

CAPITAL UNIVERSITY OF SCIENCE AND
TECHNOLOGY, ISLAMABAD



Test Data Generation for Evolutionary Mutation Testing of Object-oriented Programs

by

Muhammad Bilal Bashir

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosophy

in the

Faculty of Computing

Department of Computer Science

December 2017

Copyright © 2017 by Muhammad Bilal Bashir

All rights reserved. No part of this thesis may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, by any information storage and retrieval system without the prior written permission of the author.

Dedicated to
My Family Members



**CAPITAL UNIVERSITY OF SCIENCE & TECHNOLOGY
ISLAMABAD**

Expressway, Kahuta Road, Zone-V, Islamabad
Phone: +92-51-111-555-666 Fax: +92-51-4486705
Email: info@cust.edu.pk Website: <https://www.cust.edu.pk>

CERTIFICATE OF APPROVAL

This is to certify that the research work presented in the thesis, entitled “**Test Data Generation for Evolutionary Mutation Testing of Object-Oriented Programs**” was conducted under the supervision of **Dr. Aamer Nadeem**. No part of this thesis has been submitted anywhere else for any other degree. This thesis is submitted to the **Department of Computer Science, Capital University of Science and Technology** in partial fulfillment of the requirements for the degree of Doctor in Philosophy in the field of **Computer Science**. The open defence of the thesis was conducted on **23 November, 2017**.

Student Name : Mr. Muhammad Bilal Bashir
(PC111001)

The Examination Committee unanimously agrees to award PhD degree in the mentioned field.

Examination Committee :

(a) External Examiner 1: Dr. Muhammad Zohaib Zafar Iqbal
Associate Professor
FAST-NU, Islamabad

(b) External Examiner 2: Dr. Saad Naeem Zafar
Assistant Professor
Riphah Int. University, Islamabad

(c) Internal Examiner : Dr. Muhammad Tanvir Afzal,
Associate Professor,
CUST, Islamabad

Supervisor Name : Dr. Aamer Nadeem,
Associate Professor,
CUST, Islamabad

Name of HoD : Dr. Nayyer Masood,
Professor,
CUST, Islamabad

Name of Dean : Dr. Muhammad Abdul Qadir,
Professor,
CUST, Islamabad

AUTHOR'S DECLARATION

I, **Mr. Muhammad Bilal Bashir** (Registration No. PC111001), hereby state that my PhD thesis titled, '**Test Data Generation for Evolutionary Mutation Testing of Object-Oriented Programs**' is my own work and has not been submitted previously by me for taking any degree from Capital University of Science and Technology, Islamabad or anywhere else in the country/ world.

At any time, if my statement is found to be incorrect even after my graduation, the University has the right to withdraw my PhD Degree.



(Mr. Muhammad Bilal Bashir)

Dated: 23 November, 2017

Registration No : PC111001

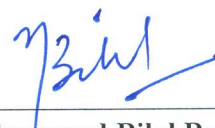
PLAGIARISM UNDERTAKING

I solemnly declare that research work presented in the thesis titled “**Test Data Generation for Evolutionary Mutation Testing of Object-Oriented Programs**” is solely my research work with no significant contribution from any other person. Small contribution/help wherever taken has been duly acknowledged and that complete thesis has been written by me.

I understand the zero tolerance policy of the HEC and Capital University of Science and Technology towards plagiarism. Therefore, I as an author of the above titled thesis declare that no portion of my thesis has been plagiarized and any material used as reference is properly referred/ cited.

I undertake that if I am found guilty of any formal plagiarism in the above titled thesis even after award of PhD Degree, the University reserves the right to withdraw/ revoke my PhD degree and that HEC and the University have the right to publish my name on the HEC/ University Website on which names of students are placed who submitted plagiarized thesis.

Dated: 23 November, 2017



(Mr. Muhammad Bilal Bashir)
Registration No. PC111001

Acknowledgements

I express my gratitude to the Almighty Allah, Who is the Most Merciful and Beneficent, Who bestowed His blessings on me and enabled me to complete this thesis.

I dedicate this thesis to my parents, wife, kids, family members and friends who are always with me, their prayers and love always supported me. My parents commitment and devotion made it possible that I am completing this thesis and my higher education today.

I further pay my gratitude to my kind and supportive supervisor Dr. Aamer Nadeem, who helped me in every step during my thesis. Without his guidance and help, I could never have been able to complete this thesis successfully. I am also grateful to Dr. Shafiq-ur-Rehman and Shaukat Ali for their valuable feedback and guidance during my thesis. I also like to say special thanks to my wife Sadaf Manzoor for her immense support and motivation. I would also like to acknowledge my senior researchers and all the members of Center for Software Dependability (CSD) for their encouragement and support. In the end again, I would like to acknowledge the support of my friends. My true gratitude and thanks to my sincere and loving family, whose prayers have always been a source of determination for me.

List of Publications

Research work appeared in the following publications:

- **Journal Publications**

1. **Bashir, M.B.**, Nadeem, A., (2017) "Improved Genetic Algorithm to Reduce Mutation Testing Cost", IEEE Access, 2017.
2. **Bashir, M.B.**, Nadeem, A., An Evolutionary Mutation Testing Tool for Java Programs eMuJava, **Under Review in Journal of Systems and Software.**

- **Conference Publications**

1. **Bashir, M.B.**, Nadeem, A., (2014 December) A State-based Fitness Function for the Integration Testing of Object-Oriented Programs. In the Proceedings of the 10th International Conference on Emerging Technologies, (ICET '14), Islamabad, Pakistan.
2. **Bashir, M.B.**, Nadeem, A., (2013 December) A Fitness Function for the Evolutionary Mutation Testing of Object-Oriented Programs, In the Proceedings of the 9th International Conference on Emerging Technologies, (ICET '13), Islamabad, Pakistan.
3. **Bashir, M.B.**, Nadeem, A., (2012 December) "Control Oriented Mutation Testing for Detection of Potential Software Bugs", In the Proceedings of 10th International Conference on Frontiers of Information Technology, (FIT 12), Islamabad, Pakistan.
4. **Bashir, M.B.**, Nadeem, A., (2012 October) "Object Oriented Mutation Testing: A Survey", In the Proceedings of the 8th International Conference on Emerging Technologies, (ICET '12), Islamabad, Pakistan.

Abstract

Software testing aims to identify bugs in the software under test to raise its quality and to ensure its correctness and usefulness for the user. It is an important phase of software development life-cycle and it consumes most of the resources in terms of time and money. Among all the activities of software testing, the test case generation consumes most of the effort and can be laborious if software testers perform it manually. Automatic generation of test cases can help in reducing the amount of resources and the effort this activity requires. Mutation testing can be very useful in generating effective test cases that can catch injected faults from the program under test. Besides that mutation testing is useful in measuring the effectiveness of given test suite. Mutation testing is computationally expensive because it requires execution of a large number of mutant programs while generating test cases. Evolutionary testing techniques (like genetic algorithm) can be used with mutation testing to reduce the computational overhead. This leads to a new testing technique named as evolutionary mutation testing that aims to automatically generate test cases using a genetic algorithm to kill non-equivalent mutants. During the test case generation, the performance of genetic algorithm depends on guidance that it gets from its fitness function.

An object-oriented program has two main components; data (object's state) and control (execution path). The fitness function should consider objects state as well as control flow information of the program under test to correctly evaluate a test case and to provide better guidance to genetic algorithm. Otherwise the process can become stuck or random at times in absence of enough guidance.

There exist several evolutionary mutation testing techniques in literature. The literature survey shows that although existing mutation based testing techniques cover some important aspects of program and use the information to evaluate fitness of a test case yet none of them uses object's state and control-flow information of program for fitness evaluation. The conventional crossover method generates two offsprings from parents by merging first segment of a parent to the second segment of the other and vice versa. This merger usually occurs at a randomly selected crossover point. This type of crossover can actually generate four offsprings by combing first segment of a parent to both the segments (first and second) of the other and vice versa.

Experiments show that this type of crossover can help exploring the input domain comprehensively. But none of the existing evolutionary testing techniques uses this method. Existing evolutionary mutation testing techniques use a fixed rate of biological mutation in genetic algorithm. In some cases more random effect is required to generate the required input values that crossover cannot produce. But since mutation rate is fixed, search process has to wait for biological mutation's turn and hence unnecessary crossover wastes some effort.

In this thesis, we target object-oriented programs and present four proposals to improve evolutionary mutation testing process. We have applied these proposals on genetic algorithm to reduce test case generation effort. The first two improvements are related to fitness function where we propose evaluating object's state and control flow information as part of the test case fitness. With the help of object's state fitness, it becomes easier to check if a given test case is able to achieve the desired state of an object or if it requires further improvement. On the other hand using control flow information of both original and mutated programs, fitness function checks if mutated program is exercising expected behavior on a test case. Sometimes this information helps in identifying logical programming errors (bugs) in the program. The third proposal is introduction of new two-way crossover method to evolve test cases using object's state fitness so they converge towards the target quickly. Finally we propose using an adaptable mutation rate during execution of genetic algorithm that adjusts according to the situation and saves testing effort.

In the later part of this thesis we present the proof of concept tool eMuJava that stands for **e**volutionary **M**utation testing of **J**ava programs. We have performed extensive experiments using this tool to compare our proposed approach with other approaches. We have also compared the results with EvoSuite, which is the most related tool we have found in literature. Our proposed approach is able to increase mutation scores ranging from 6% to 10% in comparison to other testing techniques (random testing and two variants of genetic algorithm). The results show that our proposals help improving the evolutionary mutation testing by reducing the amount of time required to achieve the targets.

Contents

Author’s Declaration	iii
Plagiarism Undertaking	iv
Acknowledgements	v
List of Publications	vi
Abstract	vii
List of Figures	xii
List of Tables	xiv
Abbreviations	xvi
1 Introduction	1
1.1 Software Testing and Automation	1
1.2 Mutation Testing	3
1.3 Research Motivation	4
1.4 Problem Statement	5
1.5 Research Contributions	6
1.6 Thesis Organization	8
2 Background	9
2.1 Software Testing Techniques	9
2.1.1 Mutation Testing	10
2.1.2 Evolutionary Testing	12
2.1.3 Evolutionary Mutation Testing	14
3 Related Work	17
3.1 Evolutionary Mutation Testing Techniques	17
3.2 Evolutionary Mutation Testing Tools	23
3.3 Conclusion of Survey	24

4	The Proposed Approach	26
4.1	Research Proposals at a Glance	26
4.2	Control-oriented Mutation Testing	29
4.2.1	Program’s Output	31
4.2.2	Method Produces an Output	32
4.2.3	Method Does Not Produce an Output	35
4.2.4	An Example	36
4.3	State-based and Control-oriented Fitness Function	38
4.3.1	State-based Reachability Cost	40
4.3.2	State-based Necessity Cost	41
4.3.3	Control-oriented Sufficiency Cost	43
4.3.4	Overall Fitness	43
4.3.5	An Example	44
4.4	Two-way Crossover Method	49
4.5	Adaptable Mutation Method	53
5	Tool Support	56
5.1	eMuJava Tool	56
5.1.1	eMuJava Architecture	58
5.1.2	eMuJava Operations	59
5.2	eMuJava Algorithms	62
5.2.1	GenMutants	63
5.2.2	GenPopulation	63
5.2.3	ExecTestCases	64
5.2.4	EvalTestCases	65
5.2.5	TWCrossoverTests	67
5.2.6	MutateTests	67
5.3	Supported Test Case Generation Techniques	68
5.4	eMuJava Design Model	69
5.5	eMuJava Configuration	69
5.6	Screenshots of eMuJava Tool	72
5.6.1	Source Code & Configuration	72
5.6.2	Configuration Editor	75
5.6.3	Mutants Viewer	75
5.6.4	Test Case Viewer	76
5.6.5	Statistics & Results	76
6	Experiments and Results Analysis	79
6.1	Test Environment	79
6.2	Initial Experiments and Results	82
6.3	Detailed Experiments and Results	85
6.3.1	Less Iterations, Higher Mutation Score	86
6.3.2	Comparison with EvoSuite	94
6.3.3	Detecting Suspicious Mutants to Raise Mutation Score	96

6.4	Statistical Analysis	98
6.4.1	<i>MannWhitney U-test</i>	99
6.4.2	Normality Test	100
6.4.3	Analysis of Initial Experiments and Results	101
6.4.4	Analysis of Detailed Experiments and Results	102
6.4.5	Effect Size Measure	103
6.5	Test Set Evaluation	104
7	Conclusion and Future Work	107
7.1	Future Work	110
	Bibliography	111
A	Literature Survey - Mutation Testing	119
A.1	Object-Oriented Mutation Testing	119
A.2	Evaluation Criteria	121
A.2.1	Cost Effective	121
A.2.2	Equivalent Mutant Detection	122
A.2.3	OO Feature Coverage	122
A.2.4	Level of Testing	123
A.2.5	State Mutation Support	123
A.2.6	Potential Mutation Operators	124
A.2.7	Tool Support	124
A.3	Surveyed Techniques	124
A.4	Analysis of Surveyed Techniques	128
A.5	Mutation Testing Tools	132
B	Literature Survey Evolutionary Testing	136
B.1	Evolutionary Testing Techniques	136
B.2	Evolutionary Testing Tools	140
C	Java Subset for eMuJava Tool	142
C.1	Java Language Subset for eMuJava	142
C.1.1	Context Free Grammar (CFG)	142
C.1.2	Keywords	142
C.1.3	Data Types	144
C.1.4	Special Symbols	144
C.1.5	Operators	144
C.1.6	Tool Assumptions	145
D	Additional Experiment Results	146

List of Figures

2.1	Mutation Testing Process Flow	11
2.2	Genetic Algorithm Process Flow	13
2.3	Fitness Function of Genetic Algorithm	14
2.4	Evolutionary Mutation Testing Process Flow	15
4.1	Research Proposals to Improve Evolutionary Mutation Testing Process	27
4.2	Control Flow Mutation Testing: Method Producing an Output [1]	33
4.3	Code Snippet to Demonstrate Using Control-flow Information in Mutation Testing: Method Producing an Output [1]	34
4.4	Control Flow Mutation Testing: Method Producing No Output [1]	35
4.5	Illustration of Control-Oriented Mutation Testing	36
4.6	Office Class with Method <code>empTaxExempt()</code> to Illustrate Control-oriented Mutation Testing	37
4.7	Java Code of <code>Mathematics</code> Class	39
4.8	Control Flow Graph of <code>smaller()</code> Method	39
4.9	Code Snippet to Demonstrate Proposed Fitness Function	45
4.10	Test Case Set	46
4.11	Code Snippet of <code>Stack</code> Class	50
4.12	Code Snippet of <code>Calculator</code> Class	54
5.1	eMuJava Architecture	57
5.2	Anatomy of a Test-case for Object-Oriented Program Testing [2]	61
5.3	<i>GenMutants</i> Algorithm	63
5.4	<i>GenPopulation</i> Algorithm	64
5.5	<i>ExecTestCases</i> Algorithm	65
5.6	<i>EvalTestCases</i> Algorithm	66
5.7	<i>TWCrossoverTests</i> Algorithm	67
5.8	<i>MutateTests</i> Algorithm	68
5.9	eMuJava Design Class Diagram	70
5.10	Main GUI - eMuJava Tool	72
5.11	eMuJava Wizard Step 1	73
5.12	eMuJava Wizard Step 2	73
5.13	eMuJava Wizard Step 3	74
5.14	eMuJava Source Code & Configuration	74
5.15	eMuJava Configuration Editor	75

