RESULTS

The standalone fault-detection effectiveness of each metamorphic relation is investigated further. Following the procedure as discussed in Section 3.2.5, the numbers of faults detected by each metamorphic relation are collected. The fault-detection data were plotted against their associate metamorphic relation for all the subject programs subsequently in Figure 4.2 to Figure 4.5.

4.3 Fault-Detection Effectiveness of Each MR

The standalone fault-detection effectiveness of each metamorphic relation is investigated further. Following the procedure as discussed in Section 3.2.5, the numbers of faults detected by each metamorphic relation are collected. The fault-detection data were plotted against their associate metamorphic relation for all the subject programs subsequently in Figure 4.2 to Figure 4.5.
These figures have metamorphic relations along the x-axis and the number of detected faults along the y-axis. The raw data associated with these figures can be found in Table B.2 to Table B.5 in Appendix B.

Metamorphic testing has been applied on TriTyp’s 320 mutants with five metamorphic relations. The test result (Figure 4.2) shows that, MR2 detects a high number of faults and MR4 detects a lower number of faults. It is observed from Figure 4.3 that, for TCAS (422 mutants), among fourteen available metamorphic relations, MR4 detects the highest number of faults while MR2 detects only one fault. Again, metamorphic testing has been applied on KNAPSACK [as mentioned in Section 3.2.4] along with its 150 mutants with ten metamorphic relations. It can be seen form figure 4.4 that, MR1 detects the lowest number of faults in comparison to the other metamorphic relations for this program. Among the eight metamorphic relations of QAP, MR3, MR7 and MR8 are capable to detect all of the 900 faults under test. On the other hand, MR1 detects a small number of faults.

Figure 4.3: Number of Faults Detected by Each MR in TCAS
Figure 4.4: Number of Faults Detected by Each MR in KNAPSACK

Figure 4.5: Number of Faults Detected by Each MR in QAP

The above discussion of the figures reveals that metamorphic relations differ in terms of fault-detection. Some metamorphic relations are able to detect a high number of faults within a program, whereas some of them failed to detect majority of the faults. This phenomenon gives the motivation to inspect the metamorphic relations further. These inspection aims to trace
out the rationale behind the dissimilarities in the fault-detection ability of metamorphic relations.

4.4 Coverage Achieved by MRs

Code coverage was collected from all the mutants during the execution of test inputs for each metamorphic relation. Thus, each metamorphic relation has a set of coverage data. The number of elements in these sets is equal to the number of mutants of that particular subject program. For example, TriTyp has a set of coverage data (5*320) which is achieved by its five metamorphic relations on 320 mutants. It is to be noted that the coverage data is calculated as a percentage. For example, line coverage is calculated as a percentage of the number of lines covered during execution of the test suite divided by the total number of lines in the subject program. The coverage data obtained by the metamorphic relations were vast, hence it is difficult to visualize. For ease of visualization, the average value of the coverage set for each metamorphic relation was calculated. The average value refers to the arithmetic mean of the coverage data achieved by a particular metamorphic relation. These values are plotted in Figures 4.6 to 4.9. These figures have metamorphic relations along the x-axis and coverage values along the y-axis. The raw data along with the maximum and minimum coverage achieved by each metamorphic relation can be found in Table B.6 to Table B.13 in Appendix B.

The observations made from Figure 4.6a and Figure 4.6b show that MR4 of TriTyp achieves the highest line and branch coverage. MR6 has the lowest
coverage and MR13 has the highest coverage among the fourteen metamorphic relations of TCAS. It is found that, MR9 achieves the highest line coverage for KNAPSACK. However, MR9 is accompanied by MR7 to achieve the highest branch coverage for KNAPSACK. In case of QAP, MR5 achieves the highest coverage among all the available metamorphic relations.

From the coverage data, it can be seen that metamorphic relations achieve different ranges of coverage. A relationship is to be investigated in the next
section between the fault-detection effectiveness and coverage of a metamorphic relation.
4.5 Relationship between Coverage and Fault-Detection Effectiveness

The identified metamorphic relations varied significantly in terms of their achieved coverage data. The metamorphic relations also detect different numbers of fault. It is difficult to analyze the relationship between coverage and fault-detection effectiveness of metamorphic relations on the basis

Figure 4.8: Code Coverage for KNAPSACK
of these raw data. Hence some statistical analysis was conducted in order to investigate the relationship.

For each subject program, collected coverage data is divided into several clusters using histogram function \[10\]. The histogram function distributes the coverage data into equally spaced clusters and returns the number of elements in each cluster. Take KNAPSACK as an example. Five clusters were created based on branch coverage (Refer to Figure 4.12b) data. These
five clusters contain almost the same number of elements (except the first cluster). Although the histogram function tries to distribute the coverage data equally into different clusters, due to non-uniformity in the collected coverage data (Details are discussed in Section 4.5.4), the number of elements in each cluster was not exactly the same.

In this study, each metamorphic relation is accompanied with a set of code coverage and a set of fault detection achieved by its pairs of source and follow up inputs on different mutant programs. In other words, each coverage data and each fault detection data will correspond to one metamorphic relation. Total numbers of metamorphic relations which are able to detect at least one fault (called Fault-detecting MRs) for each coverage cluster were investigated in order to establish the relationship between coverage and fault detection effectiveness of metamorphic relations. For Instance, the last cluster (Refer to last point in Figure 4.12b) of branch coverage for KNAPSACK is associated with 234 metamorphic relations (as correspond to 234 coverage data in that cluster). A further analysis from the experimental results reveals that, all of the 234 metamorphic relations are Fault-detecting MRs.

These numbers of total metamorphic relations and Fault-detecting MRs associated with each cluster are used in the following equations for further analysis of the experimental data. Some measures were defined for the statistical analysis. MR Probability (MRP) is defined as the probability of a metamorphic relation to be located in a particular cluster, as follows:

\[
MRP = \frac{\text{No. of MRs in a cluster}}{\text{Total No. of MRs for the program}}
\]  

(4.1)
Let us continue with the example of knapsack. The last cluster of branch coverage for KNAPSACK will yield, \( \text{MRP} = \frac{234}{1510} = 0.15 \), where 1510 is the number of code coverage data collected for KNAPSACK.

Secondly, **Fault-Detection Probability within a Cluster (FDPC)** is defined as the probability of a metamorphic relation in a particular cluster to be able to detect faults, as follows:

\[
FDPC = \frac{\text{No. of Fault-Detecting MRs in a cluster}}{\text{No. of MRs in a cluster}} \quad (4.2)
\]

Following the example of the last cluster of branch coverage of KNAPSACK, we get FDPC\( = \frac{234}{234} = 1 \)

Finally, **Fault-Detection Probability (FDP)** can be defined as the probability of a fault-detecting metamorphic relation to be located in a particular cluster, as follows:

\[
FDP = \text{MRP} \times FDPC \quad (4.3)
\]

Thus, FDP for the last cluster of branch coverage for KNAPSACK is, FDP\( = (0.15 \times 1) = 0.15 \) (as visualized in the last point of Figure 4.12b).

Figures 4.10 to 4.13 illustrate the relationship between the coverage and FDPs. In these figures, the x-axis represents the center points of each coverage cluster, while the y-axis denotes the values of FDPs.

