IMPROVING THE EFFECTIVENESS OF METAMORPHIC TESTING USING SYSTEMATIC TEST CASE GENERATION

by

Prashanta Saha

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

Doctor of Philosophy

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

May 2024

©COPYRIGHT

by

Prashanta Saha

2024

All Rights Reserved

DEDICATION

I dedicate this dissertation to my mother and my wife.

ACKNOWLEDGEMENTS

I would like to thank my PhD advisor Dr. Clemente Izurieta for the valuable comments, suggestions and guidance. I would also like to thank my previous advisor Dr. Upulee Kanewala for her collaborative ideas. I am also grateful to my PhD committee members Dr. Mike Wittie, Dr. Brendan Mumey, and Dr. John Paxton for their insightful feedback. I would like to thank my lab partner Madhusudan Srinivasan and Karishma Rahman for the effective discussions during this journey.

My research is partially funded by the award number 1656877 from the National Science Foundation.

TABLE OF CONTENTS

1. INTRODUCTION 1

1.1 Problem Decomposition 3
1.2 Contributions of the Dissertation 7
1.3 Overview of the Dissertation 9

2. BACKGROUND 10

2.1 Metamorphic Testing 10
2.2 Source Test Case Generation 12
2.2.1 Random Test Case Generation 12
2.2.2 Search Based Test Case Generation 13
2.2.2.1 Line Coverage 15
2.2.2.2 Branch Coverage 15
2.2.2.3 Weak Mutation Coverage 15
2.3 Mutation Testing 16
2.3.1 Mutant Reduction Strategies 17
2.3.1.1 Random Sampling Technique 17
2.3.1.2 Operator Based Mutant Selection 18
2.4 Scientific Applications 18
2.4.1 Numerical Applications 18
2.4.2 Machine Learning Applications 20

3. RELATED WORK 21

4. FAULT DETECTION EFFECTIVENESS OF SOURCE TEST CASE GENERATION STRATEGIES FOR METAMORPHIC TESTING 23

4.1 Contribution of Authors and Co-Authors 23
4.2 Manuscript Information 24
4.3 Abstract 25
4.4 Introduction 26
4.5 Background 27
4.5.1 Metamorphic Testing 27
4.5.2 Source Test Case Generation 28
4.5.2.1 Line Coverage 28
4.5.2.2 Branch Coverage 29
4.5.2.3 Weak Mutation 31
4.6 Evaluation Method 31

TABLE OF CONTENTS – CONTINUED

4.6.1 Code Corpus 32
4.6.2 METtester 36
4.6.3 Experimental Setup 38
4.7 Results and Discussion 39
4.7.1 Effectiveness of the Source Test Case Generation Techniques 39
4.7.2 Fault Finding Effectiveness of Combined Source Test Cases 40
4.7.3 Fault Finding Effectiveness of Individual MRs 41
4.7.4 Impact of Source Test Suite Size 42
4.8 Threats to Validity 43
4.9 Related Work 44
4.10 Conclusions & Future Work 45

5. USING METAMORPHIC RELATIONS TO IMPROVE THE EFFECTIVENESS OF AUTOMATICALLY GENERATED TEST CASES 47

5.1 Contribution of Authors and Co-Authors 47
5.2 Manuscript Information 48
5.3 Abstract 49
5.4 Introduction 50
5.5 Background 54
5.5.1 Metamorphic Testing 54
5.5.2 Automated Test Case Generation 56
5.6 Empirical Evaluation 56
5.6.1 Research Questions 56
5.6.2 Subject Programs 57
5.6.3 MR Identification 58
5.6.4 Automated Test Case Generation 61
5.6.5 Utilizing MRs to Modify Automatically Generated Test Cases 62
5.6.6 Evaluation Approach 63
5.7 Results and Discussions 65
5.8 Threats to Validity 68
5.9 Related Work 69
5.10 Conclusion and Future Work 70

TABLE OF CONTENTS – CONTINUED

6. A TEST SUITE MINIMIZATION TECHNIQUE FOR TESTING NUMERICAL PROGRAMS 72

6.1 Contribution of Authors and Co-Authors 72
6.2 Manuscript Information 73
6.3 Abstract 74
6.4 Introduction 75
6.5 Background 76
6.5.1 Metamorphic Testing 77
6.5.2 Coverage-based Test Case Generation 78
6.5.2.1 Line Coverage 78
6.5.2.2 Branch Coverage 79
6.5.2.3 WM 79
6.6 Mutation Testing 80
6.7 Test Suite Minimization Approach 80
6.7.1 Mutants Reduction Approach 81
6.7.1.1 Random Sampling Technique 82
6.7.1.2 Operator Based Mutant Selection 82
6.7.2 Effective Test Suites Selection 83
6.8 Empirical Evaluation 83
6.8.1 Research Questions 83
6.8.2 Subject Programs 84
6.8.3 MR Identification 85
6.8.4 Evaluation Approach 87
6.9 Results and Discussions 88
6.10 Threats to Validity 94
6.11 Conclusion 94

7. FAULT DETECTION EFFECTIVENESS OF METAMORPHIC RELATIONS DEVELOPED FOR TESTING SUPERVISED CLASSIFIERS 96

7.1 Contribution of Authors and Co-Authors 96
7.2 Manuscript Information 97
7.3 Abstract 98
7.4 Introduction 99
7.5 Background 101
7.5.1 Supervised Machine Learning Classifiers 101
7.5.2 K-Nearest Neighbors 101
7.6 Metamorphic Testing for Supervised Classifiers 102

TABLE OF CONTENTS – CONTINUED

7.6.1 Identified MRs for Testing KNN 104
7.7 Experimental Study 106
7.7.1 Research Questions 106
7.7.2 Source and Follow-up Test Cases 108
7.7.3 Mutation Analysis 108
7.8 Results and Discussions 110
7.9 Related Work 113
7.10 Conclusions and Future Work 115

8. MRSYN: A TEST CASE GENERATION AND MINIMIZATION APPROACH FOR TESTING MACHINE LEARNING APPLICATIONS 117

8.1 Contribution of Authors and Co-Authors 117
8.2 Manuscript Information 118
8.3 Abstract 119
8.4 Introduction 120
8.5 Background 123
8.5.1 Supervised Machine Learning Classifiers 123
8.5.2 Metamorphic Testing for Supervised Classifiers 126
8.5.3 Mutation Testing 130
8.6 Research Goals 132
8.7 Proposed Approach 136
8.7.1 Source Test Case Generation 136
8.7.2 Test Suite Minimization Approach 138
8.8 Experimental Setup 140
8.8.1 Research Questions 141
8.8.2 MRs Identification for Supervised Classifiers 142
8.8.3 Generation of Source Test Cases 142
8.8.4 Evaluation Approach 143
8.9 Results & Discussion 145
8.10 Threats to Validity 152
8.11 Related Works 153
8.12 Conclusions 154

9. CONCLUSION & FUTURE WORK 156

9.1 Conclusion 156
9.2 Future Work 158

TABLE OF CONTENTS – CONTINUED

REFERENCES CITED 160

LIST OF TABLES

Table Page

4.1 All methods with mutants kill rates and test suite size for each source test case generation technique 33

4.2 Total number of methods having the highest mutants kill rate for each source test generation techniques. 39

4.3 Total % of mutants killed after combining weak mutation, line, branch coverage, and random testing 41

4.4 Average test suites size for weak mutation, line coverage, branch coverage and random 43

5.1 Classes with Evosuite test suite & developer test suite 62

5.2 Classes with MRs and mutants generation results 64

6.1 Individual classes from five open source projects. We show counts of Methods, MRs and mutants 88

6.2 Savings obtained by operator based mutant reduction approach 90

6.3 Cost effectiveness of test suite minimization technique for MT based on test suite size 92

6.4 Percentage reduction on fault detection and code coverage after applying test suite minimization technique in MT 93

7.1 Sample data set 103

7.2 Metamorphic relations for kNN used in mutation analysis 107

7.3 Selected files for mutation analysis 110

7.4 Details of mutants 110

8.1 MRs for supervised classifiers. The significance of ✓is that particular MR supports the validation aspect of the supervised classifier. Validation aspect aim to check whether the algorithms under test meet the general user expectations or not. 127

LIST OF TABLES – CONTINUED

Table Page

8.2 Sample example of a dataset in ARFF format (Attribute-Relation File Format). This format has header and data information. The header contains attribute names and their data types (@attribute). The data contains the data set (@data). 131

8.3 Property parameters that help inform the generation of tuned source test cases. These parameters are used to generate datasets in Scikit-learn package. 134

8.4 Distribution of MRSyn datasets (# of test cases) and their characteristics 140

8.5 Details of the 4 supervised classifiers with their total lines of code and # of mutants generated. These classifiers are from Weka tool. The class and method names that we have used in this research are mentioned below. Total lines of code were calculated by excluding the comments in that method. 144

8.6 Percentage reduction on test suite size and fault detection after applying test suite minimization technique in MT 151

LIST OF FIGURES

Figure Page

1.1 Problem Decomposition....4

2.1 Illustration of Metamorphic Testing.... 11

4.1 Test source and follow-up inputs on PUT. 28

4.2 METtester architecture. 37

4.3 An example of the XML input given to METtester. 38

4.4 Total % of mutants killed by each source test suite generation technique.... 40

4.5 Venn Diagram for all the combinations of source test suites that performed best for each individual methods.... 41

4.6 % of mutants killed by all six MRs using 4 test suite strategies (Branch, Line Coverage, Weak Mutation and Random) 42

5.1 (Left) EvoSuite generated test case , (Right) Modified test case with MR in MT 52

5.2 Illustration of Metamorphic Testing 54

5.3 Mutation score of 4 classes for auto generated test suites (Evosuite(E)) and Developer test suites (D) and Metamorphic Testing, SRS = SquareRootSolver, LSS = LeastSquaresSolver, FBSS = ForwardBackSubstitutionSolver 65

5.4 Fault Detection Ratio of AGTS (Automatically Generated Test Suites) and MR augmented test suites for 3 classes. 67

6.1 Test source and follow-up inputs on PUT. 78

LIST OF FIGURES – CONTINUED

Figure | Page

6.2 Average mutation scores of mutation testing strategies. Mutation testing was done by the following strategies: total mutants set, operator based set, Random 10%, Random 20%, Random 30%, Random 40%, Random 50%, and Random 60%. The average mutation score was calculated by averaging the mutation scores of all the methods from each of the classes. 89

6.3 Comparison of average mutation scores of coverage based test suites, e.g. Line, Branch, and Weak Mutation. The average mutation score was calculated by averaging the mutation scores of all the methods from each of the classes. 91

7.1 Mutant kill rate for each MR by varying mutants size. 111

7.2 Mutant kill rate by each MR in kNN with varying data set. 113

7.3 Average mutants kill rate by each MR for MuJava and Major tool. 114

8.1 Supervised classifier application workflow. 123

8.2 Comparison between mutation scores of Random and MRSyn based datasets for 4 supervised classifier algorithms. These two datasets were applied as source test suites for MT. 145

8.3 After applying the MRSyn approach to the source test suites, the mutation scores of MRs demonstrate an increase. 147

8.4 Individual mutation scores of MRs compared with source test suites for each subject program. These MRs were selected based on their validation aspects described in Table 8.1. 148

ABSTRACT

Metamorphic testing is a well-known approach to tackle the oracle problem in software testing. This technique requires source test cases that serve as seeds for the generation of follow-up test cases. Systematic design of test cases is crucial for the test quality. Thus, source test case generation strategy can make a big impact on the fault detection effectiveness of metamorphic testing. Most of the previous studies on metamorphic testing have used either random test data or existing test cases as source test cases. There has been limited research done on systematic source test case generation for metamorphic testing. This thesis explores innovative methods for enhancing the effectiveness of Metamorphic Testing through systematic generation of source test cases. It addresses the challenge of testing complex software systems, including numerical programs and machine learning applications, where traditional testing methods are limited by the absence of a reliable oracle. By focusing on structural, mutation coverage criteria, and characteristics of machine learning datasets, the research introduces strategies to generate source test cases that are more effective in fault detection compared to random test case generation. The proposed techniques include leveraging structural and mutation coverage for numerical programs and aligning random values with machine learning properties for supervised classifier applications. These techniques are integrated into the METTester tool, automating the process and potentially reducing testing costs by minimizing the test suite without sacrificing quality. The thesis demonstrates that tailored source test case generation can significantly improve the fault detection capabilities of Metamorphic Testing, offering substantial benefits in terms of cost efficiency and reliability in software testing.

CHAPTER ONE

INTRODUCTION

Software testing is an integral part of the Software Development Life Cycle. It is a costly activity, yet essential to detect faults. More than 50% of software development costs often account for software testing [3]. Therefore, there is a certain benefit in reducing the total cost and improving the effectiveness of software testing by automating the process. The test case generation process is one of the intellectually demanding tasks in software testing. Automation of test case generation is a challenging task because it has a strong impact on the effectiveness and efficiency of the software testing process. A great amount of research effort has been performed on automated test case generation. As a result, a significant number of automated test case generation techniques have been investigated and proposed [36].

Carlos et al. [63] proposed a novel random test case generation approach. They utilized the feedback obtained during the execution of incrementally constructed test inputs, which are evaluated against predefined contracts and filters to assess their usefulness. This feedback-directed technique not only generates a suite of unit tests for assessing code contract adherence and identifying potential errors but also demonstrates superior performance in coverage and error detection when compared to both systematic and traditional undirected random test generation methods. Experimental validation on various data structures and large software libraries reveals the method's efficiency in uncovering previously undetected errors, thereby outperforming existing techniques like model checking and undirected random generation in terms of efficiency and effectiveness. Pietro et al. [11] proposed an

improved symbolic execution-based test case generation approach that overcomes limitations such as complex data structures and non-linear expressions, leading to invalid inputs, unidentified infeasible traces, and false alarms faced by traditional symbolic execution. This approach combined executable preconditions and invariant design tailored to the lazy initialization algorithm, along with rewrite rules within the symbolic executor for simplifying inverse relationships. Nikolai et al. [84] designed a tool called **Pex** for .NET programs that leverages dynamic symbolic execution to automatically generate a compact test suite with high code coverage. It monitors execution traces to understand program behavior and employs a constraint solver to find new test inputs that explore different behaviors within the program. In a notable case study on a well-tested component of the .NET runtime, **Pex** successfully identified errors, including a critical issue, demonstrating its efficiency in uncovering hidden bugs.

However, all these test case generation approaches are suitable for relatively less complicated software applications. However, complex systems such as scientific software or machine learning applications have an underlying common problem called *oracle* problem. An *oracle* is used to check whether the output produced for a given test input is correct [87]. Metamorphic Testing (MT) is a technique proposed to alleviate the *oracle* problem of software under test (SUT) [19]. The main idea is that most of the time it is easier to predict relations between outputs of a program, than understanding its input-output behavior. For example, consider a program that computes the average of a list of real numbers. It is hard to correctly predict the observed output when the input list has millions of real numbers. So, we can not validate whether the returned average is correct. But, we can permute the list of real numbers and check if the returned output matches with the previous output (considering some roundoff error tolerance). If the outputs do not match,

then there is a 'bug' (fault) in the program. This type of property is called a *metamorphic relation* (MR), which is a necessary property of the SUT and specifies a relationship between multiple inputs and their outputs [21]. MT has been successful in finding bugs in systems across various domains. MT has been successfully applied to detecting previously unknown faults in different domains such as web services, computer graphics, simulation and modeling, and embedded systems etc. [78].

To date, work done on improving the fault detection effectiveness of MT has mainly focused on developing quality MRs. However, developing such MRs is a labor-intensive task that requires the involvement of domain experts. Another, avenue to improve the fault detection effectiveness of MT, which has not been explored so far, is to systematically generate the source test cases. Most of the previous studies in MT have used randomly generated test cases or existing test cases as source test cases when conducting MT [8,20,38,79,94]. Our work shows that the effectiveness of MT can be improved by systematically generating the source test cases [76]. Such a systematic approach can also reduce the size of the test suite which will reduce the testing cost. This thesis aims to develop methods to systematically generate source test cases to improve the fault detection effectiveness of MT.

1.1 Problem Decomposition

In this section, we discuss the problem statement and knowledge questions (KQ) addressed by this thesis. Figure 1.1 depicts the practical problem and knowledge questions.

To address the stated problem of finding a better test case generation approach than random test case generation, we selected two scientific software application domains i.e., numerical programs and supervised classifier algorithms where the random test case generation approach was applied to test the applications [79]. We analyzed

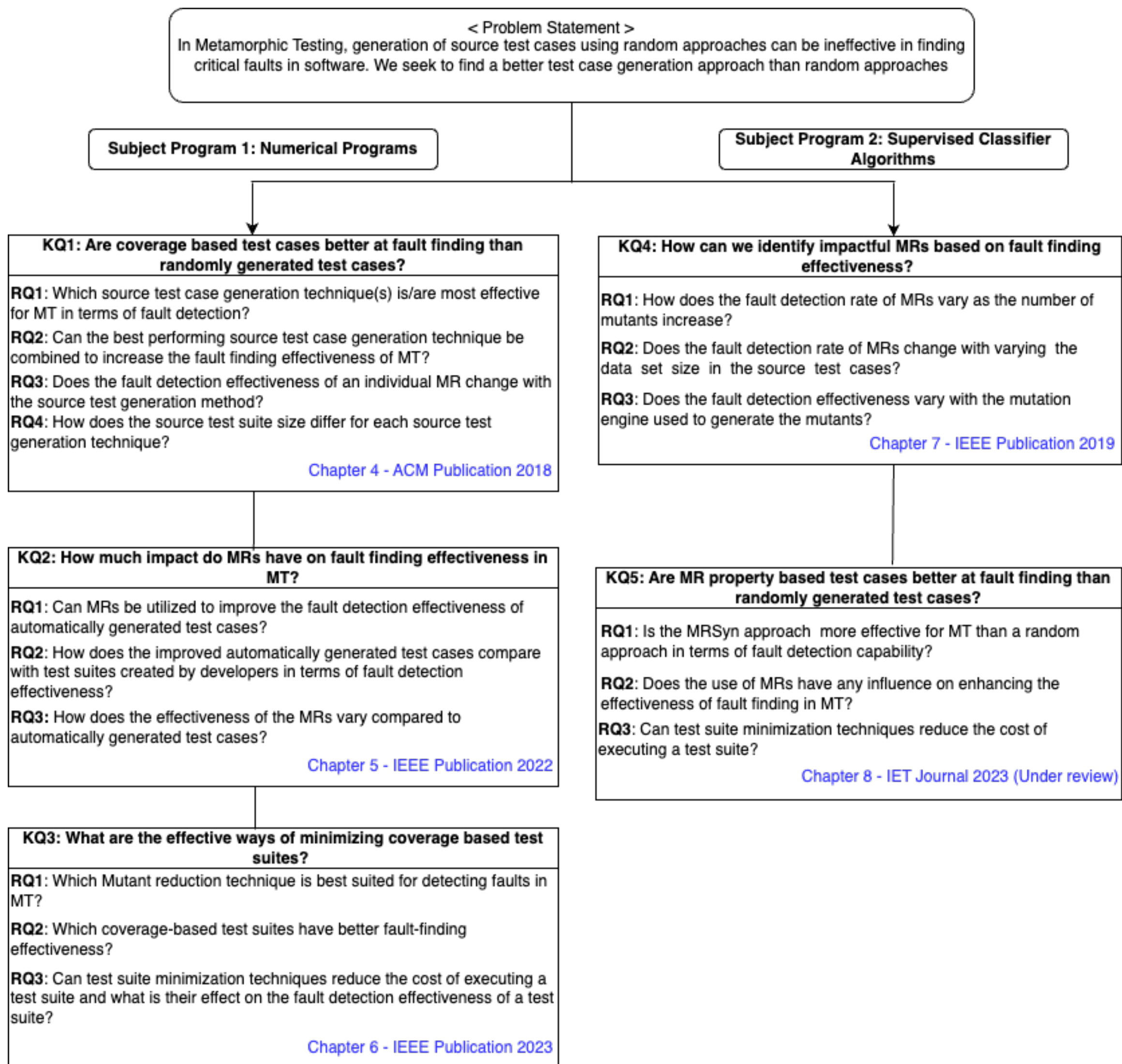


Figure 1.1: Problem Decomposition

five KQs to find an answer to our stated problem. KQ1 to KQ3 address numerical programs and KQ4 and KQ5 address supervised classifier algorithms.

For KQ1 (*Are coverage-based test cases better at fault finding than randomly generated test cases?*), we decomposed our knowledge question into four research questions (RQs):

RQ1: Which source test case generation technique(s) is/are most effective for MT in terms of fault detection?

RQ2: Can the best performing source test case generation techniques be combined to increase the fault finding effectiveness of MT?

RQ3: Does the fault detection effectiveness of an individual MR change with the source test generation method?

RQ4: How does the source test suite size differ for each source test generation technique?

The results of this study helped confirm that coverage-based test case generation approaches have better fault-finding effectiveness than random test case generation approaches in MT of numerical programs. However, since the quality of the MRs also has an impact on the effectiveness of MT, in KQ2 (*How much impact do MRs have on fault finding effectiveness of MT?*), we focus on finding impactful MRs. We decompose this knowledge question into the following three RQs:

RQ1: Can MRs be utilized to improve the fault detection effectiveness of automatically generated test cases?

RQ2: How do the improved automatically generated test cases compare with test suites created by developers in terms of fault detection effectiveness?

RQ3: How does the effectiveness of the MRs vary compared to automatically generated test cases?

Although coverage-based test cases have better fault-finding effectiveness, there can be test cases that overlap and cover the same lines of code to detect faults. This increases the time and budget of testing. In KQ3 (*What are the effective ways of minimizing coverage-based test suites?*), we answered the following three RQs which help reduce the time and budget of MT.

RQ1: Which Mutant reduction technique is best suited for detecting faults in MT?

RQ2: Which coverage-based test suites have better fault-finding effectiveness?

RQ3: Can test suite minimization techniques reduce the cost of executing a test suite and what is their effect on the fault detection effectiveness of a test suite?

After studying subject program 1: numerical programs, we move on to subject program 2: supervised classifier algorithms. To find a better test case generation approach than random test case generation, first, we answered KQ4 (*How can we identify impactful MRs based on fault finding effectiveness?*). To test supervised classifier algorithms using MT, identifying MRs plays an important role [96]. We answered this question by decomposing it into the following three RQs:

RQ1: How does the fault detection rate of MRs vary as the number of mutants increases?

RQ2: Does the fault detection rate of MRs change with varying the data set size in the source test cases?

RQ3: Does the fault detection effectiveness vary with the mutation engine used to generate the mutants?

To answer KQ5 (*Are MR property based test cases better at fault finding than randomly generated test cases?*), we use the properties of impactful MRs, developed a test case generation approach, and answered the following three following RQs:

RQ1: Is the MRSyn approach more effective for MT than a random approach in terms of fault detection capability?

RQ2: Does the use of MRs have any influence on enhancing the effectiveness of fault finding in MT?

RQ3: Can test suite minimization techniques reduce the cost of executing a test suite?

1.2 Contributions of the Dissertation

The contributions outlined in the text aim to evolve the generation of source test cases by leveraging structural, mutation coverage criteria, and characteristics of machine learning datasets. These methodologies are not only innovative but also tailored to address specific domains such as numerical programs and supervised machine learning classifier applications.

1. Utilize structural and mutation coverage criteria to generate source test cases for MRs. This proposed technique can be used to generate source test cases for testing numerical programs. Source test cases will be generated based on three coverage criteria: line, branch, and weak mutation.

2. Utilize characteristics of the machine learning (ML) dataset to generate source test cases. This proposed technique can be used to generate source test cases for testing supervised ML classifier applications. The basic idea is to generate some

random values and align those in the dataset according to some ML properties. Our main goal is to test the correctness functionality of ML.

3. Integrate these two source case generation techniques into the METTester: a Metamorphic Testing tool, to automate the source test case generation. This is a significant leap towards automating the process of source test case generation. The automation aspect is crucial for scalability and for enabling continuous testing practices, particularly in agile and DevOps environments.

The hypothesized benefits of this proposed technique over random test case generation highlight its potential to transform the testing landscape significantly:

1. By minimizing the number of source test cases needed without compromising on the quality of testing, this technique can lead to substantial cost savings. This reduction is achieved through identifying effective test cases that target critical areas of the code and application functionality, thereby eliminating redundant or less effective tests.

2. The precision in generating test cases tailored to uncover faults in specific domains (i.e. numerical programs and ML applications) enhances the likelihood of detecting errors. This precision, combined with the comprehensive coverage criteria, ensures a wide array of potential faults are tested, increasing the overall effectiveness of the testing process.

3. By providing concrete insights into the significant MRs concerning fault detection capabilities, this approach not only aids in the immediate goal of testing but also contributes to the broader knowledge base regarding effective testing strategies. This understanding can guide the development of future testing techniques and the refinement of existing methodologies.

1.3 Overview of the Dissertation

The rest of the thesis is structured as follows. Chapter 2 describes the basics of Metamorphic testing, source test case generation strategies, Mutation testing and test case reduction strategies, and scientific application domains as subject programs in this thesis. Chapter 3 discusses the related works that contributed to the test case generation for MT. Chapter 4 presents a coverage-based source test generation approach and its effectiveness in testing numerical programs using MT. Chapter 5 discusses the impact of MRs' identification to improve the effectiveness of source test cases to test numerical programs. Chapter 6 presents the effectiveness of the test suite minimization technique in MT for testing numerical programs. Chapter 7 discusses the importance of identifying good MRs for MT to test supervised algorithms. Chapter 8 presents a test case generation and minimization approach to test machine learning applications. Chapter 9 summarises all contributions and findings as the conclusions of this thesis.

CHAPTER TWO

BACKGROUND

In this chapter, we provided some background on MT, source test case generation approaches, Mutation Testing approaches, and scientific application domains used in this research.

2.1 Metamorphic Testing

Following is the typical process used for applying MT:

1. Identify MRs from the specification of the SUT. An MR $R(x_1, x_2, ..., x_n, f(x_1), f(x_2), ..., f(x_n))$ is a necessary property of the SUT and is specified over the inputs $x_1, x_2, ..., x_n$ and their corresponding outputs $f(x_1), f(x_2), ..., f(x_n)$.
2. Generate the source test inputs $x_1, x_2, ..., x_k$ and execute them on the SUT.
3. (a) Construct the follow-up test inputs $x_{k+1}, x_{k+2}, ..., x_n$ by applying a transformation specified by R to $x_1, x_2, ..., x_k, f(x_1), f(x_2), ..., f(x_k)$.
   (b) Execute the follow-up test cases.
4. Verify whether R is satisfied with the obtained $x_1, x_2, ..., x_n, f(x_1), f(x_2), ..., f(x_n)$ by executing the SUT. If R is not satisfied the MR has revealed a fault in the SUT.

Identification of MRs is typically done based on the knowledge of the subject program (step 1). Several works have been done towards automating the MR identification [47]. Any test case generation technique can be applied to generate source test cases (step 2). Special case [94] and random testing [51] techniques have

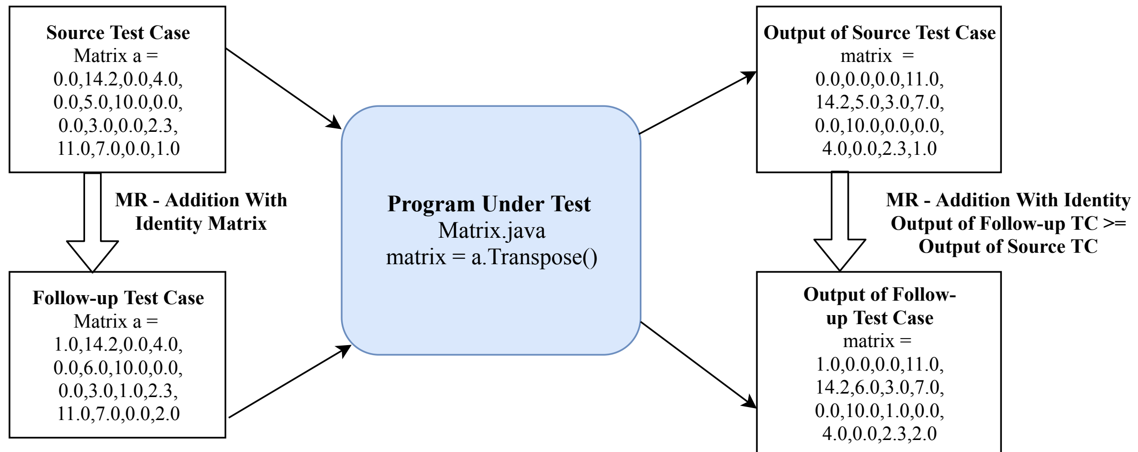


Figure 2.1: Illustration of Metamorphic Testing.

been used to generate source test cases. Further, previous studies have shown that using coverage-based test inputs as source inputs would improve the fault detection effectiveness of MT compared to random test inputs [76]. As shown in the above process, since MT checks the relationship between inputs and outputs of a test program, we can use this technique when the expected results of individual test inputs are unknown. The following example is a sample of MT process. In figure 2.1, a Java method *Transpose* from *Matrix.java* class is used to show how source and follow-up test cases perform with a program under test (PUT). The *Transpose* method transposes a matrix and returns the transposed matrix. Source test case, $a = \{(0.0, 14.2, 0.0, 4.0), (0.0, 5.0, 10.0, 0.0), (0.0, 3.0, 0.0, 2.0), (11.0, 7.0, 0.0, 1.0)\}$ is developer generated and tested on the *Transpose* method. The output for this source test case is $matrix = \{(0.0, 0.0, 0.0, 11.0), (14.2, 5.0, 3.0, 7.0), (0.0, 10.0, 0.0, 0.0), (4.0, 0.0, 2.3, 1.0)\}$. For this program, when an identity matrix is added to the input, the output should increase. This will be used as an MR to conduct MT on this PUT. An identity matrix of size 4 is added to this matrix to create a follow-up test case $a' = \{(1.0, 14.2, 0.0, 4.0), (0.0, 6.0, 10.0, 0.0), (0.0, 3.0, 1.0, 2.3), (11.0, 7.0, 0.0, 2.0)\}$ and

then execute on the PUT. The output for this follow-up test case is $matrix = \{(1.0, 0.0, 0.0, 11.0), (14.2, 6.0, 3.0, 7.0), (0.0, 10.0, 1.0, 0.0), (4.0, 0.0, 2.3, 2.0)\}$. To satisfy this MR the follow-up test output should be greater than the source output. We calculate the sum of elements of the matrices from source and follow-up outputs which is 57.5 and 61.5 respectively. In this MT example, 61.5 is greater than 57.5 then the considered MR is satisfied for this given source and follow-up test cases.

Previous studies have consistently shown that [53, 66, 82, 97] MT has many advantages. First and foremost, MT provides a test result verification mechanism during the absence of an oracle. The test results are verified against a set of MRs instead of an oracle. Besides, most MRs are simple in concept, so it is convenient to verify test results by using some simple scripts automatically.