5.16	eMuJava Mutants Viewer	76
5.17	eMuJava Test Case Viewer	77
5.18	eMuJava Statistics & Results	77
6.1	Comparison of Experiment Results among Random Testing, GA with Standard Fitness Function, and GA with Proposed Fitness Function	84
6.2	Comparison of Experiment Results among Random Testing, GA with Standard Fitness Function, and Improved Genetic Algorithm	88
6.3	Comparison among Average Executed Test Cases by Test Generation Techniques	89
6.4	Comparison to Show Progress among All Approaches while Achieving 100% Mutation Score	94
6.5	Comparison of Experiment Results Obtained from EvoSuite and eMuJava (Improved Genetic Algorithm)	97
6.6	CGPACalc Mutant with Logical Bug (Suspicious Mutant)	98
6.7	Comparison of Mutation Scores among Genetic Algorithm, EvoSuite, and Improved Genetic Algorithm	99
A.1	Mutation Operators Set [1]	120
C.1	Context Free Grammar for eMuJava Tool	143

List of Tables

6.1	List of Programs and Details	81
6.2	Results of Initial Experiments with Fixed Number of Iterations . . .	83
6.3	Results of Experiments Performed using eMuJava	87
6.4	Number of Iterations and Executed Test Cases by All Four Approaches	90
6.5	Experiment Results	96
6.6	Results of Test Set Evaluation	106
A.1	Evaluation Criteria for Cost Effective	121
A.2	Evaluation Criteria for Equivalent Mutant Detection	122
A.3	Evaluation Criteria for OO Feature Coverage	122
A.4	Evaluation Criteria for Level of Testing	123
A.5	Evaluation Criteria for State Mutation Support	123
A.6	Evaluation Criteria for Potential Mutation Operators	124
A.7	Evaluation Criteria for Tool Support	124
A.8	Analysis Table of Mutation Testing Techniques for Object-Oriented Programs	129
C.1	Supported Keywords by eMuJava	143
C.2	List of Primitive Data Types	144
C.3	List of Special Symbols	144
C.4	List of Operators	144
D.1	List of Case Studies	146
D.2	Iterations Used while Test Case Generation for <code>AutoDoor</code>	147
D.3	Iterations Used while Test Case Generation for <code>Calculator</code>	147
D.4	Iterations Used while Test Case Generation for <code>BankAccount</code>	148
D.5	Iterations Used while Test Case Generation for <code>CLI</code>	148
D.6	Iterations Used while Test Case Generation for <code>BinarySearchTree</code>	149
D.7	Iterations Used while Test Case Generation for <code>JCS</code>	149
D.8	Iterations Used while Test Case Generation for <code>CGPACalc</code>	150
D.9	Iterations Used while Test Case Generation for <code>Collections</code>	150
D.10	Iterations Used while Test Case Generation for <code>Compress</code>	151
D.11	Iterations Used while Test Case Generation for <code>Crypto</code>	151
D.12	Iterations Used while Test Case Generation for <code>CSV</code>	152
D.13	Iterations Used while Test Case Generation for <code>ElectricHeater</code> . .	153
D.14	Iterations Used while Test Case Generation for <code>HashTable</code>	154

D.15 Iterations Used while Test Case Generation for Lang	154
D.16 Iterations Used while Test Case Generation for Logging	155
D.17 Iterations Used while Test Case Generation for Math	155
D.18 Iterations Used while Test Case Generation for Stack	156
D.19 Iterations Used while Test Case Generation for TempConverter . .	156
D.20 Iterations Used while Test Case Generation for Text	157
D.21 Iterations Used while Test Case Generation for Triangle	157

Abbreviations

ABS	Absolute Value Insertion
AOR	Arithmetic Operator Replacement
EAM	Accessor Method Change
EMM	Modifier Method Change
eMuJava	Evolutionary Mutation Testing of Java programs
EOC	Reference Comparison and Content Comparison Replacement
GUI	Graphical User Interface
IOD	Overriding Method Deletion
IOP	Overriding Method Calling Position Change
IOR	Overriding Method Rename
IPC	Explicit Call of a Parent's Constructor Deletion
ISD	Super Keyword Deletion
ISI	Super Keyword Insertion
JDC	Java-supported Default Constructor Creation
JID	Member Variable Initialization Deletion
LCR	Logical Connector Replacement
LoC	Lines of Code
OMD	Overloading Method Deletion
OMR	Overloading Method Contents Replace
OOP	Object Oriented Paradigm
PNC	New Method Call With Child Class Type
ROR	Relational Operator Replacement
UOI	Unary Operator Insertion

Chapter 1

Introduction

The testing phase in software development life cycle plays a vital role in the success of a software application. Testing phase attempts to identify maximum errors from software under test so that after correction of those errors, software can work as expected. Major steps involved in software testing include generating test cases and input data, preparing the environment, test case execution, and analysis of results. The generation of test-cases and input data are time consuming and costly processes, and therefore these are required to be automated.

1.1 Software Testing and Automation

There exist two main categories of software testing techniques; black-box and white-box. In black-box testing, tester considers the program under test as a black-box that receives an input, processes it and produces an output. The tester is not concerned with the internal details of the black-box. Tester generates the test cases and test oracles using program's specification, whereas the specification is written usually in natural language or sometimes in formal specification languages. On the contrary, white-box testing techniques are more concerned with the internal details of the program, for example source code. Due to this reason white-box testing is also called code-based testing. In white-box testing, tester generates test cases to

execute certain path in the code or certain segment of the code. The test cases are generated on the basis of coverage criteria the tester wants to achieve on program under test. Specification is not required because the test cases are generated from the code itself.

Automating software testing phase and specifically test case generation process can significantly reduce the required effort. But we see that it is fairly difficult to automate black-box testing techniques for various reasons. Black-box testing techniques use specification to write test cases but specification is usually written in natural language. Natural languages are inherently ambiguous and a single word or a statement can give different meanings in different scenarios. Although formal languages exist for writing software specification but their use is uncommon. Another reason is that even sometimes the specification is incomplete or not available at all. Considering these issues, white-box testing techniques, for example, mutation testing seems more suitable for automation because it relies on code instead of specification. But automation faces new challenges as we see new programming techniques emerge with the passage of time. The object-oriented programming is the most common and popular paradigm in current era. It uses classes and objects and brings many other features like inheritance from the real world to programming the software.

Testing an object-oriented program is not straight forward and its automation is even more challenging. Testing a structured program is generally simpler as compared to testing an object-oriented program, because in structured programs, the main focus of the testers is on generation of input values [3]. A structured program comprises of functions or procedures and a function is, therefore, its basic unit of testing. On the other hand object-oriented programs consist of classes whereas each class is a blueprint of a real world concept. Unlike structured programs, writing a test case for a class involves many steps including creating an object of the class under test, generating a sequence of method calls to bring the object in a specific state, and finally calling the method under test [4]. Each method call in the test case may require generating input data to be passed as arguments, which may be primitive or non-primitive. Sometimes testing a specific method of

a class requires the object to be in a specific state, so it is important to generate a method call sequence that can bring the object in desired state. But initially, the accurate call sequence is unknown hence this process consumes effort to produce the required sequence of method calls [5].

1.2 Mutation Testing

Mutation testing is a code based testing technique but it differs from the other white-box techniques. It is a fault-based technique in which plausible faults are injected in the program under test. In this way, mutation testing can help measuring the effectiveness of a given test case set by checking how many injected faults the test set can catch. Mutation testing can also help in generating effective test cases [6, 7] that are capable of catching a set of faults in the program. So mutation testing is more concerned about faults and catching them instead of executing a certain path or a segment of source code. The concept of mutation testing was first coined around four decades ago by DeMillo, Lipton, and Sayward [8]. Mutation testing is computationally expensive by nature because it requires executing a large number of mutants (program containing a fault). The effort can be reduced only by generating effective test cases as quickly as possible

Evolutionary testing aims to automate the process of test case generation through various approaches, for instance genetic algorithm. The genetic algorithm does this by randomly generating a set of test cases initially. Then it evaluates them by considering the targets they are supposed to achieve. Later, genetic algorithm repairs the test cases, evaluates them again and repeats the process until it achieves all the targets or defined number of attempts exhaust. The structure of a test case can be different depending upon the paradigm used to develop the software. In case of object-oriented paradigm, a test case consists of two parts including invocations (constructor calls and method calls) and input data. When a class is tested in isolation each test case comprises of constructor call for object creation,

method calls to gain desired state of object, the method under test, and input data to be passed as method parameters.

Evolutionary mutation testing provides foundation to automate test case generation [6, 7] process in mutation testing to reduce its computational overhead. There exists no finite deterministic algorithm that can generate test cases for every program because test case generation is an undecidable problem. Due to this reason, we have to rely on meta-heuristic techniques (evolutionary testing) to automate test case generation for mutation testing. We use evolutionary approaches like genetic algorithm to generate test cases to kill non-equivalent mutants. For a test case to kill a mutant, it has to satisfy three conditions including reachability condition, necessity condition, and sufficiency condition. Fitness function of genetic algorithm is designed to evaluate a test case against its strength of satisfying aforementioned conditions. The process of test case generation uses test case fitness to improve it so it can converge towards the target quickly.

In evolutionary mutation testing, it is important to correctly evaluate a test case to generate effective population efficiently. Due to this reason fitness function becomes an important part of the whole testing process. In order to satisfy the three conditions to kill a mutant, in some cases the object needs to be in a desired state or otherwise test case may fail to satisfy them. Besides that, while comparing the original program with mutant program after executing a test case, control flow information can provide very useful information about program's behavior. The fitness function, therefore, should consider object's state as well as control flow information to evaluate a test case. The later phases of the genetic algorithm should intelligently utilize fitness information to evolve the test case set to achieve the targets quickly.

1.3 Research Motivation

Object-oriented programming has gained world-wide acceptance during the last two decades and almost all the types of user applications (desktop, enterprise, web,

mobile and so on) are being developed using object-oriented paradigm. It maps real world concepts to software that makes it flexible and powerful for software development. It has become increasingly important to test these systems to ensure their correctness and reliability. Test case generation for testing object-oriented programs require huge amount of effort. Due to this reason testers are looking for practical and useful automated solutions to reduce testing effort and improve fault detection rates.

Mutation testing is attractive but computationally expensive that hinders the professionals from using it to test real world programs. Search-based techniques like genetic algorithm can help in reducing mutation testing cost to a great deal [7, 9, 10]. Search-based mutation testing has gained immense attention in recent years. Over the last 5 years amount of published research in this area has rapidly increased [11]. Though it is also important to investigate if conventional genetic algorithm is sufficient to be used with mutation testing to automate test case generation process.

Therefore, we are looking to carry out this research study to improve genetic algorithm for mutation testing of object-oriented programs. As a result it will help in automatic generation of test cases to gain higher mutation scores with minimum effort (time) for object-oriented programs. We will also perform extensive experiments to validate the proposals that will show the effectiveness and usefulness of our proposed improvements.

1.4 Problem Statement

Automated test case generation is not straight forward while performing mutation testing on object-oriented programs. Evolutionary mutation testing heavily depends on the fitness function it uses for test case evaluation and for guidance to improve the test cases. So we need a fitness function, which is intelligent and comprehensive in a sense that it takes into account all the object-oriented features,

object's state for instance. Otherwise the search may get stuck or even become random, which results in consuming more time and effort for test data generation.

Mutation testing relies on program's output to judge the deviation in mutant when comparing it with the original program. The existing mutation testing techniques ignore the paths that original and mutant programs exercise during the execution of a test case. These paths are defined by the flow of control during execution and they yield important information on how the programs behave against a given test case. Mutation testing suffers badly from the existence of equivalent mutants and the research shows equivalent mutants have the tendency to mask logical software bugs that cannot be uncovered with conventional mutation testing. We need to adapt some mechanism in mutation testing so it can take advantage of control flow information. Furthermore, we also want our fitness function to be capable of using control flow information to evaluate a test case appropriately according to the program's behavior.

After accurate and comprehensive evaluation of a test case, we want the later phases (crossover and biological mutation) of genetic algorithm to use the fitness information intelligently in order to repair the test cases effectively so they converge towards the targets (killing the non-equivalent mutants) in less number of iterations. Unfortunately, this is not happening with conventional crossover and mutation methods. We want to improve crossover and mutation methods so they can utilize fitness information intelligently to repair and improve the test cases.

1.5 Research Contributions

Our contributions in this thesis are summarized below

1. We have conducted a survey on object-oriented mutation testing techniques [12]. In this survey we have covered the existing mutation testing techniques and analyzed them against a set of parameters to find out strengths and weaknesses in them.

2. We have proposed using control-flow information in mutation testing besides using program's output to compare the original and mutated programs [1]. In this study we have defined program's output and then have explained how the control flow information can be used to judge the program's behavior. Our experiments have shown that using control-flow information along with program's output can sometimes help in revealing potential software bugs from the program.
3. We have devised a novel fitness function for the evolutionary mutation testing of object-oriented programs [13]. The novel fitness function evaluates a test case against its ability to satisfy the three conditions (reachability, necessity, and sufficiency) to kill a mutant. Besides using the conventional features including approach level, branch distance, and data state differences in calculation of costs for reachability, necessity, and sufficiency conditions, we have proposed using object's state fitness and control flow information in evaluation of a test case. With the inclusion of these features, the test case generation process gets better guidance.
4. Our experiments with state-based and control-oriented fitness function have shown that although fitness function provides adequate information about the weaknesses in a test case but the standard genetic algorithm is unable to utilize the information properly hence the potential of object's state fitness is not fully exploited. So we have proposed new two-way crossover and adaptable mutation methods [14] to overcome this problem. Two-way crossover generates four offsprings from parents instead of two if a test case suffers from object's state issue. Adaptable mutation dynamically adjusts mutation rate at runtime to generate random input values frequently.
5. We have implemented a tool that we have named as eMuJava, which stands for **e**volutionary **M**utation testing of **J**ava programs. eMuJava is implementing all of our proposals and it can test Java based programs and can

generate test cases to kill mutants using four different approaches including random testing, genetic algorithm with standard fitness function, genetic algorithm with state-based & control-oriented fitness function, and genetic algorithm with state-based & control-oriented fitness function, two-way crossover, and adaptable mutation methods.

6. We have performed extensive experiments using eMuJava to validate the effectiveness of our proposed approach [14]. We have compared our proposed approach with random testing, two variants of genetic algorithm, and a tool called EvoSuite, which we have found in literature. Our experiments have shown that our proposals have potential in improving the test case generation process for evolutionary mutation testing of object-oriented programs.

1.6 Thesis Organization

The rest of this thesis is organized as follows; in chapter 2 we present brief background of the domain under study (mutation based evolutionary testing of object-oriented programs) whereas chapter 3 covers related work that includes techniques we find in the literature; chapter 4 presents our research proposals including state-based & control-oriented fitness function, two-way crossover, and adaptable mutation; chapter 5 describes about the automated solution of our proposals including its architecture and different modules, algorithms that we have devised, and some implementation details; chapter 6 presents empirical analysis of our proposed approach and details about the experiments we have performed to compare our proposed approach with others; conclusion and future work are presented in chapter 7.

Chapter 2

Background

The purpose of writing this chapter is to present brief background about testing techniques including mutation testing, evolutionary testing, and evolutionary mutation testing. The upcoming sub-sections provide information about the processes involved and the purpose of the aforementioned testing techniques.

2.1 Software Testing Techniques

Software testing is considered as the back-bone phase of software development life cycle. It plays a vital role because it helps in identifying bugs in the software under development so that a quality product can be built and delivered to its intended user. According to IEEE, software testing can be defined as: Testing is the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item [15]. Software testing requires a lot effort in terms of time and money, which makes this phase the most expensive phase of the software development life-cycle.

There are several activities involved in software testing that include preparing a test plan, test case design, test case execution, and evaluating the results. In planning usually objectives and risks of the process are identified and important decisions about the testing process are made, for example choosing the testing

strategy and defining intended targets. Test case design is the most important and time consuming phase that includes test case generation according to the targets that are to be achieved. During test case execution, test cases are run on the software under test and results are recorded, which are evaluated against the desired targets in the last phase. The process can continue until the desired results are achieved.