For ease of illustration, trend lines for the points in each figure were drawn. These trend lines were drawn in Microsoft Excel which uses the following formula:

\[
y = ce^{bx} \quad (4.4)
\]
CHAPTER 4. DATA ANALYSIS AND RESULTS

(a) FDP with Line Coverage

(b) FDP with Branch Coverage

Figure 4.10: Fault-Detection Probability of the MRs for TriTyp

Here $c$ and $b$ are constants and $e$ is the base of natural logarithm. The formula calculates the non-linear least square fit and adjusts the values of the constants $b$ and $c$ according to the data point values. Generally speaking, in both types of coverage criteria (line and branch) it can be observed that FDP normally increases with an increase in coverage value. However, few outliers were found for some scenarios. This phenomenon implies that some metamorphic relations with high coverage do not have high fault-detection
4.5.1 TriTyp

The observation from Figure 4.10a and 4.10b showed that the fault-detection probability of metamorphic relations has a very strong correlation with their code coverage for the TriTyp program. This relationship holds for both line
CHAPTER 4. DATA ANALYSIS AND RESULTS

(a) FDP with Line Coverage

(b) FDP with Branch Coverage

Figure 4.12: Fault-Detection Probability of the MRs for KNAPSACK

and branch coverage. The correlation increases exponentially as the coverage increases.

4.5.2 TCAS

The rising trend of the FDP along with their associated coverage is found to be inconsistent for this program. Although the FDP increases with the coverage for lower coverage ranges, some points within high coverage ranges
deviate from the trend lines. In Figure 4.11a, the rightmost two points are found inconsistent in the rising trend. Further inspection shows that the number of samples in the clusters represented by these two points differ by a factor of 1.5. This might caused the value of FDP to differ by 37.94% in the rightmost two points. Again in Figure 4.11b, for branch coverage, large fluctuation is visible for the third and fourth points from the left. The number of samples in the cluster represented by the third point is found 2.3
times more than that for the fourth point. This big difference in sample sizes in these coverage clusters might make their corresponding FDP to be differed by 54.2%.

4.5.3 KNAPSACK

A good exponential increase in fault-detection probability of metamorphic relations with their associated coverage data (line and branch coverage) can be observed from Figure 4.12a and 4.12b for the KNAPSACK program. The trend line shows that, the correlation increases exponentially as the coverage increases.

4.5.4 QAP

The observation from Figure 4.13a and 4.13b displays a rising trend between the FDP and coverage of metamorphic relation for the QAP program. This trend holds for both line and branch coverage. However, the leftmost point in both figures greatly deviates from the other points in the previous clusters. Further investigation reveals that the number of samples of the last cluster is at least 54.30 times larger than that of the other clusters available in these figures.

It is to be noted that in this study, the test inputs are generated randomly for the experiments. This generation process as described earlier in Section 3.2.3 was used to distribute the test inputs uniformly across the input domains. However, this cannot necessarily provide uniform distribution of the collected coverage data. Efforts have not been put to control the distri-
bution of coverage data across different clusters because this is not the goal of this study. Thus, such kind of varying sampling sizes in clusters exist, and can definitely affect the trend lines in the experiment. This fact must be taken into account while attempting future investigations in this research area.

4.6 Statistical Significance

Pearson correlation test has been performed on the calculated data of FDP for all subject programs. Pearsons correlation coefficient is one of the best methods of measuring correlation because it is based on the method of covariance. It gives information about the degree of correlation as well as the direction of the correlation. However, it is to be noted that, the Pearson correlation co-efficient is a standalone metrics to determine the correlation and does not depend on the trend lines shown earlier (as shown by the curves in Figures 4.10 to 4.13). If the Pearsons correlation coefficient value is near 1, then it is said to be a perfect correlation. If the coefficient value lies between 0.75 and 1, then it is said to be a high degree of correlation [40]. The coefficient values of the data in Figures 4.10 to 4.13 are summarized in the Table 4.1.

<table>
<thead>
<tr>
<th>Program</th>
<th>Line Coverage Vs FDP Coefficient</th>
<th>Branch Coverage Vs FDP Coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>TriTyp</td>
<td>0.84</td>
<td>0.90</td>
</tr>
<tr>
<td>TCAS</td>
<td>0.68</td>
<td>0.56</td>
</tr>
<tr>
<td>KNAPSACK</td>
<td>0.93</td>
<td>0.92</td>
</tr>
<tr>
<td>QAP</td>
<td>0.63</td>
<td>0.61</td>
</tr>
</tbody>
</table>

Table 4.1: Pearson Correlation Co-efficient Values Between Coverage and FDPs
Data from Table 4.1 shows that the correlation coefficients are greater than 0.5 for both line and branch coverage in all subject programs. For the programs TriTyp and KNAPSACK, this correlation is very strong. In both the cases of line and branch coverage, fault-detection probability of a metamorphic relation is strongly correlated with their coverage in these two programs. However, the correlation coefficient is positively correlated but they do not show a strong correlation for TCAS and QAP. As explained in Sections 4.5.2 and 4.5.4, the trend as well as the correlation value can get hampered by the sampling sizes of different clusters in code coverage data. Finally, in general, it can be said that the coverage and the FDP of metamorphic relations are positively correlated for all the programs in this study.

Thus, it can be concluded at this stage that coverage attained by metamorphic relations is a good indicator for the fault-detection effectiveness. However, it should be noted that other factors such as the structure of the subject program may have an impact on the effectiveness of metamorphic relations. In this study, the coverage ranges achieved by metamorphic relations for TCAS (between 10.6% to 93.9%) and TriTyp (between 5% to 100%) are much broader than those for the other two subject programs. Certain situations may exist where some program segments were covered by some metamorphic relations with lower coverage, but not by those with higher coverage. If faults were located in these segments, some metamorphic relations will not be able to detect them, even if they have a high coverage. In other words, when the coverage achieved by a metamorphic relation disperses in a broad range with a very low end, it cannot be guaranteed that higher code
coverage always brings higher fault-detection effectiveness. In summary, a correlation between the code coverage and the fault-detection effectiveness does exist. Code coverage can be a very good estimator of a metamorphic relation’s fault-detection effectiveness. However, it is also necessary to consider other factors, such as the program structure of the system under test.

4.7 Validity

In order to make a convincing conclusion from any type of experiments in software engineering research, it is suggested to possess a high degree of validity \[60\]. This section briefly addresses the challenges regarding to internal, external, construct and conclusion validity of this work. This discussion will aid the reader to measure how generalizable the results of this thesis are.

The internal validity of the research output lies in the implementation of the experiment. There might exist some errors while generating the test inputs and verifying the test outputs. However, to avoid any kind of error while executing these steps manually, the whole process was automated by a test driver. Moreover, the test driver has been checked by different skilled individuals. Another potential threat to validity is related to the test inputs that were used in the experiments. Twenty thousand test inputs were generated randomly for each metamorphic relation. The random generation of test inputs gives non-biased results in this study.

The main concern about the external validity of this study exists in the subject programs. The programs investigated in this study are all numeric
programs which may not be representative of all real-life programs with oracle problem. However, they were selected to demonstrate that the research outcome is applicable to different application domains with no test oracles. They were specifically selected from the domains of geometrical classification (TriTyp), conflict detection system (TCAS) and optimization (KNAPSACK and QAP). Moreover, these subject programs were taken from two different platforms. TCAS is written in C which is a procedural language; on the other hand, others are written in the object oriented programming language Java. The purpose of choosing subject programs in this way was to make this study platform independent. Automated mutant generation tools were used to avoid any biasing error, which may be introduced by manual fault generation (hand seeded fault). All possible efforts have been deployed to remove the mutants which run with the risk of being equivalent mutants. The mutants which will result in a fatal error or infinite loop were also discarded. The automated mutant generator tools were studied well before being applied into this research.