2.2 Source Test Case Generation

In MT, source test cases serve as seeds for the generation of follow-up test cases. Source test cases can be generated using any traditional testing techniques. In most of the MT approaches, researchers either use random testing or existing test suites for the source test case generation. My research goal is to find test case generation techniques that have better fault detection effectiveness.

2.2.1 Random Test Case Generation

Random testing (RT) selects test cases randomly from all possible input values. This is the poorest technique of all, according to Myers [59]. He argued that a collection of randomly selected test cases had little chance of being optimal, or close to optimal, in terms of the probability of detecting the most errors. He advised that the tester should select test cases more intelligently than RT. On the other hand, compared to other techniques, the automated implementation of RT is

much more cost-effective because it is computationally inexpensive. Based on this advantage, Duran and Wiorkowski [34] were interested in investigating RT further. They conducted a study to compare the effectiveness of RT and path-coverage testing in the context of analyzing software reliability. They found a surprising result where RT could sometimes be better. Based on the study in [34], Duran and Nafos [33] conducted an empirical study of the fault-finding capabilities of RT. They showed that RT could find errors in a reasonably high proportion of time.

In 57% of the studies with MT, RT has been applied as a source test case generation technique [78]. This is a popular source test case generation technique because of its cost-effective and straightforward approach. Another reason for using RT is that MT was considered a black box testing technique. But, in this study, my target is to explore the possibility of using this testing technique as white box testing. In addition, the new approach also has to be more effective in detecting faults than RT.

2.2.2 Search Based Test Case Generation

The application of metaheuristic search techniques for the automatic generation of test cases has been of rapid interest to many researchers in recent years. In industry, test case selection is generally a manual process, and the responsibility of which usually falls on the tester. However, this manual process is too costly, complicated, and laborious. The automation process in this area has been limited. Exhaustive enumeration of a program's input is infeasible for any reasonably sized program, yet random methods are unreliable and unlikely to exercise "deeper" software features that are not exercised by mere chance.

Previous efforts have been limited by the size and complexity of the software involved and the basic fact that, in general, test data generation is an undecidable

problem. The application of metaheuristic search techniques to test data generation is a possibility that offers much promise for these problems. Metaheuristic search techniques are high-level frameworks that utilize heuristics to find solutions to combinatorial problems at a reasonable computational cost. Such a problem may have been classified as NP-complete or NP-hard, or be a problem for which a polynomial time algorithm is known to exist but is not practical. They are not standalone algorithms in it of themselves, but rather strategies ready for adaptation to specific problems. For test data generation, this involves the transformation of test criteria into objective functions. Objective functions compare and contrast solutions of the search concerning the overall search goal. Using this information, the search is directed into potentially promising areas of the search space.

Search-based software test data generation is just one example of search-based software engineering [25, 41]. To date, metaheuristic search techniques have been applied to automate test data generation in the following areas:

- the coverage of specific program structures, as part of a structural, or white-box testing strategies;
- the exercising of some specific program feature, as described by a specification;
- attempting to automatically disprove certain grey-box properties regarding the operation of a piece of software, for example trying to stimulate error conditions, or falsify assertions relating to the software's safety;
- to verify non-functional properties, for example, the worst-case execution time of a segment of code.

Three search-based test data generation techniques proposed by EvoSuite [36] have been used as a source test case generation technique for MT in our numerical

program testing. These techniques are briefly explained below:

2.2.2.1 Line Coverage In-Line coverage [74], to cover each statement of source code, each basic code block in a method must be reached (except comments). In traditional search-based testing techniques, this reachability would be expressed by an association of branch distance [54] and approach level. The branch distance measures how different a predicate (a decision-making point) is from evaluation to an expected target result. For example, given a predicate $a == 7$ and an execution with value $a = 5$, the branch distance to the predicate value true would be $|5 - 7| = 2$, whereas execution with value $a = 6$ is closer to being true with a branch distance of $|6-7| = 1$. Branch distance can be estimated by applying a set of standard rules [50, 54]. The approach-level with regards to the control dependencies measures how distant a sole execution and the target statements.

2.2.2.2 Branch Coverage Many popular tools have implemented in practice the idea of branch coverage [74], even though this practical approach may not always match the more theoretical interpretation of covering all edges of a program's control flow. Branch coverage is often measured by maximizing the number of branches of conditional statements that are executed by a test suite. Thus, to satisfy a unit test suite for each of the branch statements, there is at least one test case that satisfies the branch predicate to *false* and at least one test case satisfies the branch predicate to *true*. In branch coverage, the fitness value is measured by calculating the closeness of the test suite to cover all the branches of a PUT.

2.2.2.3 Weak Mutation Coverage To generate test cases in test generation tools, it is preferred to satisfy the constraints or conditions rather than developers' preferred boundary cases. Small code modification is applied to the PUT in WM testing.

Then, the test generation tools are forced to generate such values that can distinguish between the original and the mutant. In mutation testing, a test case is considered to be "killed" when the execution result of the mutant version is different than the original version of the PUT. The WM criteria are satisfied when at least one test case from the unit test suite reaches the infection state of the mutant. To measure the fitness value of the WM [74], it is required to calculate infection distance concerning a set of mutation operators.

2.3 Mutation Testing

Mutation testing [29] has been used to evaluate the fault detection effectiveness of the automated test case generation approaches. Mutation testing is a fault-based testing technique that measures the effectiveness of test cases of SUT. Many experiments suggest that mutants are a proxy to the real faults for comparing testing techniques [4]. Briefly, the technique performs as follows. First, mutants are created by simply seeding faults in a program. By applying syntactic changes to its source code, new faulty versions of the original programs are generated. Each syntactic change is determined by an operator called a *mutation operator*. Test cases are then executed for the faulty and original versions of the program and checked whether they produce different responses. If the response of the mutant is different from the original program, then we say the mutant has been killed, and the test case has the ability to detect faults effectively for that program. Otherwise, the mutant remains alive. When a mutant is syntactically different but semantically identical to the original then it is referred to as an *equivalent mutant*. There are four common equivalent mutant situations: the mutant cannot be triggered, the mutant is generated from dead code, the mutant only alters the internal states, and the mutant only improves speed. The percentage of killed mutants concerning the total number of non-equivalent mutants

provides an adequacy measurement of the test suite, which is called the *mutation score.*

2.3.1 Mutant Reduction Strategies

Selecting representative subsets from a given set of mutants is the principal aim of mutant reduction strategies. This technique will practically reduce the application cost of mutation testing which will lead to a reduction in the total software testing cost. Two mutation reduction techniques have been proven more effective in recent studies [64]. But to my knowledge there is no mutation reduction techniques have been applied to evaluate the fault-finding effectiveness of source test case generation techniques in MT. Our claim in this proposal is that applying mutation reduction techniques to find a better test case generation approach is cost-effective in terms of time and budget. We will conduct an empirical study to find a better mutation reduction technique for our test case generation approach.

2.3.1.1 Random Sampling Technique

A major portion of the mutation testing demands is influenced by the generation and execution of the candidate set of mutants. By considering a small sample of mutants, a significant cost reduction can be achieved. Empirical studies have shown that a selection of 10% of mutants results in a 16% loss of the fault detection ability of the produced test sets compared to full mutation testing [93]. In our proposal, we will follow the first-order mutation testing strategies [65]. In this process, we will select x% (random x%) portion of the initial mutant set, where $x = 10, 20, 30, 40, 50$, and 60. In the empirical study, we will find out which random % of selected mutants can better represent the total mutant set.

2.3.1.2 Operator Based Mutant Selection Since mutant operators generate different numbers of mutant programs, Offutt et al. proposed $N-selective\ mutation$ theory where N is # of mutant operators [61]. In their experiment, they divided the mutant operators into three general categories based on the syntactic elements that they modify. Three categories are Replacement-of-operand operators (replace each operand in a program with each other legal operand), Expression modification operators (modify expressions by replacing operators and inserting new operators), and Statement modification operators (modify entire statements). Their experiments suggest that Expression modification operators with less number of mutants than the total mutant set can be effective and the execution time is also linear.

2.4 Scientific Applications

I am focusing on improving the fault detection effectiveness of test case generation techniques for two separate scientific application domains, numerical programs and ML applications in my doctoral research proposal.

2.4.1 Numerical Applications

Numerical programs are vital to our daily lives. They have been used not only in various theoretical disciplines but also in engineering and medical practices including mission-critical and safety-critical applications. Unfortunately, despite the importance of the quality of numerical packages, we are far from doing a good job [67].

Like other testing contexts, we usually assume that we can verify the actual outputs of numerical software against some expected results. We call the mechanism of checking the correctness of the test output as a test oracle [10, 37]. Developers of numerical software generally adopt the following mechanisms as test oracles:

1. Comparing with analytical solutions, simulation results, tabulated values, or hand-calculations [37].

2. Verifying with standard mathematical libraries or reference software packages [15].

Test oracles, however, may not be available in every program. This is the so-called oracle problem. This is especially the case for numerical software. Because of truncation errors (due to truncating an infinite series into a finite series), rounding errors (due to the digital representation of floating-point numbers) and the propagation of errors in the computing process, numerical computation introduces unwarranted errors that will affect the final results. Thus, we cannot find exact solutions to numerical problems. Instead, we can aim at bounding the errors of numerical solutions so that they satisfy given precision requirements. With the efforts of mathematicians, by applying special techniques, we can assure the precision of some numerical functions, such as elementary functions [26]. It is, however, very difficult to analyze the errors in complex numerical computation [42, 72].

There are reputable mathematical libraries, such as IMSL [1] and NAG [2], which have matured through intensive testing and real-life applications. In fact, some popular numerical libraries have been refined gradually because of errors identified throughout the operational lives of the programs. We can compare some of the results of our numerical software with these libraries or similar reference software, but how do we handle other results that may involve special features not available in standard libraries? By the same argument, tabulated values and analytical solutions may not be available in every application.

[1] http://www.vni.com/ products/imsl
[2] https://www.nag.com/content/nag-library

2.4.2 Machine Learning Applications

Many fields in scientific computing applications, such as computational biology, bioinformatics, etc. - depend on supervised ML algorithms to provide important core functionality to support solutions. For instance, over fifty real-world computational applications use support vector machines for classification [1]. As these types of applications are becoming part of our daily lives, ensuring their quality becomes even more important [55]. In such applications, formal proofs of the underlying algorithm do not always guarantee that it implements that algorithm correctly. Therefore, software testing is imperative to ensure the quality of these systems.

Quality assurance of such applications presents a challenge because conventional software testing processes do not always apply: in particular, it is difficult to detect subtle errors, faults, defects, or anomalies in many applications in these domains. After all, there is no reliable "test oracle" to indicate what the correct output should be for arbitrary input. The general class of software systems with no reliable test oracle available is sometimes known as "non-testable programs" [87]. Many of these applications fall into a category of software that Weyuker describes as "Programs which were written in order to determine the answer in the first place. There would be no need to write such programs if the correct answer were known" [87].

The majority of the research effort in the domain of ML focuses on building more accurate models that can better achieve the goal of automated learning from the real world. However, to date, very little work has been done on assuring the correctness of the software applications that perform ML operations [70, 71, 85]. Formal proofs of an algorithm's optimal quality do not guarantee that an application implements or uses the algorithm correctly, and thus software testing is necessary.

CHAPTER THREE

RELATED WORK

Most contributions on MT use either randomly generated test data or existing developer test suites for the generation of source test cases. Not much research has been done on the automatic generation of source test cases for MT. A testing approach called *Automated Metamorphic Testing* has been proposed by Gotlieb and Botella [38]. They translated the code into an equivalent constraint logic program and tried to find test cases that violated the MRs. The fault detection effectiveness of RT has been compared with "special values" as source test cases for MT by Chen et al. [20]. Special values are one type of input where the output is well-known for a particular method. But randomly generated test cases are more effective than those test cases that are derived from "special values" for MT [94]. Manually generated test suites have also been compared with the RT for MT [79]. Their experimental results showed that randomly generated test suites are more effective in detecting faults than manually designed test suites. They also observed from their results that combining RT with manually written tests provides better fault detection ability than RT only.

A genetic algorithm approach has been proposed by Batra and Sengupta [9] to generate test cases maximizing the paths traversed in the PUT for MT. The same problem has been resolved by partitioning the input domain of the PUT into multiple equivalence classes for MT [16]. They applied an algorithm that would generate test cases by covering those equivalence classes. They were able to generate source and follow-up test cases that provide a high fault detection rate. Symbolic Execution was used to construct MRs and generate their corresponding source test cases by Dong and Zhang [32]. At first, the program paths were analyzed to generate symbolic

inputs, and then, these symbolic inputs were used to construct MRs. Finally, source test cases were generated by replacing the symbolic inputs with real values.

Adaptive Random Testing (ART) has been applied over the RT to find the fault detection effectiveness of source test case generation on MT [8]. Barus et al.'s empirical study showed that ART outperforms RT in enhancing the effectiveness of MT. Another automated test case generation technique called dynamic symbolic execution (DSE) has been applied to generate the source test cases for MT [2]. Alatawi et al.'s study showed that DSE improves the coverage and fault detection rate of MT compared to RT using significantly smaller test suites.

Murphy et al. generated data with repeating values, missing values, or categorical data for testing two ML ranking applications [57]. They had generated large data sets by controlling the properties and randomness of the data. Their data generation tool was proven to be simple and reliable compared to the real-world data. Breck et al. used synthetic training data that adhere to schema constraints to trigger the hidden assumptions in the code that do not agree with the constraints [12]. They identified the data bugs while deploying the system on TFX (an end-to-end machine-learning platform at Google). They also investigated the skew in training data and new data. Perturbed Model Validation (PMV) combines MR and data mutation to detect overfitting [99]. Zhang et al. used synthetic data with known distributions to test overfitting.

CHAPTER FOUR

FAULT DETECTION EFFECTIVENESS OF SOURCE TEST CASE GENERATION STRATEGIES FOR METAMORPHIC TESTING

4.1 Contribution of Authors and Co-Authors

Manuscript in Chapter 4

Author: Prashanta Saha

Contributions: Problem identification and proposing solution, running experiment, manuscript writing, creating tables and figures. Primary writer

Co-Author: Dr. Upulee Kanewala

Contributions: Contribution in manuscript editing/writing, provided feedback, guidance and advice

4.2 Manuscript Information

Prashanta Saha and Dr. Upulee Kanewala

ACM/IEEE International Conference on Software Engineering (ICSE)

Status of Manuscript:

_____ Prepared for submission to a peer-reviewed journal

_____ Officially submitted to a peer-reviewed journal

_____ Accepted by a peer-reviewed journal

__x__ Published in a peer-reviewed journal

Association for Computing machinery (ACM)

May 27, 2018

DOI:10.1145/3193977.3193982

4.3 Abstract

Metamorphic testing is a well known approach to tackle the oracle problem in software testing. This technique requires the use of source test cases that serve as seeds for the generation of follow-up test cases. Systematic design of test cases is crucial for the test quality. Thus, source test case generation strategy can make a big impact on the fault detection effectiveness of metamorphic testing. Most of the previous studies on metamorphic testing have used either random test data or existing test cases as source test cases. There has been limited research done on systematic source test case generation for metamorphic testing. This paper provides a comprehensive evaluation on the impact of source test case generation techniques on the fault finding effectiveness of metamorphic testing. We evaluated the effectiveness of line coverage, branch coverage, weak mutation and random test generation strategies for source test case generation. The experiments are conducted with 77 methods from 4 open source code repositories. Our results show that by systematically creating source test cases, we can significantly increase the fault finding effectiveness of metamorphic testing. Further, in this paper we introduce a simple metamorphic testing tool called "METtester" that we use to conduct metamorphic testing on these methods.

4.4 Introduction

A *test oracle* [86] is a mechanism to detect the correctness of the outcomes of a program. The *oracle problem* [7] can occur when there is no oracle present for the program or it is practically infeasible to develop an oracle to verify the correctness of the computed outputs. This test oracle problem is quite frequent especially with scientific software and is one of the most challenging problems in software testing. Metamorphic testing (MT) technique was proposed to alleviate this oracle problem [17]. MT uses properties from the program under test to define metamorphic relations (MRs). A MR specifies how the outputs should change according to a specific change made into the source input. Thus, from existing test cases (named as source test cases) MRs are used to generate new test cases (named as follow-up test cases). Then the set of source and follow-up test cases are executed on the program under test and the outputs are checked according to the corresponding MRs. The program under test can be considered as faulty if a MR is violated.

Effectiveness of MT in detecting faults depends on the quality of MRs. Additionally the effectiveness of MT should also rely on the source test cases. Effectiveness of metamorphic testing can be improved by systematically generating the source test cases. Such a systematic approach can reduce the size of the test suite and could be more cost effective. Most of the previous studies in MT have used randomly generated test cases as source test data for metamorphic testing. In this study we investigated the effectiveness of line, branch coverage, weak mutation, and random testing for creating source test cases for MT.

Our experimental results show that test cases satisfying weak mutation coverage provide the best fault finding effectiveness. We also have found that combining one or more systematic source test case generation technique(s) may increase the fault

detection ability of MT.

4.5 Background

MT is a property based testing approach which aims to alleviate the oracle problem. But the effectiveness of MT not only depends on the quality of MRs but also on the source test cases. In this section we briefly discussed MT and source test generation techniques, line, branch coverage and weak mutation.

4.5.1 Metamorphic Testing

Source test cases are used in MT [17] to generate follow-up test cases using a set of MRs identified for the program under test (PUT). MRs [22] are identified based on the properties of the problem domain like the attribute of the algorithm used. We can create source test cases using techniques like random testing, structural testing or search based testing. Follow-up test cases are generated by applying the input transformation specified by the MRs. After executing the source and follow-up test cases on the PUT we can check if there is a change in the output that matches the MR, if not the MR is considered as violated. Violation of MR during testing indicates fault in the PUT. Since MT checks the relationship between inputs and outputs of a test program, we can use this technique when the expected result of a test program is not known.

For example, in figure 4.1, a Java method *add_values* is used to show how source and follow-up test cases work with a PUT. The *add_values* method sum up all the array element passed as argument. Source test case, $t = \{3, 43, 1, 54\}$ is randomly generated and tested on *add_values*. The output for this test case is 101. For this program, when a constant c is added to the input, the output should increase. This will be used as a MR to conduct MT on this PUT. A constant value 2 is added

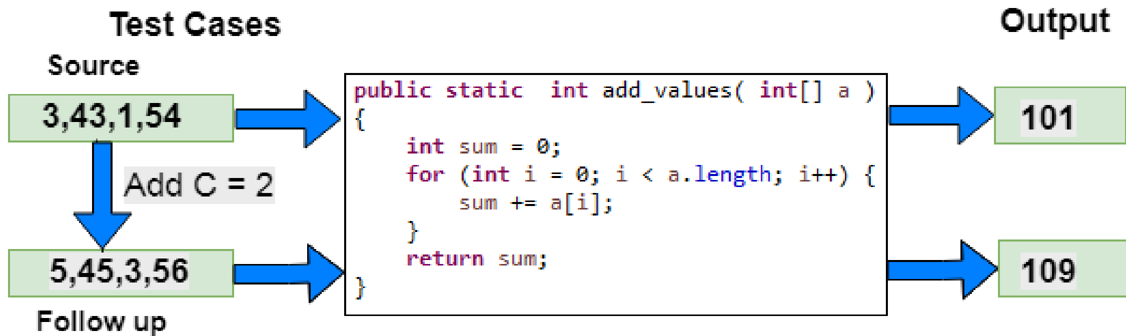


Figure 4.1: Test source and follow-up inputs on PUT.

to this array to create a follow-up test case $t' = \{5, 45, 3, 56\}$ and then run on the PUT. The output for this follow-up test case is 109. To satisfy this Addition MR the follow-up test output should be greater than the source output. In this MT example, the considered MR is satisfied for this given source and follow-up test cases.

4.5.2 Source Test Case Generation

To generate source test cases we have used the EvoSuite [36] tool. EvoSuite is a test generation tool that automatically produces test cases targeting a higher code coverage. EvoSuite uses an evolutionary search approach that evolves whole test suites with respect to an entire coverage criterion at the same time. In this paper we generated source test cases based on line, branch coverage , weak mutation and random testing. Below we briefly describe the systematic approaches used by EvoSuite to generate them.

4.5.2.1 Line Coverage In line coverage [74], to cover each line of source code, we need to make sure that each basic code block in a method is reached. In traditional search-based testing, this reachability would be expressed by a combination of branch distance [54] and approach-level. The approach-level measures how distant an individual execution and the target statement are in terms of the control

dependencies. The branch distance estimates how distant a predicate (a decision making point) is from evaluation to a desired target result. For example, given a predicate x==6 and an execution with value x = 4, the branch distance to the predicate valuing true would be $|4-6| = 2$, whereas execution with value x=5 is closer to being true with a branch distance of $|5-6| = 1$. Branch distance can be measured by applying a set of standard rules [50, 54].

In addition to test case generation, if reformation is a test suite to execute all statements then the approach level is not important, as all statements will be executed by the similar test suite. Hence, we only need to inspect the branch distances of all the branches that are related to the control dependencies of any of the statements in that class. There is a control dependency for some statements for each conditional statement in the code. It is required that the branch of the statement leading to the dependent code is executed. Hence, by executing all the tests in a test suite the line coverage fitness value can be calculated. The minimum branch distances $d_{min}(b, Suite)$ are calculated for each executed statement among all observed executions to every branch b in the collection of control dependent branches B_{CD}. Thus, the line coverage fitness function is defined as [74]:

$$f_{LC}(Suite) = v(|NCLs| - |CoveredLines|) + \sum_{b \in B_{CD}} v(d_{min}(b, Suite))$$

Where *NCLs* are the set of all statements in the class under test (CUT), *CoveredLines* are the total set of covered statements which are executed by each test case in the test suite, and *v(x)* is a normalizing function in [0,1] (e.g. $v(x) = \frac{x}{(x+1)}$) [5].

4.5.2.2 Branch Coverage The idea of covering branches is well accepted in practice and implemented in popular tools, even though the practical rationale of

branch coverage may not always match the more theoretical interpretation of covering all edges of a program's control flow. Branch coverage is often defined as maximizing the number of branches of conditional statements that are executed by a test suite. Thus, a unit test suite is considered as satisfied if and only if its at least one test case satisfies the branch predicate to *true* and at least one test case satisfies the branch predicate to *false*.

The fitness value for the branch coverage is calculated based on a criteria which is how close a test suite is to covering all branches of the CUT. The fitness value of a test suite is calculated by executing all of its test cases, keeping trail of the branch distances $d(b, Suite)$ for each of the branch in the CUT. Then [74]:

$$f_{BC}(Suite) = \sum_{b \in B} v(d(b, Suite))$$

To optimize the branch coverage the following distance is calculated, where $d_{min}(b, Suite)$ is the minimal branch distance of branch b on all executions for the test suite [74]:

$$d(b, Suite) = \begin{cases} 0 & \text{if the branch has been covered,} \\ v(d_{min}(b, Suite)) & \text{if the predicate has been executed at least twice,} \\ 1 & \text{otherwise,} \end{cases}$$

Here it is needed to cover the *true* and *false* evaluation of a predicate, so that a predicate must be executed at least twice by a test suite. If the predicate is executed only once, then in theory the searching could oscillate between *true* and *false*.

4.5.2.3 Weak Mutation Test case generation tools prefer to generate values that satisfy the constraints or conditions, rather than developers preferred values like boundary cases. In weak mutation a small code modification is applied to the CUT and then force the test generation tool to generate such values that can distinguish between the original and the mutant. If the execution of a test case on the mutant leads to a different state than the execution on the CUT than a mutant is considered to be "killed" in the weak mutation. A test suite satisfies the weak mutation criterion if and only if at least one test case kill each mutant for the CUT.

Infection distance is measured with respect to a set of mutation operator which guides to calculate the fitness value for the weak mutation criterion. Here inference of a minimal infection distance function $d_{min}(\mu, Suite)$ exists and define [74]:

$$d_w(\mu, Suite) = \begin{cases} 1 & \text{if mutant } \mu \text{ was not reached,} \\ v(d_{min}(\mu, Suite)) & \text{if mutant } \mu \text{ was reached.} \end{cases}$$

This results in the following fitness function for weak mutation [74]:

$$f_{WM}(Suite) = \sum_{\mu \in M_c} d_w(\mu, Suite)$$

Where M_c is the set of all mutants generated for the CUT.

4.6 Evaluation Method

We conducted a set of experiments to answer the following research questions:

- **RQ1: Which source test case generation technique(s) is/are most effective for MT in terms of fault detection?**

- **RQ2: Can the best performing source test case generation technique be combined to increase the fault finding effectiveness of MT?**
- **RQ3: Does the fault detection effectiveness of an individual MR change with the source test generation method?**
- **RQ4: How does the source test suite size differ for each source test generation technique?**

4.6.1 Code Corpus

We built a code corpus containing 77 functions that take numerical inputs and produce numerical outputs . We obtained these functions from the following open source projects:

- **The Colt Project[1]:** A set of open source libraries written for high-performance scientific and technical computing in Java.
- **Apache Mahout[2]:** A machine learning library written in Java.
- **Apache Commons Mathematics Library[3]:** A library of lightweight and self-contained mathematics and statistics components written in the Java.

We list these functions in Table 4.1. Functions in the code corpus perform various calculations using sets of numbers such as calculating statistics (e.g. average, standard deviation and kurtosis), calculating distances (e.g. Manhattan and Tanimoto) and searching/sorting. Lines of code of these functions varied between 4 and 52, and the number of input parameters for each function varied between 1 and 4.

[1]http://acs.lbl.gov/software/colt/
[2]https://mahout.apache.org/
[3]http://commons.apache.org/proper/commons-math/

Table 4.1: All methods with mutants kill rates and test suite size for each source test case generation technique

	Branch		weak mutation		Line		Random	
Method name	Killrate (%)	# of Test Cases	Killrate (%)	# of Test Cases	Killrate (%)	# of Test Cases	Killrate (%)	# of Test Cases
add_values	63.63	1	63.63	1	54.54	1	30	10
array_calc1	33.33	1	33.33	1	46.15	1	52.10	10
array_copy	56.00	1	64.00	1	64.00	1	0.00	10
average	38.10	1	73.80	1	42.86	1	28.20	10
bubble	51.40	1	44.95	3	36.69	1	16.90	10
cnt_zeroes	41.00	1	51.30	2	38.46	1	0.00	10
count_k	31.80	1	36.36	2	34.09	1	50.00	10
count_non_zeroes	41.00	1	48.71	2	51.28	1	22.20	10
dot_product	63.00	1	60.87	1	56.52	1	22.20	10
elementwise_max	46.30	2	68.51	3	83.33	2	0.00	10
elementwise_min	44.40	1	55.56	1	55.56	1	0.00	10
find_euc_dist	80.10	1	76.39	1	79.17	1	50	10
find_magnitude	52.10	1	75.00	1	52.10	1	8.69	10
find_max	70.80	1	50.00	1	50.00	1	70.90	10
find_max2	64.10	1	71.84	2	67.96	1	98.40	10
find_median	48.70	2	98.93	3	41.71	2	53.10	10
find_min	40.40	1	61.70	1	57.45	1	83.80	10
geometric_mean	51.20	1	53.66	1	95.12	1	65.40	10

hamming_dist	40.90	1	84.09	3	59.09	2	15.90	10
insertion_sort	43.60	1	42.55	2	37.23	1	32.65	10
manhattan_dist	53.30	1	61.36	2	53.30	1	0.00	10
mean_absolute_ error	37.50	1	41.07	2	39.29	1	0.00	10
selection_sort	41.30	1	41.30	2	39.40	1	21.60	10
sequential_search	37.20	2	25.58	3	30.23	2	37.50	10
set_min_val	51.20	2	58.14	2	30.23	1	100	10
shell_sort	43.70	1	42.51	1	43.11	1	0.00	10
variance	26.10	1	39.86	1	30.40	1	25.70	10
weighted_average	86.10	1	56.94	1	86.10	1	21.20	10
manhattan Distance	48.89	1	77.78	2	22.22	1	9.10	10
chebyshevDistance	39.08	2	43.68	5	35.63	2	2.00	10
tanimotoDistance	30.21	2	32.97	5	44.50	2	5.60	10
errorRate	61.04	3	58.44	2	58.44	2	0.00	10
sum	50.00	1	77.78	1	50.00	1	35.30	10
distance1	53.33	1	80.00	1	53.33	1	14.8	10
distanceInf	46.67	1	46.67	1	46.67	1	14.8	10
ebeadd	92.68	2	100.00	3	100.00	2	15.8	10
ebedivide	100.00	2	100.00	5	100.00	2	26.8	10
ebemultiply	100.00	2	100.00	3	92.68	2	15	10
safeNorm	14.78	1	98.63	5	97.08	4	0.8	10
scale	48.72	1	58.97	3	53.85	1	47.8	10

entropy	88.42	1	88.42	2	88.42	1	42.9	10
g	93.55	2	95.16	2	93.55	1	20.9	10
calculateAbsolute Differences	60.98	1	60.98	1	60.98	1	0	10
evaluateHoners	46.03	1	79.37	1	47.62	1	80.4	10
evaluateInternal	95.25	1	93.47	2	95.55	1	90.6	10
evaluateNewton	80.00	1	65.71	1	64.29	1	76.8	10
meanDifference	40.00	1	80.00	1	40.00	1	40	10
equals	22.50	3	27.50	4	21.25	3	100	10
chiSquare	96.41	2	96.41	2	96.41	2	65.6	10
partition	43.26	5	95.81	5	28.84	3	88.1	10
evaluateWeighted Product	30.61	2	40.82	2	42.86	2	2	10
autoCorrelation	25.20	2	93.50	2	43.09	1	79.40	10
covariance	24.84	1	23.57	1	23.57	1	86.70	10
durbinWatson	0.00	0	33.77	1	0.00	0	14.10	10
harmonicMean	74.00	1	74.00	1	76.00	1	42.50	10
kurtosis	93.84	1	93.84	1	97.16	1	34.80	10
lag1	99.55	1	32.70	1	89.55	1	33.70	10
max	51.72	1	56.90	1	51.72	1	96.60	10
meanDeviation	54.39	1	33.33	1	28.07	1	78.30	10
min	67.41	1	81.03	2	70.69	1	96.60	10
polevl	94.23	2	88.46	1	88.46	2	45.50	10
pooledMean	36.43	1	34.88	1	34.88	1	19.30	10

pooledVariance	43.08	1	47.83	1	47.83	1	31.10	10
power	53.33	1	53.33	1	53.33	1	15.80	10
product	50.00	1	50.00	1	50.00	1	94.70	10
quantile	40.13	2	40.76	2	32.48	2	40.00	10
sampleKurtosis	93.86	1	93.86	1	92.98	1	85.10	10
sampleSkew	89.47	1	89.47	1	97.37	1	89.50	10
sampleVariance	75.31	1	75.31	1	12.35	1	71.20	10
skew	93.88	1	93.88	1	93.88	1	48.80	10
square	47.37	1	47.37	1	57.89	1	5.30	10
standardize	89.26	1	89.26	1	91.95	1	77.60	10
sumOfLogarithms	75.00	1	68.75	1	68.75	1	21.90	10
sumOfPowerOf Deviations	68.75	1	52.08	1	75.00	1	64.90	10
weightedMean	77.46	1	77.46	1	77.46	1	65.00	10
weightedRMS	86.96	1	86.96	1	86.96	1	43.30	10
winsorizedMean	33.00	1	37.93	1	34.48	1	0.00	10

4.6.2 METtester

METtester [68] is a simple tool that we are developing to automate the MT process on a given Java program. This tool allows users to specify MRs and source test cases through a simple XML file. METtester transforms the source test cases according to the specified MRs and conducts MT on the given program. Figure 4.2 shows the high level architecture of the tool. Below we describe the important components of the tool:

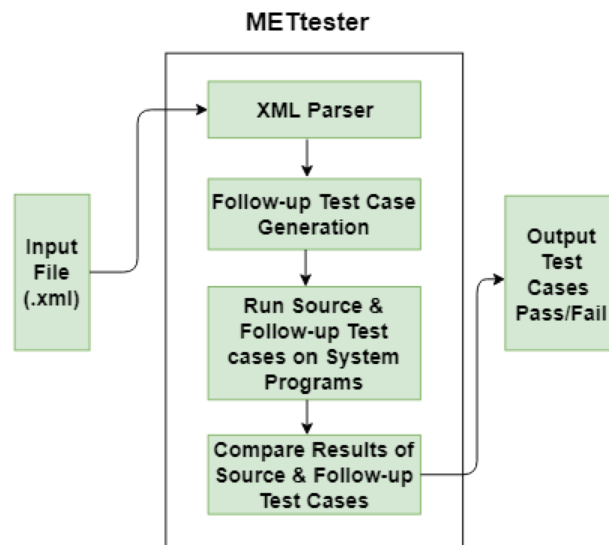


Figure 4.2: METtester architecture.