There exist two large categories of software testing techniques; black-box and white-box. In black-box testing, tester considers the program under test as a black-box that receives an input, processes it and produces an output. The tester is not concerned with the internal details of the black-box. Tester generates the test cases and test oracles using program's specification, whereas the specification is written usually in natural language or sometimes in formal specification languages. On the contrary, white-box testing techniques are more concerned with the internal details of the program, for example source code. Due to this reason white-box testing is also called code-based testing. In white-box testing, tester generates test cases to execute a certain path in the code or a certain segment of the code. The test cases are generated on the basis of coverage criteria the tester wants to achieve on program under test. Specification is usually not required because the test cases are generated by looking the code itself.

2.1.1 Mutation Testing

Mutation testing is a white-box testing technique but it differs from the other white-box testing techniques because it is more concerned about faults than code coverage. The concept of mutation testing was first coined by DeMillo, Lipton, and Sayward [8], which tests software by injecting faults in it. The mutation testing has been an area of interest for the researchers and a lot of research has been done over the last two decades or so. The purpose of mutation testing is to measure effectiveness of test cases as well as generation of effective test cases [6, 7]. It injects plausible faults in the program under test. Later the test cases are run on it to check if the test cases can detect them. This testing technique can also

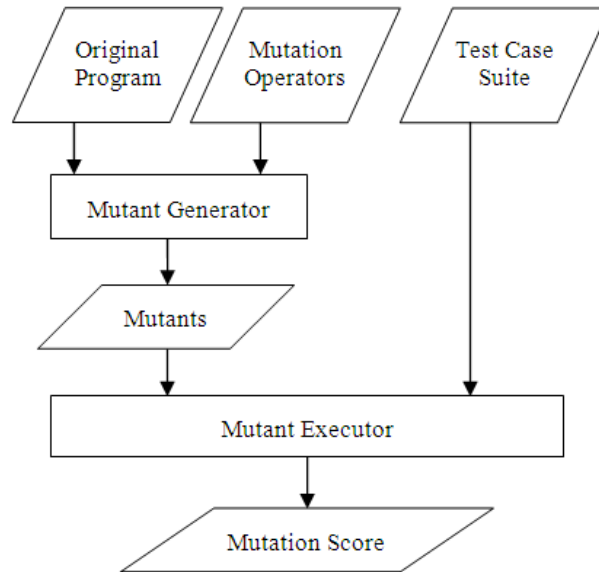


FIGURE 2.1: Mutation Testing Process Flow

be used to generate a test case set that may have potential to expose faults in the program. The figure 2.1 shows an iteration of mutation testing process including its inputs, processes, and outputs.

The first step in mutation testing is to generate mutant programs. A mutant program is a variant of original program containing only one syntactic difference. These syntactic changes are introduced using a predefined set of rules that are called mutation operators. A mutation operator can introduce a single type of change in the program whereas all the mutation operators may not be applicable to a given program. Once we have mutant programs, we execute all the test cases on them as well as on original program and observe their outputs. If a test case generates different output on mutant program as compared to the original program, then that particular mutant is considered as killed otherwise it remains alive. If all of the test cases fail to kill a given mutant, then it is called an equivalent mutant. Equivalent mutant cannot be killed because it is semantically equivalent to the original program. If test cases fail to kill all the mutants, the tester needs to re-generate a new set of test cases. The ultimate goal is to kill as many mutants as possible. The effectiveness of a test case set is determined with the help of mutation score, which is ratio of killed and all non-equivalent mutants.

Mutation testing is computationally expensive because it requires execution of a large number of mutants. The situation gets worse in presence of equivalent mutants because the mutant remains alive even after executing all the test cases. This problem is a major hindrance for the industry to accept mutation testing as a practical testing method and to use it at a high scale. Another debate has been the effectiveness of faults, which we inject during mutation analysis that whether they can represent real faults. Just et al. [16] prove through experimental study that mutants have strong correlation with real faults and they can be used as a substitute for program testing. We have presented existing work on mutation testing in Appendix A.

2.1.2 Evolutionary Testing

Evolutionary testing [3] is an approach to automatically generate test cases for the program under test. Evolutionary testing offers more than just one strategy for this purpose. Genetic algorithm, which is one of the evolutionary approaches, uses an iterative method to generate, evaluate, and repair test cases. Figure 2.2 presents the process flow of genetic algorithm [2]. Literature shows some research instances and studies produced by researchers around the globe that have taken this area a lot further than where it was in the beginning. Genetic algorithm is an iterative process that can be automated for a given type of problem like test case generation for a particular coverage criterion (test goal). The solutions go through biological crossover and mutation to gain appropriate shape, which is capable of achieving the desired test goal.

The process of genetic algorithm begins with random generation of initial population (test cases). The population size can vary depending on the scenario and requirement. The algorithm then runs the test cases on program being tested and records the execution traces along with other details like variable values, failed predicate and so on. When the test case execution completes, the test cases are evaluated using execution traces against the desired test goal. Fitness function,

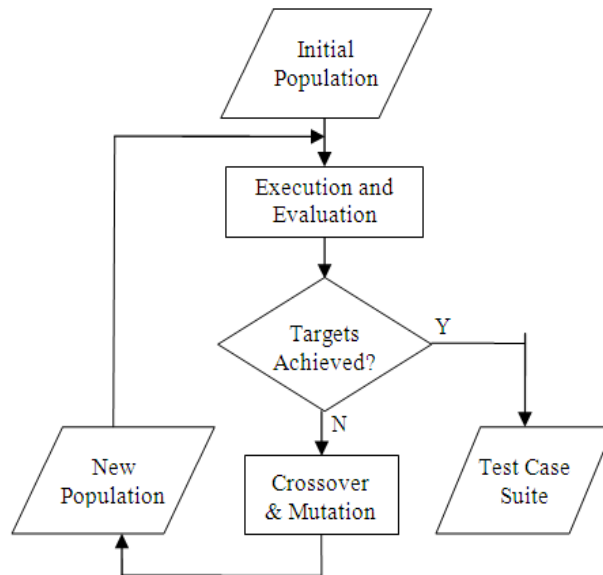


FIGURE 2.2: Genetic Algorithm Process Flow

which is one of most important part of genetic algorithm, performs the evaluation. The algorithm stops its execution if the test cases achieve the desired goal otherwise it repairs the test cases through crossover or biological mutation. Genetic algorithm continues to iterate until the desired goal is achieved or number of maximum iterations are reached. We have presented the existing work that we have found in the literature on evolutionary testing of object-oriented programs in Appendix B.

Fitness Function

The fitness function is a core part of genetic algorithm. It evaluates the test cases and assigns them a fitness value that guides the process towards the desired goal. The correct evaluation of test cases is very important otherwise the whole process may become random. The fitness function can be seen as a system that receives some inputs, process them, and produces some output as shown in figure 2.3 [2].

There are three inputs that a fitness function receives including test goals, execution traces, and test case set. Each of the test cases is evaluated using the execution traces it produces (during its execution on the program under test) against the test goal it is supposed to achieve. After completing the evaluation, fitness function assigns a non-negative cost to each test case as an output.

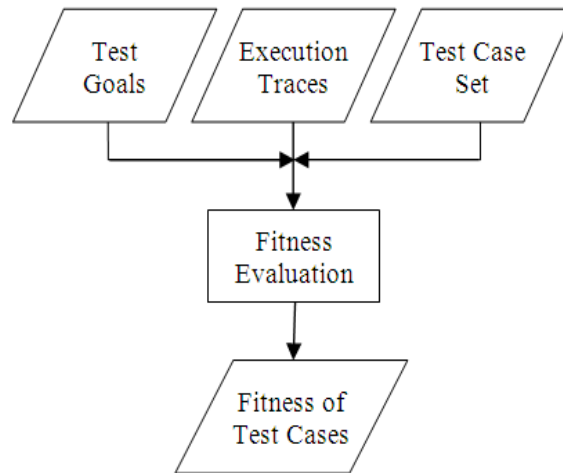


FIGURE 2.3: Fitness Function of Genetic Algorithm

2.1.3 Evolutionary Mutation Testing

The term evolutionary mutation testing was first coined by Domnguez et al. [17] to reduce mutation testing cost by mutants reduction method. We have re-used this term in same context (reduction of mutation testing cost) but we have achieved it by improving test case generation process. Another reason of using this term is that we are using two testing approaches, evolutionary testing and mutation testing, in combination for our research. Literature shows that mutation testing cost can be reduced by improving the process of test data generation [7, 9, 10]. Mutation testing is different in nature as compared to other white-box testing techniques and their coverage criteria. In mutation testing, the main target is to increase mutation score by killing all the non-equivalent mutants. So we need special kind of fitness functions to complement the mutation testing and to provide better guidance to the search. Although we find some work in the literature, which has been done in this particular area but there exists some room for more work to take full advantage of both the techniques.

The figure 2.4 presents the inputs, output, and activities involved in the testing process. The process starts with the generation of mutants that requires two inputs; original program and the mutation operators. The result is a set of mutants with injected faults and our target is to generate test cases to kill all non-equivalent

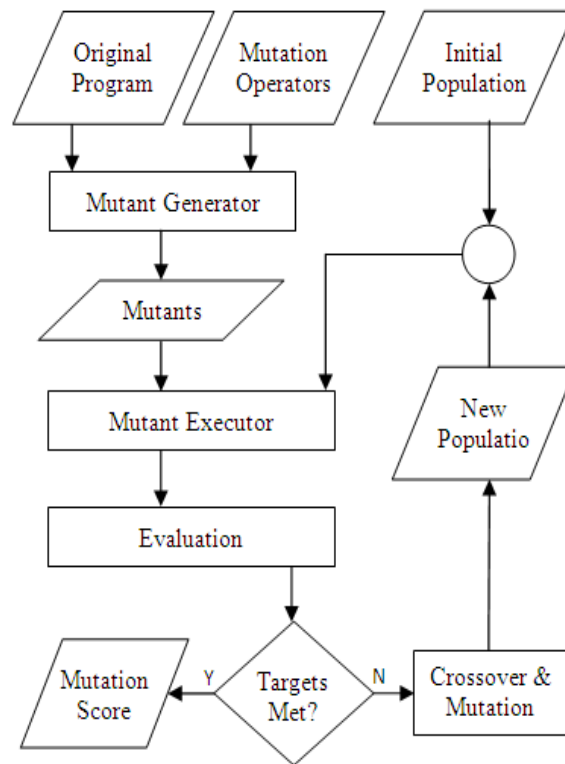


FIGURE 2.4: Evolutionary Mutation Testing Process Flow

mutants. Once we have mutants available, initial population (test cases) is generated and executed on all the mutants. The evaluation is performed to determine how many mutants are still alive and to assign fitness to test cases against them. If some mutants survive then we perform crossover and biological mutation on the test cases to generate new population on the basis of guidance that search gets from the evaluation process. This iterative process continues until all the non-equivalent mutants have been killed or maximum number of attempts has been exhausted.

If a test case fails to kill a mutant, it should be evaluated by the fitness function. The literature shows a test case should satisfy three conditions to be able to kill a mutant including; reachability condition [18], necessity condition [8], and sufficiency condition [8]. Reachability condition requires that the statement that contains the mutation must be executed. It is important for the test case to execute this statement; otherwise the behavior of original and mutated program remains the same. Secondly, the mutated statement must introduce some infection (necessity condition). That means, it must cause some change in behavior of the

program as this is the intention and purpose of introducing the mutation in the original program. Finally the infection caused by the mutation must be propagated and noticeable in the output, that is what sufficiency condition means.

Chapter 3

Related Work

The purpose of writing this chapter is to cover the literature survey that sheds some light on the research being done in the areas of object-oriented program testing using evolutionary mutation testing techniques. It provides the details about proposed work of researchers in this branch of testing.

3.1 Evolutionary Mutation Testing Techniques

The literature survey shows that, not a lot work has been done so far in evolutionary mutating testing of object-oriented programs. The researchers have mainly focused on evolutionary approaches like genetic algorithm for test case generation. We also find an object-oriented mutation testing tool that generates test data automatically. Next we provide brief description on surveyed techniques and on automated system that we have found in the literature.

L. Bottaci [18]

Bottaci [18] presents a fitness function for evolutionary mutation testing, which tries to satisfy all three conditions to kill a mutant. The fitness function assigns cost to test cases on the basis of their ability to satisfy these conditions. The reachability cost is computed by finding the difference between goal and executed path,

and calculating the branch distance on the failed predicate. The necessity cost is assigned the same way as of branch distance. Finally sufficiency cost is calculated by counting the number of same data states after the mutated statement.

M. Masud, A. Nayak, M. Zaman, N. Bansal [19]

Masud et al. [19] present a framework for performing mutation testing using genetic algorithm. Their work is basically an extension to the model proposed by Bybro [20]. In this work, research propose to partition the program into small units and then mutation is introduced in each unit independently. Before partitioning, they instrument the whole program in such a way that output of each unit is recordable. Mutations are introduced through structured mutation operators only so there is found no support for object-oriented mutation analysis in this approach. The mutations are performed through MU tool presented in the work of Bybro [20]. In the later phase test cases are generated using genetic algorithm and if a test suite fails to kill the mutant, the test cases can be improved manually or automatically. Due to the partitioning applied by the approach, this approach looks more closer to the weak mutation testing rather than strong mutation testing.

P. May, J. Timmis and K. Mander [7]

May et al. [7] conducted a research to compare the performance of immune based approaches with evolutionary approaches while generating test data for mutation testing. First they define mutation testing and provide description about its process and concepts that are involved. Later they provide description on Genetic Algorithm and Immune Inspired Algorithm and highlight their major differences and style of evolving solutions. They have also included the results of experiments that they have conducted to see which approach (Genetic Algorithm or Immune Inspired Algorithm) achieves high mutation score. For this they have developed systems that implement both the algorithms under study. The results of their experiments show that immune inspired algorithm (AIS) performs better than genetic algorithm (GA).

G. Fraser and A. Zeller [6]

Fraser and Zeller [6] design and introduce a novel fitness function that is purely object-oriented and little bit different in nature from what Bottaci [18] proposed. They use three different costs to evaluate a test case. The first cost is distance to calling function that assigns a cost to the test case that does not contain call to the function containing the mutated statement. This is a new type of cost that we do not notice in any other work so far. Also the test case structure seems different from standard object-oriented test case [4]. The second cost is distance to mutation that is calculated on the basis of approach level and branch distance, which is similar to the reachability cost propose by Bottaci [18]. The third cost is mutation impact that considers unique number of methods in mutated program for which the coverage change and noticeable state differences.

K.K. Mishra, S. Tiwari, A. Kumar, A.K. Misra [9]

Mishra et al. [9] propose a new approach for mutation testing using elitist Genetic algorithm. They extend the work of Bybro [20] and Masud [19] to generate test case for Java based programs. This approach supports unit (class) level testing and the generated test cases are in JUnit [21] format. Their approach uses test cases, which have killed some mutants, as initial population because these are good test cases and can improve performance of test case generation to a great deal.

J. Domnguez, A. Estero, A. Garca, I. Medina [17]

Domnguez et al. [17] present a novel approach to reduce number of mutants in mutation analysis. They try to find such mutants that are hard to kill because such mutants can help in generating test cases, which are better in quality as compared to the initial set of test cases. In order to reduce mutants, Domnguez et al. use evolutionary algorithms. They have also developed an automation solution called GAmera for mutants reduction. Authors have also evaluated their approach and present results of detailed experiments. The results show that instead of choosing mutants randomly, the coe-volutive genetic algorithm evolve them systematically and effectively.