In addition, the metamorphic relations were identified by different individuals; hence the identification process was not biased. While deriving the metamorphic relations, careful considerations were provided in order to cover major functionalities of the applications. Mutants are considered to be replicas of real world faults. Thus, these mutants and the non-filtered metamorphic relations (the metamorphic relations were not chosen in particular to make the consequence predicted) used in this study reduce the external validity.
CHAPTER 4. DATA ANALYSIS AND RESULTS

The construct validity of this study comes with the measurement metrics used in the experiment. Two types of coverage criteria, namely line and branch coverage were used to measure the code coverage. The code coverage reflects the execution behaviors of metamorphic relations. These two criteria are very basic code coverage criteria and they have been popularly used in practice. These two coverage criteria have been chosen for their simplicity in this preliminary study. However, other coverage criteria should be studied in future works following this study. The types of faults that are planted in the subject programs via mutation analysis and the ability to detect faults as a fair metric for testing effectiveness can be brought into question. However, mutation analysis has been shown to be a precise estimation of real-world faults [3] and is accepted as the most neutral approach for judging the effectiveness of testing techniques in general [27]. The mutants generated in this study are single fault mutants. Multiple fault mutants can be studied in future to investigate the relationship between code coverage and fault detection effectiveness of metamorphic relations.

The conclusion validity of this thesis is justified by the statistical analysis in Section 4.6. The relationship between coverage and fault-detection effectiveness of each metamorphic relations was justified using standard Pearson correlation co-efficient [40]. Pearson’s correlation may get affected by the number of data samples. However this study used a large set of coverage data as samples. Furthermore, A standard histogram function [40] has been deployed for grouping the coverage data into clusters, which ensures unbiased sampling of the data. The test inputs were generated randomly; hence
the coverage and fault-detection data collected by executing these test inputs were not controlled in any way. Moreover, coverage data were collected as percentages and not the real number of lines or branches. This measurement gives a hope that, even for a larger system (with more line of code), our technique of analyzing the relation between code coverage and fault-detection can still be applicable.

4.8 Summary

After exercising all of the available test inputs on the mutant versions of the subject programs, the test data and coverage data were collected. These data were analyzed in different perspectives to conclude this investigation. At first, the fault-detection effectiveness of metamorphic testing on the applications of interest was studied. Metamorphic testing showed 100% fault-detection for three subject programs, and 98.43% for one program (TriTyp). In the next phase, the fault-detection effectiveness of individual metamorphic relations was calculated separately. The results demonstrate that different metamorphic relations got different fault-detection performance. Coverage data were collected for each metamorphic relation on each mutant program. This revealed that metamorphic relations achieve a widespread range of coverage values even within a single subject program. In an attempt to see the relationship between the fault-detection effectiveness and code coverage of each metamorphic relation, some statistical analysis has been performed. Initial investigation showed an increase in fault-detection effectiveness along with the coverage, except certain explainable situations. These detailed analyses
are concluded such that code coverage attained by metamorphic relations is a good indicator for their fault-detection effectiveness.
Chapter 5

Conclusions and Future Directions

5.1 Related Works

Researchers have worked on the effectiveness of metamorphic relations in terms of better fault detection. The effectiveness of different metamorphic relations has been studied using some case studies by Mayer and Guderlei [44]. They found metamorphic relations with rich semantic property are strong in fault detection. However, semantic properties are qualitative properties and their richness cannot be easily measured. Thus, the findings of [44] has limited application. This study aims to find quantitative properties of metamorphic relations that can be used to identify effective metamorphic relations.

Metamorphic testing has been applied to programs of finding shortest paths and critical paths with intent for selection of good metamorphic rela-
tions by Chen et al. [15]. The authors suggested that metamorphic relations with different execution behaviors are good metamorphic relations [15]. However, no empirical work has been conducted to systematically evaluate the relationship between the execution behaviors and the fault-detection effectiveness of metamorphic relations. Code coverage measures the scale of code being covered throughout the execution of test inputs. Thus, it can be considered as an indicator for the software execution behavior [28] and a quantitative property of metamorphic relation. This thesis reports the relationship between code coverage and the effectiveness of metamorphic relations using empirical studies.

5.2 Conclusions

Metamorphic testing has been widely used as an effective method for testing systems that do not have test oracles. Metamorphic testing technique helps in alleviating the oracle problem by using a set of metamorphic relations (MRs). A metamorphic relation is an expected relationship between test inputs and test outputs for multiple executions.

Verification of the relations among the test outputs provides the evidence of correctness for the system under test. Many metamorphic relations can be identified on the basis of specifications, algorithms or implementation of the system under test. Different metamorphic relations have different effectiveness of fault-detection. A guideline for selecting metamorphic relations is important to improve the quality of metamorphic testing. Both testing time and resources can be saved by exploring metamorphic relations with high
fault-detection effectiveness. Some researchers have argued that metamorphic relations which can differ the software execution behaviors will have a high fault-detection effectiveness \cite{15}. In this work, several case studies were carried out to systematically investigate the relationship between the execution behaviors and the effectiveness of metamorphic relations. The code coverage achieved by the metamorphic relations is considered as an indicator of execution behavior. Two code coverage criteria, line coverage and branch coverage achieved by each metamorphic relation are compared against their fault-detection effectiveness.

Four case studies with different subject programs were carried out in this study. A geometrical classification system (TriTyp), an on-board aircraft collision detection and resolution system (TCAS), a program for solving the multiple knapsack problem (KNAPSACK) and an application for the quadratic assignment problem (QAP) were selected as the subject programs for this study. Different numbers of metamorphic relations were derived from these subject programs. Ten thousand pairs of test inputs were generated for each metamorphic relation of the subject programs. Mutation analysis was conducted to evaluate the fault-detection effectiveness of different metamorphic relations. Coverage data were collected by automated coverage data collection tools. Both data from mutation analysis and coverage were studied to find out the relationship between the code coverage and fault-detection effectiveness of metamorphic relations. It was found that metamorphic relations with low coverage have low effectiveness in detecting software faults. On the other hand, a high coverage shows better performance for fault-detection in most cases. However, high coverage does not necessarily imply a high fault-
detection effectiveness at all times. The experimental results depict that some metamorphic relations with a high coverage cannot detect a large number of faults. Such observations are also understandable as a metamorphic relation cannot detect a fault as long as the metamorphic relation does not execute the statement containing that fault, even if the metamorphic relation achieves a high coverage. In conclusion, high coverage indicating diverse execution behavior is a good estimator of fault-detection effectiveness of a metamorphic relation. However, a high coverage value cannot be a perfect indicator for the fault-detection effectiveness. In regression testing, previous good test inputs are chosen to re-run the system for fault detection. This study contributes to the selection and reuse of effective metamorphic relations for this area. Metamorphic relations with higher code coverage found during the early stage of software development can be reused in regression testing.

5.3 Recommendations for Future Work

The aim of this research work is to find a relationship between some factors and the fault-detection effectiveness of metamorphic relations. However, due to the time constraints of this research study, it was not possible to consider all the factors which may have impact on the fault-detection effectiveness of metamorphic relations. This work can be utilized as a guideline for further research as follows:

- Two types of coverage criteria, namely line and branch coverage were used as the code coverage criteria in this study. Other coverage criteria,
such as data-flow criteria can be studied to support the conclusion.

- In this study the coverage data were calculated as a percentage (explained in Section 4.4). This percentage value is a good indicator of the coverage. However, it does not reflect the execution path precisely. For example, 90% line coverage cannot give any indication about whether the fault has been executed during the testing process. Future work can consider collection of this data to justify the results of this study.

- This study considered single fault mutants only. Further investigation can be made with multiple fault mutants to investigate the relationship between code coverage and fault detection effectiveness of metamorphic relations.

- It was found that some metamorphic relations are able to detect a higher number of faults with fewer number of test inputs. The number of test inputs used by metamorphic relations to detect faults can be analyzed further. This analysis can be used to assign each metamorphic relation a weight for detecting faults more efficiently.