- **XML input file:** User will provide information (Figure 4.3) regarding method names to test, source test inputs, MRs, and the number of test cases to run.

- **XML file parsing:** Xmlparser class in our tool will parse information from the .xml file and process those. Then that information will be sent to the Follow-up test case generation module.

- **Follow-up test Case Generation:** In this module follow-up test cases are generated based on the provided MRs and the source test cases.

- **Execute Source & Follow-up test cases on the PUT:** After generation of the follow-up test cases METtester will run both the source and follow-up test cases individually into the system programs and return outputs from the programs.

- **Compare Source & Follow-up test results:** After getting the test results from the test program METtester will compare those results with the MR

operators mentioned in the xml file. If it satisfies the MR property then the class will flag the test case as "Pass". If it fails to satisfy the MR property class will flag it as "Fail" which means there is fault in the program.

```
<?xml version="1.0" encoding="UTF-8"?>
<TESTDESCRIPTOR>
        <METHODDESCRIPTOR>
                <METHODNAME>add_values</METHODNAME>
                <INPUTPARAM>1</INPUTPARAM>
                <INPUTDATATYPE>int,[]</INPUTDATATYPE>

                <INPUTDESCRIPTOR>
                        <INPUT>0,0,0,0,0,0,0,0,0,0,0</INPUT>
                </INPUTDESCRIPTOR>
```

Figure 4.3: An example of the XML input given to METtester.

4.6.3 Experimental Setup

For the 77 methods described in Table 4.2 we generated a total of 7446 mutated versions using the μJava mutation tool [52]. We used the following six metamorphic relations that were used in previous studies to test these functions [48]. Suppose our source test case is $X = \{x_1, x_2, x_3, ..., x_n\}$ where $x_i \geq 0, 0 \leq i \leq n$. Let source and follow-up outputs be $O(X)$ and $O(Y)$ respectively:

- **MR - Addition:** add a positive constant C to the source test case and the follow-up test case will be $Y = \{x_1 + C, x_2 + C, x_3 + C, ..., x_n + C\}$. Then $O(Y) \geq O(X)$.

- **MR - Multiplication:** multiply the source test case by a positive constant C and the follow-up test case will be $Y = \{x_1 * C, x_2 * C, x_3 * C, ..., x_n * C\}$. Then $O(Y) \geq O(X)$.

- **MR - Shuffle:** randomly permute the elements in the source test case. The follow-up test case can be $Y = \{x_3, x_1, x_n, ..., x_2\}$. Then $O(Y) = O(X)$.

- **MR - Inclusive:** include a new element $x_{n+1} \geq 0$ to the source test case and the follow-up test case will be $Y = \{x_1, x_2, x_3, ..., x_n, x_{n+1}\}$. Then $O(Y) \geq O(X)$.

- **MR - Exclusive:** exclude an existing element from the source test case and the follow-up test case will be $Y = \{x_1, x_2, x_3, ..., x_{n-1}\}$. Then $O(Y) \leq O(X)$.

- **MR - Invertive:** take the inverse of each element of source test case. Then the follow-up test case will be $Y = \{1/x_1, 1/x_2, 1/x_3, ..., 1/x_n\}$. Then $O(Y) \leq O(X)$.

For each of the methods, we used EvoSuite [36] described in section 2.2 to generate test cases targeting line, branch and weak mutation coverage. We used the generated test cases as the source test cases to conduct MT on the methods using the MRs described using METtester. Further, we randomly generated 10 test cases for each method to use as source test cases, to be used as the baseline.

Table 4.2: Total number of methods having the highest mutants kill rate for each source test generation techniques.

Total Methods	Weak mutation	Line	Branch	Random
77	41	26	29	13

4.7 Results and Discussion

4.7.1 Effectiveness of the Source Test Case Generation Techniques

Figure 4.4 shows the overall mutant killing rates for the four source test generation techniques. Among all test case generation techniques, weak mutation performed best by killing 68.7% mutants. Random tests killed 41.5% of the mutants. Table 4.2 lists the number of methods that reported the highest mutant kill rates

for each type of test generation technique. For some methods, several source test generation techniques gave the same best performance.Therefore, Figure 4.5 shows a Venn diagram of all the possible logical relations between the best performing source test generation techniques for the set of methods. Weak mutation based test generation technique reported the highest kill rate in 41 (53%) methods, whereas random testing reported the highest kill rate only in 13 (17%) methods. Therefore these results suggest that weak mutation based source test case generation is more effective in detecting faults with MT.

Figure 4.4: Total % of mutants killed by each source test suite generation technique.

4.7.2 Fault Finding Effectiveness of Combined Source Test Cases

To observe whether combining source test case generation techniques will achieve a higher fault detection rate, we combined the best performing source test generation technique, weak mutation, with the other source test generation techniques. Table 4.3 shows the total percentage of mutants killed with each combined test suite. Combination of weak mutation and random test cases has the greater percentage of mutants kill rate (74.91) than combination of line (72.87) and branch (74.6) separately

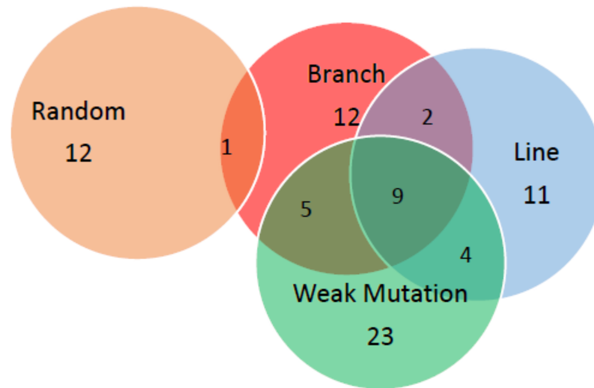


Figure 4.5: Venn Diagram for all the combinations of source test suites that performed best for each individual methods.

with weak mutation. If we combine all of the three strategies it slightly increases the total percentage of killed mutants (75.98) but there are few things to be considered, like combined test suite size.

Table 4.3: Total % of mutants killed after combining weak mutation, line, branch coverage, and random testing

Weak Mutation +Line(%)	Weak Mutation + Branch(%)	Weak Mutation + Line + Branch(%)	Weak Mutation + Random(%)
72.87	74.6	75.98	74.91

4.7.3 Fault Finding Effectiveness of Individual MRs

To see how each source test case generation technique performs with individual MRs, Figure 4.6 illustrates the percentage of mutants killed by all six MRs separately using weak mutation, line, branch coverage and random test suites. Weak mutation has the highest percentage of killed mutants in all the six MRs. Specifically with

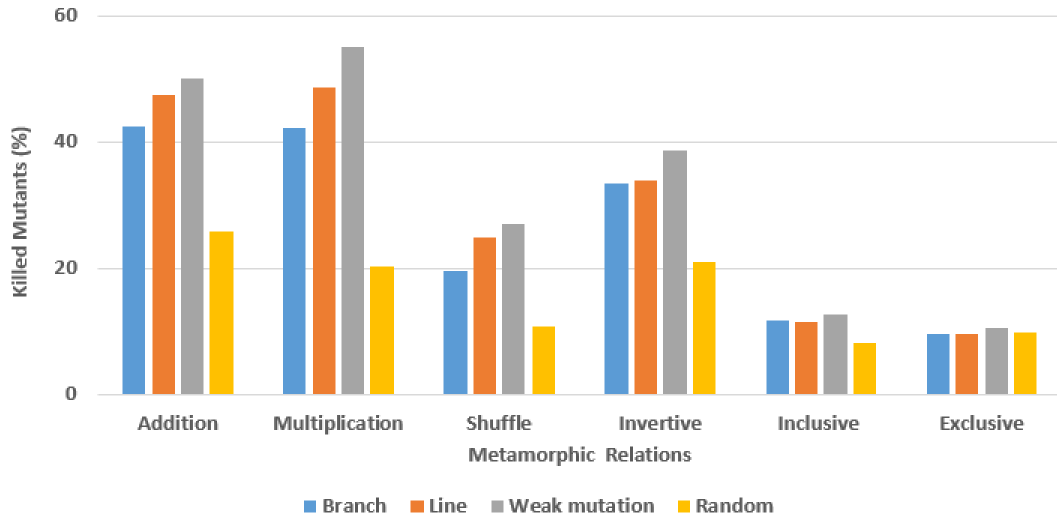


Figure 4.6: % of mutants killed by all six MRs using 4 test suite strategies (Branch, Line Coverage, Weak Mutation and Random)

multiplication and invertive MRs, the weak mutation test suite surpasses others on mutants' killing rate. But line coverage based test suites were similar to weak mutation on killing mutants with addition, shuffle, inclusive and exclusive MRs. For exclusive MR, all the test suites performed almost similarly.

4.7.4 Impact of Source Test Suite Size

Table 4.4 compares the coverage criteria in terms of the total number of tests generated, their average and median test suite size of the individual methods. In addition, in columns Smaller, Equal, and Larger we compare whether the size of the weak mutation test suites are smaller, equal or larger than those produced by other source test case generation techniques. And p-value column shows the p-value computed using the paired t-test between weak mutation - line and weak mutation -branch. We are not comparing random test suites here, because we intentionally generated 10 random test cases for each method. Weak Mutation leads to larger test suites than branch and line coverage and on average, number of test cases produced

Table 4.4: Average test suites size for weak mutation, line coverage, branch coverage and random

Test Suites	Total Number of Test Cases	Average Size	Median size	Std Dev	Smaller	Equal	Larger	p-value
Weak mutation	135	1.75	1	1.13	-	-	-	-
Line	97	1.26	1	0.67	1	45	31	3.102e-07
branch	99	1.29	1	0.59	2	49	26	1.375e-05
Random	770	10	10	0	77	0	0	-

for weak mutation are larger than those produced for branch and line coverage. The total number of test cases are also relatively larger for weak mutation compared to line and branch coverage.

4.8 Threats to Validity

Threats to *internal validity* may result from the way empirical study was carried out. EvoSuite and our experimental setup have been carefully tested, although testing can not definitely prove the absence of defects.

Threats to *construct validity* may occur because of the third party tools we have used. The EvoSuite tool has been used to generate source test cases for line, branch and weak mutation test generation techniques. Further, we used the μJava mutation

tool to create mutants for our experiment. To minimize these threats we verified that the results produced by these tools are correct by manually inspecting randomly selected outputs produced by each tool.

Threats to *external validity* were minimized by using the 77 methods was employed as case study, which is collected from 4 different open source project classes. This provides high confidence in the possibility to generalize our results to other open source software. We only used the EvoSuite tool to generate test cases for our major experiment. But we also used the JCUTE [80] tool to generate branch coverage based test suites for our initial case study and also observed similar results.

4.9 Related Work

Most contributions on MT use either random generated test data or existing test suites for the generation of source test cases. Not much research has been done on systematic generation of source test cases for MT. Gotlieb and Botella [38] presented an approach called *Automated Metamorphic Testing* where they translated the code into an equivalent constraint logic program and tried to find test cases that violates the MRs. Chen et al. [20] compared the effectiveness of random testing and "special values" as source test cases for MT. Special values are inputs where the output is well known for a particular method. Wu et al. [94] proved that random test cases are more effective than those test cases that are derived from "special values". Segura et al. [79] also compared the effectiveness of random testing with manually generated test suites for MT. Their results showed that randomly generated test suites are more effective in detecting faults than manually designed test suites. They also observed that combining random testing with manual tests provides better fault detection ability than random testing only.

Batra and Sengupta [9] proposed genetic algorithm to generate test cases

maximizing the paths traversed in the program under test for MT. Chen et al. [16] also addressed the same problem from a different prospective. They proposed partitioning the input domain of the PUT into multiple equivalence classes for MT. They proposed an algorithm which will generate test cases which will cover those equivalence classes. They were able to generate test cases that provide high fault detection rate. Symbolic Execution was used to construct MRs and their corresponding source test cases by Dong and Zhang [32]. Program paths were first analyzed to generate symbolic inputs and then, these symbolic inputs were used to construct MRs. In the final step, source test cases were generated by replacing the symbolic inputs with real values.

Barus et al. [8] applied the Adaptive Random Testing (ART) over the random testing (RT) to find the effectiveness of source test case generation on MT. Their results showed that ART outperforms RT on enhancing the effectiveness of MT. Alatawi et al. [2] used the automated test input generation technique called dynamic symbolic execution (DSE) to generate the source test inputs for metamorphic testing. Their results showed that DSE improves the coverage and fault detection rate of metamorphic testing compared to random testing using significantly smaller test suites. Compared to them, in this work, we evaluate the effectiveness of four commonly used coverage criteria for automated source test case generation.

4.10 Conclusions & Future Work

In this study we empirically evaluated the fault finding effectiveness of four different source test case generation strategies for MT: line, branch, weak mutation and random.

Our results show that weak mutation coverage based test generation can be an effective source test case generation technique for MT than the other techniques. Our results also show that the fault finding effectiveness of MT can be improved

by combining source tests generated for weak mutation coverage with randomly generated source test cases.

Further, in this paper we introduce a MT tool called "METtester." We plan to incorporate the investigated automated source test generation techniques into this tool. We also plan to extend the current case study to larger code bases and experiment with more source test generation techniques such as adaptive random test generation and data flow based test generation. Further, we plan to analyze the impact of the coverage of follow up test cases in our future research.

CHAPTER FIVE

USING METAMORPHIC RELATIONS TO IMPROVE THE EFFECTIVENESS OF AUTOMATICALLY GENERATED TEST CASES

5.1 Contribution of Authors and Co-Authors

Manuscript in Chapter 5

Author: Prashanta Saha

Contributions: Problem identification and proposing solution, running experiment, manuscript writing, creating tables and figures. Primary writer

Co-Author: Dr. Upulee Kanewala

Contributions: Contribution in manuscript editing/writing, provided feedback, guidance and advice

5.2 Manuscript Information

Prashanta Saha and Dr. Upulee Kanewala

IEEE/ACIS International Conference on Software Engineering Research, Management and Applications (SERA)

Status of Manuscript:

_____ Prepared for submission to a peer-reviewed journal

_____ Officially submitted to a peer-reviewed journal

_____ Accepted by a peer-reviewed journal

__x__ Published in a peer-reviewed journal

IEEE

June 30, 2022

DOI:10.1109/SERA54885.2022.9806786

5.3 Abstract

Automated test case generation has helped to reduce the cost of testing. However, developing effective test oracles for these automatically generated test cases still remains a challenge. Metamorphic testing (MT) has become a well-known software testing approach over the years. This testing technique can effectively alleviate the oracle problem faced when testing using metamorphic relations (MRs) to determine whether a test case is passed or failed. In this work, we conduct an empirical study on an open source linear algebra library to evaluate whether MRs can be utilized to improve the fault detection effectiveness of automatically generated test cases. Our experiment suggests that MRs can help to improve the fault detection effectiveness of automatically generated test cases.

5.4 Introduction

Software testing is an integral part of software development life cycle. Typically, testing is a costly activity yet it is essential to detect faults. As a means of reducing this cost, there has been lot of work done on automated test case generation, including the development of publicly available tools [36]. Automatically generated test suites have certain advantages over manually written test cases, in particular, saving human labor and time. Some work has shown that it is more effective to use test cases that are generated based on some coverage criteria rather than randomly generated test cases [62]. The main focus of automated test generation work thus far has been to develop efficient methods to generate test inputs to achieve a certain target such as coverage. However, there has been relatively less attention paid on utilizing effective test oracles that can improve the fault detection effectiveness of these automatically generated test cases. A *test oracle* is used to check whether the output produced for a given test case is correct or not [88]. In fact, due to the automated nature of generating test inputs, defining the oracles for these test inputs is a hard problem. Thus, many of the automatically generated test cases would contain trivial oracles, such as the assert statements that we will discuss later. This reduces the fault detection effectiveness of these test cases.

For example, consider the matrix *Power* function shown in Listing 5.4 that returns a new matrix which is the nth power of the current matrix. Figure 5.1(Left) shows a test case generated by Evosuite [36](a test case generation tool) for this function. This test case can cover 90% statements of the *Power* function from the *Matrix* class. Lines 13 to 21 are the assertions generated by Evosuite that serves as the oracles for this test case. It is easy to note that these assert statements only check for trivial properties of the output such as the the number of rows and columns of

the output matrix is not zero. Thus, these assertions do not check the accuracy of the underlying calculation, which is computing the nth power of a matrix.

```
1 public Matrix power(int n) {
2         if (n < 0) {
3             fail("The exponent should be positive: "
4             + n + ".");
5         }
6 
7         Matrix result = blankOfShape(rows, rows);
8         Matrix that = this;
9 
10        for (int i = 0; i < rows; i++) {
11            result.set(i, i, 1.0);
12        }
13
14        while (n > 0) {
15            if (n % 2 == 1) {
16                result = result.multiply(that);
17            }
18
19            n /= 2;
20            that = that.multiply(that);
21        }
22
23        return result;
24    }
```

Listing 5.1: Power function from La4j Matrix class

Metamorphic Testing (MT) is a technique proposed to alleviate the oracle problem of software under test (SUT) [19]. This is based on the idea that most of the time it is easier to develop relations between multiple inputs and outputs of a program than specifying the values of individual outputs. For example, consider a program that computes the average of a list of real numbers. It is hard to correctly predict the observed output when the input list has millions of real numbers. However, we can permute the list of real numbers and check if the returned output matches the

```java
@Test(timeout = 4000)
public void test042() throws Throwable {
    MockRandom mockRandom0 = new MockRandom();
    assertNotNull(mockRandom0);

    DenseMatrix denseMatrix0 =
    DenseMatrix.randomSymmetric(0, mockRandom0);
    assertEquals(0, denseMatrix0.columns());
    assertEquals(0, denseMatrix0.rows());
    assertNotNull(denseMatrix0);

    Matrix matrix0 = denseMatrix0.power(1293);
    assertNotSame(denseMatrix0, matrix0);
    assertNotSame(matrix0, denseMatrix0);
    assertEquals(0, denseMatrix0.columns());
    assertEquals(0, denseMatrix0.rows());
    assertEquals(0, matrix0.rows());
    assertEquals(0, matrix0.columns());
    assertTrue(matrix0.equals((Object)
                    denseMatrix0));
    assertNotNull(matrix0);
}
```

```java
@Test(timeout = 4000)
public void test042() throws Throwable {
    MockRandom mockRandom0 = new MockRandom();
    assertNotNull(mockRandom0);

    DenseMatrix denseMatrix0 =
    DenseMatrix.randomSymmetric(0, mockRandom0);
    assertEquals(0, denseMatrix0.columns());
    assertEquals(0, denseMatrix0.rows());
    assertNotNull(denseMatrix0);

    Matrix matrix0 = denseMatrix0.power(1293);

    //Matrix Multiplication - MR
    Matrix matrix1 =
            denseMatrix0.multiply(denseMatrix0);
    matrix1 = matrix1 .power(1293);
    assertTrue(matrix0.equals((Object)matrix1 ));
}
```

Figure 5.1: (Left) EvoSuite generated test case , (Right) Modified test case with MR in MT

previous output. If the outputs do not match, then there is a fault in the program. The property that specifies *when the elements of the inputs are randomly permuted the output should remain the same* is called a MR, which is a necessary property of the SUT and specifies a relationship between multiple inputs and their outputs [21].

In this work, we investigate whether we can utilize MRs to improve the fault detection effectiveness of automatically generated test cases. MRs provide an effective method to overcome the oracle problem in automatically generated test cases and verify the underlying calculations. For example, following is an Matrix Multiplication MR that should be satisfied by the program in Listing 5.4. The expected property of this MR is multiplying the input matrix with another matrix having same size and the expected return will be equal to the return of the input matrix. To incorporate the checking of this MR we modified the test case in Figure 5.1 (Right) as follows:

1. We multiplied the source test case matrix (denseMatrix0) with the same matrix (denseMatrix0) to generate the follow-up test case matrix (matrix1). (In line 15-16)

2. Next, we executed the follow-up test case matrix with the subject program (*Power* function). (In line 17)

3. Finally using the *assertTrue* JUnit assertion function we were comparing the output of source test case (matrix0) with the output of the follow-up test case (matrix1). Then we expected the resultant matrix from these two test cases are equal. (In line 18)

In this paper, we present the results of an empirical study conducted to evaluate the effectiveness of utilizing MRs with automatically generated test inputs. To this end, we generated coverage based test suites (line, branch, weak mutation coverage) using EvoSuite for several open-source software systems that implement matrix calculations

and utilized MRs to augment these automatically generated test cases. Our results show that MRs can help to increase the effectiveness of automatically generated test suites and rare cases would have the similar fault detection effectiveness as the developer written test suites.

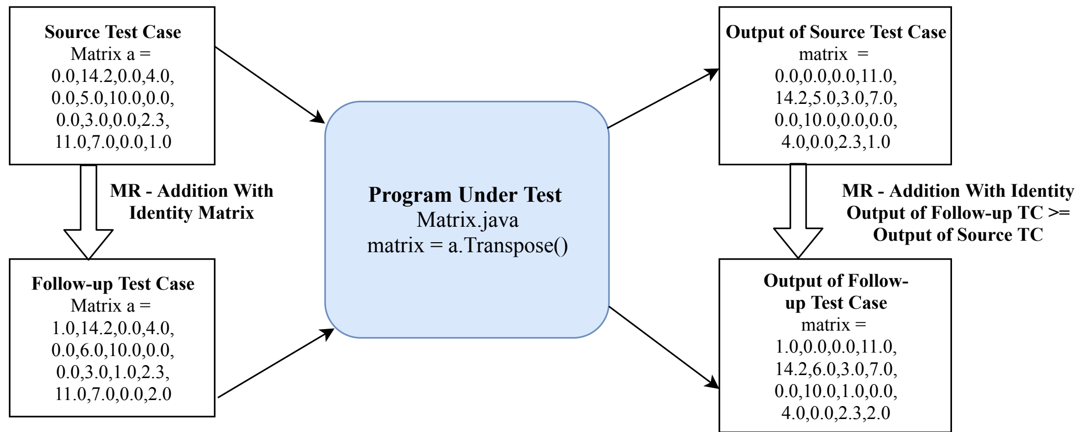


Figure 5.2: Illustration of Metamorphic Testing

5.5 Background

5.5.1 Metamorphic Testing

Following is the typical process used for applying MT:

1. Identify MRs from the specification of the SUT. An MR $R(x_1, x_2, ..., x_n, f(x_1), f(x_2), ..., f(x_n))$ is a necessary property of the SUT and is specified over the inputs $x_1, x_2, ..., x_n$ and their corresponding outputs $f(x_1), f(x_2), ..., f(x_n)$.

2. Generate the source test inputs $x_1, x_2, ..., x_k$ and execute them on the SUT.

3. Construct the follow-up test inputs $x_{k+1}, x_{k+2}, ..., x_n$ by applying transformation specified by R to $x_1, x_2, ..., x_k, f(x_1), f(x_2), ..., f(x_k)$ and execute them.

4. Verify whether R is satisfied with the obtained $x_1, x_2, ..., x_n, f(x_1), f(x_2), ..., f(x_n)$ by executing the SUT. If R is not satisfied then MR has revealed a fault in the SUT.

Step 1 (Identification of MRs), is typically done based on the knowledge of the program. Recently there have been several work done towards automating the MR identification [23, 47, 83]. In step 2 (Generation of source test cases), any test case generation technique can be applied. Previous studies have used special case [94] and random testing [51] techniques to generate source test cases. Further, previous studies have shown that using coverage-based test inputs as source inputs would improve the fault detection effectiveness of MT compared to random test inputs [76]. As shown in the above process, since MT checks the relationship between inputs and outputs of a test program, we can use this technique when the expected results of individual test inputs are unknown.

The following example is a sample of MT process. In Figure 5.2, a Java method *Transpose* from *Matrix.java* class is used to show how source and follow-up test cases perform with a PUT. The *Transpose* method transposes a matrix and returns the transposed matrix. Source test case, $a = \{(0.0, 14.2, 0.0, 4.0), (0.0, 5.0, 10.0, 0.0), (0.0, 3.0, 0.0, 2.0), (11.0, 7.0, 0.0, 1.0)\}$ is developer generated and tested on *Transpose* method. The output for this source test case is $matrix = \{(0.0, 0.0, 0.0, 11.0), (14.2, 5.0, 3.0, 7.0), (0.0, 10.0, 0.0, 0.0), (4.0, 0.0, 2.3, 1.0)\}$. For this program, when an identity matrix is added to the input, the output should increase. This will be used as a MR to conduct MT on this PUT. An identity matrix of size 4 is added to this matrix to create a follow-up test case $a^{'} = \{(1.0, 14.2, 0.0, 4.0), (0.0, 6.0, 10.0, 0.0), (0.0, 3.0, 1.0, 2.3), (11.0, 7.0, 0.0, 2.0)\}$ and then execute on the PUT. The output for this follow-up test case is $matrix = \{(1.0, 0.0, 0.0, 11.0), (14.2, 6.0, 3.0, 7.0), (0.0, 10.0, 1.0, 0.0), (4.0, 0.0, 2.3, 2.0)\}$. To satisfy this MR the follow-up test output should be greater than

the source output. We calculate the sum of elements of the matrices from source and follow-up outputs which is 57.5 and 61.5 respectively. In this MT example, 61.5 is greater than 57.5. Hence, the considered MR is satisfied for this given source and follow-up test cases.

Previous studies [53,66,82,97] have consistently shown that metamorphic testing has many advantages. First and foremost, MT provides a test result verification mechanism in the absence of an oracle. The test results are verified against a set of MRs instead of an oracle. Besides, most MRs are simple in concepts, so it is convenient to verify test results by using some simple scripts automatically.

5.5.2 Automated Test Case Generation

In this work, we used EvoSuite [36] as the automated test case generation tool. It automatically produces test cases targeting a high coverage such as line, branch, and weak Mutation coverage. EvoSuite uses an evolutionary search approach that evolves whole test suites with respect to an entire coverage criterion at the same time.

5.6 Empirical Evaluation

5.6.1 Research Questions

We conducted a set of experiments to answer the following research questions:

1. **RQ1: Can MRs be utilized to improve the fault detection effectiveness of automatically generated test cases?**

2. **RQ2: How does the improved automatically generated test cases compare with test suites created by developers in terms of fault detection effectiveness?**

3. **RQ3: How does the effectiveness of the MRs vary compared to automatically generated test cases?**

5.6.2 Subject Programs

In this experiment we used four classes from *la4j*[1] (version 0.6.0) open-source Java library. la4j is a linear algebra library that provides matrix and vector implementations and algorithms, was one of the software packages used for evaluating the performance of automated testing tools [36]. We used the following Java classes from la4j in this study:

- **Matrix.java:** This class has methods to perform matrix operations. We picked 20 methods to conduct our experiment on them. The description of these 20 methods is available in this GitHub repository[2].

- **LeastSquaresSolver.java:** This solver method is the least squares approximation of linear functions to data. In this algorithm for approximation of $\mathbf{Ax} = \mathbf{b}$ equation QR decomposition have been applied.

- **ForwardBackSubstitutionSolver.java:** This class represents the process of solving a system of linear algebraic equations using Forward Back Substitution method. This algorithm used to solve $\mathbf{LUx} = \mathbf{b}$, where $\mathbf{L}$ is lower triangular with units on the diagonal and $\mathbf{U}\ (= \mathbf{DV})$ is upper triangular. And $\mathbf{b}$ a given vector [3].

- **SquareRootSolver.java:**This class represents Square Root method for solving linear systems [4]. This algorithm solves the matrix equation $\mathbf{Au} = \mathbf{g}$. for $\mathbf{u}$,

[1] http://la4j.org/
[2] https://github.com/ps073006/ConfRepo
[3] https://algowiki-project.org/en/Forward_substitution
[4] http://mathworld.wolfram.com/SquareRootMethod.html

with $\mathbf{A}$ a $\mathbf{p}\times\mathbf{p}$ symmetric matrix and $\mathbf{g}$ a given vector.

5.6.3 MR Identification

We developed the following 10 MRs for testing the functions in the Matrix.java class. Not all these MRs are satisfied by each of these functions. Total list of these functions and the specific MRs satisfied by them can be found in this GitHub repository[4]. (In all cases we assume that Matrix A comprises only non-negative numbers).

- **MR1 - Scalar Addition:** Let A be the initial input matrix to a program P, and b be a positive scalar. Let A' be the follow-up input matrix where $A' = \forall i, j \in b + A_{i,j}$. Let the output of P for A be O (i.e. $P(A) = O$) and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' \geq \sum_{i,j} O$.

- **MR2 - Addition With Identity Matrix:** Let A be the initial input matrix to a program P, and I be an identity matrix. Let A' be the follow-up input matrix where $A' = \forall i, j \in I_{i,j} + A_{i,j}$. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' \geq \sum_{i,j} O$.

- **MR3 - Scalar Multiplication:** Let A be the initial input matrix to a program P, and b be a positive scalar. Let A' be the follow-up input matrix where $A' = \forall i, j \in b.A_{i,j}$. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' \geq \sum_{i,j} O$.