M. Papadakis and N. Malevris [22]

Papadakis and Malevris [22] propose a fitness function for evolutionary mutation testing. Their fitness function assigns four costs to a test case. The first two costs are approach level and branch distance that they borrow from the work of Wegener et al. [23]. The third cost is predicate mutation distance that requires mutation distance, which is more or less same as of branch distance of failing condition. The impact distance (fourth cost) they use is an attempt to complement sufficiency condition. They borrow this from the research of Fraser and Zeller [6].

S. Subramanian, R. Natarajan [24]

Subramanian and Natarajan [24] propose a new approach that uses mutant gene algorithm. Mutant gene algorithm combines both mutation testing and genetic algorithm. Initially the test cases are generated randomly and then they are minimized through mutation gene algorithm. In every iteration, test cases are executed and evaluated using a fitness function. Then using rank based selection, test cases are selected for single-point crossover to generate new population for next iteration. Authors claim that their minimization approach outperforms other approaches in complex scenarios.

J. Louzada, C. Camilo-Junior, A. Vincenzi, C. Rodrigues [25]

Louzada et al. [25] propose a new approach for performing search-based mutation testing that uses elitist genetic algorithm for test data generation. Their approach tries to generate test cases that can kill maximum mutants to gain high mutation score. Authors have performed experiments on small scale on Java programs. The results show that their approach performs better than random testing.

Y. Ali and F. Benmaiza [26]

Ali and Benmaiza [26] propose an approach for testing object-oriented programs using search-based algorithm (genetic algorithm) and as a criteria they choose mutation analysis. Their approach automatically generates test cases to kill mutants, which are generated using class under test. Genetic algorithm uses mutation as

a fitness function and tries to maximize it through evolution. Authors have performed experiments on Java programs and the results show that their technique managed to reach 97.73% mutation score.

M. Rad and S. Bahrekazemi [27]

Rad and Bahrekazemi [27] study four evolutionary methods to improve test data generation for mutation testing. Authors start the discussion with mutation testing and present mutation operators proposed by [28]. Further they shed some light on problems of mutation testing. Then they present all the studied evolutionary algorithms including genetic algorithm, bacteriological, particle swarm optimization, and evolutionary quantum. Authors have performed experiments using Matlab [29] on a Java program. For experimentation they pick `Tritype` program that receives three inputs and tell the type of a triangle as output. With results of experiments, authors state that evolutionary algorithms can reduce mutation testing cost.

G. Fraser and A. Arcuri [10]

Fraser and Arcuri [10] present a new scalable approach to generate test cases for the programs. They use mutation testing as coverage criteria to generate test cases. In this work they apply new optimization approaches to reduce the total effort, which is required for mutation testing. Using these optimization approaches first they monitor the state infection conditions to avoid redundant execution of test cases to kill mutants and secondly, instead of generating test cases for individual mutants, they do it for all the targets all at once to save time. In this research they have performed extensive experiments using the EvoSuite [30] tool. For experimentation, they have extended the tool to support the new optimization methods. Their experiments show good results and the results indicate that optimization methods can help reducing mutation testing effort to a great deal.

J. Miguel, M. Vivanti, A. Arcuri, G. Fraser [31]

Miguel et al. [31] present an empirical study that try to answer some research questions regarding evolutionary mutation testing. In this research authors try to

answer some questions like if targets (mutants) should be investigated one by one or the testing approach should try to generate test cases all at once. They also want to investigate if the search process should only consider the targets, which are not already covered. For experiments, authors have picked 100 Java classes and performed detailed experiments. The results are quite interesting and show that some targets need to have more specific attention and should be covered independently. On the other hand most of the targets (mutants) can be investigated together.

N. Jatana, B. Suri, S. Misra, P. Kumar, A.R. Choudhury [32]

Jatana et al. [32] investigate the application of particle swarm optimization (PSO) in the area of mutation based evolutionary testing. Authors start the discussion with genetic algorithm, which is also a popular evolutionary approach and has been widely used. Genetic algorithm has already been applied by researchers to generate test data for mutation analysis. In this paper authors have tried to explore if particle swarm optimization can also be equally effective as genetic algorithm has been. To empirically prove that authors have applied PSO on some benchmark C programs and computed results. The results show that PSO has found killing many substantial mutants. On the basis of these experiments, authors have claimed that PSO can also be a good choice for test data generation in evolutionary mutation testing.

R. Silva, S. Rocio, S. Souza, P. Srgio [11]

Silva et al. [11] present a detailed literature survey of existing studies on search-based mutation testing. They have performed a thorough survey on all the related work and covered 263 research studies in the area of evolutionary and mutation testing. In this study they have tried to identify those areas that still need attention and problems that are still unsolved. They have also covered various different fitness functions that have been designed for search-based mutation analysis. The survey shows that so far search-based techniques have been mainly applied for the optimization of test data generation, selection of effective mutation operators, and mutant generation. Out of many, the researchers have used mainly five

techniques including genetic algorithm, ant colony, hill climbing, bacteriological algorithm, and simulated annealing. Authors have highlighted that still areas that need more attention are identification of equivalent mutants, experimental studies and application of this method to different domains like concurrent programs.

P. Delgado, I. Medina, S. Segura, A. Garca, J. Jose [33]

Delgado et al. [33] present their research that investigate generation of hard mutants through evolutionary algorithms. Authors initially present details about mutation analysis and evolutionary algorithm (genetic algorithm). They also provide class mutation operators for C++ programs in this paper. Then a system call GiGAn is presented along side its architecture. Authors have conducted brief experiments to see if genetic algorithm can produce stronger mutants rather than choosing them randomly. Although authors have claimed that genetic algorithm works well but the amount of experiments in this study are too few to advocate this claim.

3.2 Evolutionary Mutation Testing Tools

The above mentioned techniques are those that we have found in literature on evolutionary mutation testing of object-oriented programs. We have found only one tool that has been implemented and available for testing in this area that we present next.

EvoSuite [30]

Fraser and Arcuri [34] have developed an automated solution that generates test data for object-oriented programs. The tool is named as EvoSuite that has been developed in Java programming language to test programs written in Java. EvoSuite is a white-box testing tool that performs unit level testing and generates test cases for a class (unit). It can test as many classes as tester wants but the test cases are generated for every class individually. EvoSuite supports branch

coverage as well as mutation based coverage criteria. EvoSuite integrates two optimization techniques to reduce the overhead of mutation testing; one it monitors the state infection conditions, second it attempts to generate whole test suite by killing as much mutants as possible rather picking up them one by one [10]. It is a freeware tool, which is available for download from the internet [30]. The tool can be used in two ways; firstly it can be integrated with Eclipse [35] which is a popular tool for programming and secondly EvoSuite can also be used as a stand-alone testing tool with the help of commands. We have briefly used it for generating test cases for mutation based coverage criterion and we have found that it produces promising and interesting results. Though it requires more detailed analysis of test cases that it produces to judge its effectiveness.

3.3 Conclusion of Survey

As we have seen earlier in this section, a number of techniques exist for evolutionary mutation testing in the literature. The most important feature of mutation based evolutionary testing is its capability of finding optimal solution (test case) and the goal is to achieve it as quickly as possible. The better the fitness evaluation is, more chances are there for the search process to find the required solution in less number of iterations.

After analyzing the approaches that we find in the literature, we come to know that they are experiencing some issues, which we have explained below. The techniques proposed by Bottaci [18] and Papadakis and Malevris [22] mainly focus on structured paradigm so they are incapable of supporting evolutionary mutating testing of object-oriented programs. The reason being, these approaches do not provide support for object-oriented features like inheritance, polymorphism and so on. The work of Fraser and Zeller [6] supports object-oriented paradigm but it does not consider object's state, which is an important object-oriented features and the evolutionary search process may suffer from object's state problem [36]. The technique proposed by Mishra et al. [9] supports only unit level testing of Java

programs hence they cannot test all object-oriented features through mutations. Fraser and Arcuri [34] have developed a powerful tool, which they have called EvoSuite. It can test Java programs at unit level so it cannot test all object-oriented features. EvoSuite can apply limited set of mutation operators that are designed for structured paradigm. To summarize we can state that the existing approaches either support structured paradigm or they provide limited support to object-oriented features. The existing object-oriented techniques are vulnerable of suffering from object's state problem as they do not consider it while evaluating a test case. Due to these reasons their fitness functions also are not capable enough to provide accurate guidance.

Secondly, the existing techniques rely on the program's output only and do not use control flow information generated by a test case after execution. Control flow information can be very useful in determining the behavior of a program especially in case of mutation testing where mutated program is compared with original program. Sometimes this information can expose logical programming errors that are introduced by the programmers unintentionally. Lastly, using fitness information intelligently also increases the chances of finding the solution quickly, especially in situations where object's state matters. For this conventional crossover and mutation methods cannot help exploring the input domain comprehensively. A more sophisticated crossover and mutation methods are required.

Chapter 4

The Proposed Approach

This chapter presents our proposals to improve mutation testing and evolutionary mutation testing of object-oriented programs. The proposals include control-oriented mutation testing [1], state-based & control-oriented fitness function [13], and two-way crossover & adaptable mutation methods [14]. All of these proposed techniques are specifically designed for object-oriented paradigm.

4.1 Research Proposals at a Glance

Before we go in the details of research proposals, in this section, we present a brief description of our proposed modifications in mutation testing and evolutionary mutation testing approaches. This will help the reader to understand the issues we have addressed in this research, how our proposals are related to each other, and how they improve the test case generation process. Figure 4.1 presents the process flow of evolutionary mutation testing with highlighted activities that we have improved in this research. As an evolutionary algorithm, we have chosen genetic algorithm for our research. Genetic algorithm is the most widely used meta-heuristic technique applied in the domain of mutation testing [11]. There exists several variants of genetic algorithm and its fitness function, which are applicable

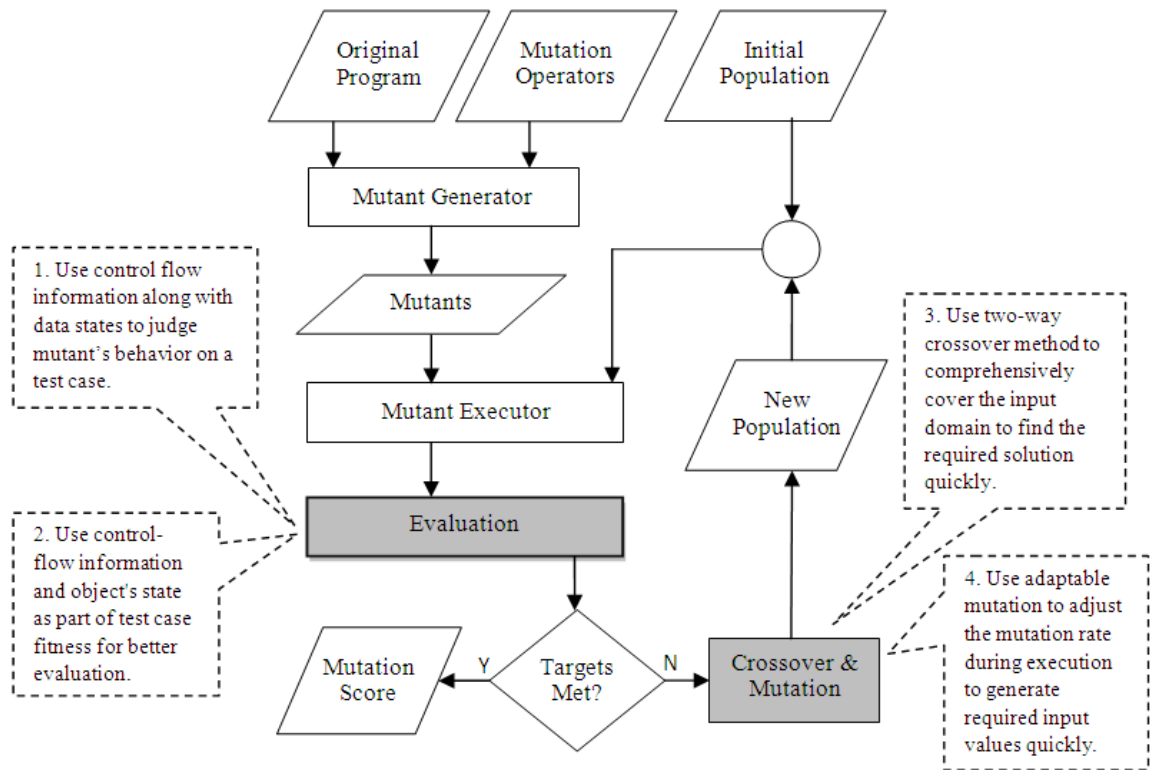


FIGURE 4.1: Research Proposals to Improve Evolutionary Mutation Testing Process

in different domains of software testing. Next we present brief description about each proposal before going in the detail in subsequent sections.

Catching Suspicious Mutants Using Control-flow Information: Mutation testing techniques that we find in the literature compare the output of original and mutant programs to decide if a test case is able to catch the injected fault or not. But the behavior of a program is defined by output as well as flow of control. If we ignore control flow of a program while analyzing its behavior on a test case, we may end up losing important information. So besides using data flow information, we have proposed that control-flow information should also be used to compare behaviors of original and mutant programs against a test case. We call this approach as *control-oriented mutation testing*. Sometimes original and mutant programs produce same output but mutant program suspiciously exercises different execution path. This may happen due to presence of some logical programming mistake or a potential bug in the code.

Guiding Search Using Objects State & Control-flow Information: Secondly, we

have designed a novel fitness function that can evaluate a test case, which is written for mutation based evaluation testing of an object-oriented program. The fitness function considers all three conditions that need to be satisfied to kill a mutant including reachability, necessity, and sufficiency conditions. We have called the proposed fitness function as *state-based & control-oriented fitness function* because we have included object's state and control-flow information in it as part of test case fitness. Both the parameters are based on our previous work including state-based fitness function [2] and control-oriented mutation testing [1]. Our proposed fitness function evaluates a test case thoroughly and provides better guidance to the search process so it can converge to the target quickly.

New Crossover Method to Produce Effective Offsprings: Our third proposal is about crossover method, which helps in covering the input domain in better way especially when object's state matters. The new crossover method considers object's state and generates offsprings (new test cases) that have the capability to gain required object state, which is required to achieve the target (kill a mutant). We have called our proposed crossover method as *two-way crossover method* for genetic algorithm. The two-way crossover method works in conjunction with our proposed state-based & control-oriented fitness function. It uses the fitness information produced by the fitness function to decide how to perform crossover to generate new test cases.

Adaptable Mutation to Generate Frequent Random Inputs: Fourth proposal is about biological mutation, which helps generating random input values more frequently. The new method considers object's state and changes the mutation rate at runtime and we call it *adaptable mutation method*. If all the test cases have 0.0 as state fitness, it means their objects are able to gain desired state but the input parameters of method under test are probably causing the test case to fail. So crossover in this situation may result in wasting effort because crossover does not change input parameters. The mutation can change input parameters of methods so increasing its frequency can help in generating the required input parameters quickly.

In the next sub-sections, we have presented all of our proposals in detail. Our proposed modifications for GA may not work well with other evolutionary algorithms like AIS so some adjustments may need to be done in order to check effectiveness of our proposals with other evolutionary approaches.

4.2 Control-oriented Mutation Testing

We propose to use flow of control of the program in mutation testing that can uncover potential bugs present in the code. The details of our proposed technique are presented here in this section with the help of an example.

Mutation testing techniques that we find in the literature compare the output of original and mutant programs to decide if a test case is able to catch the injected fault or not. But the behavior of a program is defined by output as well as flow of control. If we ignore control flow of a program while analyzing its behavior on a test case, we may end up losing important information. So when a test case is executed, besides the output, the control flow information should also be compared. We propose and present a new technique for mutation testing that considers both of these behavioral elements (output and control flow information) to decide if a mutant is killed or alive. With this modification in mutation testing, a comprehensive comparison of both the programs can be made. Sometimes on a test case, original and mutant programs produce same output but mutant program suspiciously exercises different execution path. This may happen due to some logical mistake (potential bug) in the code, which is usually made by the programmer.