- Metamorphic relations responsible for the same fault-detection can be grouped together and can be inspected further to reveal characteristics which are common in these metamorphic relations. This study can help in providing guideline for deriving metamorphic relations.
Bibliography


01, Department of Computer Science, Hong Kong University of Science and Technology, 1998.


[34] D. Hamlet. When only random testing will do. In *Proceedings of the 1st international workshop on Random testing (RT ’06).*


[42] Y. S. Ma, J. Offutt, and Y. R. Kwon. Mujava: an automated class muta-


[49] C. Murphy, K. Shen, and G. Kaiser. Using jml runtime assertion check-
ing to automate metamorphic testing in applications without test or-
cles. In Proceedings of the 2009 International Conference on Software 


[51] G. J. Myers, C. Sandler, T. Badgett, and T. Thomas. The Art of Soft-

the conformance of java classes against algebraic specifications. In Pro-
ceedings of the International Conference Formal Methods and Software 

[53] M. Pezz’e and M. Young. Software Testing and Analysis: Process, Prin-

test oracles for reactive systems. In Proceedings of the 14th international 

In Proceedings of the symposium on Testing, analysis, and verification 


Appendix A

Metamorphic Relations for The Subject Programs

A.1 TriTyp

On the basis of basic mathematical properties of geometrical classification five metamorphic relations have been devised for TriTyp. They are as follows:

MR1: multiplicative property
Given a source test input \( L_s = \{ l_1, l_2, l_3 \} \), its test output is \( O_s \) which indicates the type of the triangle can that be formed with the given lengths. In order to generate the follow up test input, multiply each length of the source test input set \( L_s \) with a randomly selected number \( n \) from \( N \), where \( N \) is a set of numbers from 1 to 10. The follow up test input thus obtained is \( L_f = \{ l_1 \times n, l_2 \times n, l_3 \times n \} \). The test output of follow up test input, \( O_f \) will be equal to \( O_s \).
APPENDIX A. METAMORPHIC RELATIONS FOR THE SUBJECT PROGRAMS

MR2: permutative property
Given a source test input \( L_s = \{l_1, l_2, l_3\} \), its test output is \( O_s \) which indicates the type of the triangle that can be formed with the given lengths. Permuting the source test input set \( L_s \) can generate the follow up test input, say \( L_f = \{l_3, l_1, l_2\} \). The follow up test output \( O_f \) will be equal to \( O_s \).

MR3: additive property
Given a source test input \( L_s = \{l_1, l_2, l_3\} \), its test output is \( O_s \) which indicates the type of the triangle that can be formed with the given lengths. Add a variable that is higher than the maximum length of the source test input set \( L_s \) with each length of \( L_s \) in order to generate the follow up test input, say \( L_f = \{l_1 + \text{max}(L_s) + C, l_2 + \text{max}(L_s) + C, l_3 + \text{max}(L_s) + C\} \). Here the addition of a constant \( C \), which can be any random value greater than zero ensures the lengths in the follow up test input must be greater than the maximum length of the source test input. The test output of follow up test input, \( O_f \) will be equal to \( O_s \).

MR4: triangle property
Given a source test input \( L_s = \{l_1, l_2, l_3\} \), its test output is \( O_s \) which indicates the type of the triangle that can be formed with the given lengths. Subtract \( C = \text{min}(L_s) \) that is the minimum length of the source test input set \( L_s \) from each length of \( L_s \) in order to generate the follow up test input, say \( L_f = \{l_1 - C, l_2 - C, l_3 - C\} \). The test output of follow up test input, \( O_f \) will be (i) If \( O_s = 4 \), then \( O_f = O_s \) (ii) else \( O_f \neq O_s \).
APPENDIX A. METAMORPHIC RELATIONS FOR THE SUBJECT PROGRAMS

MR5: triangle property
Given a source test input $L_s = \{l_1, l_2, l_3\}$, its test output is $O_s$ which indicates the type of the triangle that can be formed with the given lengths. Substitute any length of the source test input set $L_s$ with an arbitrary length from $L_s$ in order to generate the follow up test input, say $L_f = \{l_1, l_2, l_3\}$. The test output of follow up test input, $O_f$ will be (i) If $O_s \in \{2, 4\}$, then $O_f \in \{2, 3, 4\}$ (ii) If $O_s = 3$, then $O_f = O_s$ (iii) else $O_f \in \{2, 4\}$.

A.2 TCAS
All possible attempts are given to identify 14 metamorphic relations for TCAS, as described below. The source and follow up test inputs are denoted by $T_s$ and $T_f$, respectively. The test outputs of the source and follow up test inputs are denoted by $O_s$ and $O_f$, respectively.

MR1: Given that the intruder aircraft does not have the TCAS system, if $T_s$ and $T_f$ only differ in whether the intruder aircraft has an intention or not, it should have the relation $O_f = O_s$.

MR2: Given that the intruder aircraft does not have the TCAS system, if $T_s$ and $T_f$ only differ in whether the report describing the presence of any intruder is valid or not, it should have the relation $O_f = O_s$.

MR3: Given that the intruder aircraft does not have any intention, and the report describing the presence of any intruder is valid, if $T_s$ and $T_f$ only differ in whether the intruder aircraft has the TCAS system or not, it should
have the relation $O_f = O_s$.

The next eight metamorphic relations (MR4-MR11) have an additional prerequisite that includes the following conditions.

1. The TCAS system on the controlled aircraft has a high confidence, and
2. The vertical converging speed is not larger than 600, and
3. The current vertical separation between the two aircrafts at the closest point will be larger than 600 if the controlled aircraft maintains its trajectory, and
4. (a) The intruder aircraft does not have the TCAS system, or
   (b) i. the intruder aircraft does not have any intention, and
      ii. the report describing the presence of any intruder is valid.

**MR4:** Given that the current altitude of the controlled aircraft is smaller than that of the intruder aircraft, and the vertical separation between two aircrafts will be smaller than the threshold value if the controlled aircraft initiates a downward manoeuvre, if $T_s$ and $T_f$ differ in the relation between the calculated inhibit biased climb and the vertical separation between two aircrafts where the controlled aircraft initiates a downward manoeuvre, it should have the relation $O_f \neq O_s$.

**MR5:** Given that the current altitude of the controlled aircraft is not smaller than that of the intruder aircraft and the vertical separation between two aircrafts will not be smaller than the threshold value if the controlled
aircraft initiates an upward manoeuvre, if $T_s$ and $T_f$ differ in the relation between the calculated inhibit biased climb and the vertical separation between two aircrafts where the controlled aircraft initiates a downward manoeuvre, it should have the relation $O_f \neq O_s$.

**MR6:** Given that the vertical separation between two aircrafts will be no larger than the calculated inhibit biased climb and smaller than the threshold value if the controlled aircraft initiates a downward manoeuvre, if $T_s$ and $T_f$ differ in the relation between the current altitudes of the two aircrafts, it should have the relation $O_f \neq O_s$.

**MR7:** Given that the vertical separation between two aircrafts will not be smaller than the calculated inhibit biased climb if the controlled aircraft initiates a downward manoeuvre and not larger than the threshold value if the controlled aircraft initiates an upward manoeuvre, if $T_s$ and $T_f$ differ in the relation between the current altitudes of the two aircrafts, it should have the relation $O_f \neq O_s$.

**MR8:** Given that the vertical separation between two aircrafts will be smaller than the threshold value if the controlled aircraft initiates a downward manoeuvre, if $T_s$ and $T_f$ differ in the relation between the calculated inhibit biased climb and the vertical separation between two aircrafts where the controlled aircraft initiates a downward manoeuvre, it should have the relation: if $O_s=0$, $O_f \in \{0, 1, 2\}$; otherwise, $O_f \neq O_s$. 