- **MR4 - Multiplication With Identity Matrix:** Let A be the initial input matrix to a program P, and I be an identity matrix. Let A' be the follow-up input matrix where $A' = \forall i, j \in I_{i,j}.A_{i,j}$. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' = \sum_{i,j} O$.

- **MR5 - Transpose:** Let A be the initial input matrix to a program P. Let A' be the follow-up input matrix where $A' = \forall i, j \in A^T_{i,j} = A_{j,i}$. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' = \sum_{i,j} O$.

- **MR6 - Matrix Addition:** Let A be the initial input matrix to a program P. Let A' be the follow-up input matrix where $A' = \forall i, j \in A_{i,j} + A_{i,j}$. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' \geq \sum_{i,j} O$.

- **MR7 - Matrix Multiplication:** Let A be the initial input matrix to a program P. Let A' be the follow-up input matrix where $A' = \forall i, j \in A_{i,j}.A_{i,j}$. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' \geq \sum_{i,j} O$.

- **MR8 - Permute Column:** Let A be the initial input matrix to a program P with $j = 1, 2, 3, .., n$ columns. Let A' be the follow-up input matrix after permuting the column positions of A. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' = \sum_{i,j} O$.

- **MR9 - Permute Row:** Let A be the initial input matrix to a program P with $i = 1, 2, 3, .., n$ rows. Let A' be the follow-up input matrix after permuting the row positions of A. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' = \sum_{i,j} O$.

- **MR10 - Permute Element:** Let A be the initial input matrix to a program P with $j = 1, 2, 3, .., n$ columns and $i = 1, 2, 3, .., n$ rows. Rows and columns have to be same size. Let A' be the follow-up input matrix after permuting $A_{i,n}$ element with $A_{n,j}$ element. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' = \sum_{i,j} O$.

We also identified 6 MRs for 3 solver classes. In these 3 classes one matrix and one vector value acted as parameters for the source test case. The particular method we were testing in these 3 classes is *Solve* method. These 6 MRs are explained below:

- **MR11 - Multiplication:** Let A be the input matrix and v be a input vector to a program P. Their source test execution output is vector $P(A, v) = O$. After multiplying a positive scalar constant b with both, the follow-up matrix will be $A' = \forall i, j \in b.A_{i,j}$ and vector $v' = \forall i \in b.v_i$. And their follow-up output is vector $P(A', v') = O'$. Then, expected output relation is $\sum_{i,j} O' = \sum_{i,j} O$.

- **MR12 - Permute Row Element:** Let A be the input matrix with $i = 1, 2, 3, .., n$ rows and v be a input vector with $i = 1, 2, 3, .., n$ elements to a program P. Their test execution output is vector $P(A, v) = O$. The follow-up matrix will be A' after permuting the row positions and vector v' after permuting elements. Then, their executed test output is vector $P(A', v') = O'$. And expected output relation is $\sum_{i,j} O' = \sum_{i,j} O$.

- **MR13 - Matrix Vector Addition:** Let A be a input matrix with $i = 1, 2, 3, .., n$ rows and v be a input vector with $i = 1, 2, 3, .., n$ elements to a program P. Their test execution output is vector $P(A, v) = O$. The follow-up matrix will be $A' = \forall i, j \in A_{i,j} + A_{i,j}$ and vector $v' = \forall i \in v_i + v_i$. Their executed test output is vector $P(A', v') = O'$. Then, expected output relation is $\sum_{i,j} O' = \sum_{i,j} O$.

- **MR14 - Multiplication With Transpose Matrix:** Let A be a input matrix with $i = 1, 2, 3, .., n$ rows and v be a input vector with $i = 1, 2, 3, .., n$ elements to a program P. Their test execution output is vector $P(A, v) = O$. The follow-up matrix will be $A' = \forall i, j \in A^T_{i,j}.A_{i,j}$ and vector $v' = \forall i \in A^T_{i,j}.v_i$. Their

executed test output is vector $P(A', v') = O'$. Then, expected output relation is $\sum_{i,j} O' = \sum_{i,j} O$.

- **MR15 - Multiplication With Identity Matrix:** Let A be a input matrix with $i = 1, 2, 3, .., n$ rows and v be a input vector with $i = 1, 2, 3, .., n$ elements to a program P. Their test execution output is vector $P(A, v) = O$. The follow-up matrix will be $A' = \forall i, j \in I_{i,j}.A_{i,j}$ with an identity matrix I. and vector $v' = v$. And their executed test output is vector $P(A', v') = O'$. Then, expected output relation is $\sum_{i,j} O' = \sum_{i,j} O$.

- **MR16 - Multiplication With Negative:** Let A be a input matrix and v be a input vector to a program P. Their test execution output is vector $P(A, v) = O$. For follow-up test input, we multiply a negative constant b to both and the follow-up matrix will be $A' = \forall i, j \in b.A_{i,j}$ and $v' = \forall i \in b.v_i$. And their executed test output is vector $P(A', v') = O'$. Then, expected output relation is $\sum_{i,j} O' = \sum_{i,j} O$.

5.6.4 Automated Test Case Generation

For each of the classes as mentioned above, we used EvoSuite [36] commandline tool to generate test cases targeting line, branch, and weak mutation coverage. In this experiment, we used EvoSuite *1.0.6v*. In commandline, for *assertion_strategy* parameter we have used all available strategies (e.g. Mutation, Unit). A test *assertion* is a predicate which compares some aspects of the observed behavior of a function against the expected behavior. Some types of test *assertions* are related to arrays and standard container classes. We ran each coverage criterion (e.g., line, branch, and weak mutation) separately using the *criterion* parameter. Based on the coverage criterion EvoSuite generated separate *.java* files with all the *JUnit* test cases. We ran the generated *JUnit* test cases with the original programs, which generate a pass/fail

report of test cases. From that report, we removed test cases, which were checking *Undeclared exceptions* e.g. *NullPointerException, IllegalArgumentException.* It is not feasible to apply MT for those test cases since they were throwing exceptions. In Table 5.1 we have listed our test suite sizes separately for all classes. EvoSuite column has the total test suite generated by line, branch, and weak mutation coverage.

Table 5.1: Classes with Evosuite test suite & developer test suite

Class name	Evosuite	Developer
Matrix.java	37	40
LeastSquaresSolver.java	4	11
ForwardBackSubstitutionSolver.java	8	7
SquareRootSolver.java	5	6

5.6.5 Utilizing MRs to Modify Automatically Generated Test Cases

1. To generate the follow-up test cases we modified the automatically generated test case based on each MR. We modified the Evosuite generated *.java* files with follow-up test cases (Fig 5.1(Right)). This follow-up test case insertion process was manually completed. (line 15-17)

2. We also added new assert statement to compare the source and follow-up test outputs.(line 18)

3. We executed those source and follow-up test cases on the original programs and verified the MR properties. If any MR property did not hold for any test input, we excluded that MR for that particular input.

5.6.6 Evaluation Approach

We used mutation testing to measure the fault detection effectiveness of the automatically generated test cases and the test cases enhanced with MRs. Mutation Testing [29] is a fault-based testing technique that measures the effectiveness of test cases and many experiments suggest that mutants act as a proxy to real faults for comparing testing techniques [4]. Briefly, the technique performs as follows. First, mutants are created by simply seeding faults in a program. By applying syntactic changes to its source code, new faulty versions of the original programs are generated. Each syntactic change is determined by a operator called a *mutation operator*. Test cases are then executed for the faulty and original versions of the program and checked whether they produce different responses. If the test output of the mutant is different from the original program, then we say the mutant is *killed*. Otherwise, the mutant remains *alive*. When a mutant is syntactically different but semantically identical to the original program then it is referred to as an *equivalent mutant*. There are 4 common equivalent mutant situations: the mutant cannot be triggered, the mutant is generated from dead code, the mutant only alters the internal states, and the mutant only improves speed. To detect the original program and the mutated programs are equivalent is undecidable [14]. The percentage of killed mutants with respect to the total number of non-equivalent mutants provides an adequacy measurement of the test suite, which is called the *mutation score*. There is another adequacy measurement metric used to measure the efficiency of MRs, called *fault detection ratio* [94]. In MT, fault detection ratio is calculated by the ratio of the test case that detects a fault by an MR and the total number of Metamorphic tests generated from source test cases.

Table 5.2: Classes with MRs and mutants generation results

Class name	Lines of Code	#Mutants	#of remaining Mutants	# of MRs
Matrix.java	2210	884	363	11
LeastSquaresSolver.java	95	89	39	6
ForwardBackSubstitutionSolver.java	95	92	39	6
SquareRootSolver.java	106	55	32	6

In our evaluation, we used PIT[5] tool to systematically generate 1120 mutants for the programs described in Section 5.6.2. In Table 5.2, we list the total number of mutants generated by the PIT tool for each class. Due to the high number of mutants, identifying equivalent mutant manually was not practical. Thus, we used the following filtering approach to identify mutants to be used in the experiment: we executed the EvoSuite generated test suites and developer test suites on the generated mutants and filtered out the mutants that caused compilation errors, run-time exceptions. We also filtered out any mutants that were passing both coverage based and developer test suites. Column two in Table 5.2, lists the number of remaining mutants used in the experiment after the filtering process. Thus the *mutation score* is calculated by the ratio of the killed mutants to the remaining total mutants after filtering. Our full evaluation approach with source code repository is available in *GitHub*[6].

[5]https://pitest.org/
[6]https://github.com/ps073006/matrixmt

Below we discuss the results of our experiments and provide answers to our research questions:

1. **Effectiveness of MR over the automatically generated test suites:**

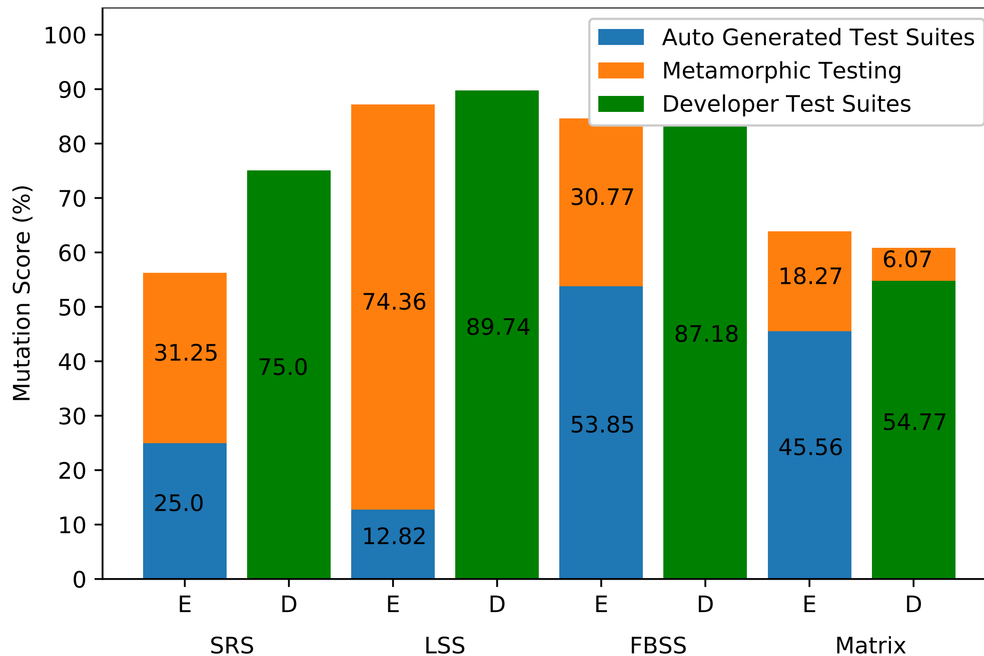


Figure 5.3: Mutation score of 4 classes for auto generated test suites (Evosuite(E)) and Developer test suites (D) and Metamorphic Testing, SRS = SquareRootSolver, LSS = LeastSquaresSolver, FBSS = ForwardBackSubstitutionSolver

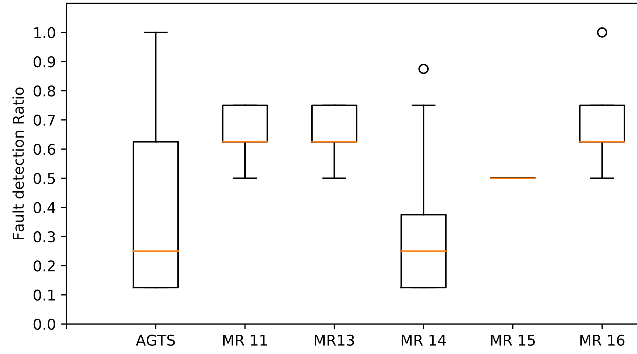
Figure 5.3 shows the mutation score of automatically generated test suites by EvoSuite (columns denoted by E) and Developer generated test suites [7] (columns denoted by D). We also show the increase of mutation score achieved by these test suites after augmenting the corresponding test cases with applicable MRs as described in Section 5.6.5. EvoSuite columns show the combined mutation scores of the test suites generated by all the three strategies (line, branch, and weak mutation), and for MT

[7] https://github.com/vkostyukov/la4j

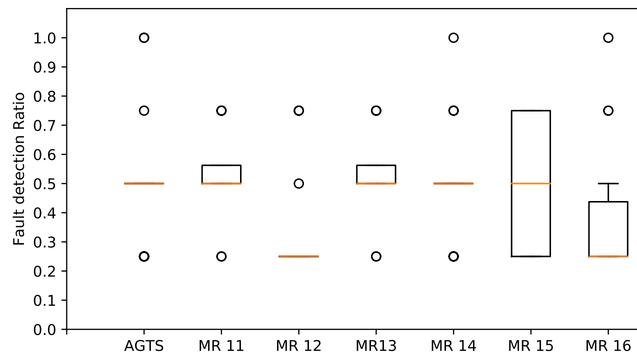
the combined mutation scores of all the MRs are used. From Evosuite columns, we see a significant increases in mutation score when augmented with MRs, but for *Matrix* class, the increase of mutation score is relatively low compared to the other three classes. Note that the MR lists are different for *Matrix* class than the other 3 classes. The fault detection effectiveness of MT is determined by the MRs used for testing as well as the source test cases used to execute those MRs. Thus, further investigation is required to determine the exact reason for the reduction of fault detection in the Matrix class.

2. Effectiveness of improved automatically generated test suites compared to the developer written test suites: From Figure 5.3, there was no additional mutant killed by the test cases augmented by MRs with the developer test suites, except for the matrix class. Further, MR augmented test cases from matrix class only killed 6.07% additional mutants. Thus, MRs did not help to improve the fault detection effectiveness of developer test suites as they did with automatically generated test suites.

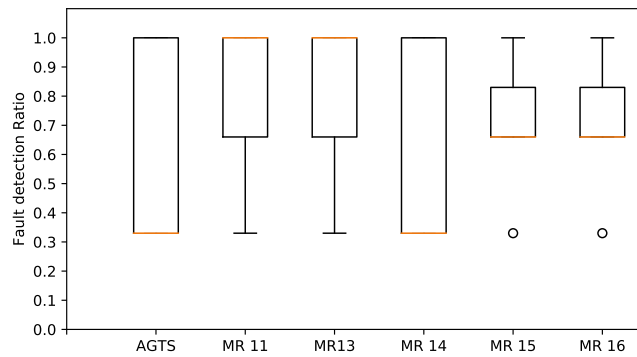
Developer test suites (Table 5.1) are generated based on the knowledge of the specification and often developers try to cover majority of the branches and utilize boundary cases to generate test cases. Therefore, it is not surprising that fault detection effectiveness did not improve with the MR augmentation. However, since our goal is to improve the effectiveness of automatically generated test suites using MT, we use this developer test suite as a benchmark of our experiments. From Figure 5.3, except for *SquareRootSolver* class, the mutation score of the MR augmented automatically generated test cases are significantly close to the mutation score of the developer test suite. This evidence suggests that MRs can be utilized to improve the fault detection effectiveness of automatically generated test suites up to a level that closely matches the fault detection effectiveness of developer test suites.



(a) ForwardBackSubstitutionSolver.



(b) LeastSquaresSolver.



(c) SquareRootSolver.

Figure 5.4: Fault Detection Ratio of AGTS (Automatically Generated Test Suites) and MR augmented test suites for 3 classes.

3. Fault detection effectiveness of the MRs compared to automatically generated test suites: Figure 5.4 shows the box plots of fault detection ratio of the automatically generated test suites and MR augmented test suites (the specific MR used for augmentation is listed as the label header) for the 3 solver classes. The box plots show that majority of the MR augmented test suites either outperform or perform equally compared to the automatically generated test suites. For *SquareRootSolver* and *ForwardBackSubstitutionSolver* class, MR11, MR13, MR15, MR16 test suites have median fault detection ratio over 0.6 which is twice as automatically generated test suites (~ 0.3). The box plots for the *LeastSquaresSolver* class show that there are lots of outliers for both the automatically generated test suites and MR augmented test suites. For this class, MR11, MR13, MR14, MR15 have similar median fault detection ratio (0.5) as the automatically generated test suites. Size of the outliers suggest that the performance of MR augmented test suites or automatically generated test cases are not consistent in the *LeastSquaresSolver* class.

For *SquareRootSolver* and *ForwardBackSubstitutionSolver* class, we can see that MR12 is not supported for both classes. From the above experimental outcome, we can deduce that majority of the MR augmented test suites have the ability to kill more mutants compared to automatically generated test cases.

5.8 Threats to Validity

Threats to *internal validity* may result from the way the empirical study was carried out. EvoSuite and our experimental setup have been carefully tested, although testing can not prove the absence of defects. Construction of MRs can still be error prone since we have manually identified and verified the MRs against the programs.

Threats to *construct validity* may occur because of the third-party tools we have

used. The EvoSuite tool has been used to generate source test cases for line, branch, and weak mutation test generation techniques. Further, we used the PIT mutation tool to create mutants for our experiment. To minimize these threats, we verified that the results produced by these tools are correct by manually inspecting randomly selected outputs produced by each tool.

Threats to *external validity* were minimized by using the 4 classes as case studies, which was performing different matrix operations. This provides high confidence in the possibility of generalizing our results to other open-source softwares. We only used the EvoSuite tool to generate test cases for our major experiment.

5.9 Related Work

Most contributions on MT use either randomly generated test data or existing developer test suites for the generation of source test cases. Not much research has been done on automatically generation of source test cases for MT. Gotlieb and Botella [38] presented an approach called *Automated Metamorphic Testing*. Using this technique, they translated the code into an equivalent constraint logic program and tried to find test cases that violate the MRs. Chen et al. [20] compared the fault detection effectiveness of random testing and "special values" as source test cases for MT. Special values are one type of inputs where the output is well known for a particular method. But Wu et al. [94] proved that randomly generated test cases are more effective than those test cases that are derived from "special values" for MT. Segura et al. [79] also compared the fault detection effectiveness of random testing with manually generated test suites for MT. Their experimental results showed that randomly generated test suites are more effective in detecting faults than manually designed test suites. They also observed from their results that combining random testing with manually written tests provides better fault detection

ability than random testing only.

Batra and Sengupta [9] proposed a genetic algorithm approach to generate test cases maximizing the paths traversed in the PUT for MT. Chen et al. [16] also resolved the same problem from a different perspective. They proposed partitioning the input domain of the PUT into multiple equivalence classes for MT. They applied an algorithm that will generate test cases by covering those equivalence classes. They were able to generate source and follow-up test cases that provide a high fault detection rate. Symbolic Execution was used to construct MRs and generate their corresponding source test cases by Dong and Zhang [32]. At first, the program paths were analyzed to generate symbolic inputs, and then, these symbolic inputs were used to construct MRs. Finally, source test cases were generated by replacing the symbolic inputs with real values. Saha et al. [76] applied a coverage-based testing technique to generate test cases for MT. They compared their results with randomly generated test cases, and it outperforms the effectiveness of randomly generated test suite. Compared to their research, in this work, we evaluate the improvement of the fault detection effectiveness of MRs over the automatically generated test suites, specifically 3 commonly used coverage criteria (Line, Branch, and Weak Mutation).

5.10 Conclusion and Future Work

In this study, we empirically evaluated whether the fault detection effectiveness of automatically generated test suites can be improved using MRs. Our results show that augmenting automatically generated test cases with MRs can improve their fault detection capability since they provide a method to improve upon the trivial oracles used in these test cases. Our case study also shows that once the automatically generated test cases are augmented with MRs, their fault detection effectiveness is comparable to the developer test suites. This empirical study results also suggest

that identifying strong MRs is also important, which will help to increase the fault detection capability of automatically generated test suite where it fails to perform efficiently.

In the future we will extend this experiment with other automatic test case generation techniques like Adaptive Random Testing. To fully automate MR augmentation of automatically generated test cases, our plan is to integrate the MR identification approach [47] with source test case generation process. Finally, we will implement this automated test case generation process to publicly available METTester [68] tool (A Metamorphic Testing tool to test scientific applications).

CHAPTER SIX

A TEST SUITE MINIMIZATION TECHNIQUE FOR TESTING NUMERICAL PROGRAMS

6.1 Contribution of Authors and Co-Authors

Manuscript in Chapter 8

Author: Prashanta Saha

Contributions: Problem identification and proposing solution, running experiment, manuscript writing, creating tables and figures. Primary writer

Co-Authors: Dr. Clemente Izurieta, Dr. Upulee Kanewala

Contributions: Contribution in manuscript editing/writing, provided feedback, guidance and advice

6.2 Manuscript Information

Prashanta Saha, Dr. Clemente Izurieta and Dr. Upulee Kanewala

IEEE/ACIS International Conference on Software Engineering Research, Management and Applications (SERA)

Status of Manuscript:

_____ Prepared for submission to a peer-reviewed journal

_____ Officially submitted to a peer-reviewed journal

_____ Accepted by a peer-reviewed journal

__x__ Published in a peer-reviewed journal

IEEE

August 3, 2023

DOI:10.1109/SERA57763.2023.10197757

6.3 Abstract

Metamorphic testing is a technique that uses metamorphic relations (i.e., necessary properties of the software under test), to construct new test cases (i.e., follow-up test cases), from existing test cases (i.e., source test cases). Metamorphic testing allows for the verification of testing results without the need of *test oracles* (a mechanism to detect the correctness of the outcomes of a program), and it has been widely used in many application domains to detect real-world faults. Numerous investigations have been conducted to further improve the effectiveness of metamorphic testing. Recent studies have emerged suggesting a new research direction on the generation and selection of source test cases that are effective in fault detection. Herein, we present two important findings: i) a mutant reduction strategy that is applied to increase the testing efficiency of source test cases, and ii) a test suite minimization technique to help reduce the testing costs without trading off fault-finding effectiveness. To validate our results, an empirical study was conducted to demonstrate the increase in efficiency and fault-finding effectiveness of source test cases. The results from the experiment provide evidence to support our claims.

6.4 Introduction

Metamorphic Testing (MT) is a technique used to alleviate the *oracle* problem of software under test (SUT) [19]. A *test oracle* is a mechanism used to detect the correctness of the outcomes of a program [87]. In most cases of software behavior, it is easier to predict relationships between the elements of the output of a program, than to characterize the precise output given some input. For example, consider a program that computes the average of a list of real numbers. It is hard to correctly predict the observed output when the input has infinite possibilities. Thus, using this approach, we cannot validate whether the returned average is correct. However, we can permute the list of real numbers used in the input, and check to see if the returned output matches the output from the original test. If the outputs do not match, then there is a potential 'bug' (fault) in the program. This type of property is called a *metamorphic relation* (MR), a necessary property of the SUT that specifies a relationship between multiple test inputs and their outputs [21]. Thus, from existing test cases (i.e., source test cases) MRs are used to generate new test cases (i.e., follow-up test cases). The set of source and follow-up test cases are then executed on the SUT and the outputs are checked according to the corresponding MRs. The SUT can be considered faulty if an MR is violated.

MT has been successful in finding bugs in systems across various domains and has been successfully applied to detecting previously unknown faults in different domains such as web services, computer graphics, simulation and modeling, embedded systems etc. [78]. To date, work done on improving the fault detection effectiveness of MT has mainly focused on developing quality MRs [78]. However, developing such MRs is a labor-intensive task that requires the involvement of domain experts. Another, avenue to improve the fault detection effectiveness of MT, which has not, to our

knowledge, been explored thus far, is to systematically generate the source test cases. In fact, most of the previous studies in MT have used randomly generated test cases or existing test cases as source test cases when conducting MT [8, 20, 38, 79, 94]. Our previous work showed that the effectiveness of MT can be improved by systematically generating the source test cases based on some coverage criteria such as line, branch, and weak mutation (WM) [76]. But, it is sub-optimal to use a combination of all the coverage-based techniques to test numerical programs. A problem arises because test cases generated based on separate coverage criteria can be redundant, which means that there can be test case overlaps with the same code coverage as well as the same mutant killing rate making this approach inefficient.

In our research, we selected numerical programs as our target SUT since this domain has been relatively unexplored in the MT research [78]. The goal of this research is to select efficient and effective coverage-based test suites. To achieve this goal, we have divided our approach into two parts. First, we perform mutation analysis using a reduced set of mutants. Mutation analysis is necessary since we will apply it as our performance measurement metric to measure the fault-finding effectiveness of our test suites. This approach reduces the time and budget of the entire testing process. Second, we select the best coverage-based test suites among line, branch, and WM based on their cost-effectiveness and fault-finding effectiveness. In this way, we find the set of test suites with better efficiency and similar fault-finding effectiveness.

6.5 Background

MT is a testing technique that aims to alleviate the oracle problem. However, the effectiveness of MT not only depends on the quality of MRs but also on source test cases. In this section, we discuss MT and source test case generation techniques.

Specifically, we discuss line, branch, and WM coverage.

6.5.1 Metamorphic Testing

Source test cases are used in MT to generate follow-up test cases using a set of MRs identified for the program under test (PUT) [17]. MRs are identified based on the properties of the problem domain, such that said properties uniquely identify some expected behavior of the problem [22]. For example, there exist some unique characteristics of weather systems that help testers to find the correct MRs. We can create source test cases using techniques like random testing [20], structural testing [31], or search-based testing [76]. Follow-up test cases are generated by applying the input transformation specified by the MRs. After executing the source and follow-up test cases on the PUT we can check to see if the MR was violated. The violation of the MR during testing indicates fault in the PUT. Since MT checks that the relationships between the inputs and outputs of a test program are maintained under the conditions of an MR, we can use this technique when the expected result of a test program is unknown. For example, in figure 6.1, a Java method *add_values* is used to illustrate how source and follow-up test cases work within a PUT. The *add_values* method aggregates the array elements passed as an argument. The source test case, $t = \{3, 43, 1, 54\}$ is randomly generated and tested on *add_values*. The output of this test case is 101. In this program, we would expect that when a constant c is added to every element in the input collection, the output should increase accordingly. This expected behavior is used to generate an MR to conduct MT on this PUT. A constant value 2 is added to each element in the collection to create a follow-up test case $t' = \{5, 45, 3, 56\}$ that is then run on the PUT. The output of the follow-up test case is 109. To satisfy this *Addition* MR, the follow-up test output should be greater than the source output. In this MT example, the *Addition* MR is satisfied for the given

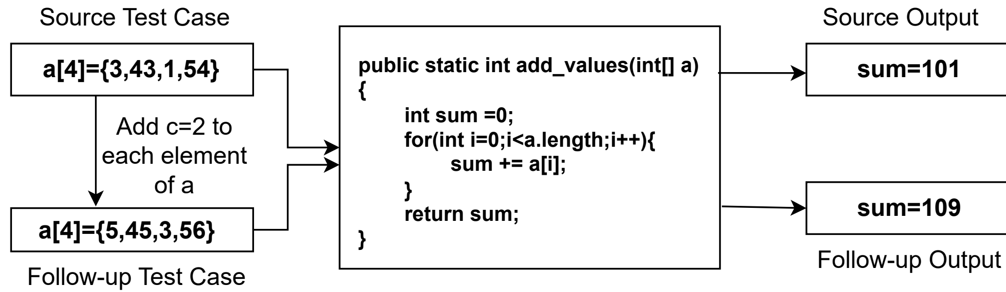


Figure 6.1: Test source and follow-up inputs on PUT.

source and follow-up test cases.

6.5.2 Coverage-based Test Case Generation

In this work, we used EvoSuite as the automated test case generation tool [36]. EvoSuite automatically generates test cases with coverage criteria e.g. line, branch, and WM approaches. EvoSuite uses an evolutionary search approach that simultaneously evolves test suites with respect to an entire coverage criterion. Below, we briefly describe the systematic approaches used by EvoSuite to generate coverage-based test suites.

6.5.2.1 Line Coverage

In Line coverage, to cover each statement of source code, it is required that each basic code block in a method is reached (except comments) [74]. In traditional search-based testing techniques, this reachability would be expressed by an association of branch distance and approach-level [54]. The branch distance measures how different a predicate (i.e., a decision-making point) is from evaluation to an expected target result. For example, given a predicate, $a == 7$ and an execution where the value of $a = 5$, the branch distance to the predicate evaluating to true would be $|5 - 7| = 2$, whereas execution where the value of $a = 6$ is closer to being true with a branch distance of $|6 - 7| = 1$. Branch distance can be estimated by applying

a set of standard rules [50, 54]. The approach level measures the closest point of a given execution to the target node. If any test suite executes all the statements of a method, then the approach level will be 0, which means it will become insignificant.

6.5.2.2 Branch Coverage Many popular tools have implemented in practice the idea of branch coverage, even though this practical approach may not always match the more theoretical interpretation of covering all edges of a program's control flow [36, 74, 80]. Branch coverage is often measured by maximizing the number of branches of conditional statements that are executed by a test suite. Thus, to satisfy a unit test suite for each of the branch statements, there is at least one test case that satisfies the branch predicate to *false* and at least one test case that satisfies the branch predicate to *true*. In branch coverage, the fitness function of a test suite is to cover all the branches in a method. This value is measured by calculating the closeness with which a test suite covers all the branches of a PUT.

6.5.2.3 WM When generating test cases from test generation tools, the preferred practice is to satisfy the constraints or conditions (i.e. in WM when a test case reaches the mutated statement of a method) rather than a developer's preferred boundary cases [36]. In WM testing, small code modifications are applied to the PUT. Then, the test generation tools are forced to generate values that can distinguish between the original test case and the mutant test case. In mutation testing, a test case is considered "killed" when the execution result of the mutant version is different than the original version of the PUT. The WM criteria are satisfied when at least one test case from the unit test suite reaches the infection state of the mutant. To measure the fitness value of the WM, it is required to calculate the infection distance with respect to a set of mutation operators [74].