As the evolutionary mutation testing begins, it generates all possible mutants from the program under test using mutation operators the tester provides as input. The testing process continues and tries to kill all non-equivalent mutants through test cases, which it generates automatically using genetic algorithm (GA). The GA considers one mutant at a time as a target. The test cases are then executed on original and mutant programs while output as well as execution traces are

logged. On completion of test case execution, GA compares output and traces of all the test cases to decide on mutant's status. If there exists at least one test case that produces different output as well as different execution path on original and mutated program, the mutant is declared as killed and GA picks up the next target (mutant) and continues its execution. Otherwise if GA finds a test case that produces same output but different execution path or different output but same execution path, GA declares it as suspicious mutant. But if there is no such test case and for all the test cases, output as well as execution path remains the same, the mutant remains alive. The GA then goes on to run next iteration and tries to kill the mutant by repairing test cases. If the maximum number of iterations complete but mutant remains alive then GA gives up on the current mutant and picks the next one and continues its execution. There are some mutation operators that can affect the flow of control in mutants, for example ROR (Relational Operator Replacement). So comparing control flow information for such mutants can help in identifying piece of code containing a bug in method under test. Because if mutant does not produce different output even though it exercises different execution path there is a chance the program may have some logical error in it. Below we present a list of mutation operators that can change the execution path;

Mutation Operators for Structured Paradigm (Offutt et al. [37] & Barbosa et al. [38])

- *ROR*: Relational operator replacement
- *LCR*: Logical connector replacement

Mutation Operators for Object-Oriented Paradigm (Offutt et al. [39])

- *IOR*: Overriding method rename
- *IOP*: Overriding method calling position change
- *IOD*: Overriding method deletion

- *ISD*: super keyword deletion
- *ISI*: super keyword insertion
- *PNC*: new method call with child class type
- *IPC*: Explicit call of a parent's constructor deletion
- *OMD*: Overloading method deletion
- *OMR*: Overloading method contents replace
- *JDC*: Java-supported default constructor creation
- *JID*: Member variable initialization deletion
- *EMM*: Modifier method change
- *EAM*: Accessor method change
- *EOC*: Reference comparison and content comparison replacement

In object-oriented programs, every class can have methods whereas each method contains the code. So a class is tested method-wise because each method represents some behavior of the class. To test that behavior test cases are generated whereas each test case first instantiate an object of the class under test. Then a sequence of method calls is generated followed by the method under test. Methods in a class may or may not generate some output. In coming sub-section we present our proposed idea of using control flow information in mutation testing but before that we shed some light on program's output.

4.2.1 Program's Output

A program generates an output after processing the data, which it receives as input. In case of structured paradigm where program's code is spread across many functions, we consider the returned value of the function (containing mutated statement) and variables passed by reference (if any) as output. This is not as

straight-forward in case of object-oriented programs because methods in a class can not only return a value, they can also change the object's state by redefining its state variables. Due to this salient feature of object-oriented programs, a question arises that how should the program's output be determined after execution of a test case in mutation testing. The survey of existing techniques provides no information and the question remains unanswered. In our research we answer this question by explicitly defining program's output and we have considered both (object's state after function completes its execution and returned value of the function) as output of the program. In coming sections, if we state a function produces some output then it means the function either returns some value or it redefines one or more object's state variables or both of these.

It is interesting to note that a method in a class may not always produce some output. So how mutation testing will behave to this situation? In next subsections we present our idea of using control flow information in mutation testing and explain how mutation testing will work when method of a class does and does not produce some output.

4.2.2 Method Produces an Output

We explain this concept further with the help of figure 4.2. In first case if the method under test produces an output, we will compare its output and execution path with the original method. If for at least one test case original and mutated methods produce different output and exercise different execution path, we declare it as *killed*. Otherwise if for at least one test case original and mutated methods produce same output but exercise different execution path or produce different output but exercise same execution path, we declare such a mutant as *suspicious*. The suspicious mutant alarms the tester to review the code for any possible bug in it. Finally if there is no test case for which original and mutated methods produce either different output or exercise different execution path, we declare such a mutant as *alive*.

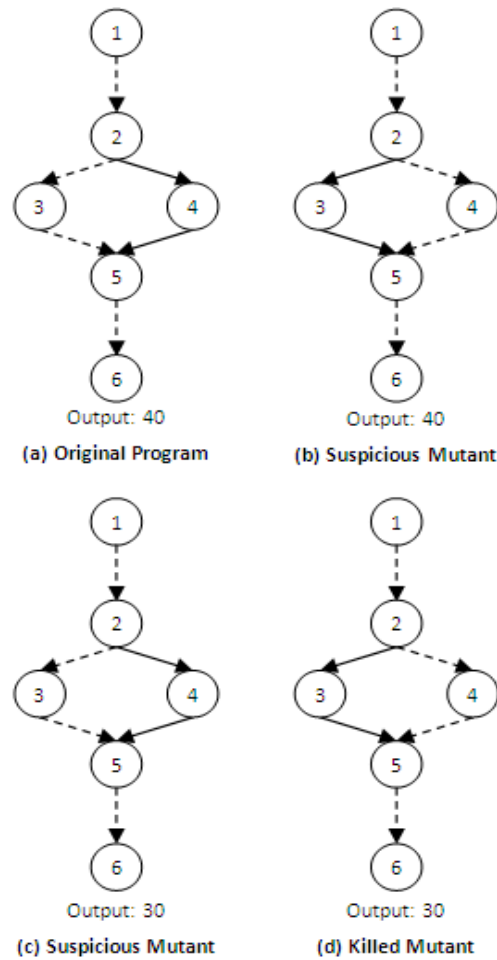


FIGURE 4.2: Control Flow Mutation Testing: Method Producing an Output [1]

Suppose there is a program P whose control flow graph G is presented in figure 4.2 having 6 nodes. When a test case T is executed on program P it produces execution path shown with the help of dotted arrows in figure 4.2(a) and produces output as 40. The control flow graphs in figures including 4.2(b), 4.2(c), and 4.2(d) represent three different cases when the test case T is executed on mutant M generated from program P . In figure 4.2(b) we show that mutant exercises different execution path but produces same output (40). In figure 4.2(c), mutant exercises same path but produces different output (30). As per our proposal both the cases shown in figure 4.2(b) and 4(c) are examples of **suspicious mutants**. In the last case shown in figure 4.2(d), the mutant exercises different execution path and produces different output (30) hence it is said to be *killed*. If a mutant exercises same execution path and produces same output that remains *alive*.

<pre> int z; int compare(int x, int y) { 1 if(x < y) { 2 z = x; 3 } else { 4 z = x; 5 } 6 return z; } </pre>	<pre> int z; int compare(int x, int y) { 1 if(x >= y) { 2 z = x; 3 } else { 4 z = x; 5 } 6 return z; } </pre>
(a) Original Program	(b) Mutated Program

FIGURE 4.3: Code Snippet to Demonstrate Using Control-flow Information in Mutation Testing: Method Producing an Output [1]

Using control flow information in mutation testing can be useful in finding logical errors (bug) present in the program. Next we present an example to explain how it works. Figure 4.3(a) provides definition of a function `compare()` accepting two parameters of integer type and returns an integer value to the calling function.

The `compare()` function computes the value of data member `z`, which is declared outside the scope of the function. The function assigns a value to `z` (line 2 and 4) after making a comparison between its two arguments (`x` and `y`). Instead of assigning value of variable `y` to `z` at line 4, the function assigns `x` again. Hence the function will always return the value of variable `x` at line 6. The figure 4.3(b) presents mutated version of the `compare()` function.

After applying the ROR (relation operator replacement) operator the predicate `x<y` on line 1 is replaced by `x>=y`. This mutation will cause the original and mutant versions to always exercise different execution paths with any value of `x` and `y`. So if original program executes body of `if` statement, the mutant will execute the `else` part and vice versa. But since there is a logical error in the program (`x` defines `z` on line 2 and 4) the output of both the functions will be the same. The problem here is that existing mutation testing techniques will declare this mutant as alive because no test case can kill it. In this situation when we apply our proposed technique, it will declare this mutant as **suspicious** and will help in uncovering the error, the programmer made.

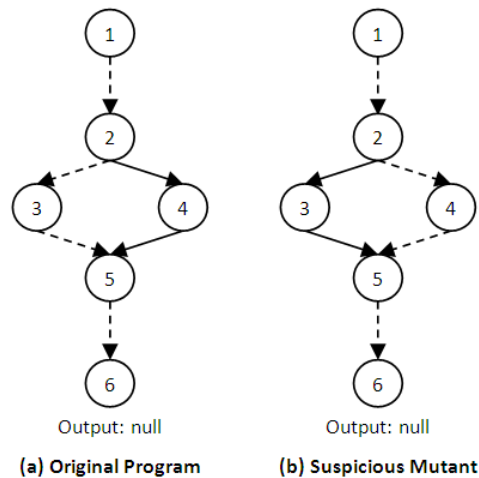


FIGURE 4.4: Control Flow Mutation Testing: Method Producing No Output [1]

4.2.3 Method Does Not Produce an Output

If the method containing mutation does not produce some output we suggest comparing execution paths to decide if the mutant has been killed or if it stays alive. The reason being if we stay on conventional approach and try to compare output of the programs or even object's states, we will not be able to decide on mutant's status and mutant will remain alive. No test case will be able to kill such a mutant and eventually this will be declared as equivalent mutant and a lot of computational effort will go in vain.

Suppose there is a program P whose control flow graph G is presented in figure 4.4 having 6 nodes. When a test case T is executed on program P it produces execution path shown with the help of dotted arrows in figure 4.4(a) and produces no output. The control flow graph in figure 4.4(b) represents the execution path generated by the test case T when executed on mutant M . Logically, if a method does not produce an output, ideally it uses no data inside its body hence execution path should remain the same therefore, if a mutant exercises different execution path, we declare it as **suspicious mutant**.

Next we present an example to illustrate how our control-oriented mutation testing works.

```

public class Employee {
1  private float salary;
2  public Employee( float s ) {
3      salary = s;
4  }
5  public String isTaxExempted() {
6      if( salary < 30000 ) {
7          return "Yes";
8      } else {
9          return "No";
10     }
11 }
}

```

(a) Employee Class

```

public class Manager extends Employee {
1  public Manager( float s ) {
2      super( s );
3  }
}

```

(b) Manager Class

FIGURE 4.5: Illustration of Control-Oriented Mutation Testing

4.2.4 An Example

In this section we show the working of our proposed technique using an example. We apply "PNC: new method call with child class type" operator in our example. Consider the code snippet presented in figure 4.5;

Figure 4.5(a) provides definition of `Employee` class that declares a data member `salary` of type `float` at line 1. The class provides a parameterized constructor (line 2 to 4) that initializes `salary` with the parameter it receives. The `Employee` declares and defines a method `isTaxExempted()` (line 5 to 10) that determines if an employee is exempted from tax or not. Figure 4.5(b) provides definition of `Manager` class, which is a sub-class of `Employee`. `Manager` also provides a parameterized constructor. `Manager` inherits `isTaxExempted()` from `Employee` hence `Employee.isTaxExempted()` is invoked whether a reference contains an object of `Employee` class or `Manager` class.

Now consider code example of figure 4.6 that presents original and modified versions of method `empTaxExempt()`, which is defined in `Office` class.

```
public class Office {
1  public String empTaxExempt(float salary) {
2      Employee emp = new Employee( salary );
3      return emp.isTaxExempted();
4  }
}
```

(a) Office Class – Original Program

```
public class Office {
1  public String empTaxExempt(float salary) {
2      Employee emp = new Manager( salary );
3      return emp.isTaxExempted();
4  }
}
```

(b) Office Class – Mutant Program

FIGURE 4.6: Office Class with Method `empTaxExempt()` to Illustrate Control-oriented Mutation Testing

The figure 4.6(a) provides definition of `Office` class having a function `empTaxExempt()` that accepts a `float` type parameter and returns a `String` value to the caller. On line 2 of the function, it creates an object of `Employee` class and invokes the method `isTaxExempted()` on it, which also returns a `String` value. The figure 4.6(b) presents modified version of the code in figure 4.6(a). The modified version contains just one change at line 2 where using polymorphism concept, an object of `Manager` class is assigned to `Employee` reference. This is an application of operator "PNC: new method call with child class type".

If we run a test case on both versions of the program presented in figure 4.6, we find that in both cases, `Employee.isTaxExempted()` is executed even though in figure 4.6(b) the reference holds the object of `Manager` class. The reason being, `Manager` class does not override this function rather it inherits it from the parent. So in both cases the output remains the same although in mutant program the control passes through different nodes as it invokes the constructor of `Manager` class at line 2 instead of `Employee` class. As per our proposal we declare such a mutant as **suspicious** whereas conventional techniques will tag the mutant as alive.

4.3 State-based and Control-oriented Fitness Function

In this section we present a novel fitness function, which we have designed for evolutionary mutation testing of object-oriented programs. The novel fitness function is specifically designed to work with Genetic algorithm.

Our proposed fitness function evaluates three costs against three conditions that a test case tries to satisfy to kill a mutant and as used by Bottaci [18]. We have extended the conditions used by Botacci [18] and have incorporated two additional parameters in them for better evaluation. The two parameters are based on our previous work including state-based fitness function [2] and control-oriented mutation testing [1]. Our proposed fitness function suits well for object-oriented testing because we propose isolating state of the object from branch distance while calculating costs for reachability and necessity conditions. It helps evaluating if the object is in desired state. We also propose comparing flow of control besides output of the original and mutant programs. Using control flow information in mutation testing can help finding out logical errors in the program under test. Fixing these errors can result in killing more mutants that eventually raises the mutation score.

The figure 4.7 defines a class with name `Mathematics` that offers a default constructor for object creation. The class also provides a function `smaller()` that accepts two parameters of type `long`. The function finds the smaller value between the two passed to it as parameters, computes its square and returns the result to the caller. If both the values are equal then the square of first parameter is computed and returned.

The `Mathematics.smaller()` function is represented with the help of control flow graph in figure 4.8. Suppose there is a mutation introduced in the code shown in figure 4.7 such that on line 6, a constant is assigned to variable `small` instead of `x`. The control flow graph in figure 4.8 is logically sub-divided in three segments to show the costs associated to a test case in evolutionary mutation testing.


```

public class Mathematics {

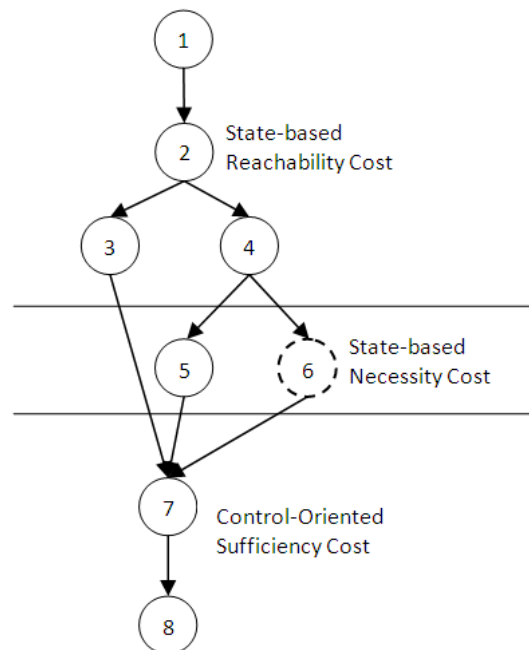
    public Mathematics() { }

    public long smaller(long x, long y) {
        1. long small = -1;
        2. if( x<y ) {
        3.     small = x;
        4. } else if( x>y ) {
        5.     small = y;
           } else {
        6.     small = x;
           } //END if-else
        7. small = small * small;
        8. return small;
    } //END smaller()

} //END Mathematics

```

FIGURE 4.7: Java Code of Mathematics Class

FIGURE 4.8: Control Flow Graph of `smaller()` Method

In figure 4.8, the control flow graph is sub-divided with the help of two lines. This division is done to represent piece of code that is considered while evaluating fitness (costs) of a test case. If the test case fails to execute mutated statement, State-based Reachability Cost is evaluated. Otherwise if test case does execute but no infection is introduced in the mutant, State-based Necessity Cost is computed. Eventually if infection is introduced but not propagated to the output then Control-oriented Sufficiency Cost is calculated for the test case.