80
**APPENDIX A. METAMORPHIC RELATIONS FOR THE SUBJECT PROGRAMS**

**MR9:** Given that the vertical separation between two aircrafts will not be smaller than the threshold value if the controlled aircraft initiates an upward manoeuvre, if $T_s$ and $T_f$ differ in the relation between the calculated inhibit biased climb and the vertical separation between two aircrafts where the controlled aircraft initiates a downward manoeuvre, it should have the relation: if $O_s=0$, $O_f \in \{0, 1, 2\}$; otherwise, $O_f \neq O_s$.

**MR10:** Given that the vertical separation between two aircrafts will be smaller than the threshold value if the controlled aircraft initiates a downward manoeuvre, if $T_s$ and $T_f$ differ in the relation between the current altitudes of the two aircrafts, it should have the relation: if $O_s=0$, $O_f \in \{0, 1, 2\}$; otherwise, $O_f \neq O_s$.

**MR11:** Given that the vertical separation between two aircrafts will not be smaller than the threshold value if the controlled aircraft initiates an upward manoeuvre, if $T_s$ and $T_f$ differ in the relation between the current altitudes of the two aircrafts, it should have the relation: if $O_s=0$, $O_f \in \{0, 1, 2\}$; otherwise, $O_f \neq O_s$.

**MR12:** Given that other parameters can be randomly changed, if $T_s$ and $T_f$ differ in whether the TCAS system on the controlled aircraft has a high confidence or not, it should have the relation: if $O_s=0$, $O_f \in \{0, 1, 2\}$; otherwise, $O_f \neq O_s$.

**MR13:** Given that other parameters can be randomly changed, if $T_s$ and $T_f$ differ in whether the vertical converging speed is larger than 600 or
not, it should have the relation: if $O_s=0$, $O_f \in \{0, 1, 2\}$; otherwise, $O_f \not= O_s$.

**MR14:** Given that other parameters can be randomly changed, if $T_s$ and $T_f$ differ in whether the current vertical separation between the two aircrafts at the closest point will be larger than 600 or not where the controlled aircraft maintains its trajectory, it should have the relation: if $O_s=0$, $O_f \in \{0, 1, 2\}$; otherwise, $O_f \not= O_s$.

### A.3 KNAPSACK

Ten metamorphic relations were identified for KNAPSACK as follows. In the following, the source test input is denoted as $T_s=\{P, W, C\}$, where $P=\{p_1, p_2, \ldots, p_n\}$, $W=\{w_1, w_2, \ldots, w_n\}$, and $C=\{c_1, c_2, \ldots, c_m\}$. The test output of the source test input is denoted as $O_s=\{Y, TP\}$, where $Y=\{y_1, y_2, \ldots, y_n\}$ and $TP$ is a positive integer representing the total profit.

**MR1:** changing the order of items

Given a source test input $T_s=\{P, W, C\}$, its test output is $O_s=\{Y, TP\}$.

Swap items $k$ and $r$ ($l \leq k \leq n$) where $p_k \not= p_r$ or $w_k \not= w_r$, then it get a follow up test input $T_f=\{P', W', C\}$ where $P'=\{p_1, \ldots, p_r, \ldots, p_k \ldots, p_n\}$ and $W'=\{w_1, \ldots, w_r, \ldots, w_k \ldots, w_n\}$. The test output of $T_f$ is $O_f=\{TP', Y'\}$. It should have $Y'=\{y_1, \ldots, y_r, \ldots, y_k \ldots, y_n\}$ and $TP'=TP$.

**MR2:** increasing the profits of items in first knapsack

Given a source test input $T_s=\{P, W, C\}$, its test output is $O_s=\{Y, TP\}$. 

82
Increase the profit of item \( k \) in \( 1^{st} \) knapsack where \( y_k = 1 \), that is \( p'_k = p_k + c \) (\( c \) is a positive number), then the follow up test input will be \( T_f = \{ P', W, C \} \) where \( P' = \{ p_1, \ldots, p'_k, \ldots, p_n \} \). The test output of \( T_f \) is \( O_f = \{ TP', Y' \} \) where \( Y' = \{ y'_1, \ldots, y'_j, \ldots, y'_n \} \). It should have \( y_j \cdot y'_j = 0 \) if and only if \( y_j = y'_j = 0 \); \( TP' = TP + c \).

**MR3: increasing the unselected item’s weights**

Given a source test input \( T_s = \{ P, W, C \} \), its test output is \( O_s = \{ Y, TP \} \).

Increase the weight of unselected item \( k \) where \( y_k = 1 \), that is \( w'_k = w_k + c \) (\( c \) is a positive number), then the follow up test input will be \( T_f = \{ P, W', C \} \) where \( W' = \{ w_1, \ldots, w'_k, \ldots, w_n \} \). The test output of \( T_f \) is \( O_f = \{ TP', Y' \} \) where \( Y' = \{ y'_1, \ldots, y'_j, \ldots, y'_n \} \). It should have \( y_j \cdot y'_j = 0 \) if and only if \( y_j = y'_j = 0 \); \( TP' = TP \).

**MR4: decreasing the unselected item’s profits**

Given a source test input \( T_s = \{ P, W, C \} \), its test output is \( O_s = \{ Y, TP \} \).

Decrease the profit of unselected item \( j \) where \( y_k = 1 \), that is \( p'_j = p_j - c \) (\( c \) is a positive number), then the follow up test input will be \( T_f = \{ P', W, C \} \) where \( P' = \{ p_1, \ldots, p'_k, \ldots, p_n \} \). The test output of \( T_f \) is \( O_f = \{ TP', Y' \} \), where \( Y' = \{ y'_1, \ldots, y'_j, \ldots, y'_n \} \). It should have \( y_j \cdot y'_j = 0 \) if and only if \( y_j = y'_j = 0 \); \( TP' = TP \).

**MR5: decreasing the capacity of 1st knapsack to the total weights of its selected items**

Given a source test input \( T_s = \{ P, W, C \} \), its test output is \( O_s = \{ Y, TP \} \).

Change the capacity of 1st knapsack, that is \( c_1 = \sum w_j e_j \) where \( e_j = 1 \) when
P R O G R A M S

$e_j = 1$; otherwise $e_j = 0$, then the system get the follow up test input $T_f = \{P, W, C'\}$ where $C' = \{c_1, ..., c'_i, ..., c_m\}$. The test output of $T_f$ is $O_f = \{TP', Y'\}$. It should have $Y' = Y$ and $TP' = TP$.

MR6: adding an item with the least profit of selected items
and the largest weight of all the items
Given a source test input $T_s = \{P, W, C\}$, its test output is $O_s = \{Y, TP\}$.
Add a new item $k = n + 1$, where $p_k = \min (p_j)$ when $y_j \neq 0$ and $w_k = \max (w_j)$ ($j = 1, 2, ..., n$), then the the follow up test input will be $T_f = \{P, W, C'\}$, where $P' = \{p_1, ..., p'_k, p_k\}$ and $W' = \{w_1, ..., w'_n, w_k\}$. The test output of $T_f$ is $O_f = \{TP', Y'\}$. It should have $Y' = \{y_1, ..., y_n, 0\}$ and $TP' = TP + c$.

MR7: deleting the unselected items
Given a source test input $T_s = \{P, W, C\}$, its test output is $O_s = \{Y, TP\}$.
Delete an unselected item $k$ where $y_k = 0$, then the follow up test input will be $T_f = \{P', W', C\}$, where $P' = \{p_1, ..., p'_k, p_{k+1}, ..., p_n\}$ and $W' = \{w_1, ..., w_{k-1}, w_{k+1}, ..., w_n\}$. The test output of $T_f$ is $O_f = \{TP', Y'\}$, where $Y' = \{y'_1, ..., y_{k-1}, y_{k+1}, ..., y'_n\}$. It should have $y_j = 0$ if and only if $y_j = y'_j = 0$ ($j \neq k$); $TP' = TP$.