6.6 Mutation Testing

Mutation testing has been used to evaluate the fault detection effectiveness of the automated test case generation approaches [29]. Mutation testing is a fault-based testing technique that measures the effectiveness of the test cases of the SUT. Many experiments suggest the usage of mutants as a proxy to real faults when comparing testing techniques [4]. Briefly, the testing technique follows these steps: First, mutants are created by simply seeding faults into a program. By applying syntactic changes to the original source code, new faulty versions of the original programs are generated. Each syntactic change is determined by an operator called a *mutation operator*. Test cases are then executed for the faulty and the original versions of the program and checked to see whether they produce different responses. If the response of the mutant is different from the original program, then we say that the mutant has been killed, and the test case is deemed to have the ability to detect faults effectively for that program. Otherwise, the mutant remains alive. When a mutant is syntactically different but semantically identical to the original program, it is referred to as an *equivalent mutant*. There are four common equivalent mutant situations: the mutant cannot be triggered, the mutant is generated from dead code, the mutant only alters the internal states of a program, and the mutant only improves the speed of execution of a program. The percentage of killed mutants to the total number of non-equivalent mutants provides an adequacy measurement of the test suite, which is called the *mutation score*.

6.7 Test Suite Minimization Approach

As systems evolve, their test suites are modified to accommodate new functionality. It is possible that redundant test cases (i.e., test cases for components that

are already covered by other existing test cases) are introduced as test suites grow. Test suite minimization techniques address this problem by seeking to permanently remove redundant test cases in a test suite. The goal is to create a more efficient test suite, that is, smaller in size but effective at finding faults. The minimization process is typically accomplished without the knowledge of the changes in the new version of the program [92].

Our goal for this research is to select efficient and effective coverage-based test suites. To achieve this goal, we divided our approach into two parts. In the first part, we performed mutation analysis using a reduced set of mutants. Mutation analysis is necessary because we apply it as our performance measurement metric to assess the fault-finding effectiveness of our test suites. This approach saves the time and budget of the entire testing process. In the second part, we selected the best test suites among line, branch, and WM coverage based on their cost and fault-finding effectiveness. This allowed us to find reduced test suites with better efficiency and similar effectiveness.

6.7.1 Mutants Reduction Approach

Selecting representative subsets from a given set of mutants is the principal aim of mutant reduction strategies. This technique reduces the application cost of mutation testing which leads to reductions in the total cost of software testing. Recent studies have shown two mutation reduction techniques that have proven to be highly effective [64]. But to the best of our knowledge, there are no mutation reduction techniques that have been applied to evaluate the fault-finding effectiveness of source test case generation techniques in MT. We claim that applying mutation reduction techniques to find better test case generation approaches is cost-effective in terms of time and budget. Thus, our goal is to find a better mutation reduction technique for

our test case generation approach.

6.7.1.1 Random Sampling Technique A major portion of the mutation testing demands is influenced by the generation and execution of the candidate set of mutants. By considering a small sample of mutants, a significant cost reduction can be achieved. Empirical studies have shown that a selection of 10% of mutants results in a 16% loss in the fault detection ability of the produced test sets when compared to full mutation testing [93]. In this study, we followed first-order mutation testing strategies [65] and selected a random x% portion of the initial mutants set, where $x = 10, 20, 30, 40, 50$, and 60. Our target was to find out which random % of selected mutants is a better representative of the total mutants set.

6.7.1.2 Operator Based Mutant Selection Since mutant operators generate different numbers of mutant programs, Offutt et al. proposed $N-\textit{selective mutation}$ theory, where N is # of mutant operators [61]. In their experiment, they divided the mutant operators into three general categories based on the syntactic elements that they modify. The three categories are *Replacement-of-operand* operators (i.e., replace each operand in a program with each other legal operand), *Expression modification* operators (i.e., modify expressions by replacing operators and inserting new operators), and *Statement modification* operators (i.e., modify entire statements). Their experiments suggest that *Expression modification* operators with a smaller number of mutants than the total mutant set can be effective and the execution time is also shown to be linear. In this study, we applied *Expression modification* operators as mutants set to do the mutation analysis.

6.7.2 Effective Test Suites Selection

In preliminary work, we showed that coverage-based test cases have better fault detection effectiveness than randomly generated test cases [76]. However, it is not feasible to use all of the coverage-based techniques together to test numerical programs. This is because the process is time-consuming and test cases are repetitive regarding code coverage. Further, the fault detection effectiveness of test suites can vary based on the methods used. Therefore, we need an effective approach to help select better test suites when approaching SUT.

Mutation testing has been proven to be an effective approach to assessing the fault detection effectiveness of test case generation techniques. In our approach, we run mutation testing on the SUT and applied the test cases generated by the coverage-based test case generation techniques. After that, we measured the mutation score for each of the techniques. The test case generation technique with the highest mutation score was selected as a source test suite to test SUT.

6.8 Empirical Evaluation

This section describes the design of the empirical evaluation approach: the research questions we will answer for our experiments, the description of the subject programs selected for the evaluation, description of the identified MRs for the subject programs and the evaluation process of the case study.

6.8.1 Research Questions

- **RQ1: Which Mutant reduction technique is best suited for detecting faults in MT?**

- **RQ2: Which coverage-based test suites have better fault-finding**

effectiveness?

- **RQ3: Can test suite minimization techniques reduce the cost of executing a test suite and what is their effect on the fault detection effectiveness of a test suite?**

6.8.2 Subject Programs

We built a code corpus containing 96 methods that take numerical inputs and produce numerical outputs. We obtained these functions from the following open-source projects:

- **The Colt Project[1]:** A set of open source libraries written for high-performance scientific and technical computing in Java.
- **Apache Mahout[2]:** A machine learning library written in Java.
- **Apache Commons Mathematics Library[3]:** A library of lightweight and self-contained mathematics and statistics components written in Java.
- **Matrix.java:** This class has methods that perform matrix operations. We selected 20 methods randomly from the class and conducted our experiment on them. The description of these 20 methods is available in this GitHub repository[4].

Functions in the code corpus perform various calculations using sets of numbers such as calculating statistics (e.g., average, standard deviation, and kurtosis), calculating distances (e.g., Manhattan and Tanimoto), and searching/sorting. The total number

[1] http://acs.lbl.gov/software/colt/
[2] https://mahout.apache.org/
[3] http://commons.apache.org/proper/commons-math/
[4] https://github.com/ps073006/ConfRepo

of lines of code for these functions varied between 4 and 52, and the number of input parameters for each function varied between 1 and 4.

6.8.3 MR Identification

We selected all six MRs that were used in previous studies to test methods from the first 3 projects mentioned in section 6.8.2 [76]. Suppose our source test case is $X = \{x_1, x_2, x_3, ..., x_n\}$ where $x_i \geq 0, 0 \leq i \leq n$. Let source and follow-up outputs be $O(X)$ and $O(Y)$ respectively:

- **Addition:** add a positive constant C to the source test case yielding the follow-up test case $Y = \{x_1 + C, x_2 + C, x_3 + C, ..., x_n + C\}$. Then $O(Y) \geq O(X)$.
- **Multiplication:** multiply the source test case by a positive constant C yielding the follow-up test case $Y = \{x_1 * C, x_2 * C, x_3 * C, ..., x_n * C\}$. Then $O(Y) \geq O(X)$.
- **Shuffle:** randomly permute the elements in the source test case. The follow-up test case can then be $Y = \{x_3, x_1, x_n, ..., x_2\}$. Then $O(Y) = O(X)$.
- **Inclusive:** include a new element $x_{n+1} \geq 0$ in the source test case yielding the follow-up test case $Y = \{x_1, x_2, x_3, ..., x_n, x_{n+1}\}$. Then $O(Y) \geq O(X)$.
- **Exclusive:** exclude an existing element from the source test case yielding the follow-up test case $Y = \{x_1, x_2, x_3, ..., x_{n-1}\}$. Then $O(Y) \leq O(X)$.
- **Inversion:** take the inverse of each element of the source test case. Then the follow-up test case will be $Y = \{1/x_1, 1/x_2, 1/x_3, ..., 1/x_n\}$. Then $O(Y) \leq O(X)$.

We identified and developed the following ten MRs for testing the functions in the Matrix.java class. We have verified if the MRs satisfies each of the methods from the class and found out not all these MRs are satisfied by each of these methods.

The entire list of methods and the specific MRs satisfied by them can be found in this GitHub repository[4]. In all cases, we assume that Matrix A comprises only non-negative numbers.

- **Scalar Addition:** Let A be the initial input matrix to a program P, and b be a positive scalar. Let A' be the follow-up input matrix where $A' = \forall i, j \in b + A_{i,j}$. Let the output of P for A be O (i.e. $P(A) = O$) and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' \geq \sum_{i,j} O$.

- **Addition With Identity Matrix:** Let A be the initial input matrix to a program P, and I be an identity matrix. Let A' be the follow-up input matrix where $A' = \forall i, j \in I_{i,j} + A_{i,j}$. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' \geq \sum_{i,j} O$.

- **Scalar Multiplication:** Let A be the initial input matrix to a program P, and b be a positive scalar. Let A' be the follow-up input matrix where $A' = \forall i, j \in b.A_{i,j}$. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' \geq \sum_{i,j} O$.

- **Multiplication With Identity Matrix:** Let A be the initial input matrix to a program P, and I be an identity matrix. Let A' be the follow-up input matrix where $A' = \forall i, j \in I_{i,j}.A_{i,j}$. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' = \sum_{i,j} O$.

- **Transpose:** Let A be the initial input matrix to a program P. Let A' be the follow-up input matrix where $A' = \forall i, j \in A^T_{i,j} = A_{j,i}$. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' = \sum_{i,j} O$.

- **Matrix Addition:** Let A be the initial input matrix to a program P. Let A' be the follow-up input matrix where $A' = \forall i, j \in A_{i,j} + A_{i,j}$. Let $P(A) = O$ and

$P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' \geq \sum_{i,j} O$.

- **Matrix Multiplication:** Let A be the initial input matrix to a program P. Let A' be the follow-up input matrix where $A' = \forall i, j \in A_{i,j}.A_{i,j}$. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' \geq \sum_{i,j} O$.

- **Column Permutation:** Let A be the initial input matrix to a program P with $j = 1, 2, 3, .., n$ columns. Let A' be the follow-up input matrix after permuting the column positions of A. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' = \sum_{i,j} O$.

- **Row Permutation:** Let A be the initial input matrix to a program P with $i = 1, 2, 3, .., n$ rows. Let A' be the follow-up input matrix after permuting the row positions of A. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' = \sum_{i,j} O$.

- **Element Permutation:** Let A be the initial input matrix to a program P with $j = 1, 2, 3, .., n$ columns and $i = 1, 2, 3, .., n$ rows. Rows and columns have to be same size. Let A' be the follow-up input matrix after permuting $A_{i,n}$ element with $A_{n,j}$ element. Let $P(A) = O$ and $P(A') = O'$. Then the expected output relation is $\sum_{i,j} O' = \sum_{i,j} O$.

6.8.4 Evaluation Approach

In our evaluation, we applied the μJava[5], and PIT[6] tools to systematically generate mutants for our subject programs. For the 96 methods from the five open-source java projects, we generated a total of 8330 mutated versions using the mutation tools. Mutant distributions for the five classes are shown in Table 6.1. We ran

[5]https://github.com/jeffoffutt/muJava
[6]https://pitest.org/

mutation testing on the five classes and exclude the mutants that cause compilation errors, runtime exceptions, and equivalent mutants. We also manually verified all the MRs for each subject programs.

Table 6.1: Individual classes from five open source projects. We show counts of Methods, MRs and mutants

Class name	# Methods	# Mutants	# MRs
MethodsCollection2.java	28	1875	6
MethodsFromMahout.java	5	409	6
MethodsFromApacheMath.java	18	2248	6
MethodsFromColt.java	25	2914	6
Matrix.java	20	884	11

6.9 Results and Discussions

Below we discuss the results of our experiments and provide answers to our research questions:

RQ1. Which Mutant reduction technique is best suited for detecting faults in MT? Figure 6.2 displays the average mutation scores (in %) per mutant reduction strategies for each class from SUT. The columns of Table 6.2 "Total Mutants" and "Selected Mutants" allow us to derive the mutant reduction percentage shown in the third column. We excluded Matrix.java class while analyzing the answer of RQ1 because we utilized the PIT tool to generate mutants for this class and this tool does not provide the mutant operators' names. So, we could not distinguish expression modification operators from the mutants set. The most interesting aspect of the figure 6.2 is that the operator-based mutation strategy always detects more

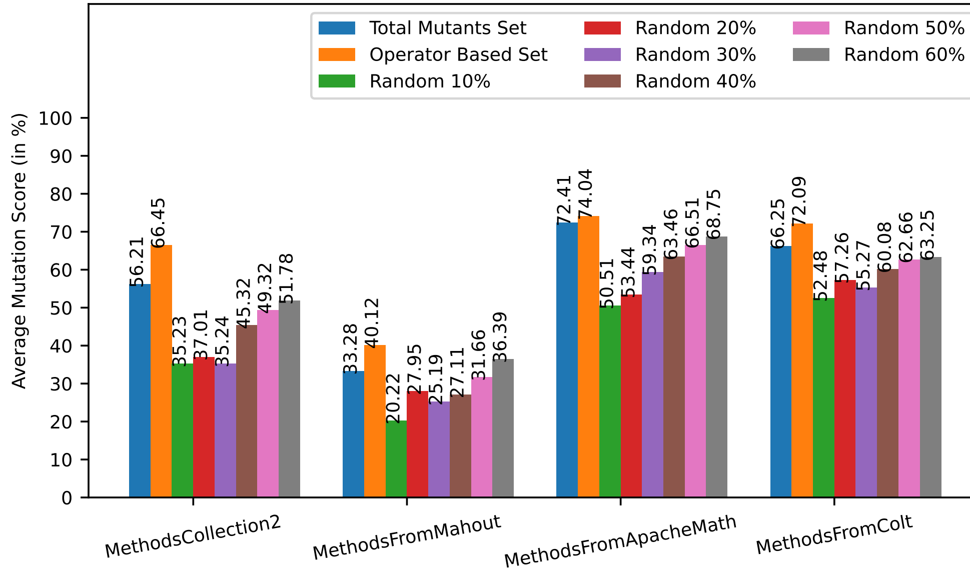


Figure 6.2: Average mutation scores of mutation testing strategies. Mutation testing was done by the following strategies: total mutants set, operator based set, Random 10%, Random 20%, Random 30%, Random 40%, Random 50%, and Random 60%. The average mutation score was calculated by averaging the mutation scores of all the methods from each of the classes.

faults than any other strategy and in the majority of cases this situation is statistically significant (p-value<0.05) (pairwise T-test results are available here[7]). Although testing with total mutants set (7446 mutants) has a comparatively high mutation score than randomly selected mutants set, but it is not as high as the operator-based mutation strategy. Mutation strategies where randomly selected mutants are generated (ranging from 10% to 60%) have comparatively low mutation scores.

Table 6.2 shows the savings obtained using the operator-based mutation strategy in terms of the number of mutants. The column “Percentage Saved” was computed by subtracting the number of ”Selected Mutants” from the number of ”Total Mutants”

[7]https://github.com/ps073006/ConfRepo

Table 6.2: Savings obtained by operator based mutant reduction approach

Class name	Total Mutants	Selected Mutants	Percentage Saved
MethodsCollection2.java	1875	818	56.37
MethodsFromMahout.java	409	163	60.15
MethodsFromApacheMath.java	2248	723	67.84
MethodsFromColt.java	2914	994	65.89
Total	7446	2698	63.77

and dividing the difference by the number of "Total Mutants." The operator-based mutation strategy sets save anywhere between 56% to 67% of the total mutants that are generated across the subject programs. The expression modification operators (e.g., AORB, AORS, LOR, ROR, AOIU, COI, LOI) from the μJava tool are used for the operator-based mutation strategy.

RQ2. Which coverage-based test suites have better fault-finding effectiveness? Figure 6.3 shows the average mutation scores of the coverage-based test suites (Line, Branch, and WM) generated for the MT to test the five subject programs. To answer this RQ we combined the mutation scores of MT (source & follow-up test cases) for each subject programs.

The most interesting aspect of the figure is that WM-based test suites always detect more faults than any other test suites and in the majority of the subject programs. These results are statistically significant ($p\text{-value}<0.05$)(pairwise T-test results are available here[8]). However, in MethodsfromColt.java class, the difference in mutation scores is really small between the branch and WM test suites. **RQ3.**

[8]https://github.com/ps073006/ConfRepo

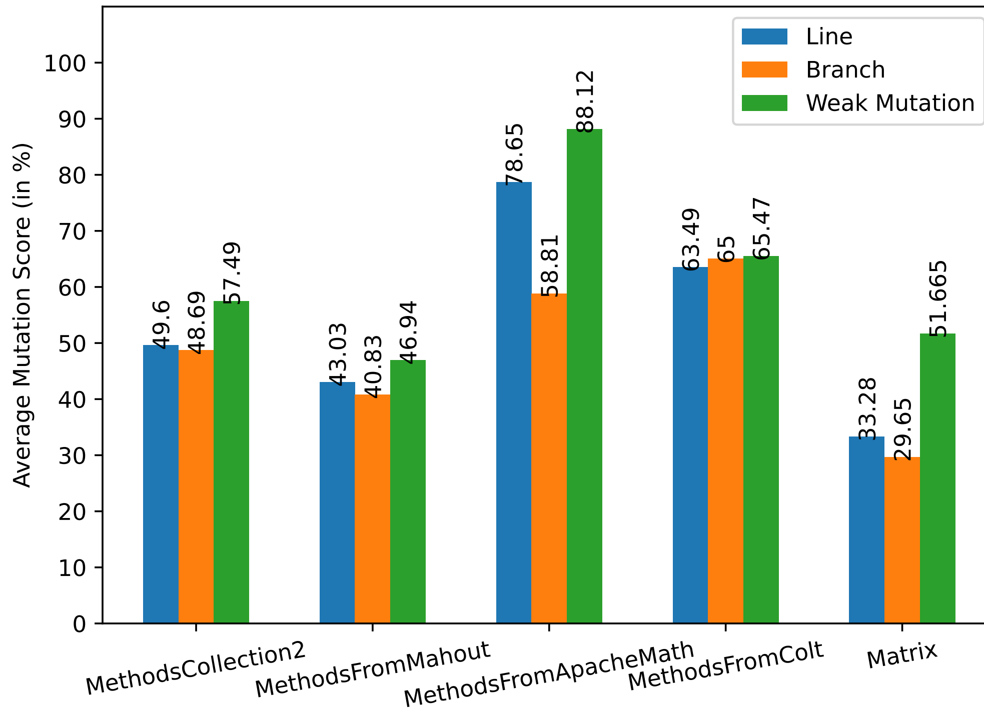


Figure 6.3: Comparison of average mutation scores of coverage based test suites, e.g. Line, Branch, and Weak Mutation. The average mutation score was calculated by averaging the mutation scores of all the methods from each of the classes.

Can test suite minimization techniques reduce the cost of executing a test suite and what is their effect on the fault detection effectiveness of a test suite? Table 6.3 reports the percentage reduction in test suite size for each of the five subject classes for the combined test suite of line, branch, and WM as well as WM coverage criteria separately. The results show that test suite minimization can significantly reduce the size of a test suite. The results also show that the WM coverage criteria yield a larger reduction in test suite size (53.13–65.43%). Results show that there is little variation in the reduction in test suite size, which means the performance of the WM test suites is linear across the subject programs.

Table 6.3: Cost effectiveness of test suite minimization technique for MT based on test suite size

	Total Test Suite Size		
Class Name	Combined	Weak Mutation	% Reduced
MethodsCollection2.java	111	47	57.66
MethodsFromMahout.java	32	15	53.13
MethodsFromApacheMath.java	104	44	57.69
MethodsFromColt.java	81	28	65.43
Matrix.java	37	14	62.16

Table 6.4 reports the percentage reduction in fault detection effectiveness and code coverage of the WM coverage criteria. Minimized test suites of the WM coverage criteria perform well overall (1.44-14.14%) with fault detection effectiveness reduction as compared to the combined test suites. Also, the performance remains the same (0% reduction) for the code coverage with the minimized test suites of the WM coverage criteria as compared to the combined test suites.

Minimized test suites with WM can produce significant reductions in test suite size, resulting in significant savings in test execution costs. When using WM coverage criteria as a source test case generation technique for MT, test suite minimization causes an average reduction in fault detection effectiveness that is less than 9%. This makes the approach potentially useful in practice when testing time is limited and the system is not critical.

Table 6.4: Percentage reduction on fault detection and code coverage after applying test suite minimization technique in MT

A: Percentage reduction on code coverage

	Code Coverage		
Class Name	Combined	WM	% Reduced
MethodsCollection2.java	99.7	99.7	0
MethodsFromMahout.java	99.32	99.32	0
MethodsFromApacheMath.java	98.4	98.4	0
MethodsFromColt.java	99.16	99.16	0
Matrix.java	97.49	97.49	0

B: Percentage reduction on fault detection

	Fault Detection (Mutation score %)		
Class Name	Combined	WM	% Reduced
MethodsCollection2.java	64.21	57.49	10.47
MethodsFromMahout.java	54.28	46.94	13.52
MethodsFromApacheMath.java	89.41	88.12	1.44
MethodsFromColt.java	76.25	65.47	14.14
Matrix.java	53.71	51.67	3.8

6.10 Threats to Validity

We have followed Wohlin et al. guidelines while discussing the threats to the validity of our empirical study [91].

Threats to *internal validity* cause and effects may result from the way in which the empirical study was carried out. To increase our confidence in the experimental setup and mitigate this threat, we ran our experiments 10 times with the same setup.

Threats to *construct validity* may occur because of the third-party tools we have used. The EvoSuite tool was used to generate source test cases for line, branch, and WM test generation techniques. Further, we used the μJava and PIT mutation tool to create mutants for our experiment. To minimize these threats we verified that the results produced by these tools are correct by manually inspecting randomly selected outputs produced by each tool.

Threats to *external validity* were minimized by using the 96 methods from 5 different open-source project classes. This provides high confidence that the generalization of our results to other open-source software is appropriate. We only used the EvoSuite tool to generate test cases for our major experiment. But we also used the JCUTE [9] tool to generate branch coverage-based test suites for our initial case study and also observed similar results [80].

6.11 Conclusion

This paper presented an empirical study on the effects of test suite minimization on MT. Also, we present a hybrid approach to the reduced mutants set finding and the reduction in fault-finding effectiveness problems on numerical programs. The study

[9]https://github.com/osl/jcute

used open source java projects which have been applied in previous studies, and test suites created using a systematic state-of-the-art approach. The mutated version of the code can be comparable with real-world faults.

The study showed that an operator-based mutant reduction technique can significantly reduce the mutant set size for mutation testing. This technique also keeps the mutation score comparable to the original mutant set. Practically this approach helps to reduce the total testing costs. However, we still need to perform industrial case studies to continue to scale the solutions presented in this work. Industrial case studies are still required to increase the power of these results.

The results also revealed that using minimized test suites with WM coverage criteria for MT provides a trade-off between the reduction in execution cost (53.13–65.43%) and the reduction in fault finding effectiveness (1.44–14.14%) that might be suitable in certain contexts in non-critical systems, where testing time and resources are limited.

Surprisingly, to date, there are no case studies that report on the impact of test suite minimization on fault detection effectiveness for MT. But, there were few case studies reported on test case prioritization for MT. This empirical study is the elemental step in this direction. Our future goal is to apply our proposed approach on real fault based programs such as Defects4J [10].

[10] https://github.com/rjust/defects4j

CHAPTER SEVEN

FAULT DETECTION EFFECTIVENESS OF METAMORPHIC RELATIONS DEVELOPED FOR TESTING SUPERVISED CLASSIFIERS

7.1 Contribution of Authors and Co-Authors

Manuscript in Chapter 7

Author: Prashanta Saha

Contributions: Problem identification and proposing solution, running experiment, manuscript writing, creating tables and figures. Primary writer

Co-Author: Dr. Upulee Kanewala

Contributions: Contribution in manuscript editing/writing, provided feedback, guidance and advice

7.2 Manuscript Information

Prashanta Saha and Dr. Upulee Kanewala

IEEE International Conference On Artificial Intelligence Testing (AITest)

Status of Manuscript:

_____ Prepared for submission to a peer-reviewed journal

_____ Officially submitted to a peer-reviewed journal

_____ Accepted by a peer-reviewed journal

__x__ Published in a peer-reviewed journal

IEEE

May 20, 2019

DOI:10.1109/AITest.2019.00019

7.3 Abstract

In machine learning, supervised classifiers are used to obtain predictions for unlabeled data by inferring prediction functions using labeled data. Supervised classifiers are widely applied in domains such as computational biology, computational physics and healthcare to make critical decisions. However, it is often hard to test supervised classifiers since the expected answers are unknown. This is commonly known as the *oracle problem* and metamorphic testing (MT) has been used to test such programs. In MT, metamorphic relations (MRs) are developed from intrinsic characteristics of the software under test (SUT). These MRs are used to generate test data and to verify the correctness of the test results without the presence of a test oracle. Effectiveness of MT heavily depends on the MRs used for testing. In this paper we have conducted an extensive empirical study to evaluate the fault detection effectiveness of MRs that have been used in multiple previous studies to test supervised classifiers. Our study uses a total of 709 reachable mutants generated by multiple mutation engines and uses data sets with varying characteristics to test the SUT. Our results reveal that only 14.8% of these mutants are detected using the MRs and that the fault detection effectiveness of these MRs do not scale with the increased number of mutants when compared to what was reported in previous studies.

7.4 Introduction

Supervised classifiers are widely used for making predictions in diverse domains. For instance, over fifty real world computational applications use support vector machines for classification [1]. As these types of applications are becoming part of our daily life, ensuring their quality becomes even more important [55]. In such applications, formal proofs of the underlying algorithm does not always guarantee that it implements that algorithm correctly. Therefore, software testing is imperative to assure the quality of these systems.

Often, conventional software testing approaches may not be feasible for assuring the quality of supervised classifiers because of the absence of a test oracle that determines the correctness of produced test outputs. This class of software applications is often referred to as "non-testable programs" [87]. Further, usually supervised classifiers are not 100% accurate. Thus, an incorrect prediction does not necessarily mean that there is a fault in the program. These characteristics of supervised classifiers make it hard to detect subtle faults in these applications.

To date, limited work has been done on systematic testing of software systems that incorporate machine learning. Among them, metamorphic testing (MT) has been used widely for testing software applications that uses supervised machine learning algorithms [30, 35, 58, 96]. MT uses metamorphic relations (MRs) for testing the software under test (SUT), where MRs act as partial oracle [21, 24]. A MR specifies how the outputs should change according to a specific change made to the input. Thus, from existing test cases (named as *source test cases*) MRs are used to generate new test cases (named as *follow-up test cases*). If the changes found between the outputs of the source and follow-up test cases are not as expected according to the MR, then there is a defect in the SUT. Thus, using MRs we can address the oracle

problem presented by supervised machine learning classifiers.

Several previous studies have defined MRs for testing supervised classifiers. Xie et al. proposed a set of MRs based on user expectations to validate supervised classifiers [96]. Dwarakanath et al. developed MRs for two image classifiers that are based on support vector machines and deep learning [35]. Ding et al. developed three levels of MRs to test and validate a deep learning framework [30]. The evaluations conducted in these studies to measure the fault detection effectiveness of the developed MRs is fairly limited due to the number of mutants used. For example Xie et al. used 24 mutants and Dwarakanath et al. used 22 mutants in total. These numbers are significantly low especially considering the number of classes and the number of lines of source code involved with these SUTs.

To overcome this limitation, in this paper, we report the findings of a large scale experiment that we conducted to evaluate the fault detection effectiveness of MRs developed for supervised classifiers. In this experiment we used a total of 709 reachable mutants (i.e. the mutated statement in the mutant was executed with the test cases) that were generated for a real world supervised classifier implementation from Weka [89]. Our results show a significant reduction of the fault detection effectiveness with the increased number of mutants.

Rest of the paper organized as follows: Section 7.5 describes the background of this work, including an overview of supervised machine learning and the k-Nearest Neighbors algorithm, which is used as the SUT in this study. Section 7.6 discusses more about MT and the MRs used for testing. In Section 7.7 we discuss the details of our experimental approach and mutation analysis. Section 7.8 presents the results and their analysis. Section 7.9 identifies the related work and Section 7.10 contains our conclusions and future work.

7.5 Background

7.5.1 Supervised Machine Learning Classifiers

Supervised classification is the task of deducing a function from labeled training data such that it can be used to predict unknown labels on test data. Training data can be represented by two vectors of size k. One vector is the training samples $S =< s_0, s_1, ..., s_{k-1} >$ and the other one is the class labels $C =< c_0, c_1, .., c_{k-1} >$ where, c_i is the class label for s_i. Each sample s_i has m features from which the prediction function will be learned. Class labels are a finite set and each class label c_i is an element of it, i.e. $c \in L =< l_0, l_1, ..., l_{n-1} >$, where n is the number of class labels [96].

Supervised machine learning applications execute in two phases: the *training phase* and *testing phase*. In the *training phase*, a set of training samples are used by a supervised classification algorithm to learn a prediction function. To develop the prediction function, the supervised learning algorithm would analyze how the attributes relate to the class label. In the *testing phase*, the prediction function is applied to unseen data known as the *test set*, where the class labels are unknown. The application attempts to predict the class label for each instance in the test set using the learned prediction function [96]. Some of the commonly used supervised classification algorithms are K-Nearest Neighbors [28], Naive Bayes [73] and Support Vector Machine [1].

7.5.2 K-Nearest Neighbors

This study uses an open-source implementation of the K-Nearest Neighbors (kNN) algorithm as the SUT. kNN is particularly chosen due to its popularity in the machine learning community and is used in domains such as recommendation

systems, semantic searching and anomaly detection etc. Further, Xie et al. used kNN in their study and using the same algorithm would allow us to do a comparison with their results [96]. However, the MT approach discussed here should be applicable to other supervised learning algorithm implementations.

In kNN, for a sample training set S, each sample set has m attributes,$< att_0, att_1, ..., att_{m-1} >$, and also n classes,$< l_o, l_1, ..., l_{n-1} >$. The sample test data is t_s,$< a_0, a_1, ..., a_{m-1} >$. kNN computes the distance between each sample training set and the test case. Euclidean distance metric is one of the most popular approach to measure distance. For sample $s_i \in S$, the value for each attribute is $< sa_0, sa_1, ..., sa_{m-1} >$. And the euclidean distance formula is:

$$dist(s_i, t_s) = \sqrt{\sum_{j}^{m-1} (sa_j - a_j)^2}$$

Once the distance is calculated, kNN selects the k nearest training samples for the test data after sorting all the distances. These k samples from the training set are considered as the *k-nearest neighbors* of the test case. Then, kNN calculates the proportion of each label in the selected k-nearest neighbors. The class label with the highest proportion is predicted as the label for the test data.

7.6 Metamorphic Testing for Supervised Classifiers

Often, programs exhibit properties such that if the test input is changed in a way that the new output can be predicted based on the original output. In MT, such properties (known as MRs) are used as partial oracles to conduct testing [17, 22]. In practice one can easily apply MT. As the first step, it is necessary to identify MRs that can relate multiple pairs of the inputs and outputs of the SUT.