In coming sections, we present all three types of fitness values associated with a test case along with our proposed changes to demonstrate how our proposals improve the fitness evaluation.

4.3.1 State-based Reachability Cost

In mutation testing, a test case should execute the line of code containing mutation. In case of failure, the test case is assigned a cost that guides the process towards reaching the mutated statement. Until test case executes the mutated statement, whatever cost we assign to other two segments of fitness, it makes no difference.

In object-oriented program, executing a specific statement sometimes require the object to be in a specific state. For this reason a test case makes calls to various functions randomly on the object. If the function call sequence is not accurate the object may not gain required state and fail to execute the mutated statement. Here, the fitness function should highlight this discrepancy in test case through the fitness it assigns to test case. The existing techniques do not take into account object's state as an important factor hence they can suffer from object's state problem. Bashir and Nadeem [2] propose separating object's state fitness from the branch distance of local variables in genetic algorithm. This way the fitness function properly tells the search process if the object has gained the desired state or if function call sequence in test case needs some modification.

We suggest using the idea of isolating object's state fitness from local variables fitness in evolutionary mutation testing. We call the new cost as *State-based Reachability Cost* or simply SbRC. Thus the proposed cost comprises of state fitness, approach level, and branch distance. This way if a test case fails to execute mutated statement, it gets appropriate fitness value that guides the search accurately. Following is the general representation of this cost;

$$SbRC(T) = (state_fitness, coverage_fitness) \dots(4.1)$$

The equation (4.1) shows that for test case T, the SbRC comprises of two sub-costs including state and coverage fitness. The formulas for calculating these costs are mentioned below;

$$state_fitness = branch_distance(states) \dots(4.2)$$

The *state_fitness* is computed with the help of normalized branch distance, which is calculated for state variables of the object that are found in the program node from where the test case loses its way to the target. The *coverage_fitness* is calculated using following formula;

$$coverage_fitness = [approach_level, branch_distance(vars)] \dots(4.3)$$

The value for *approach_level* is computed by counting the dependent nodes that a test case fails to execute whereas *branch_distance* is computed on local variables. The fitness function computes these costs at the program node from where the test case loses its way to the target.

4.3.2 State-based Necessity Cost

After the test case executes the mutated statement, the mutant must exhibit some deviation in the behavior due to the mutation. If this does not happen, the mutant becomes logically equivalent to the original program. In this situation the fitness function should calculate and assign a cost to the test case. If the test case does not execute the mutated statement, necessity cost of the test case can be any non-zero value. Otherwise if the test case does execute the mutated statement but the

mutation does not cause some infection in the program, necessity cost should be properly calculated and assigned.

The literature survey shows that all the techniques use only branch distance of variables to compute necessity cost. This method works in situations where conventional mutation operators are applied but fails to compute the fitness when object-oriented mutation operators are involved. There are number of mutation operators that do not use variables at all rather they operate on classes and objects. For instance, there are set of operators that cover encapsulation, inheritance, and polymorphism features and they do not involve making any change to the variables of the program. So the necessity cost should be calculated by considering type of the mutation operator.

We propose calculating necessity cost on the basis of the type of mutation operator applied to the mutant. Since there are two types of mutation operators including conventional and object-oriented, we suggest using two different methods to compute necessity cost and we call it as *State-based Necessity Cost* or SbNC in short.

1. SbNC for Conventional Operators: If a conventional mutation operator (for example arithmetic operator replacement) is applied on the code, we suggest computing SbNC with the help of normalized branch distance. The branch distance shall be calculated for variables that are found in the mutated statement like variables declared in a function, returned values of the function or even object's state variables.
2. SbNC for Object-Oriented Operators: If an object-oriented operator (for example overriding method deletion) is applied on the code, we suggest checking if the object type has changed. If the mutation is able to change the type of the object involved in mutated statement, the SbNC should be "0". We do this because when the type of an object is changed, a new object is introduced and will have entirely different states. But if the object type does not change, we suggest computing normalized branch distance, which should be computed on state variables of the concerned object.

4.3.3 Control-oriented Sufficiency Cost

Once the test case executes mutated statement and that mutation causes some change in the mutant program, its effect should be propagated towards the output. Because if this does not happen, the mutant will produce same output and will remain alive. The fitness function then computes the sufficiency cost for the test case that tells how far the deviation propagates after the mutated statement. Bottaci [18] proposes to compute this cost by counting the number of nodes in the control flow graph (from bottom to top) that contain same data states. The fitness function considers those program nodes that exist between the mutated statement and the output node. This method is very simple and effective but it ignores an important aspect of the program's behavior; the flow of control. Bashir and Nadeem [1] propose to use control flow information in mutation testing and shows how useful it can be (Section 4.2). They suggest comparing flow of control of mutant program with original program besides the output.

We give a new name to the proposed method of calculating sufficiency cost and call it as *Control-oriented Sufficiency Cost* or simply CoSC. We propose comparing flow of control of mutant program with the original one besides comparing the data states. The comparison should occur right after the mutated statement till the output node of the program. If the comparison indicates that mutant program produces same output after exhaling different execution path or produces different output after exercising same execution path, we call that mutant as suspicious mutant. Such mutants can help in detecting logical errors from the program being tested.

4.3.4 Overall Fitness

The general representation of the fitness for a test case can be formed by placing together all three costs we have discussed above. All the costs should be kept separate from each other for better representation of the fitness instead of adding them up to generate a single fitness value. From the single fitness value, the

search process cannot determine which part of the test case needs improvement. The general formula of fitness representation is mentioned below;

$$fitness(t) = [SbRC(t); SbNC(t); CoSC(t)] \dots(4.4)$$

4.3.5 An Example

Now we present a code example to demonstrate how our proposed fitness function works. We will generate a mutant and will show how the test cases generated to kill that mutant will be evaluated using our approach. The figure 4.9 provides definition of `GPACalculator` class that declares five data members; `gpa`, `totalPoints`, `totalCredits`, `grades[]`, and `creditHours`. `GPACalculator` provides a constructor and three functions to define data members. The `calculateGPA()` is most important function that calculates GPA for a given exam result. Initially the function computes total earned credit hours and earned points (line 1 to 11). Then the function computes `gpa` using `totalPoints` and `totalCredits` (line 13). The value of `gpa` is then used to evaluate if the student has passed the exam or not. Using this code snippet now we show how our fitness function works.

Mutants

Suppose a mutant is generated using the program in figure 4.9 by applying AOR mutation operator. After applying the mutation, we replace the multiplication (*) sign with addition (+) on line 8 of `calculateGPA()` function. The resultant statement will look like as shown below;

```
8. pts = 2 + creditHours[g]; //AOR Mutation (+)
```

Before we show how our proposed fitness function operates, we want to highlight an important aspect of the program in figure 4.9. The `calculateGPA()` function has a logical error on line 16 because instead of returning "Failed" the function returns "Passed" again. Both the original and mutant programs contain this error hence whatever changes we make in the computation of `gpa`, the function

```

public class GPACalculator {

    float gpa;
    int totalPoints, totalCredits;
    int creditHours[];
    char grades[];

    public Result() {
        gpa = 0.0f;
        totalPoints = 0;
        totalCredits = 0;
        creditHours = 0;
    } //Result() ENDS

    public void setTotalPoints(int p) {
        totalPoints = p;
    } //setTotalPoints() ENDS

    public void setCreditHours(int ch) {
        totalCredits = ch;
    } //setCreditHours() ENDS

    Public void setExamResult(char g[], int ch[]) {
        grades = g;
        creditHours = ch;
    } //setExamResult() ENDS

    public String calculateGPA() {
1. for(int g=0 ; g<grades.length ; g++) {
2.     int pts = 0;
3.     if( grades[g]=='A' ) {
4.         pts = 4 * creditHours[g];
5.     } else if( grades[g]=='B' ) {
6.         pts = 3 * creditHourss[g];
7.     } else if( grades[g]=='C' ) {
8.         pts = 2 * creditHours[g]; //AOR Mutation (+)
9.     } else {
10.        pts = 0;
11.    } //if-else ENDS
12. totalPoints += pts;
13. totalCredits += creditHours[g];
14. } //for ENDS
15. if( totalCredits>0 ) {
16.     gpa = totalPoints / totalCredits;
17. } //if ENDS
18. if( gpa>=2.5 ) {
19.     return "Passed";
20. } else {
21.     return "Passed"; //Logical error (bug)
22. } //if-else ENDS
23. } //calculateGPA() ENDS
24. } //GPACalculator ENDS

```

FIGURE 4.9: Code Snippet to Demonstrate Proposed Fitness Function

<p>Test Case - T1 <code>\$t1=GPAcalculator():\$t1.setTotalPoints(int):\$t1.setCreditHours(int):t1.setExamResult(char[],int[]): t1.calculateGPA() @ 0, 0, {'A','B'}, {3,3}</code></p> <p>Test Case - T2 <code>\$t2=GPAcalculator():\$t2.setTotalPoints(int):\$t2.setCreditHours(int):t2.setExamResult(char[],int[]): t2.calculateGPA() @ 0, 0, {'C','C'}, {2,2}</code></p> <p>Test Case - T3 <code>\$t3=GPAcalculator():\$t3.setTotalPoints(int):\$t3.setCreditHours(int):t3.setExamResult(char[],int[]): t3.calculateGPA() @ 0, 0, {'C','B'}, {3,3}</code></p>

FIGURE 4.10: Test Case Set

always returns "Passed". Due to this reason almost all the mutants generated from `calculateGPA()` function remain alive and cannot be killed.

Test Cases

We take three instances of test cases shown in figure 4.10 that will attempt to kill the mutant we have generated from the program presented in figure 4.9. If an instance does not kill the mutant we will show how our proposed fitness function evaluates it.

We use a specific format for presenting the test cases that Tonella [4] proposes in his work. When we execute them on the program (shown in figure 4.9) and on its modified version we notice that all three test cases produce same output on both versions, which is the string value "Passed". The reason being, the program under test contains a logical error.

State-based Reachability Cost (SbRC)

The T1 fails to satisfy reachability condition because it does not execute mutated statement, which is on line 8. Actually for a test case to execute this statement, the `grades[]` array should contain one 'C' in it. Only then predicate of `if` statement at line 7 can be satisfied. The mutated statement is enclosed by two conditions (line 1 and line 7) and T1 executes only one of them (line 1) so *approach_level* of T1 is 1. The predicate at line 7 involves state variable `grades[]` so as per our proposal we calculate *state_fitness* for T1, which we assume for now is 0.5. The

branch_distance remains 0 here because the predicate does not contain any local variable of the function. The SbRC for T1 is as follows;

$$SbRC(T1) = (0.5, [1, 0])$$

The fitness shows that T1 fails to execute the mutated statement because object in T1 is not in desired state. The other two test cases execute the mutated statement because the state variable `grades[]` contain a value 'C' so the SbRC for T2 and T3 is as follows;

$$SbRC(T2) = (0, [0, 0])$$

$$SbRC(T3) = (0, [0, 0])$$

State-based Necessity Cost (SbNC)

Unless a test case executes mutated statement, the state-based necessity condition cannot be satisfied. The T1 fails to reach the mutated statement hence it cannot satisfy this condition either so we can assign any constant 'c' to T1 as its SbNC;

$$SbNC(T1) = c$$

If we analyze T2, we find that it executes mutated statement hence it is a strong candidate of satisfying state-based necessity condition. But as we know T2 has both values in `creditHours[]` as '2' so either program computes "`2 * creditHours[g]`" in original version or "`2 + creditHours[g]`" in modified version, both expression result in value 4. That means although T2 satisfies SbRC yet it does not satisfy SbNC because mutated statement causes no infection in the modified version of program. So the fitness function computes SbNC for T2 and for this it computes *branch_distance* of state variable `creditHours[]` because it is involved in mutated statement. For now we assume a value of 0.5;

$$SbNC(T2) = 0.5$$

The case with T3 is simple because it satisfies both SbRC by executing the mutated statement and also SbNC by introducing infection in the modified version of the program so fitness function assigns 0 as SbNC to T3;

$$SbNC(T3) = 0$$

Control-oriented Sufficiency Cost (CoSC)

Until and unless a test case executes mutated statement, it cannot satisfy control-oriented sufficiency cost. The T1 fails to reach the mutated statement hence it cannot satisfy this condition either so we can assign any constant 'c' to T1 as its CoSC. Since test case does not execute mutated statement, so no change can be observed so we declare it as normal mutant;

$$CoSC(T1) = (c, normal)$$

The T2 also face similar situation where it although executes the mutated statement but it does not cause any change. In this case fitness function assigns any constant value 'c' as CoSC to T2. Since no change is introduced in the mutant, the status of mutant remains normal;

$$CoSC(T2) = (c, normal)$$

The T3 satisfies SbNC but we notice that the output of original and mutant programs still remains the same. This happens because of the presence of logical error in the program under test at line 16. So the fitness function computes CoSC for T3 and compares data states of both the versions after the program node that contains mutation. The fitness function finds only one equal data state so it assigns '1' to T3 as its CoSC. After that fitness function compares flow of control and finds that on original program T3 executes 12, 13, 14, and 15 whereas on mutant program, it executes 12, 13, 14, and 16. Since T3 produces same output on both programs but exercises different execution path, it tags the mutant as suspicious;

$$CoSC(T3) = (1, suspicious)$$

Overall Fitness

After presenting the procedure of how our proposed fitness function computes costs for different conditions, we present overall fitness of all the test cases as follow;

$fitness(T1) = [(0.5, [1, 0]) ; c ; (c, normal)]$

$fitness(T2) = [(0, [0, 0]) ; 0.5 ; (c, normal)]$

$fitness(T3) = [(0, [0, 0]) ; 0 ; (1, suspicious)]$

The above representation seems too complicated at first look but this kind of comprehensive and detailed representation not only enhances the understandability about fitness of the test case but it also guides the search quite well. The long fitness value is capable to represent fitness of different aspects of a test case in such a way that all the strengths and weaknesses of the test case are highlighted in it.

4.4 Two-way Crossover Method

In this section we present new two-way crossover method for genetic algorithm that uses objects state fitness to produce new test cases. To demonstrate how our proposed two-way crossover works and helps the targets to converge quickly, we have used an example of `Stack` class. The two-way crossover method works in conjunction with our proposed fitness function (Section 4.3).