MR8: deleting an item in the first knapsack and decreasing the first knapsack’s capacity with deleted item’s weight
Given a source test input $T_s = \{P, W, C\}$, its test output is $O_s = \{Y, TP\}$.
Delete first knapsack’s selected item $k$, where $y_k = 1$ and minus the capacity of first knapsack with $k$ th item’s weight, then the the follow up test input will be $T_f = \{P', W', C'\}$, where $P' = \{p_1, ..., p_{k-1}, p_{k+1}, ..., p_n\}$, $W' = \{w_1,
... , \, w_{k+1}, \, ..., \, w_n} \) and \( C' = \{ c_1 - w_k, \, ..., \, c_m \} \). The test output of \( T_f \) is \( O_f = \{ TP', \, Y' \} \), where \( Y' = \{ y'_1, \, ..., \, y'_k, \, y'_{k+1}, \, ..., \, y'_n \} \). It should have \( y_j, y'_j = 0 \) if and only if \( y_j = y'_j = 0 \) (\( j \neq k \)); \( TP' = TP - p_k \).

**MR9:** combining two items in the first knapsack

Given a source test input \( T_s = \{ P, \, W, \, C \} \), its test output is \( O_s = \{ Y, \, TP \} \). Combine item \( k \) and item \( r \) (\( 1 \leq k < r \leq n \)) where \( y_k = y_r = i \neq 0 \), then the the follow up test input will be \( T_f = \{ P', \, W', \, C \} \), where \( P' = \{ p_1, \, ..., \, p_k + p_r, ... p_{r-1}, \, p_{r+1}, \, ..., \, p_n \} \), \( W' = \{ w_1, \, ..., \, w_k + w_r, \, ..., \, w_{r-1}, \, w_{r+1}, \, ..., \, w_n \} \). The test output of \( T_f \) is \( O_f = \{ TP', \, Y' \} \), where \( Y' = \{ y'_1, \, ..., \, y'_k, \, ... y'^{r-1}, \, y'^{r+1}, \, ..., \, y'_n \} \). It should have \( TP' = TP \). Because there might be another optimal solution available, then the test output \( Y \) is uncertain.

**MR10:** deleting the first knapsacks as well as its selected items

Given a source test input \( T_s = \{ P, \, W, \, C \} \), its test output is \( O_s = \{ Y, \, TP \} \). Split 1st knapsack and its selected items out from the given set \( T_s \), supposing \( N \) items are split out and their total profit is \( \sum_k^N p_k \), then the the follow up test input will be \( T_f = \{ P', \, W', \, C \} \), where \( P' = \{ p'_1, \, ..., \, p'_j, ... p'^{n-N} \} \), \( W' = \{ w'_1, \, ..., \, w'_j, ... w'^{n-N} \} \) and \( C' = \{ c_2, \, ..., \, c_i, \, ..., \, c_m \} \). The test output of \( T_f \) is \( O_f = \{ TP', \, Y' \} \), where \( TP' = TP - \sum_k^N p_k \).

### A.4 QAP

Eight metamorphic relations have been derived for QAP. Among them, MR1 to MR6 are similar metamorphic relations used in [17]. However, for simplicity and ease of understanding of this dissertation’s reader, they are rewritten.
here. MR1 and MR2 refer to the section 2.1, MR3 and MR4 refer to section 2.2 and MR5 and MR6 refer to section 2.3 of the referred paper [17]. Interested readers can go to the reference for further understanding. In the description of metamorphic relations below, the source test input is defined by $T_s = \{W_s, D_s\}$.

**MR1: interchange facilities (weight)**
Interchanging the facility among them will not change the outcome of QAP solver. This will only change the names of facilities in the assignments. Given a source test input $T_s$, the total cost of the optimal solution can be denoted by $TC_s$. Interchanging the facilities will yield another pool of facilities in the follow up test input $T_f$ for this metamorphic relation. The total cost of optimal solution in the follow up input can be denoted by $TC_f$, where $TC_s = TC_f$.

**MR2: interchange locations (distance)**
The interchanging of locations in the QAP does not affect the nature of the solutions as well. Given a source test input $T_s$, the total cost of the optimal solution can be denoted by $TC_s$. Interchanging the locations in the same way as done with the facilities in MR1 will yield another pool of locations in the follow up test input $T_f$ for this metamorphic relation. The total cost of optimal solution in the follow up input can be denoted by $TC_f$, where $TC_s = TC_f$.

**MR3: addition of another facility without any weight**
The addition of another facility and location to the existing pool will obvi-
ous change the current optimal solution and the final cost. However, this change can be controlled in some ways. One approach is the addition of another facility with zero weight and a location which is too far from the existing locations in the source test input. Given a source test input $T_s$, the total cost of the optimal solution can be denoted by $TC_s$. Addition of a new facility with zero weight $w_z$ and a long distance $d_z$ will generate the follow up test input $T_f$. The total cost of optimal solution in the follow up input can be denoted by $TC_f$, where $TC_s = TC_f$.

**MR4: addition of another facility with some random weight**

One existing facility $i$ is chosen randomly, and a weight smaller than all existing non-zero weights is assigned between the new facility $n$ with weights between the new and existing locations other than $w_i$ set to 0. Given a source test input $T_s$, the total cost of the optimal solution can be denoted by $TC_s$. Addition of a new facility with a weight $w_n$ as stated in the previous paragraph and a location with a high distance $d_h$ value, will yield the follow up test input $T_f$. The total cost of optimal solution in the follow up input can be denoted by $TC_f$, where $TC_f > TC_s$.

**MR5: addition of another pool of facilities and locations without any weights**

Given a source test input $T_s$, the total cost of the optimal solution can be denoted by $TC_s$. Another pool of facilities and locations can be merged with weight $W_m$ and distance $D_m$ in order to generate follow up test input $T_f$. While merging the two sets facilities from different sets should have zero
weights and a high distances among them. The total cost of optimal solution in the follow up input can be denoted by $TC_f$, where $TC_f=TC_s+TC_m$.

**MR6: addition of another pool of facilities and locations some random weights**
Given a source test input $T_s$, the total cost of the optimal solution can be denoted by $TC_s$. Another pool of facilities and locations can be merged $W_m$ and $D_m$ in order to generate follow up test input $T_f$. While merging the two sets, there should be one small weight between a pair of facilities, one in each pool. The total cost of optimal solution in the follow up input can be denoted by $TC_f$, where $TC_f>TC_s+TC_m$.

**MR7: increasing the weights of a pair of facilities**
Given a source test input $T_s$, the total cost of the optimal solution can be denoted by $TC_s$. Increasing the weights of a pair of facilities will yield follow up test input $s_f$, where one random pair has been selected from $W_s$ and increased by some constant $c$. The total cost of optimal solution in the follow up input can be denoted by $TC_f$, where $TC_f>TC_s$.