Then, source test cases are generated using techniques like random testing,

Table 7.1: Sample data set

<table>
<tr>
<td>@attribute pictures numeric
@attribute paragraphs numeric
@attribute files numeric
@attribute files2 numeric
@attribute profit {0,1,2,3,4}

@data
45,3,16,38,0
15,87,89,46,4
59,77,94,11,0
86,89,94,15,2
80,28,94,11,4
23,12,47,41,1
94,15,22,15,0
95,26,97,76,3
50,90,0,72,2
33,46,47,95,0</td>
<td>@attribute pictures numeric
@attribute paragraphs numeric
@attribute files numeric
@attribute files2 numeric
@attribute profit {0,1,2,3,4}

@data
6,40,8,89,0</td>
</tr>
</table>

structural testing or search based testing and the corresponding follow-up test cases are constructed based on the MRs. In our previous studies we investigated how the fault detection effectiveness of MT varies with various source test case generation techniques such as different structural coverage based approaches and our results show that coverage based source test case generation outperforms randomly generated source test cases [76]. After executing the source and the follow-up test cases on the SUT we can check if there is a change in the output that matches the MR, if not the MR is considered as violated. Violation of a MR during testing indicates faults in the SUT. Since MT checks relationship between inputs and outputs of a test program, we can use this technique when the expected result of individual test output is unknown.

For example, *Consistence with affine transformation* MR described in Section 7.6.1 can be used to test a kNN classifier. Source test case for kNN can be randomly generated (see Table 7.1 for an example - training data on the left and test data on the right). After executing this source test case, the output will be the class label predicted for that test case which is 0 for this example. To generate follow-up test case, we apply the input transformation described in the above MR, where an arbitrary affine transformation function is applied to the attributes of both the training data and test data. After executing the follow-up test case the output is 0 which is the predicted class label for the transformed test data. To satisfy this MR both the source and follow-up test case outputs should be same. Therefore, in this example, the considered MR is satisfied for the given source and follow-up test cases.

7.6.1 Identified MRs for Testing KNN

Murphy et al. [58] suggested six MRs (additive, multiplicative, permutative, invertive, inclusive and exclusive) that can be applied to machine learning applications including both supervised and unsupervised ML. Xie et al. developed a set of MRs

based on the user expectations of supervised classifiers [96]. In our study, we mainly use the MRs developed by Xie et al. to test kNN. Some variations of the same MRs are used in some recent studies as well [35]. Below we provide a brief description of these MRs (formal definitions can be found in [96]).

MR1: Consistence with affine transformation. If we apply some affine transformation function, $f(x) = kx + b, (k \neq 0)$, to every value x in some subset of features in the training and testing data, and then create a new model using this data, the predictions made by the model should be unchanged.

MR2: Permutation of the attribute. If we permute the order of the attributes, or features, of all the samples in the training and testing data, the result of the predictions of the test data should not change.

MR3: Addition of uninformative attributes. If we add some new feature that is equally associated with all classes, the predictions of the test data should not be changed.

MR4: Consistence with re-prediction. Suppose we predict some test case t as class l_i. If we append t to our training data and re-create the model, t should still be classified as class l_i.

MR5: Additional training sample. Suppose we predict some test case t as class l_i. If we duplicate all samples of class l_i in our training data and re-classify our test data, t should still be classified as l_i. More generally, every test case predicted as class l_i should still be predicted as class l_i with the duplicated samples.

MR6: Addition of classes by re-labeling samples. For some test cases not of class l_i, we switch the class label from x to x^*. Then every test case predicted as class l_i should still be predicted as class l_i with the re-labeled samples.

MR7: Permutation of class labels. If we permute the order of the class labels with some random permutation $p(l_i)$ where l_i is a class label, all test cases

which were predicted as l_i should now be predicted as $p(l_i)$.

MR8: Addition of informative attribute. If we add some new feature that is strongly associated with one class, l_i, then for every prediction that was class l_i, the prediction with this new attribute should also be class l_i.

MR9: Addition of classes by duplicating samples. Suppose we duplicate every class except for n, and give them all a new class. For example, if we originally had class labels of 1, 2, 3, and 4, then we would create class labels of 1, 1*, 2, 2*, 3, 4, 4*. Then every test case predicted as class l_i (class 3 in this example) should still be predicted as class l_i with the duplicated samples.

MR10: Removal of classes. If we remove some class l_i, the remaining predictions should remain unchanged.

MR11: Removal of samples. If we remove samples that have not been predicted as class l_i, then all cases which were predicted as l_i should remain unchanged.

When using these MRs for testing kNN, it is important to select the appropriate value for k (i.e. number of nearest neighbours) such that the MR becomes a necessary property for kNN. Table 7.2 shows the k values used with each MR for verification of kNN [96].

7.7 Experimental Study

The goal of this experimental study is to conduct an in-depth evaluation of the fault detection effectiveness of the MRs listed in Section 7.6.1. We used the kNN implementation in Weka 3.5.7 as the SUT [89].

7.7.1 Research Questions

We conducted a set of experiments to answer the following research questions:

Table 7.2: Metamorphic relations for kNN used in mutation analysis

k	Metamorphic relation
k=3	MR1 Consistence with affine transformation
	MR2 Permutation of the attribute
	MR3 Addition of uninformative attributes
	MR4 Consistence with re-prediction
	MR5 Additional training sample
	MR6 Addition of classes by re-labeling samples
k=1	MR7 Permutation of class labels
	MR8 Addition of informative attribute
	MR9 Addition of classes by duplicating samples
	MR10 Removal of classes
	MR11 Removal of samples

1. **How does the fault detection rate of MRs vary as the number of mutants increase?** As we mentioned above the fault detection effectiveness of these MRs were measured using a limited set of mutants in previous studies [96]. But it is important to use a reasonable number of mutants to evaluate the fault detection effectiveness using the mutation killing rate. Therefore we increase the number of mutants significantly and measure the killing rate.

2. **Does the fault detection rate of MRs change with varying the data set size in the source test cases?** There are two major components in MT that determines its fault detection effectiveness: the MRs and the source test cases. We examined whether varying the data set size of the source test case can effect the mutant killing rate for a given MR.

3. **Does the fault detection effectiveness vary with the mutation engine**

used to generate the mutants? As we discussed above the underlying process used by MuJava and Major for generating mutants is different. The purpose of this research question is to see whether that one category of mutants are hard to kill than the other.

7.7.2 Source and Follow-up Test Cases

In an individual source test case there is a training set and a test set. Similar to Xie et al. [96], we used a random approach to generate these source test cases. In each training and test set, there are four numerical attributes that are named as: $\{pictures, paragraphs, files, files2\}$. The class label $profit$ can have five values $\{0, 1, 2, 3, 4\}$. The value of each attribute is randomly selected and ranges within [0, 100]. The value of the class labels are also selected randomly. The training set size ranges within [10, 200]. Table 7.1 shows a sample source test case, with the training data set on the left and test data set on the right.

We transform the source test cases to obtain the corresponding follow-up test cases according to the MRs described in Section 7.6.1. Multiple source and follow-up test case pairs are generated for each MR by varying the number of samples in the training data set as well as the size of source test case.

We executed all the source and follow-up test case pairs on the original kNN implementation and validated the outputs against each MR before proceeding to the mutation analysis described below. The original kNN implementation did not report any MR violations.

7.7.3 Mutation Analysis

To evaluate the fault detection effectiveness of the MRs described in Section 7.6.1, we use a mutation engine to systematically inject defects into the SUT. Mutation testing has been extensively used to evaluate fault detection effectiveness,

as many experiments suggest that mutants are a proxy to the real faults for comparing testing techniques [4]. As we mentioned above, previous studies used mutation analysis to evaluate the fault detection effectiveness of MRs [96]. But, the number of mutants used in the mutation analysis is quite low compared to the size of the SUT's used in those experiments.

In our evaluation, we applied MuJava [52] and Major [45] tools to systematically generate mutants for kNN in Weka-3.5.7. MuJava is a powerful and automatic mutation analysis system, which can support both method-level and class-level mutation operators. MuJava provides various types of mutants, including inter-class, intra-class, inter-method and intra-method level of mutants. In this experiment, we only included the intra-method level of mutants. Major is a mutation testing framework which manipulates the abstract syntax tree of the SUT. Similar to MuJava, we only used the intra-method level mutant operators from the Major tool.

MuJava and Major allows users to define which parts of the source code needs to be mutated. Since, Weka is a large scale software with about 16.4M source code and our experiments only focuses on the functionality of the kNN classifier, we only selected the class files which are directly related to the kNN classifier according to its hierarchy structure. Table 7.3 shows the names of the selected class files in our mutation analysis.

We generated all possible mutants for the 6 class files in Table 7.3. After excluding the mutants that caused compilation errors, runtime exception as well as equivalent mutants, we have obtained a total of 1500 mutants from the MuJava and Major mutation tools. From those mutants we identified 609 mutants generated by MuJava that are reachable by the test cases that we described above. From the mutants generated by Major, we randomly selected 100 mutants that are reachable by the test cases due to time limitations. The distribution of mutants between the

Table 7.3: Selected files for mutation analysis

kNN
weka.classifiers.lazy.IBk.java
weka.core.Attribute.java
weka.core.neighboursearch.LinearNNSearch.java
weka.core.neighboursearch.NearestNeighbourSearch.java
weka.core.NormalizableDistance.java
weka.core.EuclideanDistance.java

two tools are described in Table 7.4.

Table 7.4: Details of mutants

Tool Name	Total # of mutants generated	# of mutants used
MuJava	2383	609
Major	987	100

7.8 Results and Discussions

Below we discuss the results of our experiments and provide answers to our research questions.

1. How does the fault detection rate of MRs vary as the number of mutants increase? Out of the 709 mutants (609 MuJava + 100 Major) used in this experiment, only 105 (14.8%) mutants could be killed using the MRs. This is a significant decrease in the mutation killing rate compared to what is reported in Xie et al., where 19 out of the 24 (79%) mutants were killed by the same set of MRs [96]. We think that the reason for this significant decrease in the mutation

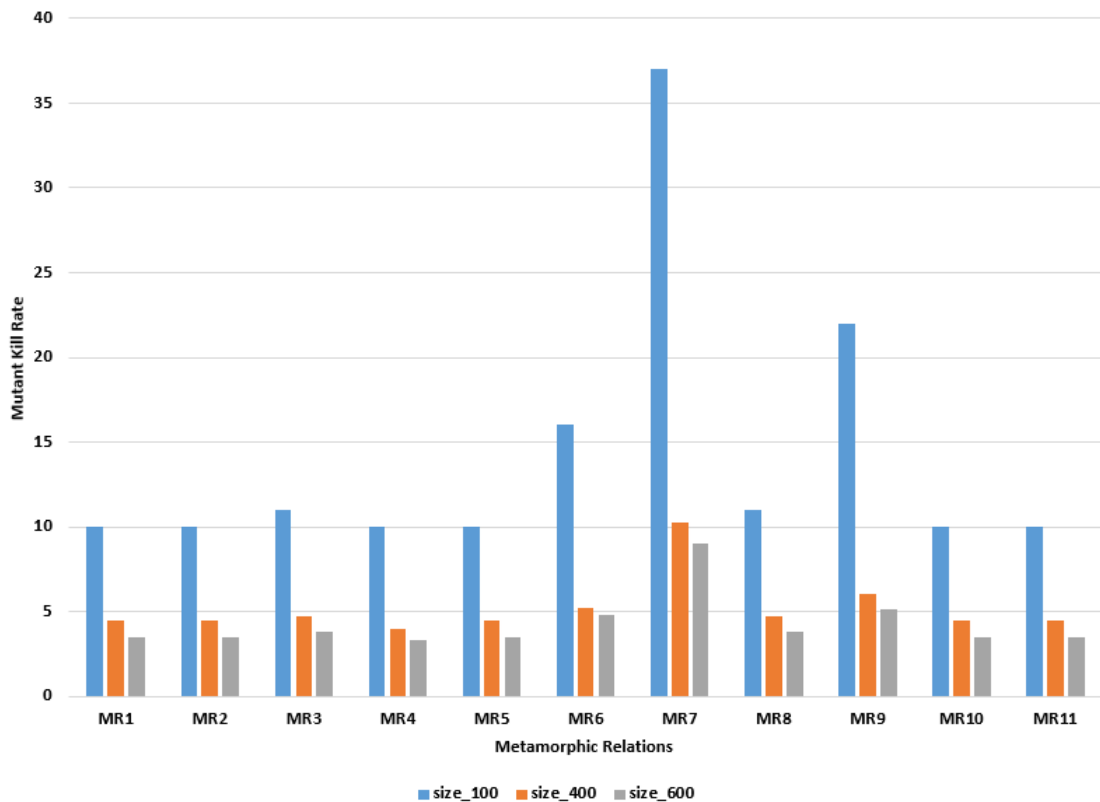


Figure 7.1: Mutant kill rate for each MR by varying mutants size.

killing rate is due to the fact that Xie et al. used a selected set of mutation operators to generate the mutants used in their experiment and those mutants do not provide a good representation of the potential faults in the SUT.

To further evaluate how the mutation killing rate varies when increasing the number of mutants, we executed the MRs with mutant sets of size 100, 400 and 600 that are randomly selected from the mutants generated by the MuJava tool. We used 10 randomly generated source test cases for executing each of the MRs. Figure 7.1 shows the mutant kill rates for each MR for the three mutant sets. As, shown in Figure 7.1, mutant kill rate for all the MRs reduced when the number of mutants were increased. In particular, the killing rates for sizes 400 and 600 are significantly lower compared to size 100 for MR6, MR7, and MR9.

2. Does the fault detection rate of MRs change with varying the data set size in the source test cases? The goal of this research question is to identify whether fault detection effectiveness of MRs vary with the size of the data sets used as the source test case. In this experiment, we created data sets of 18 different sizes where the number of samples vary from 30 to 200. These data sets were executed on 100 mutants that were randomly selected from the MuJava mutants.

Figure 7.2 shows the mutant killing rate for each MR with varying number of samples in the source test cases. It is interesting to note that mutants killing rate is low for all the MRs across the different data sets ranging between 10% and 37%. Only MR7 and MR9 is showing some variation in the killing rate which is 6% and 4%, respectively. On the other hand, the rest of MRs have a constant mutants killing rate despite the difference in data set sizes used as the source test cases. In summary, varying the size of the random sample data has no significant effect on the fault detection effectiveness of the considered MRs. But it might be possible to increase the fault detection effectiveness by generating test data based on some test coverage criterion as we discussed in our previous study [76].

3. Does the fault detection effectiveness vary with the mutation engine used to generate the mutants? In order to answer this research question, we used mutants from MuJava and Major mutation tools. We used 10 randomly generated sample data sets as source test cases for each MR and executed them on a set of 100 randomly selected mutants from MuJava and a set of 100 randomly selected mutants from Major. We report the results of this evaluation in Figure 7.3.

As shown in Figure 7.3, the overall mutant killing rate on the MuJava and Major mutants is 43.6% and 35.1%, respectively. When comparing the results at the individual MR level, it is noticeable that there is some consistency in the killing rate for each MR between these two tools. For example, for both tools, MR7 and

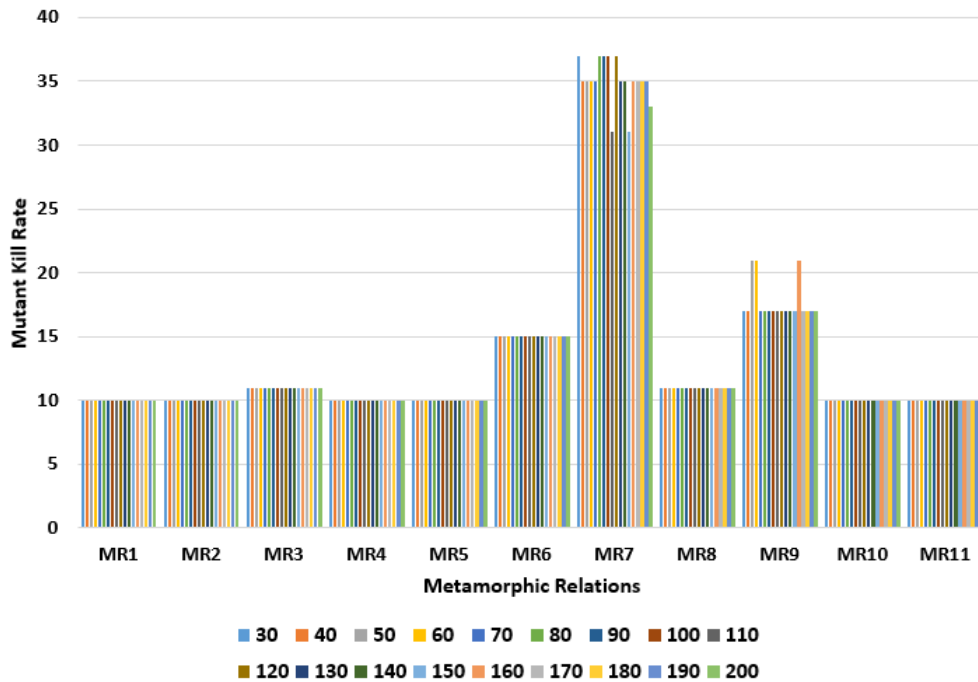


Figure 7.2: Mutant kill rate by each MR in kNN with varying data set.

MR9 have comparatively higher mutants killing rate than the other MRs. Also it is interesting to note that for all the MRs except MR7, killing rate of Major mutants is higher than that of the MuJava mutants even though the overall killing rate is higher for MuJava mutants. This indicates that the mutation killing is dominated by MR7. In summary, overall MuJava mutants are easier to kill, while with majority of MRs Major mutants are easier to kill.

7.9 Related Work

There has been a significant amount of work done on applying machine learning to solve software testing problems compared to testing machine learning applications [13]. For example, machine learning has been used to predict likely MRs for a given programs [40, 46, 49, 69].

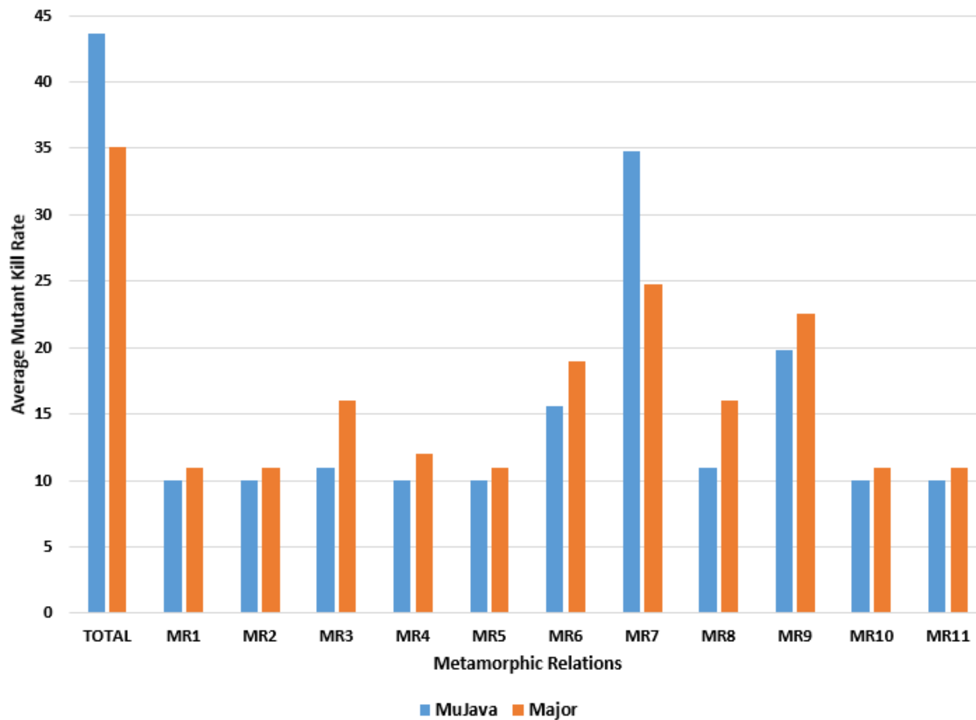


Figure 7.3: Average mutants kill rate by each MR for MuJava and Major tool.

MT has been applied to test different types of machine learning applications [58]. A case study done on real world machine learning application framework shows that MRs can effectively detect faults [96]. A recent work [35] investigated the application of metamorphic testing to test complex machine learning algorithms such as SVMs with non-linear kernels and deep residual neural networks (ResNET). The technique was able to successfully detect mutants in open-source machine learning applications.

MRs has been proven to be a core element of MT. In image processing applications MT was used by Tahir et al. [44]. They have shown that only few MRs that are related to specific images are more effective in detecting faults than others. Regardless of conducting MT, MRs have been used for the augmentation of the machine learning models [98]. Here MRs were identified based on properties of the input data and the usage of the binary classification model. Hui et al. [43] has

proposed a semi automated MT approach for GIS testing that used the superficial area calculation program to illustrate the process of the testing approach. They have developed a MR model to generate compound MRs.

Some research efforts are reported on how to identify effective MRs. Asrafi et al. [6] have observed a correlation between the test code coverage achieved by an MR and its fault detection effectiveness. In object oriented software testing a method of constructing MRs based on algebraic specification has been proposed [101]. This method provides low MRs redundancy and improves the efficiency of software testing. μMT [81] a MR construction tool that uses data mutation to construct an input relation and the generic mapping rule associated with each mutation operator to construct output relation.

7.10 Conclusions and Future Work

Previous studies have developed MRs for conducting MT on supervised classifiers. But, a major drawback of these studies is the limited number of mutants used to evaluate their fault detection effectiveness. In this paper, we empirically evaluated the fault detection effectiveness of MRs developed for supervised classifiers using a set of 709 reachable mutants, which is a significant increase in the number of mutants compared to what is used in the previous studies.

Our study shows that the MRs identified based on user expectations of supervised classifiers are not as effective in detecting faults as claimed in previous studies. Out of the 709 mutants only 14.8% of mutants could be detected using these MRs. Our study also shows that changing the size of randomly generated data used as source test cases does not have an effect on the fault detection effectiveness of these MRs.

In the future, we plan to develop MRs based on specific algorithmic properties

of commonly used supervised classifiers. We think such MRs will have higher fault detection effectiveness compared to the ones we investigated in this study. We also plan to investigate ways to develop more effective source test cases for this domain using various data distributions. Further, we plan to extend this experiment to other machine learning algorithms including deep learning algorithms.

CHAPTER EIGHT

MRSYN: A TEST CASE GENERATION AND MINIMIZATION APPROACH FOR TESTING MACHINE LEARNING APPLICATIONS

8.1 Contribution of Authors and Co-Authors

Manuscript in Chapter 8

Author: Prashanta Saha

Contributions: Problem identification and proposing solution, running experiment, manuscript writing, creating tables and figures. Primary writer

Co-Authors: Dr. Clemente Izurieta, Dr. Upulee Kanewala

Contributions: Contribution in manuscript editing/writing, provided feedback, guidance and advice

8.2 Manuscript Information

Prashanta Saha, Dr. Clemente Izurieta and Dr. Upulee Kanewala

IET Software

Status of Manuscript:

_____ Prepared for submission to a peer-reviewed journal

__x__ Officially submitted to a peer-reviewed journal

_____ Accepted by a peer-reviewed journal

_____ Published in a peer-reviewed journal

Wiley Publishers

8.3 Abstract

Machine learning (ML) algorithms, particularly supervised classifiers, are extensively employed in various domains for making predictions and decision-making tasks. However, ensuring the correctness and reliability of these systems is challenging due to the absence of formal proofs for their underlying algorithms. Software testing becomes imperative to validate the quality and accuracy of supervised classifier applications. Conventional testing techniques face limitations when testing supervised classifiers, primarily due to the absence of a reliable test oracle and the probabilistic nature of these models. Metamorphic Testing (MT) offers a promising approach to address these challenges by leveraging metamorphic relations (MRs) to test the probabilistic behavior of supervised classifiers. However, effective test case generation for MT in the context of supervised classifiers remains a challenge. This research proposes the MRSyn (**M**etamorphic **R**elation-based **Syn**thetic dataset generation) approach, a novel test case generation and minimization technique for MT. The MRSyn approach combines the test case generation technique using important properties of supervised classifier algorithms and test suite minimization techniques to generate a reduced set of source test cases with enhanced fault detection capability. Empirical experiments demonstrate the superiority of the MRSyn approach over random test case generation in terms of fault detection capability for supervised classifiers. The results also show a significant increase in mutation scores when MRSyn-generated test cases are augmented with applicable MRs. This research contributes to improving the reliability and accuracy of supervised classifiers by optimizing the testing process and enhancing the quality assurance process for these critical applications.

8.4 Introduction

Machine learning (ML) algorithms, specifically supervised classifiers, commonly employed in various domains for making predictions, play a crucial role in many real-world applications. For instance, over fifty real-world computational applications use support vector machines for classification [1]. However, ensuring the correctness and reliability of these systems is challenging due to the absence of formal proofs for the underlying algorithms. As a result, software testing becomes imperative to validate the quality and accuracy of supervised classifier applications.

Conventional testing techniques face limitations in regard to testing supervised classifiers. One major challenge is the absence of a reliable test oracle [88], which is responsible for determining the correctness of the test outputs. This characteristic makes supervised classifiers "non-testable programs" since their correctness cannot be easily determined. Additionally, supervised classifiers are not expected to be 100% accurate, meaning that incorrect predictions do not necessarily indicate faults in the program. These factors make it difficult to detect subtle faults in supervised classifier applications.

Metamorphic Testing (MT) offers a promising approach to address the challenges associated with testing supervised classifiers. MT leverages the idea of using metamorphic relations (MRs) to test the probabilistic behavior of these classifiers. MRs are properties that hold true over multiple executions of a program when specific inputs undergo predefined transformations. By applying MT, it becomes possible to identify faults in supervised classifiers by detecting discrepancies in the output predictions based on the expected behavior defined by the MRs.

In the context of MT, test case generation is a critical aspect. Traditional testing approaches, such as Random Testing [39], have been widely used for test case

generation. However, previous studies have indicated that randomly generated source test cases (train and test data) for testing supervised classifiers are not effective in detecting injected faults in the program [77]. This limitation calls for the development of more effective test case generation approaches specifically tailored to the verification of the source code of supervised classifiers.

The goal of this research is to devise an effective test case generation approach using MT for verifying the source code of supervised classifiers. By leveraging MRs and their ability to capture the expected behavior of supervised classifiers, the proposed approach aims to enhance the fault detection capability and improve the overall quality assurance process for these applications.

Synthetic datasets can be generated as a test suite to test supervised classifiers. *scikit-learn* uses critical properties of ML to generate synthetic datasets. But, not all of these properties are critical to the validation aspect of supervised classifiers. We have developed an effective approach for identifying and selecting the most critical properties that will generate meaningful source test cases. Through this research, it is anticipated that the developed test case generation approach will enable more thorough testing of supervised classifiers, leading to improved reliability and accuracy in their predictions. This, in turn, will enhance the trust and confidence in the performance of supervised classifiers across various domains and applications.

Testing a large test suite can indeed be time-consuming and resource-intensive, especially in the context of machine learning models where the test suite can consist of a large number of test cases. To address this challenge, a test suite minimization approach is employed, which aims to reduce the number of test cases while preserving the effectiveness of the test suite.

The objective of a test suite minimization approach is to optimize the testing process by identifying a subset of test cases that adequately represent the behavior

and characteristics of the larger test suite. The goal is to reduce redundancy and eliminate unnecessary test cases without compromising the ability to detect faults or accurately evaluate the performance of the machine learning model.

Test suite minimization techniques typically employ various strategies to achieve this goal. These strategies can include techniques such as prioritization, selection, and reduction. Prioritization techniques assign priorities to test cases based on certain criteria, such as code coverage or fault detection capability, and select the most important test cases for execution. Selection techniques involve selecting a subset of representative test cases that cover diverse scenarios or capture critical aspects of the system under test. Reduction techniques aim to eliminate redundant or overlapping test cases while preserving the overall test coverage. In this research, we are considering only the reduction strategy as our test suite minimization approach. In this research, we propose a novel approach called MRSyn for testing supervised classifier models. The MRSyn approach combines test case generation and minimization techniques within the framework of MT. The main objective is to generate a set of source test cases that not only cover diverse scenarios but also have a reduced number of test cases while maintaining their fault detection capability.

The MRSyn approach has the potential to significantly improve the efficiency and effectiveness of testing supervised classifier models. By generating a reduced set of test cases with high fault detection capability, it reduces the time and resources required for testing while maintaining the ability to detect faults. Additionally, the minimized test suite enhances the overall quality assurance process by focusing on the most relevant and effective test cases.

The rest of the paper is organised as follows: Section 8.5 introduces the background on subject programs, MT and mutation testing. Section 8.6 describes our research goals. Section 8.7 presents our MRSyn approach. Section 8.8 shows how

the experiments are set up. Section 8.9 summarises the empirical results and provides discussion. Section 8.10 lists threats to validity. Section 8.11 discusses the related work. And finally, Section 8.12 concludes our work.

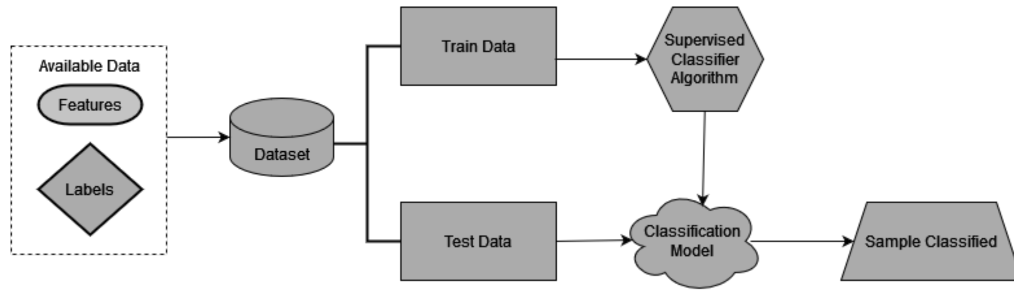


Figure 8.1: Supervised classifier application workflow.

8.5 Background

8.5.1 Supervised Machine Learning Classifiers

Supervised classification is the task of deducing a function from labeled training data such that it can be used to predict unknown labels on test data. Training data can be represented by two vectors of size k. One vector is the training samples $S =< s_0, s_1, ..., s_{k-1} >$ and the other one is the class labels $C =< c_0, c_1, .., c_{k-1} >$ where c_i is the class label for s_i. Each sample s_i has m features from which the prediction function will be learned. Class labels are a finite set and each class label c_i is an element of it, i.e., $c \in L =< l_0, l_1, ..., l_{n-1} >$, where n is the number of class labels [96].