In our previous work [13] we propose a novel fitness function for evolutionary mutation testing of object-oriented programs (see Section 4.3 for details). We introduce two new values in a test case fitness including object's state and control flow information. Both of these values are calculated from execution traces of program under test and are represented as separate costs in fitness. With initial experiments and analysis of our approach we discover that although the proposed fitness function provides information about the weakness in a test case (whether object in the test case has gained desired state through state fitness) yet the next phase of Genetic algorithm (crossover) fails to utilize it due to its inherent nature. Whether Genetic algorithm uses one-point crossover or two-point crossover, it does not consider object's state fitness to produce offsprings (new test cases for next iteration). The process has to wait until biological mutation uses state fitness to form new method call sequence in a test case, which may help object in gaining

```
public class Stack {
    private int top = 0;
    private int[] elements;
    ...
    ...
    public int pop() {
        1. int element = -1;
        2. if( top>=2 ) {
        3.     top = top - 1;
        4.     element = elements[ top ];
        5. } //END if STATEMENT
        6. return element;
    } //END pop() METHOD
    ...
    ...
} //END Stack CLASS
```

FIGURE 4.11: Code Snippet of Stack Class

desired state. If we can enable crossover to use the object's state fitness to produce offsprings, we may be able to push mutation score further up in limited number of iterations. For this we have proposed a new method for performing crossover using state fitness information.

Consider code snippet of **Stack** class in figure 4.11. The class has two private data members including **top** pointer and **elements[]** array to contain integers in stack formation. There are several methods defined in the class to simulate stack's behavior but we are interested in only **pop()** method definition for now. The **pop()** method pops out the top most element from the stack but there is an additional constraint on it. This method will remove and return the top element, only if there are at least 2 elements present in the stack object.

Now we generate a mutant from **Stack** class such that the line 3 of **pop()** method contains a mutation. We want to generate a test case that will kill that mutant by satisfying all three conditions (reachability, necessity, and sufficiency). For a test case to kill this mutant, the object of **Stack** class is required to invoke **push(int)** method (its definition is not shown in figure 4.11) for at least 2 times before invoking **pop()** method otherwise test case will fail to execute the mutated statement. Suppose the genetic algorithm generates following two test cases as initial population;

Initial Population

$T1: \$s=Stack():s.push(int):s.mX():s.mY():s.pop() @ 22$

$T2: \$s=Stack():s.mX():s.push(int):s.mZ():s.mA():s.pop() @ 6$

Both the test cases cannot kill the mutant because they contain just one call to `push(int)` method hence object does not gain the desired state before calling `pop()`. Since both the test cases fail to achieve the target, fitness function comes into play for their evaluation. Our proposed fitness function [13] will be able to highlight the discrepancy in them by assigning a non-zero value to *state_fitness* as shown below;

Fitness of Initial Population

$Fitness(T1): [(0.5, [1, 0]) ; c ; (c, normal)]$

$Fitness(T2): [(0.5, [1, 0]) ; c ; (c, normal)]$

The 0.5 fitness shows that object is not in desired state and method call sequence needs modification. After that genetic algorithm performs crossover to form new population for next iteration. The one-point crossover picks a random point and bisects both the test cases on that. The new test cases are generated by merging the first part of first test case to the second part of second test case and similarly by merging first part of second test case and second part of first test case as shown below;

Current Population

$T1: \$s=Stack():s.push(int):\underline{s.mX():s.mY():s.pop()} @ 22$

$T2: \underline{s.mX():s.push(int):s.mZ():s.mA():s.pop()} @ 6$

New Population

$T1: \$s=Stack():s.push(int):s.mZ():s.mA():s.pop() @ 22$

$T2: \$s=Stack():s.mX():s.push(int):s.mX():s.mY():s.pop() @ 6$

If we carefully analyze the new population the new test cases have the same issue as the old test cases had. The reason of this problem is that the crossover does not consider the state fitness and does not try to repair test cases accordingly. To solve this problem, we propose a new method of performing crossover, which we

call **two-way crossover** that takes into account object's state fitness to form new test cases.

In order to perform crossover, first we use tournament selection method to pick the good test cases from population. Then we form pairs of test cases from the selected test cases and crossover is performed on the pairs. In our proposed two-way crossover method, we merge both the segments of first test case to both the segments of second test case as shown below;

Current Population

T1: \$s=Stack():s.push(int):s.mX():s.mY():s.pop() @ 22

T2: \$s=Stack():s.mX():s.push(int):s.mZ():s.mA():s.pop() @ 6

New Population

T1: \$s=Stack():s.push(int):s.mX():s.push(int):s.pop() @ 22, 6

T2: \$s=Stack():s.push(int):s.mZ():s.mA():s.pop() @ 22

T3: \$s=Stack():s.mZ(): s.mA():s.mX():s.mY():s.pop() @

T4: \$s=Stack():s.mX():s.push(int):s.mX():s.mY():s.pop() @ 6

The two-way crossover doubles the test cases because it merges both the segments of each test case with both the segments of other. The new set of test cases contains a test case (T1) that gains desired object state before invoking `pop()` method. Similarly even strict state requirement can be fulfilled in next iteration by further crossing over test cases of this population. This `Stack` class has a simple state requirement but experiments have shown that the two-way crossover works equally well when situation demands even more complicated object's state.

The two-way crossover has tendency to generate more test cases for next iteration as compared to conventional crossover. Every pair involved in crossover can generate up to four new test cases after performing two-way crossover. We adapt two ways to control this rapid increase in test cases. First we perform two-way crossover only on those test cases whose objects have state problem and have a non-zero state fitness. So if in a pair of test cases, both test cases have state fitness as zero, we do not perform two-way crossover on them. The reason is that

if test case does not have state problem, this means the issue lies within the variable values passed as argument to method under test that needs modification and crossover cannot help here. Secondly, in worst-case scenario, if all the test cases of a given iteration have non-zero state fitness, then the next iteration will receive double amount of test cases and so on. To control this, when genetic algorithm performs biological mutation to ensure search does not get stuck in local optima, we reduce number of test cases back to the size of population, tester sets in the beginning. By these two ways, we control the number of test cases during execution of Genetic algorithm.

4.5 Adaptable Mutation Method

Mutation operation in genetic algorithm prevents the search process to get stuck in local optima. It is not possible to set one fix mutation rate that can produce best results as every program is different in nature and requires different type and range of input data. If mutation rate is too high, it will become close to random testing and if mutation rate is too low, the test cases will become similar after a certain number of iterations.

In evolutionary mutation testing, genetic algorithm can experience similar situation. Consider the code snippet presented in figure 4.12 of `Calculator` class in which `add()` method is defined. This method takes two integer parameters, adds them, and returns the result to caller. At line 1 it checks if either of two parameters is non-zero and if so then it performs the addition operation. Now on this condition a mutant can possibly be generated by applying relational operator replacement (ROR) as follows;

```
if( a==0 || b!=0 )
```

In order to kill the mutant having above mentioned condition as compared to the one that `Calculator.add()` method has, either `a` and `b` both have to be 0 or `a` has to be 1 and `b` has to be 0. For all other possible values the mutant remains

```
public class Calculator {  
    private int result;  
    public Calculator() {  
        result = 0;  
    } //END Calculator() CONSTRUCTOR  
    public int add( int a, int b ) {  
1.   if( a!=0 || b!=0 ) {  
2.       result = a+b;  
3.   } //END if STATEMENT  
4.   return result;  
    } //END add() METHOD  
} //END Calculator CLASS
```

FIGURE 4.12: Code Snippet of Calculator Class

alive. In situations like these, crossover cannot help in generating the required inputs because method under test needs specific set of values. Instead frequent mutations can produce required input values rather quickly. On the other hand there are certain applications like **Stack** (figure 4.11) that needs some time to evolve test cases to satisfy complex predicates to kill the mutants. In this case high mutation rate can result in diverging the search away from its target. So we can state that a fixed mutation rate in genetic algorithm can cause problems hence we need an adaptable mutation frequency that can adapt to the situation as the situation demands.

Literature survey shows some research has been done by Baudry [40], Xie et al. [41], Dharsana and Askarunisha [42], Alsmadi [43], and Wang et al. [44] that provides basis to pick the right configuration for genetic algorithm and adapt mutation rate as per situation. Mainly these approaches decide about pre-maturity in population by analyzing the solutions (test cases) and comparing them whereas some approaches are specifically for structured paradigm. These approaches are not appropriate to help in object-oriented programs and for mutation based evolutionary testing where the goal is a mutant not just a statement or a path to be executed.

In this case our state-based & control-oriented fitness function provides comprehensive information to genetic algorithm to decide if mutation rate needs an adjustment for next iteration or current configuration is fine. For understanding,

consider the following general fitness of a test case 't';

$$fitness(t) = [SbRC(t); SbNC(t); CoSC(t)]$$

$$fitness(t) = [state_fitness(t), coverage_fitness(t); SbNC(t); CoSC(t)]$$

Once iteration completes, we propose analyzing fitness of all the test cases if the mutant (target) remains alive. If *state_fitness* of all the test cases remain 0.0, it shows test cases have gained required state or the program under test does not have state requirement. Now crossover of test cases cannot help because it does not modify arguments of method under test. In such a case increasing mutation frequency can save effort in producing required values for method under test. We propose decreasing interval between two mutations by 1 after unsuccessful iteration for programs that do not have state requirement or having 0.0 as *state_fitness*.

Chapter 5

Tool Support

In this chapter, we present the automation details related to our proposed work. The proposals have been implemented in a tool called eMuJava (**e**volutionary **M**utation testing of **J**ava programs). This tool is efficient and is able to evaluate the fitness of test cases by considering the object's state fitness. eMuJava can classify mutants as suspicious or normal on the basis of control-flow information obtained through execution traces. Besides that this tool performs two-way crossover and adaptable methods using state-based fitness of a test case. eMuJava also supports multiple methods for test case generation including random testing, genetic algorithm with standard fitness function, genetic algorithm with state-based & control-oriented fitness function, and genetic algorithm with state-based & control-oriented fitness function, two-way crossover, and adaptable mutation methods. This chapter presents automation details including algorithms used, tool architecture, responsibilities of different tool components, supported test case generation method, and screenshots of GUIs (Graphical User Interface).

5.1 eMuJava Tool

eMuJava has been implemented in Java programming language and is capable of testing Java based programs. eMuJava does not rely on any other third party

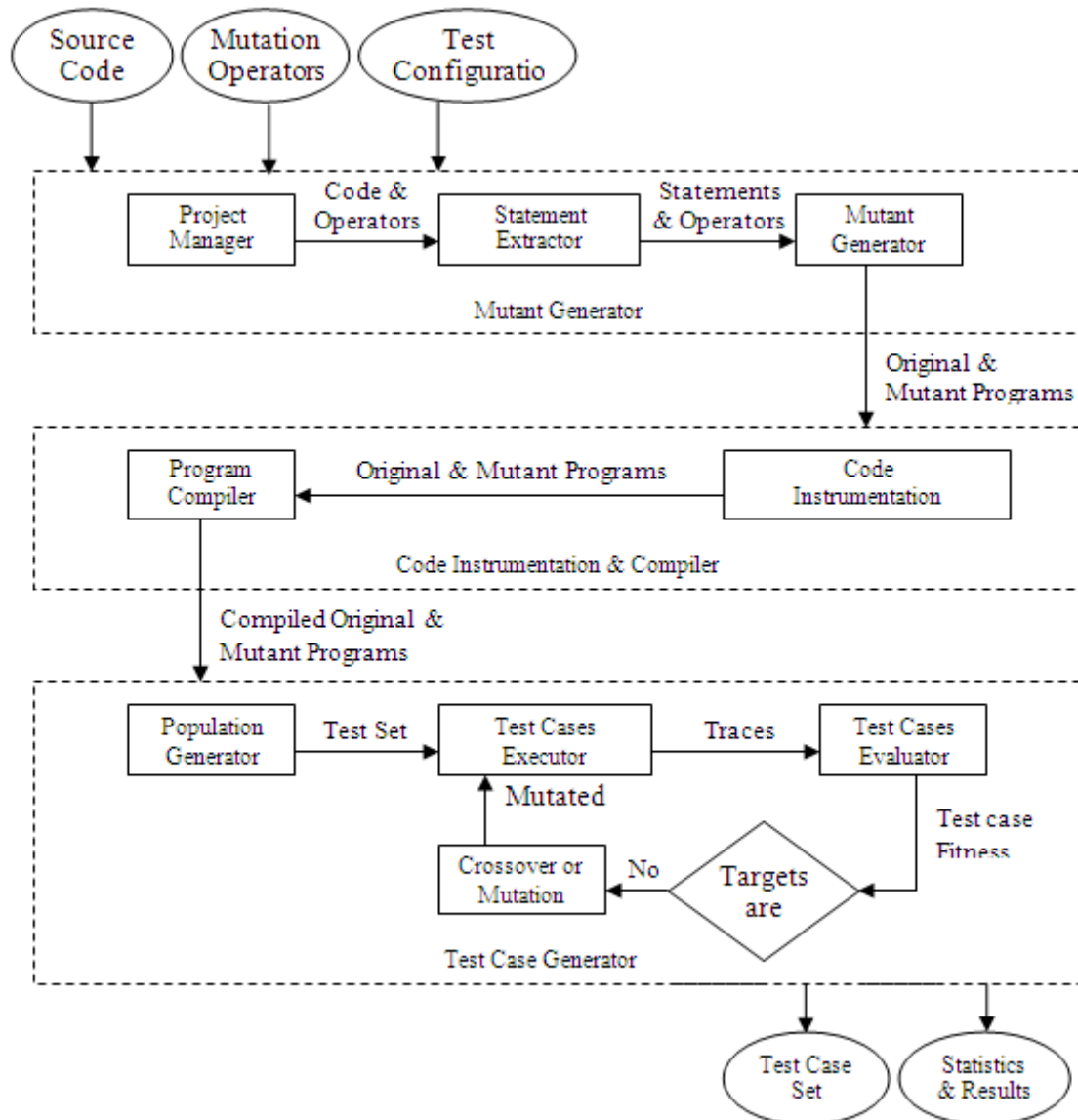


FIGURE 5.1: eMuJava Architecture

or freeware component that makes it a complete and self-dependent automation. The figure 5.1 presents architecture in terms of modules that the tool consists of. There are three large modules that collectively work and share information among them to perform testing of the programs. eMuJava is a prototype tool and supports limited set of Java constructs (see Appendix C for more details). It provides support for 10 mutation operators at this time out of which 5 are for structured paradigm and 5 are for object-oriented paradigm. eMuJava cannot test more than 2 Java classes at a time. Next we present details about eMuJava tool including its inputs, outputs, and responsibilities of each component.

5.1.1 eMuJava Architecture

The implementation is complex in nature and based on three large components (Mutant Generator, Code Analyzer and Instrumentation, Test Case Generator) with a set of responsibilities assigned to each of them. Main inputs of eMuJava include Java program under test and mutation operators whereas after processing it produces test case set and results of experiment as result. Next we present inputs, outputs, and responsibilities of each component of the tool in detail.

Mutant Generator

The first component of tool is Mutant Generator from where the process of test case generation begins. It takes two inputs from the environment (usually provided by the tester) including source code to be tested and the mutation operators. User can load the source code in the tool by browsing through the directory structure of testers computer and mutation operators can be selected by choosing them from the list provided on tool interface. The mutant generator then starts generating all possible mutants using the mutation operators. This process can take a while to complete because mutants can be large in number depending upon the Java classes under test and mutation operators selected by the tester. The execution of mutant generator completes on generating all possible mutants whereas each mutant contains single mutation.

Code Analyzer & Instrumentation

The output of Mutant Generator becomes the input of this component. The major responsibility of this component is to perform analysis of the code and instrument it so later on when the test cases are run, the traces can be generated and logged. To perform these responsibilities, the component relies on its sub-components. Initially, scanner and parser tokenize the code and parse the tokens to collect information about the statements they appear in. After that the class members extractor group the pieces back to form more meaningful segments like data members, constructors, methods or functions. Towards the end, additional code is inserted by this component to keep track of execution flow when test cases

are run on the code. This component generates instrumented Java code with no syntax error so it can be compiled and later executed. Finally, the instrumented mutants are compiled by this component.