**MR8: increasing the distances of a pair of locations**
Given a source test input $T_s$, the total cost of the optimal solution can be denoted by $TC_s$. Increasing the distances of a pair of facilities will yield follow up test input $T_f$, where one random pair has been selected from $D_s$ and increased by some constant $c$. The total cost of optimal solution in the follow up input can be denoted by $TC_f$, where $TC_f>TC_s$. 
Appendix B

Experimental Data Values

<table>
<thead>
<tr>
<th>Program</th>
<th>Faults Under Test</th>
<th>Number of Faults Detected</th>
<th>Percentage of Success%</th>
</tr>
</thead>
<tbody>
<tr>
<td>TriTyp</td>
<td>320</td>
<td>315</td>
<td>98.43</td>
</tr>
<tr>
<td>TCAS</td>
<td>422</td>
<td>422</td>
<td>100</td>
</tr>
<tr>
<td>KNAPSACK</td>
<td>151</td>
<td>151</td>
<td>100</td>
</tr>
<tr>
<td>QAP</td>
<td>900</td>
<td>900</td>
<td>100</td>
</tr>
</tbody>
</table>

Table B.1: Fault-Detection Effectiveness of Metamorphic Testing

<table>
<thead>
<tr>
<th>MR</th>
<th>Number of Faults Detected</th>
<th>Percentage of Success%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR1</td>
<td>59</td>
<td>18.44</td>
</tr>
<tr>
<td>MR2</td>
<td>211</td>
<td>67.81</td>
</tr>
<tr>
<td>MR3</td>
<td>120</td>
<td>37.5</td>
</tr>
<tr>
<td>MR4</td>
<td>28</td>
<td>8.75</td>
</tr>
<tr>
<td>MR5</td>
<td>71</td>
<td>22.18</td>
</tr>
</tbody>
</table>

Table B.2: Fault-Detection by Individual MRs of TriTyp
### APPENDIX B. EXPERIMENTAL DATA VALUES

<table>
<thead>
<tr>
<th>MR</th>
<th>Number of Faults Detected</th>
<th>Percentage of Success%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR1</td>
<td>20</td>
<td>4.74</td>
</tr>
<tr>
<td>MR2</td>
<td>1</td>
<td>0.24</td>
</tr>
<tr>
<td>MR3</td>
<td>34</td>
<td>7.82</td>
</tr>
<tr>
<td>MR4</td>
<td>268</td>
<td>63.51</td>
</tr>
<tr>
<td>MR5</td>
<td>218</td>
<td>51.66</td>
</tr>
<tr>
<td>MR6</td>
<td>257</td>
<td>60.90</td>
</tr>
<tr>
<td>MR7</td>
<td>194</td>
<td>45.97</td>
</tr>
<tr>
<td>MR8</td>
<td>50</td>
<td>11.85</td>
</tr>
<tr>
<td>MR9</td>
<td>96</td>
<td>22.79</td>
</tr>
<tr>
<td>MR10</td>
<td>34</td>
<td>8.06</td>
</tr>
<tr>
<td>MR11</td>
<td>35</td>
<td>8.29</td>
</tr>
<tr>
<td>MR12</td>
<td>27</td>
<td>6.40</td>
</tr>
<tr>
<td>MR13</td>
<td>43</td>
<td>10.19</td>
</tr>
<tr>
<td>MR14</td>
<td>23</td>
<td>5.45</td>
</tr>
</tbody>
</table>

Table B.3: Fault-Detection by Individual MRs of TCAS

<table>
<thead>
<tr>
<th>MR</th>
<th>Number of Faults Detected</th>
<th>Percentage of Success%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR1</td>
<td>63</td>
<td>41.72</td>
</tr>
<tr>
<td>MR2</td>
<td>144</td>
<td>95.36</td>
</tr>
<tr>
<td>MR3</td>
<td>151</td>
<td>100</td>
</tr>
<tr>
<td>MR4</td>
<td>151</td>
<td>100</td>
</tr>
<tr>
<td>MR5</td>
<td>151</td>
<td>100</td>
</tr>
<tr>
<td>MR6</td>
<td>149</td>
<td>98.67</td>
</tr>
<tr>
<td>MR7</td>
<td>151</td>
<td>100</td>
</tr>
<tr>
<td>MR8</td>
<td>151</td>
<td>100</td>
</tr>
<tr>
<td>MR9</td>
<td>151</td>
<td>100</td>
</tr>
<tr>
<td>MR10</td>
<td>151</td>
<td>100</td>
</tr>
</tbody>
</table>

Table B.4: Fault-Detection by Individual MRs of KNAPSACK

<table>
<thead>
<tr>
<th>MR</th>
<th>Number of Faults Detected</th>
<th>Percentage of Success%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR1</td>
<td>511</td>
<td>56.78</td>
</tr>
<tr>
<td>MR2</td>
<td>520</td>
<td>57.78</td>
</tr>
<tr>
<td>MR3</td>
<td>900</td>
<td>100</td>
</tr>
<tr>
<td>MR4</td>
<td>895</td>
<td>99.44</td>
</tr>
<tr>
<td>MR5</td>
<td>881</td>
<td>97.89</td>
</tr>
<tr>
<td>MR6</td>
<td>877</td>
<td>97.44</td>
</tr>
<tr>
<td>MR7</td>
<td>900</td>
<td>100</td>
</tr>
<tr>
<td>MR8</td>
<td>900</td>
<td>100</td>
</tr>
</tbody>
</table>

Table B.5: Fault-Detection by Individual MRs of QAP

<table>
<thead>
<tr>
<th>MR</th>
<th>Maximum%</th>
<th>Minimum%</th>
<th>Average%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR1</td>
<td>100</td>
<td>26</td>
<td>84.87</td>
</tr>
<tr>
<td>MR2</td>
<td>100</td>
<td>26</td>
<td>85.31</td>
</tr>
<tr>
<td>MR3</td>
<td>100</td>
<td>26</td>
<td>85.03</td>
</tr>
<tr>
<td>MR4</td>
<td>100</td>
<td>26</td>
<td>92.55</td>
</tr>
<tr>
<td>MR5</td>
<td>100</td>
<td>26</td>
<td>88.87</td>
</tr>
</tbody>
</table>

Table B.6: Line Coverage Achieved by Individual MRs of TritTyp
## APPENDIX B. EXPERIMENTAL DATA VALUES

### Table B.7: Branch Coverage Achieved by Individual MRs of TriTyp

<table>
<thead>
<tr>
<th>MR</th>
<th>Maximum%</th>
<th>Minimum%</th>
<th>Average%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR1</td>
<td>100</td>
<td>5</td>
<td>87.01</td>
</tr>
<tr>
<td>MR2</td>
<td>100</td>
<td>5</td>
<td>87.34</td>
</tr>
<tr>
<td>MR3</td>
<td>100</td>
<td>5</td>
<td>87.18</td>
</tr>
<tr>
<td>MR4</td>
<td>100</td>
<td>5</td>
<td>93.94</td>
</tr>
<tr>
<td>MR5</td>
<td>100</td>
<td>5</td>
<td>88.61</td>
</tr>
</tbody>
</table>

### Table B.8: Line Coverage Achieved by Individual MRs of TCAS

<table>
<thead>
<tr>
<th>MR</th>
<th>Maximum%</th>
<th>Minimum%</th>
<th>Average%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR1</td>
<td>89.2</td>
<td>81.5</td>
<td>85.88</td>
</tr>
<tr>
<td>MR2</td>
<td>89.2</td>
<td>81.5</td>
<td>85.82</td>
</tr>
<tr>
<td>MR3</td>
<td>89.2</td>
<td>81.5</td>
<td>86.26</td>
</tr>
<tr>
<td>MR4</td>
<td>89.2</td>
<td>80</td>
<td>85.07</td>
</tr>
<tr>
<td>MR5</td>
<td>89.7</td>
<td>80</td>
<td>85.07</td>
</tr>
<tr>
<td>MR6</td>
<td>89.2</td>
<td>73.4</td>
<td>80.31</td>
</tr>
<tr>
<td>MR7</td>
<td>89.2</td>
<td>46.2</td>
<td>82.31</td>
</tr>
<tr>
<td>MR8</td>
<td>89.2</td>
<td>46.2</td>
<td>85.56</td>
</tr>
<tr>
<td>MR9</td>
<td>89.2</td>
<td>46.2</td>
<td>86.48</td>
</tr>
<tr>
<td>MR10</td>
<td>89.2</td>
<td>81.2</td>
<td>85.47</td>
</tr>
<tr>
<td>MR11</td>
<td>89.2</td>
<td>81.2</td>
<td>86.48</td>
</tr>
<tr>
<td>MR12</td>
<td>90.8</td>
<td>46.2</td>
<td>86.92</td>
</tr>
<tr>
<td>MR13</td>
<td>90.8</td>
<td>46.2</td>
<td>88.15</td>
</tr>
<tr>
<td>MR14</td>
<td>89.2</td>
<td>46.2</td>
<td>86.81</td>
</tr>
</tbody>
</table>