Supervised machine learning applications execute in two phases: the *training phase* and *testing phase* (Figure 8.1). In the *training phase*, a set of training samples are used by a supervised classification algorithm to learn a prediction function. To develop the prediction function, the supervised learning algorithm would analyze how the attributes relate to the class label. In the *testing phase*, the prediction function

is applied to unseen data, known as the *test set*, where the class labels are unknown. The application attempts to predict the class label for each instance in the test set using the learned prediction function [96]. In this paper, some of the commonly used supervised classification algorithms such as K-Nearest Neighbor [27], Naive Bayes [73], Neural Networks [60] and Support Vector Machine [1] are studied as subject programs.

K-Nearest Neighbors (kNN), In kNN, for a sample training set S, each sample set has m attributes $< att_0, att_1, ..., att_{m-1} >$, and n classes,$< l_o, l_1, ..., l_{n-1} >$. The sample test data is t_s,$< a_0, a_1, ..., a_{m-1} >$. kNN computes the distance between each sample training set and the test case. Euclidean distance metric is one of the most popular approaches to measure distance. For sample $s_i \in S$, the value for each attribute is $< sa_0, sa_1, ..., sa_{m-1} >$. And the euclidean distance formula is [96]:

$$dist(s_i, t_s) = \sqrt{\sum_{j}^{m-1} (sa_j - a_j)^2}$$

Once the distance is calculated, kNN selects the k nearest training samples for the test data after sorting all distances. These k samples from the training set are considered as the *k-nearest neighbors* of the test case. Then, kNN calculates the proportion of each label in the selected k-nearest neighbors. The class label with the highest proportion is predicted as the label for the test data.

In **Naive Bayes Classifier (NBC)**, suppose for a sample training set S, each sample set has m attributes $< att_0, att_1, ..., att_{m-1} >$, and n classes,$< l_o, l_1, ..., l_{n-1} >$. The sample test data is t_s,$< a_0, a_1, ..., a_{m-1} >$. The label of t_s is called l_{ts} which is to be predicted by NBC. The equation of calculating the probability of l_{ts} to be of class l_k [96]:

$$P(l_{ts} = l_k | a_0 a_1 ... a_{m-1}) = \frac{p(l_k) \prod_j P(a_j | l_{ts} = l_k)}{\sum_i P(l_i) \prod_j P(a_j | l_{ts} = l_i)}$$

After computing the probability for each $l_i \in \{l_0, l_1, ..., ln-1\}$, NBC assigns the highest probability label l_k, as the label of the test case t_s.

In **Support Vector Machine (SVM)**, the basic idea is to maximize the distance between two classes. Let's consider a binary classification problem, with a sample set $z = \{x_i, y_i\}^m{}_{i=1}$, where $x_i \in \mathbb{R}^n$ and $y_i \in \{-1, 1\}$. Then z consists of two classes with the following sets, $I = \{i|y_i = 1\}$ and $II = \{i|y_i = -1\}$. Let, $\mathcal{H}$ be a hyperplane given by $w^T x + b = 0$ with $w \in \mathbb{R}^n, ||w|| = 1$, and $b \in \mathbb{R}$. So, I and II are separable by hyperplane $\mathcal{H}$ if for $i = 1, ..., m$, [1]

$$\begin{cases} w^T x_i + b > 0, & \forall i \in I, \\ w^T x_i + b < 0, & \forall i \in II. \end{cases}$$

In this case, $y_i(w^T x_i + b)$ gives the distance between point x_i and the hyperplane $\mathcal{H}$. Then the distance of each class to hyperplane $\mathcal{H}$ is defined as,

$$t_I(w, b) = min_{i \in I}\{y_i(w^T x_i + b)\}, t_{II}(w, b) = min_{i \in II}\{y_i(w^T x_i + b)\}.$$

The corresponding classification hyperplane is obtained by,

$$max_{||w||=1,b}\{t_I(w, b) + t_{II}(w, b)\},$$

which can be equivalently posed into the well-known SVM formulation.

In **Multilayer Perceptron (MLP)**, a perceptron is a basic unit that receives a set of input signals $\{x_i\}$ and emits a signal y calculated by applying an activation function σ to a weighted sum of its input; $y = \sigma(\sum_{i=1}^n w_i x_i)$. An NN is a multi-layered network of perceptrons. All these input signals $\{x_i\}$ are fed into perceptrons in the hidden layer, and then all the signals from perceptrons are input to the output layer.

Let G and H be two activation functions. Given $D-$dimensional vector x and M perceptrons consisting of the hidden layer, signals $\mathcal{K} = 1, .., \mathcal{R}$ from the output layer is mathematically defined as below, [60]

$$y_k(W; x) = H(\sum_{j=0}^{M} v_{kj} G(\sum_{i=0}^{D} w_{ji} x_i))$$

where v_{kj} and w_{ji} are weights, which are compactly written as W.

8.5.2 Metamorphic Testing for Supervised Classifiers

Often, programs exhibit properties such that if the test input is changed, then the new output can be predicted based on the original output. In MT, such properties (known as MRs) are used as partial oracles to conduct testing [18]. In practice, one can easily apply MT. As the first step, it is necessary to identify MRs that can relate multiple pairs of the inputs and outputs of the SUT. Then, source test cases are generated using techniques like random testing, structural testing, or search-based testing, and the corresponding follow-up test cases are constructed based on the MRs. In previous studies, we investigated how the fault detection effectiveness of MT varies with various source test case generation techniques such as different structural coverage-based approaches, and our results show that coverage-based source test case generation outperforms randomly generated source test cases [76]. After executing the source and follow-up test cases on the SUT we can check if there is a corresponding change in the output that matches the MR, if not, the MR is considered violated. Violation of an MR during testing indicates faults in the SUT. Since MT checks the relationship between the inputs and outputs of a test program, we can use this technique when the expected result of an individual test output is unknown.

For example, the *MR1: Consistence with affine transformation* MR described in

Table 8.1 can be used to test a kNN classifier. The source test case for a kNN can be randomly generated (see Table 8.2 for an example - training data on the left and test data on the right). After executing this source test case, the output will be the class label predicted for that test case, which is 0 for this example. To generate follow-up test cases, we apply the input transformation based on *MR1: Consistence with affine transformation* MR, where an arbitrary affine transformation function is applied to the attributes of both the training data and test data. After executing the follow-up test case, the output is 0, which is the predicted class label for the transformed test data. To satisfy this MR, both the source and follow-up test case outputs should be same. Therefore, in this example, the considered MR is satisfied for the given source and follow-up test cases.

Table 8.1: MRs for supervised classifiers. The significance of ✓is that particular MR supports the validation aspect of the supervised classifier. Validation aspect aim to check whether the algorithms under test meet the general user expectations or not.

Metamorphic Relations(MRs)	**kNN**	**NBC**	**SVM**	**MLP**	**Description**
MR1: Consistent with affine transformation	✓	✓	✓		If we apply some affine transformation function, $f(x) = kx + b, (k \neq 0)$, to every value x in some subset of features in the training and testing data, and then create a new model using this data, the predictions made by the model should be unchanged.

MR2: Permutation of class labels		✓	✓	✓	If we permute the order of the class labels with some random permutation $p(l_i)$ where l_i is a class label, all test cases which were predicted as l_i should now be predicted as $p(l_i)$.
MR3: Addition of informative attribute		✓	✓	✓	If we add some new feature that is strongly associated with one class, l_i, then for every prediction that was class l_i, the prediction with this new attribute should also be class l_i.
MR4: Addition of uninformative attributes	✓	✓		✓	If we add some new feature that is equally associated with all classes, the predictions of the test data should not be changed.
MR5: Permutation of the attribute	✓	✓	✓		If we permute the order of the attributes, or features, of all the samples in the training and testing data, the result of the predictions of the test data should not change.

MR6: Consistence with re-prediction	✓			✓	Suppose we predict some test case t as class l_i. If we append t to our training data and re-create the model, t should still be classified as class l_i.
MR7: Additional training sample	✓	✓	✓		Suppose we predict some test case t as class l_i. If we duplicate all samples of class l_i in our training data and re-classify our test data, t should still be classified as l_i. More generally, every test case predicted as class l_i should still be predicted as class l_i with the duplicated samples.
MR8: Addition of classes by re-labeling samples	✓				For some test cases not of class l_i, we switch the class label from x to x^*. Then every test case predicted as class l_i should still be predicted as class l_i with the re-labeled samples.

MR9: Addition of classes by duplicating samples		✓			Suppose we duplicate every class except for n, and give them all a new class. For example, if we originally had class labels of 1, 2, 3, and 4, then we would create class labels of 1, 1*, 2, 2*, 3, 4, 4*. Then every test case predicted as class l_i (class 3 in this example) should still be predicted as class l_i with the duplicated samples.
MR10: Removal of classes		✓			If we remove some class l_i, the remaining predictions should remain unchanged.
MR11: Removal of samples			✓		If we remove samples that have not been predicted as class l_i, then all cases which were predicted as l_i should remain unchanged.

8.5.3 Mutation Testing

Mutation testing [29] is a fault-based testing technique that measures the effectiveness of test cases associated with SUTs and has been used to evaluate the fault detection effectiveness of automated test case generation approaches. Briefly, the technique can be described as follows. Initially, mutants are generated by introducing faults into a program. This is achieved through the application of syntactic changes to the program's source code, resulting in new faulty versions. Each syntactic alteration

Table 8.2: Sample example of a dataset in ARFF format (Attribute-Relation File Format). This format has header and data information. The header contains attribute names and their data types (@attribute). The data contains the data set (@data).

Training Data	**Test Data**
@attribute pictures numeric	@attribute pictures numeric
@attribute paragraphs numeric	@attribute paragraphs numeric
@attribute files numeric	@attribute files numeric
@attribute files2 numeric	@attribute files2 numeric
@attribute profit{0,1,2,3,4}	@attribute profit{0,1,2,3,4}
@data	@data
45,3,16,38,0	6,40,8,89,0
15,87,89,46,4	
59,77,94,11,0	
86,89,94,15,2	
80,28,94,11,4	
23,12,47,41,1	
94,15,22,15,0	
95,26,97,76,3	
50,90,0,72,2	
33,46,47,95,0	

is dictated by a *mutation operator*. Subsequently, test cases are executed for both the original and faulty versions of the program, and their responses are compared. If the mutant produces a different response than the original, it is considered killed, signifying the test case's effectiveness in detecting faults. Conversely, if the mutant's response is identical to the original program, it is labeled as an *equivalent mutant*. Common scenarios for equivalent mutants include those that cannot be triggered, arise from dead code, only modify internal states, or solely enhance speed. The *mutation score*, calculated as the percentage of killed mutants among the total number of non-equivalent mutants, serves as a metric for the adequacy of the test suite.

8.6 Research Goals

ML is a type of artificial intelligence technique that makes decisions or predictions from data [56]. From the perspective of training data characteristics, ML includes Supervised learning, Unsupervised learning, and Reinforcement learning. Desired properties of ML programs can be classified into functional (i.e., correctness and model relevance) and non-functional (i.e., efficiency, robustness, fairness, interpretability) categories [100]. Though these properties are not strictly independent of each other, their external manifestation of behavior is different. In this research, the correctness property of ML system testing is targeted. Correctness measures the fundamental functional accuracy of an ML system.

Definition (Correctness.) Let U represent the future unlabeled data and let x be a data instance in U. Let m be the ML model that we want to test. $m(x)$ is the predicted label for x and $c(x)$ is the true label. The model correctness $E(m)$ is the probability that $m(x)$ and $c(x)$ are identical [100]:

$$E(m) = Pr_{x \sim U}[m(x) = c(x)]$$

There exist multitude of challenges in systematically validating the correctness of ML models. Consider an arbitrary ML model M that accepts an input I conforming to the grammar G and classifies I in one of the categories $\{C_1, C_2, C_3, ..., C_n\}$. Firstly, without precisely knowing the correct categorization of input I, it is not possible to validate the model M. In other words, the validation of ML models faces the oracle problem [88] in software testing. Secondly, there has been a significant effort in the software engineering research community to design test input generation strategies. The insight behind such directed strategies is to uncover bugs faster. However, to date, very little work has been done to assure the correctness of the software applications that perform machine learning. Formal proofs of an algorithm's optimal quality do not guarantee that an application implements or uses the algorithm correctly, and thus software testing is necessary. Since the oracle problem exists in ML model testing, MT has been proven to be an effective approach to test any ML system [95]. In MT, we require source and follow-up test cases to test any program. There are many open-source datasets available that can be used as source test cases to test supervised classifiers' applications, but their effectiveness in detecting faults is yet to be proven with MT techniques. Random test data generation [95] is another approach that has been used to generate source test cases for MT, however, and because of the probabilistic prediction nature of supervised classifiers, this approach does not effectively reveal faults.

Our research goal focuses on proposing two approaches for testing supervised ML models: test case generation and test suite minimization. Specifically:

- Test Case Generation Approach: In the context of supervised ML models, test case generation involves creating input data and corresponding expected outputs to evaluate the correctness and effectiveness of the models. An effective test case generation approach aims to generate a diverse and representative set of

test cases that cover various scenarios and edge cases. By proposing an effective test case generation approach, we aim to improve the quality and coverage of the test suite, leading to better evaluation and verification of supervised ML models.

- Test Suite Minimization Approach: Testing every single test case in a large test suite can be time-consuming and resource-intensive. Therefore, a test suite minimization approach aims to reduce the number of test cases while maintaining the effectiveness of the test suite. By proposing a test suite minimization approach, we aim to optimize the testing process by reducing the number of test cases without sacrificing the ability to detect faults or evaluate the ML model's performance.

Both the test case generation approach and the test suite minimization approach contribute to improving the efficiency and effectiveness of testing supervised ML models. These approaches can help ensure comprehensive coverage, identify potential issues or weaknesses in the models, and provide a cost-effective testing strategy for evaluating and verifying the correctness and performance of supervised ML models.

Table 8.3: Property parameters that help inform the generation of tuned source test cases. These parameters are used to generate datasets in Scikit-learn package.

Properties	**Description**
Samples	This parameter represents the total number of samples that will be generated for the data set.

Features	This parameter represents the total number of features that will be generated for the data set. These include informative features, redundant features, duplicate features, and useless features.
Informative Features	Each class is composed of a number of Gaussian clusters, each located around the vertices of a hypercube in a subspace of dimension. For each cluster, informative features are drawn independently from N(0, 1) and then randomly linearly combined within each cluster in order to add covariance. The clusters are then placed on the vertices of the hypercube.
Redundant Features	This feature is generated as random linear combinations of the informative features.
Repeated Features	This feature is drawn randomly from the informative and the redundant features.
Classes	This parameter represents the number of classes (or labels) of the classification problem.
n_clusters_per_class	This parameter denotes the number of clusters per class that will hold.
weights	This parameter represents the proportions of samples assigned to each class. Based on this property the balance of the classes will be decided.
Flip Sample	This floating point parameter value will decide the fraction of samples whose class is assigned randomly.

Class Separator	This floating point parameter represents the factor multiplying the hypercube size. Larger values spread out the clusters/classes and make the classification task easier.
Hypercube	This boolean value parameter has two outcomes. If True, the clusters are put on the vertices of a hypercube. If False, the clusters are put on the vertices of a random polytope.
Shift	Shift features by the specified value. If None, then features are shifted by a random value drawn in the class separator parameter.
Shuffle	This feature shuffles the samples and the features.

8.7 Proposed Approach

8.7.1 Source Test Case Generation

While real-world datasets are valuable for testing the overall performance and generalization of ML models, they do not always provide sufficient coverage of specific scenarios or functionalities that need to be tested. Generating controlled datasets with specific properties can offer additional control over the data and help evaluate the behavior of ML models under different conditions.

The idea of generating random values and placing them in a dataset according to specific parameters is a straightforward approach to creating controlled datasets. By defining the properties or characteristics we want to test, we can generate synthetic data that exhibits those properties. This allows us to have more control over the data and tailor it to specific testing scenarios.

In the context of ML, *scikit-learn* is a widely used Python library that provides various functionalities for ML tasks, including dataset generation. *scikit-learn* includes methods to generate datasets based on critical properties of ML algorithms. These properties are often related to statistical distributions, class imbalances, feature correlations, or other important characteristics of the data that can impact the performance of ML models.

By leveraging the functionality of *scikit-learn*, we can generate synthetic datasets that possess critical properties relevant to the ML algorithms being tested. This approach enhances the reliability of testing by providing more control over the data and allowing targeted evaluation of specific functionalities or scenarios.

Each step of the process is described as follows:

1. **Centroids and Class Separation:** To create clusters representing different classes, centroids are generated for each cluster. The number of centroids is determined by the number of classes in the dataset. By introducing some randomness, the centroids can be slightly varied within each cluster. The class separator parameter controls how closely the centroids are placed together, affecting the overlap between clusters.

2. **Gaussian Distribution and Informative Features:** The data is generated using a Gaussian distribution with a variance of 1. This ensures that the data follows a normal distribution. Additionally, informative features are incorporated into the dataset. These features contribute significantly to the separation and distinction between different classes.

3. **Splitting Data into Classes:** The generated data is then split into the desired number of classes. This is achieved by shifting portions of the data to each

cluster. The shift in data points helps create distinct clusters corresponding to different classes.

4. **Correlations among Informative Features:** Correlations among informative features are introduced by multiplying the feature matrix with a randomly generated covariance matrix. This step ensures that the informative features are not independent of each other, reflecting realistic scenarios where features may exhibit correlations.

5. **Redundant Features and Other Characteristics:** The function further enhances the dataset by adding redundant features. These features may have correlations among themselves but do not break the correlations among the informative features. Additionally, the dataset may include useless features, repeated features, and associated weights to simulate more complex and realistic datasets.

By utilizing the property parameters from Table 8.3, we can control the characteristics and properties of the synthetic datasets generated. These parameters provide flexibility in adjusting the cluster separation, overlap, feature correlations, and other relevant characteristics, allowing us to create diverse and controlled datasets for testing and evaluating ML algorithms. Overall, this approach enables the generation of synthetic datasets with specific properties, providing a means to systematically explore the behavior and performance of ML models under different conditions and scenarios.

8.7.2 Test Suite Minimization Approach

Test suite minimization techniques are commonly used to reduce the cost associated with executing test suites. However, it is essential to investigate whether a

minimized source test suite maintains similar fault detection capability as the original source test suite when applying Metamorphic Testing (MT). Our second research goal is to reduce the software testing cost by minimizing the test suite size. Our idea is to reduce the generated source test suite size based on their fault detection effectiveness. We can measure the fault detection effectiveness by using the mutation score as a metric. Additionally, instead of using a total mutant set, our plan is to use a popular mutant reduction strategy to reduce the mutant set size. A mutant reduction strategy technique will practically reduce the application cost of mutation testing, which will lead to a reduction in the total software testing cost.

Offutt et al. introduced the $N - \textit{selective mutation}$ theory to address the variability in the number of mutant programs generated by different mutant operators [61]. In their experimental approach, they categorized mutant operators into three groups based on the syntactic elements they affect. These categories include Replacement-of-operand operators, which involve replacing each operand in a program with other legal operands; Expression modification operators, focused on modifying expressions through the replacement and insertion of operators; and Statement modification operators, designed to alter entire statements. The findings of their experiments indicate that Expression modification operators, even when generating a smaller number of mutants compared to the total mutant set, can be effective. Furthermore, they observed a linear relationship between the execution time and the use of Expression modification operators.

With this reduced number of mutants, we will rank the mutation scores of the respective source test cases. After that, we will pick the higher-ranked source test cases based on a threshold level (decided by the developer/tester) for the metamorphic testing [75].

By reducing the number of test cases in a large test suite, a test suite

minimization approach offers several benefits. It can significantly reduce the execution time and resource requirements, enabling faster and more efficient testing processes. Moreover, it can facilitate the identification of critical faults or issues more quickly by focusing on the most relevant test cases. This, in turn, allows for timely bug fixes or performance improvements.

Table 8.4: Distribution of MRSyn datasets (# of test cases) and their characteristics

Property parameters	**# of Test Cases**	**Characteristics**
Informative Features	11	1 informative feature was used in 1 dataset, 2 informative features were used in 3 datasets and 3 informative features were used in 7 datasets
n_Clusters_per_class	27	1, 2 and 3 clusters per class parameters were individually applied in 9 datasets
Class Separator	27	1, 3 and 5 class separator parameters were individually applied in 9 datasets
Hypercube	20	Boolean values 'True' and 'False' of the Hypercube parameter were equally applied to 20 datasets

8.8 Experimental Setup

This section describes the experimental setup, three formally stated research questions, a description of the identified MRs for the subject programs, the approaches we took to generate source test cases, and the evaluation process of the experiment.

8.8.1 Research Questions

We identified three research questions that will help us test the effectiveness of the MRSyn approach over the random data generation approach.

RQ1: Is the MRSyn approach more effective for MT than a random approach in terms of fault detection capability?

The motivation for this research question is to determine if the MRSyn approach has better fault detection capabilities than randomly generated source test cases. To evaluate the fault detection capability of these approaches, we create a comprehensive set of faulty versions of our subject programs, herein referred to as mutants. We then apply both the MRSyn approach and the random approach to the mutants' set and analyze their performance in terms of fault detection capabilities.

RQ2: Does the use of MRs have any influence on enhancing the effectiveness of fault finding in MT?

The motivation behind this question stems from the potential impact of MRs in improving the effectiveness of fault detection in MT. We analyze the mutation score obtained from the application of MT with MRs to evaluate the impact of MRs on the fault detection effectiveness. We compare the mutation score obtained by using MRs to the mutation score obtained without using MRs (i.e., mutation testing with source test cases only) to assess if there is an increase in fault detection effectiveness.

RQ3: Can test suite minimization techniques reduce the cost of executing a test suite?

The motivation for this research question is to determine if a minimized source test suite has similar fault detection capabilities as the original source test suite while applying MT. We compare the fault detection capability of the original source test suite and the minimized test suite based on the mutation score. Additionally, we analyze the reduction in test suite size and execution time to evaluate the cost

reduction potential.

8.8.2 MRs Identification for Supervised Classifiers

Identifying MRs is an important step in MT of ML classifiers. Murphy et al. [57] suggested six MRs (additive, multiplicative, permutative, invertive, inclusive, and exclusive) that can be applied to machine learning applications including both supervised and unsupervised ML. Later, 11 MRs were identified by Xie et al. [96] (Described in Table 8.1). These MRs were derived from the specification of a particular algorithm under test, or from the users' general expectations for the classifiers. Each of the proposed MRs either targets the verification or the validation aspect of testing the supervised classifier algorithms under test. The MRs targeting verification aspect aims to check whether the algorithms under test adhere to the necessary characteristics (from the implementation perspective) expected from the algorithms, whereas the MRs targeting validation aspect aims to check whether the algorithms under test meet the general user expectations or not. We selected MRs for this research based on the validation aspect.

8.8.3 Generation of Source Test Cases

To conduct this experiment, we generated source test cases using two different approaches: random generation and the MRSyn approach. We discuss each approach and its process in detail.

Random Generation: For random generation, we used a random number generator tool to create 85 pairs of training and testing datasets. These sets were generated randomly, meaning the input data for the training and testing sets were chosen without any specific pattern or criteria.

MRSyn Approach: The MRSyn approach involves the generation of source test cases based on a set of property parameters. In this experiment, we randomly

selected four property parameters, as described in Table 8.3, to generate 85 datasets using the *scikit-learn* package in Python. The *scikit-learn* package provides various functionalities for ML, including dataset generation. Each dataset generated using the MRSyn approach is unbiased, meaning the sample sizes are equal for all class labels. This ensures that the generated datasets are balanced and representative of the underlying data distribution.

Table 8.2 represents a sample source test case, where the left side of the table displays the training dataset, and the right side displays the corresponding test dataset. These source test cases, generated through random generation and the MRSyn approach, serve as input data for the MT.

Table 8.4 provides the distribution of the MRSyn-generated source test cases. It lists the counts of test cases as well as the characteristics present in the generated datasets. This information helps in understanding the distribution and characteristics of the datasets created using the MRSyn approach.

8.8.4 Evaluation Approach

MT is a testing technique used to verify the behavior of programs when the input undergoes certain changes or transformations. It is based on the idea that if a program is correct, then certain relationships should hold between the outputs of the source test case and the follow-up test case, even if the inputs are transformed.

In the context of MT, mutation analysis can be used as an evaluation technique. After applying the *Major* mutation tool to generate mutants for the subject programs (Table 8.5), the source and follow-up test suite were executed on both the original program and its mutants. The purpose is to determine whether the mutant and the original program produce the same outputs for the given inputs. If a mutant produces different outputs compared to the original program, it is considered a "killed" mutant

Table 8.5: Details of the 4 supervised classifiers with their total lines of code and # of mutants generated. These classifiers are from Weka tool. The class and method names that we have used in this research are mentioned below. Total lines of code were calculated by excluding the comments in that method.

Subject Programs	**Method Names**	**Lines of Code**	**# of Mutants Used**
k Nearest Neighbors (KNN)	weka.classifiers.lazy.IBk. buildClassifier	28	125
Naive Bayes (NBC)	weka.classifiers.bayes. NaiveBayes.buildClassifier	53	104
Support Vector Machine (SVM)	weka.classifiers.functions.SMO. buildClassifier	68	136
Neural Network (MLP)	weka.classifiers.functions. MultilayerPerceptron.buildClassifier	196	130

since it indicates a fault in the mutated code. Conversely, if the mutant produces the same output as the original program, it is considered a "surviving" mutant.

The mutation score is then calculated as the ratio of the number of killed mutants to the total number of mutants. It provides a measure of how effective the test suite is at detecting faults introduced by the mutations. A higher mutation score indicates a more effective test suite, as it has detected a larger proportion of mutants.

Consider a test suite t composed of test cases, i.e., pairs of source and follow–up test cases. The Mutation Score (MS) of t is calculated as follows:

$$MS(t) = M_k / M_t$$

where M_k is the number of killed mutants by the test suite in t, and M_t is the total number of mutants.

By applying MT in combination with mutation analysis, we can evaluate the quality of the test suite in terms of its ability to detect faults introduced by the mutants generated by the *Major* mutation tool.

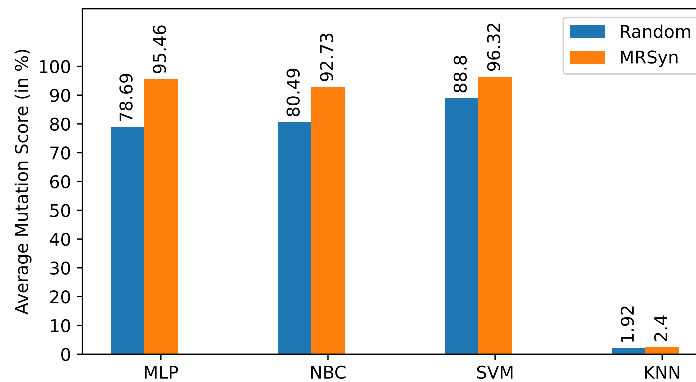


Figure 8.2: Comparison between mutation scores of Random and MRSyn based datasets for 4 supervised classifier algorithms. These two datasets were applied as source test suites for MT.

8.9 Results & Discussion

Below we discuss the results from our experiments and provide answers to the research questions:

RQ1: Is the MRSyn approach more effective for MT than a random approach in terms of fault detection capability?

The objective was to compare the mutation scores obtained using the MRSyn approach and the random approach for four supervised classifier algorithms: *MLP*, *Naive*, *SVM*, and *kNN*. A total of 85 random datasets and 85 MRSyn generated datasets were executed separately for this comparison.

Figure 8.2 presents the comparison of mutation scores between the MRSyn approach and the random approach for each of the four supervised classifier algorithms. The results show that in three of the algorithms, namely, *MLP*, *NBC*, *SVM*, the MRSyn approach outperformed the random approach in terms of mutation score which is statistically significant (p-value$<$0.05). This suggests that the test cases generated using the MRSyn approach were more effective in detecting faults than the randomly generated test cases.

However, an interesting observation was made for the *kNN* algorithm. In both the MRSyn approach and the random approach, the mutation scores were found to be relatively low. To investigate this further, we examined the code coverage achieved by both approaches.

Upon analyzing the code coverage, it was discovered that in the majority of cases, neither the MRSyn approach nor the random approach were able to cover the statements in the code where the mutants were located. This implies that the test cases generated by both approaches did not adequately exercise the parts of the code containing the mutants for the *kNN* algorithm. Consequently, the fault detection capability of both approaches was limited for this particular algorithm.

The low mutation scores for the *kNN* algorithm could be attributed to various factors, such as the specific characteristics of the algorithm, the nature of the mutants introduced, or the limitations of the test case generation process. It is crucial to further investigate and understand these factors to improve the fault detection effectiveness for the *kNN* algorithm in future studies.

Overall, while the MRSyn approach demonstrated better performance than the random approach in three out of the four supervised classifier algorithms, both approaches faced challenges in achieving adequate code coverage for the *kNN* algorithm. These findings highlight the importance of considering code coverage and

conducting further investigations to enhance the fault detection capability of both approaches, particularly in cases where specific algorithms or code segments pose difficulties for the test case generation process.

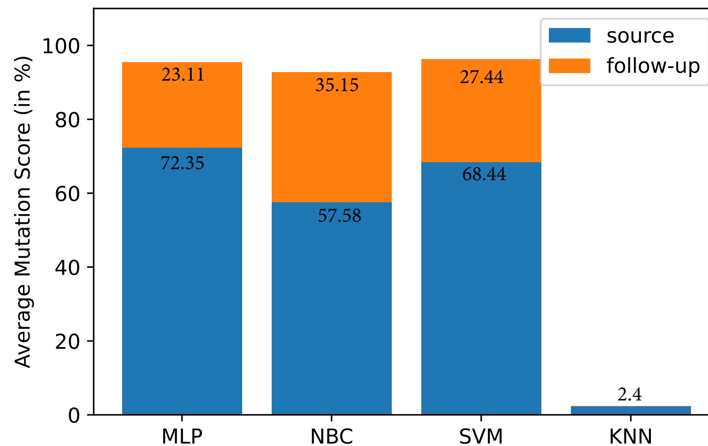
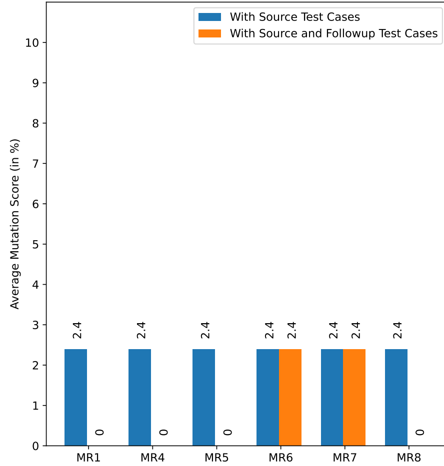


Figure 8.3: After applying the MRSyn approach to the source test suites, the mutation scores of MRs demonstrate an increase.