### Table B.9: Branch Coverage Achieved by Individual MRs of TCAS

<table>
<thead>
<tr>
<th>MR</th>
<th>Maximum%</th>
<th>Minimum%</th>
<th>Average%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR1</td>
<td>90.9</td>
<td>53</td>
<td>69.94</td>
</tr>
<tr>
<td>MR2</td>
<td>90.9</td>
<td>53</td>
<td>69.48</td>
</tr>
<tr>
<td>MR3</td>
<td>90.9</td>
<td>43.8</td>
<td>70.44</td>
</tr>
<tr>
<td>MR4</td>
<td>92.2</td>
<td>43.9</td>
<td>56.38</td>
</tr>
<tr>
<td>MR5</td>
<td>92.2</td>
<td>48.4</td>
<td>52.42</td>
</tr>
<tr>
<td>MR6</td>
<td>92.2</td>
<td>41.9</td>
<td>52.69</td>
</tr>
<tr>
<td>MR7</td>
<td>90.9</td>
<td>19.7</td>
<td>51.28</td>
</tr>
<tr>
<td>MR8</td>
<td>93.9</td>
<td>19.7</td>
<td>70.26</td>
</tr>
<tr>
<td>MR9</td>
<td>93.9</td>
<td>19</td>
<td>73.93</td>
</tr>
<tr>
<td>MR10</td>
<td>90.9</td>
<td>51.6</td>
<td>69.89</td>
</tr>
<tr>
<td>MR11</td>
<td>92.2</td>
<td>47</td>
<td>73.96</td>
</tr>
<tr>
<td>MR12</td>
<td>93.9</td>
<td>13.6</td>
<td>77.03</td>
</tr>
<tr>
<td>MR13</td>
<td>93.9</td>
<td>10.9</td>
<td>87.92</td>
</tr>
<tr>
<td>MR14</td>
<td>93.9</td>
<td>10.6</td>
<td>78.32</td>
</tr>
</tbody>
</table>
### APPENDIX B. EXPERIMENTAL DATA VALUES

<table>
<thead>
<tr>
<th>MR</th>
<th>Maximum%</th>
<th>Minimum%</th>
<th>Average%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR1</td>
<td>88</td>
<td>73</td>
<td>85.2</td>
</tr>
<tr>
<td>MR2</td>
<td>88</td>
<td>73</td>
<td>85.22</td>
</tr>
<tr>
<td>MR3</td>
<td>88</td>
<td>74</td>
<td>85.28</td>
</tr>
<tr>
<td>MR4</td>
<td>87</td>
<td>73</td>
<td>85.18</td>
</tr>
<tr>
<td>MR5</td>
<td>89</td>
<td>75</td>
<td>86.31</td>
</tr>
<tr>
<td>MR6</td>
<td>88</td>
<td>73</td>
<td>85.21</td>
</tr>
<tr>
<td>MR7</td>
<td>88</td>
<td>74</td>
<td>86.5</td>
</tr>
<tr>
<td>MR8</td>
<td>88</td>
<td>74</td>
<td>85.28</td>
</tr>
<tr>
<td>MR9</td>
<td>91</td>
<td>78</td>
<td>87.66</td>
</tr>
<tr>
<td>MR10</td>
<td>88</td>
<td>75</td>
<td>86.03</td>
</tr>
</tbody>
</table>

Table B.10: Line Coverage Achieved by Individual MRs of KNAPSACK

<table>
<thead>
<tr>
<th>MR</th>
<th>Maximum%</th>
<th>Minimum%</th>
<th>Average%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR1</td>
<td>84</td>
<td>65</td>
<td>80.18</td>
</tr>
<tr>
<td>MR2</td>
<td>85</td>
<td>65</td>
<td>80.2</td>
</tr>
<tr>
<td>MR3</td>
<td>84</td>
<td>66</td>
<td>81.18</td>
</tr>
<tr>
<td>MR4</td>
<td>84</td>
<td>65</td>
<td>80.16</td>
</tr>
<tr>
<td>MR5</td>
<td>85</td>
<td>67</td>
<td>81.29</td>
</tr>
<tr>
<td>MR6</td>
<td>84</td>
<td>65</td>
<td>80.21</td>
</tr>
<tr>
<td>MR7</td>
<td>87</td>
<td>67</td>
<td>82.49</td>
</tr>
<tr>
<td>MR8</td>
<td>84</td>
<td>66</td>
<td>80.68</td>
</tr>
<tr>
<td>MR9</td>
<td>86</td>
<td>71</td>
<td>82.52</td>
</tr>
<tr>
<td>MR10</td>
<td>85</td>
<td>68</td>
<td>82.38</td>
</tr>
</tbody>
</table>

Table B.11: Branch Coverage Achieved by Individual MRs of KNAPSACK

<table>
<thead>
<tr>
<th>MR</th>
<th>Maximum%</th>
<th>Minimum%</th>
<th>Average%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR1</td>
<td>100</td>
<td>52</td>
<td>98.47</td>
</tr>
<tr>
<td>MR2</td>
<td>100</td>
<td>52</td>
<td>98.47</td>
</tr>
<tr>
<td>MR3</td>
<td>100</td>
<td>52</td>
<td>98.49</td>
</tr>
<tr>
<td>MR4</td>
<td>100</td>
<td>52</td>
<td>98.49</td>
</tr>
<tr>
<td>MR5</td>
<td>100</td>
<td>52</td>
<td>98.51</td>
</tr>
<tr>
<td>MR6</td>
<td>100</td>
<td>52</td>
<td>98.51</td>
</tr>
<tr>
<td>MR7</td>
<td>100</td>
<td>52</td>
<td>98.49</td>
</tr>
<tr>
<td>MR8</td>
<td>100</td>
<td>52</td>
<td>98.49</td>
</tr>
</tbody>
</table>

Table B.12: Line Coverage Achieved by Individual MRs of QAP

<table>
<thead>
<tr>
<th>MR</th>
<th>Maximum%</th>
<th>Minimum%</th>
<th>Average%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR1</td>
<td>99</td>
<td>40</td>
<td>97.34</td>
</tr>
<tr>
<td>MR2</td>
<td>99</td>
<td>40</td>
<td>97.33</td>
</tr>
<tr>
<td>MR3</td>
<td>99</td>
<td>40</td>
<td>97.37</td>
</tr>
<tr>
<td>MR4</td>
<td>99</td>
<td>40</td>
<td>97.35</td>
</tr>
<tr>
<td>MR5</td>
<td>99</td>
<td>40</td>
<td>97.38</td>
</tr>
<tr>
<td>MR6</td>
<td>99</td>
<td>40</td>
<td>97.40</td>
</tr>
<tr>
<td>MR7</td>
<td>99</td>
<td>40</td>
<td>97.35</td>
</tr>
<tr>
<td>MR8</td>
<td>99</td>
<td>40</td>
<td>97.35</td>
</tr>
</tbody>
</table>

Table B.13: Branch Coverage Achieved by Individual MRs of QAP
Appendix C

Publication