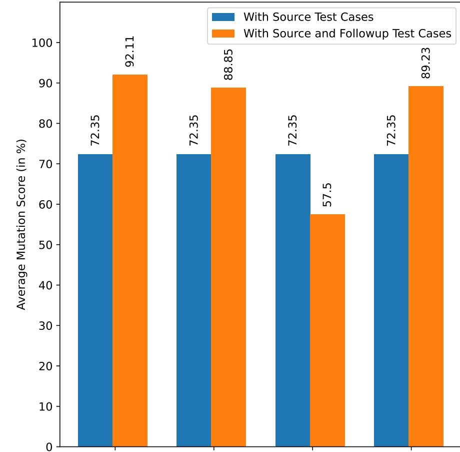
RQ2: Does the use of MRs have any influence on enhancing the effectiveness of fault finding in MT?

Figure 8.3 illustrates the mutation scores of automatically generated source test suites using the MRSyn approach. Additionally, the figure displays the increase in mutation scores achieved by augmenting MT with the applicable MRs as described in Table 8.1. Each column represents a supervised classifier algorithm considered as the subject program in the evaluation.

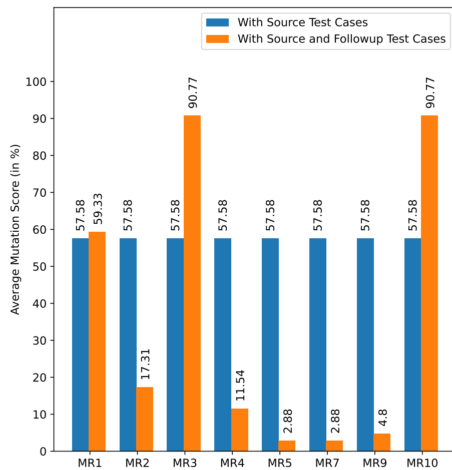
In the first three columns, a significant increase in mutation scores is observed when the source test suites are augmented with MRs. This indicates that the addition of MRs enhances the fault detection capability of MT for these algorithms. However,



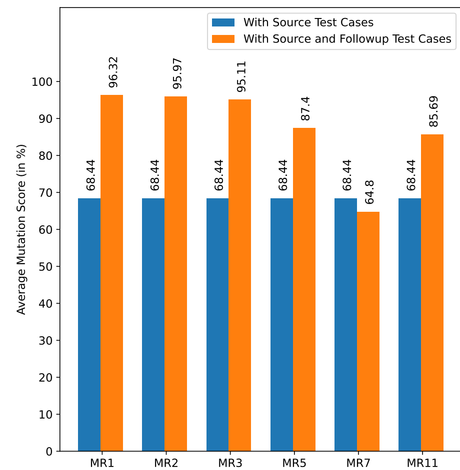
(a) kNN



(b) MLP



(c) NBC



(d) SVM

Figure 8.4: Individual mutation scores of MRs compared with source test suites for each subject program. These MRs were selected based on their validation aspects described in Table 8.1.

in the case of the *kNN* classifier algorithm, there is no observable increment in the mutation score compared to the other three algorithms. It is important to note that even the mutation score of the source test suites generated by the MRSyn approach is low for the *kNN* classifier algorithm.

These findings suggest that the fault detection effectiveness of MT is influenced by both the MRs used for testing and the quality of the source test cases employed to execute those MRs. The increase in mutation scores achieved by augmenting with MRs indicates that effective source test suites can enhance the effectiveness of MRs in detecting faults. However, the lack of improvement in the mutation score for the *kNN* classifier algorithm raises the need for further investigation to determine the exact reason for the poor performance in fault detection effectiveness.

Possible factors contributing to the observed reduction in fault detection for the *kNN* classifier algorithm could include the specific characteristics of the algorithm, the nature of the identified MRs, or limitations in the test case generation process. Detailed analysis and investigation to identify these factors and address them are necessary.

The findings depicted in Figure 8.4 provide insights into the individual fault finding capability of MRs when applied with the source test cases generated by the MRSyn approach. Specifically, the performance of MRs in detecting mutants is compared to that of the source test cases when applying MT on subject programs utilizing the *MLP*, *SVM*, *NBC*, and *kNN* classifiers.

For the *MLP* and *SVM* classifiers, a majority of the MRs demonstrate improved fault finding capabilities compared to the source test cases. When subjected to MT, these MRs display a higher mutant detection rate. Conversely, in the case of the *NBC* classifier, MR1, MR3, and MR10 yield similar results in mutant detection when compared to the source test cases. However, for the *kNN* classifier, all MRs except

MR6 and MR7 fail to detect any mutants.

These findings emphasize the significance of identifying effective MRs to enhance the fault finding capabilities of test suites. Moreover, generating high-quality source test cases also proves to be a crucial factor in the success of Metamorphic Testing. By meticulously selecting MRs and ensuring the generation of robust source test cases, the effectiveness of MT can be maximized, leading to improved software testing outcomes.

Overall, these observations shed light on the importance of both MR selection and source test case generation in Metamorphic Testing, underscoring their role in enhancing the effectiveness and fault-finding capabilities of the testing process.

RQ3: Can test suite minimization techniques reduce the cost of executing a test suite?

Table 8.6 presents the findings on the percentage reduction in test suite size and fault detection for the combined test suites and the reduced selected property parameter criteria separately. The results highlight the significant impact of test suite minimization in reducing the size of the test suite. In particular, the *n_clusters_per_class* property parameter criteria result in a substantial reduction of 68.24% in test suite size.

Furthermore, Table 8.6-B provides insights into the percentage reduction in fault detection effectiveness, specifically for the *n_clusters_per_class* property parameter. The minimized test suites generated based on the *n_clusters_per_class* property demonstrate a promising overall performance, with a fault detection effectiveness reduction of less than 6% compared to the combined test suites.

The results indicate that utilizing the *n_clusters_per_class* parameter from the Scikit-learn package as a source test case generation technique for MT can lead to test suites that exhibit comparable fault detection capability to larger test suites.

Table 8.6: Percentage reduction on test suite size and fault detection after applying test suite minimization technique in MT

A: Percentage reduction on test suite size

Subject Programs	Combined	After Reduction (Selected Property Parameters)	% Reduced
kNN	85	27 (n_clusters_per_class)	68.24
MLP	85	27 (n_clusters_per_class)	68.24
NBC	85	27 (n_clusters_per_class)	68.24
SVM	85	27 (n_clusters_per_class)	68.24

B: Percentage reduction on fault detection

Subject Programs	Combined	After Reduction	% Reduced
kNN	2.4	2.4	0
MLP	95.46	90.34	5.36
NBC	92.73	92.31	0.45
SVM	96.32	93.85	2.56

This significant reduction in test suite size translates to substantial savings in test execution costs.

It is worth noting that the average reduction in fault detection effectiveness of less than 6% implies that the approach of test suite minimization based on the *n_clusters_per_class* property parameter can be practically useful in scenarios where the testing time is limited and the system under test is not critical. The approach allows for efficient testing while still maintaining an acceptable level of fault detection

effectiveness.

Overall, the findings highlight the potential benefits of utilizing the *n_clusters_per_class* property parameter for test suite minimization in MT, enabling significant reductions in test suite size and associated costs, without compromising the overall fault detection effectiveness within practical contexts.

8.10 Threats to Validity

In our empirical study, we have taken measures to address potential threats to the validity of our findings, following the guidelines proposed by Wohlin et al. [90].

To mitigate threats to *internal validity*, which involve the causal relationship between factors in the study, we ensured the reliability of our experimental setup. We conducted our experiments 10 times using the same configuration, which increases our confidence in the obtained results and minimizes the impact of random variations.

Threats to *construct validity* were considered, which pertain to the measurement and operationalization of variables. We utilized third-party tools, such as the Scikit-learn package in Python for generating source test cases and the Major mutation tool for creating mutants. To minimize potential threats arising from these tools, we conducted manual inspections of randomly selected tool outputs. By verifying the correctness of the results, we increase confidence in the validity of the tools' outputs and their impact on our study.

In terms of *external validity*, which concerns the generalizability of the study's findings, we employed four different supervised classifier algorithms from the Weka tool. This choice ensures a diverse range of classifiers and enhances the likelihood that our results can be generalized to other open-source software projects. However, it is important to note that we specifically used the Scikit-learn package for generating test cases in our main experiment.

By considering these threats and taking appropriate measures to mitigate them, we have striven to ensure the internal, construct, and external validity of our empirical study. This increases the reliability and generalizability of our findings, enhancing the overall quality and trustworthiness of our research.

8.11 Related Works

Prior research in the field of metamorphic testing (MT) applied to machine learning (ML) applications has predominantly focused on generating modified test cases to evaluate the quality and performance of ML models. These studies have explored various modifications to the dataset, such as repeated values, missing values, categorical data, and synthetic data with known distributions.

For instance, Murphy et al. conducted experiments to test two ML ranking applications using generated data with controlled properties and randomness. Their data generation tool proved to be a reliable and simplified alternative to real-world data for testing purposes [57].

Breck et al. employed synthetic training data adhering to schema constraints to expose hidden assumptions in ML code that do not align with those constraints. They identified data bugs during the deployment of the system on TFX, an end-to-end ML platform at Google. They also investigated data skewness in both training and new data [12].

Zhang et al. focused on testing overfitting in ML models by using synthetic data with known distributions. They proposed a technique called Perturbed Model Validation (PMV), which involved injecting noise into the training data. PMV combined the principles of metamorphic relationships and data mutation to detect overfitting in ML models [99].

However, none of these studies have specifically addressed the verification of

software applications implementing supervised classifiers. The objective of our research is to verify the correctness of software applications for supervised classifiers using MT as a testing technique. By applying MT to the software applications of supervised classifiers, we aim to uncover potential faults, assess the reliability of the implementations, and validate the expected behavior of these applications. This targeted approach distinguishes our work from previous studies that have primarily focused on evaluating ML models and their data inputs.

8.12 Conclusions

This research paper proposes the MRSyn approach, a novel test case generation and minimization technique for testing supervised classifier models using MT. The motivation for this research stems from the challenges associated with testing supervised classifiers, such as the absence of a reliable test oracle and the probabilistic nature of these models.

The MRSyn approach leverages the important properties of supervised ML alogrithms to guide the test case generation process. MRs define the expected relationships between inputs and outputs when specific transformations are applied. By systematically applying these transformations to the source test cases, a set of related test cases is generated. The approach also incorporates test suite minimization techniques to reduce the number of test cases while maintaining their fault detection capability.

Through empirical experiments and analysis, the effectiveness of the MRSyn approach is evaluated. The results demonstrate that the MRSyn approach outperforms random test case generation in terms of fault detection capability for three out of four supervised classifier algorithms tested. Additionally, the MRSyn-generated test cases, when augmented with applicable MRs, show a significant increase in mutation

scores, indicating improved fault detection effectiveness.

The findings highlight the importance of effective test case generation in the context of supervised classifier models. By utilizing MRs and applying test suite minimization techniques, the MRSyn approach offers a more efficient and effective testing process. It reduces the number of test cases while maintaining the fault detection capability, thereby optimizing the testing process and improving the reliability and accuracy of supervised classifier models.

Overall, this research contributes to the field of software testing by addressing the challenges specific to testing supervised classifier models. The MRSyn approach provides a valuable methodology for generating and minimizing test cases, enabling more thorough testing, and enhancing the quality assurance process for these critical applications.

CHAPTER NINE

CONCLUSION & FUTURE WORK

9.1 Conclusion

This thesis has contributed significant enhancements to the application of coverage-based test case generation for testing numerical programs and MR property-based test case generation for testing supervised classifier algorithms with complex input types. In detail, the contributions are:

- Demonstrating the effectiveness of the line, branch, and weak mutation coverage-based test case generation approaches on 4 open source code repositories using MT
- Investigating the importance of MRs' identification for MT on *LA4J* Java classes
- Investigating a mutant reduction strategy that is applied to increase the testing efficiency of source test cases, and a test suite minimization technique to help reduce the testing costs without trading off fault-finding effectiveness
- Investigating the fault detection effectiveness of MRs while testing the applications of supervised classifiers using MT
- Developing a test case generation and minimization technique for MT to test supervised classifier algorithms

Chapter 4 demonstrated the effectiveness of systematic source test case generation in enhancing the fault detection capabilities of metamorphic testing, a testing strategy addressing the oracle problem in software testing. Through evaluating line

coverage, branch coverage, weak mutation, and random test generation strategies across 77 methods from four open-source code repositories, the study demonstrated that a systematic approach to generating source test cases significantly improves the effectiveness of metamorphic testing. Additionally, we introduced "METtester" tool to facilitate the application of these findings by conducting metamorphic testing on the specified methods.

Chapter 5 investigated the challenge of developing effective test oracles for automatically generated test cases, focusing on the utilization of metamorphic testing (MT), a technique known for its potential to mitigate the oracle problem through metamorphic relations (MRs). An empirical study conducted on an open-source linear algebra library *LA4J*, assesses the impact of MRs on enhancing the fault detection capabilities of these test cases. The findings indicate that employing MRs can indeed significantly improve the fault detection effectiveness of automatically generated test cases.

Chapter 6 proposed a mutant reduction strategy to enhance the efficiency of source test cases and a test suite minimization technique aimed at lowering testing costs while maintaining fault detection capabilities. An empirical study validates these methods, showing that they improve both the efficiency and effectiveness of source test cases in fault detection.

Chapter 7 investigated the effectiveness of MT in identifying faults in supervised classifiers, a common challenge due to the oracle problem in machine learning applications. Through an empirical study involving 709 mutants generated from different mutation engines and various datasets, the research assessed the fault detection capability of MRs previously used in testing supervised classifiers. The findings indicate that the MRs tested only detected 14.8% of the mutants, suggesting that the effectiveness of these MRs does not improve proportionally with an increase

in the number of mutants, challenging previous studies' results.

Chapter 8 introduced the MRSyn approach, a novel method for generating and minimizing test cases for MT to test supervised classifiers, addressing the challenges of testing these probabilistic models without a reliable test oracle. By leveraging important properties of supervised classifiers, MRSyn efficiently creates a condensed set of source test cases that exhibit an improved ability to detect faults. Empirical experiments have shown that test cases generated by MRSyn significantly outperform those created through random generation methods in identifying faults within supervised classifiers. The results underscore the potential of MRSyn to enhance the reliability and accuracy of supervised classifiers by refining the test case generation process and thereby advancing the quality assurance of these essential machine learning applications.

To summarize, this thesis introduces two separate source test case generation approaches for MT to test numerical programs and supervised classifier algorithm-based software applications.

9.2 Future Work

This dissertation has contributed to develop two novel source test case generation techniques for MT. The contributions of this work pave the way for more in-depth studies and technological advancements in these areas. Future work could focus on the following key aspects:

- **Exploration of Additional Coverage Criteria:** While this thesis has demonstrated the effectiveness of line, branch, and weak mutation coverage in generating test cases, future work could explore additional coverage criteria such as path, condition, and decision coverage. Investigating these criteria

could uncover new insights into optimizing test case generation for numerical programs.

- **Expansion of Test Suite Minimization Techniques:** The proposed test suite minimization technique has shown promise in reducing testing costs without compromising fault detection effectiveness. Future research could expand on this work by developing more sophisticated minimization algorithms that further optimize the trade-off between cost and effectiveness, potentially incorporating machine learning models to predict the impact of different minimization strategies.

- **Broader Application and Evaluation of the MRSyn Approach:** The MRSyn approach has demonstrated significant potential in generating effective test cases for supervised classifiers. Future work could apply MRSyn to a wider range of machine learning models and applications specifically unsupervised classifier algorithms, evaluating its effectiveness across different contexts and exploring modifications to the approach that could enhance its applicability and performance.

- **Empirical Studies in Industry Contexts:** Conducting empirical studies in industry contexts could provide valuable insights into the practical challenges and benefits of applying the techniques developed in this thesis. Collaborating with industry partners to test real-world applications could help to validate the effectiveness of these approaches in diverse and complex software environments, driving further improvements and adaptations.

In conclusion, this thesis lays the groundwork for a range of future research opportunities aimed at advancing the state of the art in software testing for numerical programs and supervised classifier algorithms.

REFERENCES CITED

[1] Svm application list. `http://www.clopinet.com/isabelle/Projects/SVM/\applist.html`.

[2] E. Alatawi, T. Miller, and H. Søndergaard. Generating source inputs for metamorphic testing using dynamic symbolic execution. In *2016 IEEE/ACM 1st International Workshop on Metamorphic Testing (MET)*, pages 19–25, May 2016.

[3] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, J. Jenny Li, and H. Zhu. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978 – 2001, 2013.

[4] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? [software testing]. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 402–411, May 2005.

[5] A. Arcuri. It does matter how you normalise the branch distance in search based software testing. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 205–214, April 2010.

[6] M. Asrafi, H. Liu, and F. Kuo. On testing effectiveness of metamorphic relations: A case study. In *2011 Fifth International Conference on Secure Software Integration and Reliability Improvement*, pages 147–156, June 2011.

[7] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015.

[8] A. C. Barus, T. Y. Chen, F. C. Kuo, H. Liu, and H. W. Schmidt. The impact of source test case selection on the effectiveness of metamorphic testing. In *2016 IEEE/ACM 1st International Workshop on Metamorphic Testing (MET)*, pages 5–11, May 2016.

[9] G. Batra and J. Sengupta. An efficient metamorphic testing technique using genetic algorithm. In S. Dua, S. Sahni, and D. P. Goyal, editors, *Information Intelligence, Systems, Technology and Management*, pages 180–188, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[10] B. Beizer. *Software Testing Techniques (2nd Ed.)*. Van Nostrand Reinhold Co., USA, 1990.

[11] P. Braione, G. Denaro, and M. Pezzè. Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 411–421, New York, NY, USA, 2013. Association for Computing Machinery.

[12] E. Breck, M. Zinkevich, N. Polyzotis, S. Whang, and S. Roy. Data validation for machine learning. In *Proceedings of SysML*, 2019.

[13] L. C. Briand. Novel applications of machine learning in software testing. In *2008 The Eighth International Conference on Quality Software*, pages 3–10, Aug 2008.

[14] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, Mar 1982.

[15] B. Butler, M. Cox, A. Forbes, S. Hannaby, and P. Harris. *A methodology for testing classes of approximation and optimization software*, pages 138–151. Springer US, Boston, MA, 1997.

[16] L. Chen, L. Cai, J. Liu, Z. Liu, S. Wei, and P. Liu. An optimized method for generating cases of metamorphic testing. In *2012 6th International Conference on New Trends in Information Science, Service Science and Data Mining (ISSDM2012)*, pages 439–443, Oct 2012.

[17] T. Y. Chen. Metamorphic testing: A simple method for alleviating the test oracle problem. In *Proceedings of the 10th International Workshop on Automation of Software Test*, AST '15, pages 53–54, Piscataway, NJ, USA, 2015. IEEE Press.

[18] T. Y. Chen. Metamorphic testing: A simple method for alleviating the test oracle problem. In *2015 IEEE/ACM 10th International Workshop on Automation of Software Test*, pages 53–54, 2015.

[19] T. Y. Chen, S. C. Cheung, and S. W. Yiu. Metamorphic testing: a new approach for generating next test cases. 1998.

[20] T. Y. Chen, F. Kuo, Y. Liu, and A. Tang. Metamorphic testing and testing with special values. In *4th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004), 15-16 September 2004, Chicago, IL, USA*, pages 128–134, 2004.

[21] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Comput. Surv.*, 51(1):4:1–4:27, Jan. 2018.

[22] T. Y. Chen, F. C. Kuo, D. Towey, and Z. Q. Zhou. Metamorphic testing: Applications and integration with other methods: Tutorial synopsis. In *2012 12th International Conference on Quality Software*, pages 285–288, Aug 2012.

[23] T. Y. Chen, P.-L. Poon, and X. Xie. Metric: Metamorphic relation identification based on the category-choice framework. *Journal of Systems and Software*, 116:177–190, 2016.

[24] T. Y. Chen, T. H. Tse, and Z. Zhou. Fault-based testing without the need of oracles. *Information & Software Technology*, 45:1–9, 2003.

[25] J. Clarke, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *IEE Proceedings - Software*, 150(3):161–175, 2003.

[26] W. J. Cody. *Software Manual for the Elementary Functions (Prentice-Hall Series in Computational Mathematics)*. Prentice-Hall, Inc., USA, 1980.

[27] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[28] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Trans. Inf. Theor.*, 13(1):21–27, Sept. 2006.

[29] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, Apr. 1978.

[30] J. Ding, X. Kang, and X.-H. Hu. Validating a deep learning framework by metamorphic testing. In *Proceedings of the 2Nd International Workshop on Metamorphic Testing*, MET '17, pages 28–34, Piscataway, NJ, USA, 2017. IEEE Press.

[31] J. Ding, T. Wu, J. Q. Lu, and X.-H. Hu. Self-checked metamorphic testing of an image processing program. In *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*, pages 190–197, 2010.

[32] G. Dong, T. Guo, and P. Zhang. Security assurance with program path analysis and metamorphic testing. In *2013 IEEE 4th International Conference on Software Engineering and Service Science*, pages 193–197, May 2013.

[33] J. W. Duran and S. Ntafos. A report on random testing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, page 179–183. IEEE Press, 1981.

[34] J. W. Duran and J. J. Wiorkowski. Quantifying software validity by sampling. *IEEE Transactions on Reliability*, R-29(2):141–144, 1980.

[35] A. Dwarakanath, M. Ahuja, S. Sikand, R. M. Rao, R. P. J. C. Bose, N. Dubash, and S. Podder. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 118–128, New York, NY, USA, 2018. ACM.

[36] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM.

[37] A. Gosain and G. Sharma. A survey of dynamic program analysis techniques and tools. In S. C. Satapathy, B. N. Biswal, S. K. Udgata, and J. Mandal, editors, *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*, pages 113–122, Cham, 2015. Springer International Publishing.

[38] A. Gotlieb and B. Botella. Automated metamorphic testing. In *Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003*, pages 34–40, Nov 2003.

[39] R. Hamlet. *Random Testing*. John Wiley & Sons, Ltd, 2002.

[40] B. Hardin and U. Kanewala. Using semi-supervised learning for predicting metamorphic relations. In *Proceedings of the 3rd International Workshop on Metamorphic Testing*, MET '18, pages 14–17, New York, NY, USA, 2018. ACM.

[41] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, 45(1), Dec. 2012.

[42] N. Higham. Numerical computation: Methods, software, and analysis. *IEEE Computational Science and Engineering*, 5:79–79, 1998.

[43] Z. Hui and S. Huang. Experience report: How do metamorphic relations perform in geographic information systems testing. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 598–599, June 2016.

[44] T. Jameel, M. Lin, and L. Chao. Test oracles based on metamorphic relations for image processing applications. In *2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 1–6, June 2015.

[45] R. Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 433–436, New York, NY, USA, 2014. ACM.

[46] U. Kanewala and J. Bieman. Using machine learning techniques to detect metamorphic relations for programs without test oracles. In *In Proc. 24th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 1–10, Pasadena, California, USA, Nov. 2013.

[47] U. Kanewala and J. M. Bieman. Using machine learning techniques to detect metamorphic relations for programs without test oracles. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 1–10, Nov 2013.

[48] U. Kanewala, J. M. Bieman, and A. Ben-Hur. Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels. *Software Testing, Verification and Reliability*, 26(3):245–269, 2016. stvr.1594.

[49] U. Kanewala, J. M. Bieman, and A. Ben-Hur. Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels. *Softw. Test., Verif. Reliab.*, 26(3):245–269, 2016.

[50] B. Korel. Automated software test data generation. *IEEE Trans. Softw. Eng.*, 16(8):870–879, Aug. 1990.

[51] H. Liu, X. Xie, J. Yang, Y. Lu, and T. Y. Chen. Adaptive random testing by exclusion through test profile. In *2010 10th International Conference on Quality Software*, pages 92–101, July 2010.

[52] Y.-S. Ma, J. Offutt, and Y. R. Kwon. Mujava: An automated class mutation system: Research articles. *Softw. Test. Verif. Reliab.*, 15(2):97–133, June 2005.

[53] J. Mayer and R. Guderlei. An empirical study on the selection of good metamorphic relations. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, volume 1, pages 475–484, Sep. 2006.

[54] P. McMinn. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004.

[55] R. S. Michalski, J. G. Carbonell, and T. M. Mitchell. *Machine Learning: An Artificial Intelligence Approach*. Springer Publishing Company, Incorporated, 2013.

[56] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012.

[57] C. Murphy, G. Kaiser, and M. Arias. Parameterizing random test data according to equivalence classes. In *Proceedings of the 2nd International Workshop on Random Testing: Co-Located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, RT '07, page 38–41, New York, NY, USA, 2007. Association for Computing Machinery.

[58] C. Murphy, G. E. Kaiser, L. Hu, and L. Wu. Properties of machine learning applications for use in metamorphic testing. In *SEKE*, 2008.

[59] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. Wiley Publishing, 3rd edition, 2011.

[60] S. Nakajima. Generalized oracle for testing machine learning computer programs. In A. Cerone and M. Roveri, editors, *Software Engineering and Formal Methods*, pages 174–179, Cham, 2018. Springer International Publishing.

[61] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, Apr. 1996.

[62] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007. ACM.

[63] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*, pages 75–84, 2007.

[64] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman. Chapter six - mutation testing advances: An analysis and survey. volume 112 of *Advances in Computers*, pages 275–378. Elsevier, 2019.

[65] M. Papadakis and N. Malevris. An empirical evaluation of the first and second order mutation testing strategies. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 90–99, 2010.

[66] Peng Wu. Iterative metamorphic testing. In *29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, volume 1, pages 19–24, July 2005.

[67] J. C. T. Pool. Is numerical software relevant? is it too late to worry about quality? In *Proceedings of the IFIP TC2/WG2.5 Working Conference on Quality of Numerical Software: Assessment and Enhancement*, page 3–11, GBR, 1997. Chapman & Hall, Ltd.

[68] P. S. (ps073006) and U. Kanewala. Msu-stlab/mettester 1.0.0, Jan. 2018.

[69] K. Rahman and U. Kanewala. Predicting metamorphic relations for matrix calculation programs. In *Proceedings of the 3rd International Workshop on Metamorphic Testing*, MET '18, pages 10–13, New York, NY, USA, 2018. ACM.

[70] F. u. Rehman and C. Izurieta. Statistical metamorphic testing of neural network based intrusion detection systems. In *2021 IEEE International Conference on Cyber Security and Resilience (CSR)*, pages 20–26, 2021.

[71] F. U. Rehman and C. Izurieta. An approach for verifying and validating clustering based anomaly detection systems using metamorphic testing. In *2022 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 12–18, 2022.

[72] J. R. Rice. *Numerical Methods, Software and Analysis.* McGraw-Hill, Inc., USA, 1983.

[73] I. Rish. An empirical study of the naive bayes classifier. Technical report, 2001.

[74] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri. *Combining Multiple Coverage Criteria in Search-Based Unit Test Generation*, pages 93–108. Springer International Publishing, Cham, 2015.

[75] P. Saha, C. Izurieta, and U. Kanewala. A test suite minimization technique for testing numerical programs. In *2023 IEEE/ACIS 21st International Conference on Software Engineering Research, Management and Applications (SERA)*, pages 61–68, 2023.

[76] P. Saha and U. Kanewala. Fault detection effectiveness of source test case generation strategies for metamorphic testing. In *Proceedings of the 3rd International Workshop on Metamorphic Testing*, MET '18, pages 2–9, New York, NY, USA, 2018. ACM.

[77] P. Saha and U. Kanewala. Fault detection effectiveness of metamorphic relations developed for testing supervised classifiers. In *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 157–164, 2019.

[78] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824, 2016.

[79] S. Segura, R. M. Hierons, D. Benavides, and A. Ruiz-Cortés. Automated metamorphic testing on the analyses of feature models. *Information and Software Technology*, 53(3):245 – 258, 2011.

[80] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In T. Ball and R. B. Jones, editors, *CAV*, pages 419–423, 2006.

[81] C. Sun, Y. Liu, Z. Wang, and W. K. Chan. μmt: A data mutation directed metamorphic relation acquisition methodology. In *2016 IEEE/ACM*

1st International Workshop on Metamorphic Testing (MET), pages 12–18, May 2016.

[82] C. Sun, G. Wang, B. Mu, H. Liu, Z. Wang, and T. Y. Chen. Metamorphic testing for web services: Framework and a case study. In *2011 IEEE International Conference on Web Services*, pages 283–290, July 2011.

[83] C.-A. Sun, A. Fu, P.-L. Poon, X. Xie, H. Liu, and T. Y. Chen. Metric^{+}+: A metamorphic relation identification technique based on input plus output domains. *IEEE Transactions on Software Engineering*, 47(9):1764–1785, 2021.

[84] N. Tillmann and J. de Halleux. Pex–white box test generation for .net. In B. Beckert and R. Hähnle, editors, *Tests and Proofs*, pages 134–153, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[85] F. Ur Rehman and C. Izurieta. Mt4uml: Metamorphic testing for unsupervised machine learning. In *2022 9th Swiss Conference on Data Science (SDS)*, pages 26–32, 2022.

[86] E. Weyuker. On testing non-testable programs. 25, 11 1982.

[87] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[88] E. J. Weyuker. On Testing Non-Testable Programs. *The Computer Journal*, 25(4):465–470, 11 1982.

[89] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, San Francisco, 2 edition, 2005.

[90] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*. The Kluwer International Series In Software Engineering. Springer, Germany, 2000.

[91] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

[92] W. Wong, J. Horgan, A. Mathur, and A. Pasquini. Test set size minimization and fault detection effectiveness: a case study in a space application. In *Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)*, pages 522–528, 1997.

[93] W. E. Wong. *On Mutation and Data Flow*. PhD thesis, USA, 1993. UMI Order No. GAX94-20921.

[94] P. Wu, S. Xiao-Chun, T. Jiang-Jun, and L. Hui-Min. Metamorphic testing and special case testing: A case study. 16, 07 2005.

[95] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software*, 84(4):544–558, 2011. The Ninth International Conference on Quality Software.

[96] X. Xie, J. W. K. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen. Testing and validating machine learning classifiers by metamorphic testing. *J. Syst. Softw.*, 84(4):544–558, Apr. 2011.

[97] X. Xie, W. E. Wong, T. Y. Chen, and B. Xu. Spectrum-based fault localization: Testing oracles are no longer mandatory. In *2011 11th International Conference on Quality Software*, pages 1–10, July 2011.

[98] L. Xu, D. Towey, A. P. French, S. Benford, Z. Q. Zhou, and T. Y. Chen. Enhancing supervised classifications with metamorphic relations. In *2018 IEEE/ACM 3rd International Workshop on Metamorphic Testing (MET)*, pages 46–53, May 2018.

[99] J. Zhang, E. T. Barr, B. Guedj, M. Harman, and J. Shawe-Taylor. Perturbed Model Validation: A New Framework to Validate Model Relevance. working paper or preprint, May 2019.

[100] J. M. Zhang, M. Harman, L. Ma, and Y. Liu. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[101] X. Zhang, L. Yu, and X. Hou. A method of metamorphic relations constructing for object-oriented software testing. In *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 399–406, May 2016.