Test Case Generator

This is the third and last component of eMuJava and it receives compiled instrumented mutants as input and starts the process of test case generation. eMuJava can use four different approaches for test case generation; random testing, genetic algorithm with standard fitness function, genetic algorithm with state-based & control-oriented fitness function, and genetic algorithm with state-based & control-oriented fitness function, two-way crossover, and adaptable mutation methods. This component receives the approach to be used for test case generation as input from the user along with other configuration details including number of test cases to be used as initial population, number of iterations to perform, crossover and mutation rate and so on. This component produces final test case set, results, and statistics of the experiment.

5.1.2 eMuJava Operations

This section explains how eMuJava performs various operations to conduct an experiment. The details of testing process from the input it receives until the output it produces.

Mutant Generation

Once eMuJava completes the process of identifying statements and tokens that form a given statement, it starts generating mutants. eMuJava generates mutants on the basis of mutation operators, the tester provides as input. eMuJava offers ten mutation operators such that five are structured and five are object-oriented mutation operators. The five mutation operators for structured paradigm are chosen from the work Offutt et. al [37]. We have chosen these mutation operators because Wong et. al [45] prove that the impact of applying these selective mutation

operators is almost equal to applying the whole mutation operators set. The selective mutation operators include the following:

- Absolute value insertion (ABS)
- Arithmetic operator replacement (AOR)
- Logical connector replacement (LCR)
- Relational operator replacement (ROR)
- Unary operator insertion (UOI)

The object-oriented mutation operators are chosen from the research of Offutt et. al [39]. We have chosen these operators such that we have a representation for every object-oriented feature:

- Overriding method deletion (IOD)
- new method call with child class type (PNC)
- Overloading method deletion (OMD)
- Member variable initialization deletion (JID)
- Reference comparison and content comparison replacement (EOC)

Population Generation

When eMuJava begins the test case generation process, it generates initial population of solutions (test cases) randomly. Tester provides the size of the population as input whereas the default size is 50. For every given mutant, eMuJava looks for the method containing the mutated statement and that method becomes the method under test. We have used the test case template proposed by Tonella [4] as shown in the figure 5.2.

The tool generates the test cases using the same template in which it creates an object of class under test followed by a random number of calls to some other

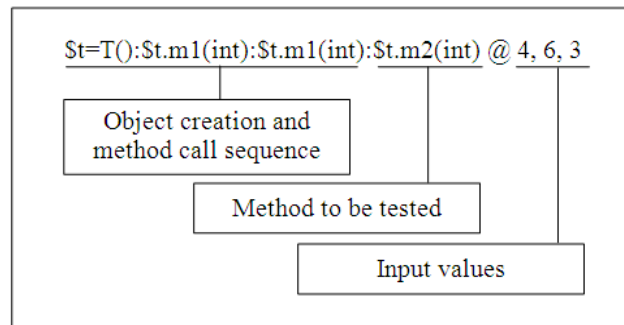


FIGURE 5.2: Anatomy of a Test-case for Object-Oriented Program Testing [2]

method(s) of the same class and finally it calls to the method under test (method with mutation). Later the tool generates required input parameters randomly. eMuJava supports all the primitive data types of Java programming language as well as generation of String literals of random length. If the class under test depends on the object of another class then the tool generates the object of other class and makes call to some methods to gain the desired state.

Next we present details about three more operations that eMuJava performs including fitness evaluation, crossover, and biological mutation. These are standard operations, which genetic algorithm performs; therefore, eMuJava does not perform them when tester chooses to use random testing for test case generation.

Fitness Evaluation

Our tool evaluates fitness of a test case using execution traces the test case generates. It evaluates fitness by two methods depending upon the approach tester chooses for test case generation. If tester chooses to generate test cases using standard Genetic algorithm, then eMuJava uses fitness evaluation method of Bottaci [18]. On the other hand eMuJava uses our state-based and control-oriented fitness function [13] for test case evaluation.

Crossover

On completion of the iteration, if the target (mutant) remains alive, the test cases are gone through crossover to generate off-springs for next iteration. eMuJava uses tournament selection method to select test cases for crossover, after which, pairs of test cases are formed. The tool uses two methods to perform crossover

depending on type of test case generation strategy user chooses in the beginning of the process. First one is single point crossover, which is performed on a randomly chosen point from the test cases. The second method is two-way crossover, which we have proposed (Section 4.4) and it uses state fitness of test cases to perform crossover.

Biological Mutation

The biological mutation helps minimizing the chances of the search process getting stuck in local optima. Mainly on a test case, the tool performs two types of mutations. If a test case has state fitness of 0.0 that means the object is already in a desired state and its method call sequence does not need a change. In this case eMuJava only mutates the input parameters of the method under test. Besides that eMuJava increases mutation frequency as the test cases need correct combination of input values for the method under test. This is what we call adaptable mutation method that adapts to the situation by increasing the mutation frequency when required. In other case if the object is not in the desired state (state fitness has a non-zero value) then the tool not only changes the input parameters of method under but it also changes the method call sequence. One or more method invocations may be introduced as well as one or more method invocation may be removed from the sequence. Once the mutation operation completes on all the test cases, new population is ready for next iteration of genetic algorithm.

5.2 eMuJava Algorithms

In this section we present the algorithms that we have devised to automate the proposals implemented in eMuJava. These algorithms have been divided into six sections including GenMutants, GenPopulation, ExecTestCases, EvalTestCases, TWCrossoverTests, and MutateTests. The GenMutants generates mutants, GenPopulation generates initial population, and ExecTestCases executes test cases


```

Algorithm:    GenMutants
Input:      class names CN
                mutation operators MO
Output:     set of mutants MS
Declare:    CN instance CN1
                MO instance M
Begin:      GenMutants
  1. for each mutation operator M in MO do
  2.   find application of M in CN1
  3.   if M applies on CN1 then
  4.     for each variation of M do
  5.       make copy of CN1
  6.       apply variation of M on copy of CN1
  7.       add copy of CN1 to MS
  8.     end for
  9.   end if
 10. end for
 11. return MS
end GenMutants

```

FIGURE 5.3: *GenMutants* Algorithm

and records the execution traces. Then EvalTestCases uses traces to evaluate fitness of test cases and if mutant remains alive, TWCrossoverTests and MutateTests algorithms repair test cases.

5.2.1 GenMutants

The figure 5.3 shows pseudo code of the GenMutants algorithm. The GenMutants algorithm generates mutants by applying the mutation operators set provided by the user on program under test. Input of this algorithm includes program under test, which comprises of one or two Java classes and mutation operators including operators for structured and object-oriented paradigm. This algorithm applies mutation operators on the code and generates all possible mutants by applying all possible variations of a given operator.

5.2.2 GenPopulation

The figure 5.4 shows the pseudo code of the GenPopulation algorithm. The GenPopulation algorithm generates initial population. It takes population size, class

```

Algorithm:   GenPopulation
Input:     class name CN
               constructors list CL
               methods list ML
               population size P
Output:    test case set TCSet
Declare:   CNinstance CN1
               TestCaseinstance T
Begin:     GenPopulation
1.   for each method M in ML do
2.     for 1 to P do
3.       select C from CL randomly
4.       instantiate CN1 using C and add to test case T
5.       generate integer N for number of methods in call sequence randomly
6.       for 1 to N do
7.         select M from ML randomly where M does not have mutation
8.         generate arguments for M randomly
9.         add M to test case T
10.      end for
11.      generate arguments for M that contains mutation
12.      Add M to test case T
13.      add test case T to TCSet
14.    end for
15.  end for
16.  return TCSet
end GenPopulation

```

FIGURE 5.4: *GenPopulation* Algorithm

name, constructors and methods list and produces a set of test cases to be executed on the class under test. The *GenPopulation* generates number of test cases for each target (mutant to be killed) equal to population size. It randomly selects the constructor from the list of available constructors and randomly adds method call sequence before adding a call to method under test. The method under test is one that contains mutated statement. The algorithm randomly generates required arguments for the selected constructor and methods.

5.2.3 ExecTestCases

The figure 5.5 shows the *ExecTestCases* algorithm that generates, compiles, and executes the *Driver* class. The *Driver* class is one that contains a test case, which we want to run on original and mutated programs. After executing the *Driver* class, the algorithm records the execution traces as well. The input of *ExecTestCases* algorithm contains set of test cases and instrumented class under

Algorithm:	ExecTestCases
Input:	instrumented source code test case set <i>TCSet</i>
Output:	execution traces <i>Traces</i>
Declare:	driver file <i>Driver</i>
Begin:	ExecTestCases
	1. add header for <i>Driver</i> class
	2. add header for <i>Main()</i> method
	3. for each <i>T</i> in <i>TCSet</i> do
	4. add <i>T</i> in <i>Main()</i> method
	5. end for
	6. add terminator for <i>Main()</i> method
	7. add terminator for <i>Driver</i> class
	8. compile <i>Driver</i> class
	9. execute <i>Driver</i> class
	10. record execution <i>Traces</i> of <i>Driver</i> class
	11. return <i>Traces</i>
	end ExecTestCases

FIGURE 5.5: *ExecTestCases* Algorithm

test. After receiving the input, the algorithm generates one `Driver` class for each test case and then compiles the class. On successful compilation, the algorithm executes the `Driver` class and records the execution traces, which is the output of this algorithm as well.

5.2.4 EvalTestCases

The figure 5.6 shows the `EvalTestCases` algorithm. The `EvalTestCases` takes execution traces, target (mutant to be killed), and set of test cases as input. On the basis of target the test cases are evaluated. For a given test case, algorithm evaluates three costs against its ability to fulfill three conditions (reachability, necessity, and sufficiency) to meet a target. The algorithm also uses object's state and control-flow information to evaluate a test case. The output of this algorithm is fitness set of test cases.

```

Algorithm: EvalTestCases
Input: test case set TCSet
          target mutant Traget
          execution traces Traces
Output: test case evaluation results Fitness
Declare: test case fitness set Fitness
          test case fitness F as String
          approach_level as integer
          state_fitness as double
          coverage_fitness as double
          reachability_cost as String
          necessity_cost as double
          sufficiency_cost as String

Begin: EvalTestCases
1. for each T in TCSet do
2.   read traces of T from Traces
3.   if T fails to execute mutation then
4.     evaluate approach_level
5.     if predicate contains state variables then
6.       evaluate branch_distance of state variables as state_fitness
7.     end if
8.     if predicate contains local variables then
9.       evaluate branch_distance of local variables as coverage_fitness
10.    end if
11.    assign approach_level, state_fitness, coverage_fitness as reachability_cost
12.    assign a constant C to necessity_cost
13.    assign a constant C to sufficiency_cost
14.  else if T fails to introduce infection then
15.    assign reachability_cost as 0
16.    if mutation involves object then
17.      evaluate branch_distance of object's state variables as necessity_cost
18.    else if mutation does not involve object then
19.      evaluate branch_distance as necessity_cost
20.    end if-else
21.    assign a constant C to sufficiency_cost
22.  else if T fails to propagate infection in output then
23.    assign reachability_cost as 0
24.    assign necessity_cost as 0
25.    count the nodes having same values after mutation as sufficiency_cost
26.    if execution path differs in original and mutated programs then
27.      Add flag with T as suspicious
28.    end if
29.  else do
30.    assign reachability_cost = necessity_cost = sufficiency_cost = 0
31.  end if-else-if
32.  F = [reachability_cost; necessity_cost; sufficiency_cost]
33.  add F to Fitness set
34. end for
35. return Fitness
end EvalTestCases

```

FIGURE 5.6: *EvalTestCases* Algorithm

```

Algorithm:    TWCrossoverTests
Input:      test case set TCSet
Output:    test case set TCSet
Declare:   test case pair T1 and T2 as TestCase
Begin: TWCrossoverTests
  1. for each pair of test case in TCSet do
  2.   pick T1 and T2 from TCSet
  3.   if T1 and T2 have state_fitness as 0 then
  4.     perform two-way crossover and add all 4 offsprings in TCSet
  5.   else
  6.     perform conventional crossover and add both offsprings in TCSet
  7.   end if-else
  8. end for
  9. return TCSet
end TWCrossoverTests

```

FIGURE 5.7: *TWCrossoverTests* Algorithm

5.2.5 TWCrossoverTests

The figure 5.7 shows the pseudo code of the algorithm. The *TWCrossoverTests* algorithm performs crossover between pair of selected test cases. The *TWCrossoverTests* receives test cases and their fitness as input. The test cases are selected on the basis of their fitness using tournament selection. The test cases, which are extremely weak are filtered and dropped before crossover. *TWCrossoverTests* algorithm pairs them up such that each test case in a pair either has 0 *state_fitness* or non-zero *state_fitness*. If the pair of test cases have 0 *state_fitness*, the algorithm performs single-point crossover on them and generates two new test cases. If the pair of test cases have non-zero *state_fitness*, *TWCrossoverTests* performs two-way crossover on them and generates four new offsprings. The output of this algorithm is new test case set, which it generates after performing crossover.

5.2.6 MutateTests

The figure 5.8 shows the *MutateTestCases* algorithm. If the target (mutant) is not achieved the *MutateTestCases* algorithm is used to mutate the test cases to achieve the target. The *MutateTestCases* algorithm takes test case set as input and produces mutated set of test cases as output. This algorithm analyzes every test case and mutates arguments of method under test if the *state_fitness* is 0

```

Algorithm:    MutateTests
Input:      test case set TCSet
Output:     mutated test cases MTCSet
Declare:    test case T
Begin: MutateTests
  1. for each T in TCSet do
  2.   if T.state_fitness is 0 then
  3.     mutate argument of method under test in T
  4.     increase mutation rate
  5.   else
  6.     mutate method call sequence of T
  7.   end if-else
  8.   add test case T in MTCSet
  9. end for
 10. return MTCSet
end MutateTests

```

FIGURE 5.8: *MutateTests* Algorithm

and increase mutation frequency. Otherwise the algorithm mutates method call sequence if *state_fitness* is non-zero.

5.3 Supported Test Case Generation Techniques

eMuJava supports following four techniques to generate test cases for mutation based testing.

1. *Random Testing*: The test cases and input data are generated randomly in all iterations. If iteration fails to kill a mutant, the tool does not perform either crossover or mutation to repair the test cases rather it generates new population of test cases for next iteration.
2. *Genetic Algorithm with Standard Fitness Function*: This approach uses genetic algorithm to generate and evolve test cases. In first iteration test cases and input data are generated randomly. If a mutant (target) remains alive after the iteration, eMuJava evaluates test cases using standard fitness function and then it performs single-point crossover and after certain number of iterations, it performs biological mutation to repair the test cases.

3. *Genetic Algorithm with State-based & Control-oriented Fitness Function:* This approach is identical to the approach we have discussed in 2) but the only difference it has, lies in its fitness function. This variant of genetic algorithm implements state-based & control-oriented fitness function [13] for evaluating test cases.
4. *Genetic Algorithm with State-based & Control-oriented Fitness Function, Two-way Crossover, and Adaptable Mutation Methods:* This approach is also similar to the approach we present in 2) but besides genetic algorithm, it implements state-based & control-oriented fitness function, two-way crossover, adaptable mutation methods. Hence this approach implements all of our proposals of this research.

5.4 eMuJava Design Model

This section presents the design class diagram of eMuJava tool in figure 5.9. Class diagram shows business classes, their attributes, operations, and associations among these classes.

5.5 eMuJava Configuration

eMuJava provides some useful configurations to perform the experiments efficiently. It allows the tester to provide test configuration testing; loading the Java class(es) for testing, selection of mutation operators, selection of test case generation approach, population size, maximum number of iterations, crossover type, and biological mutation rate. Besides that there are some other notable configuration options that eMuJava provides to the tester. The are listed below;

- *Integer Range:* eMuJava uses this range to generate integer numbers to be passed as arguments to the method calls in a test case. The integer remains between 0 and specified range. For example if tester sets the range be '100'

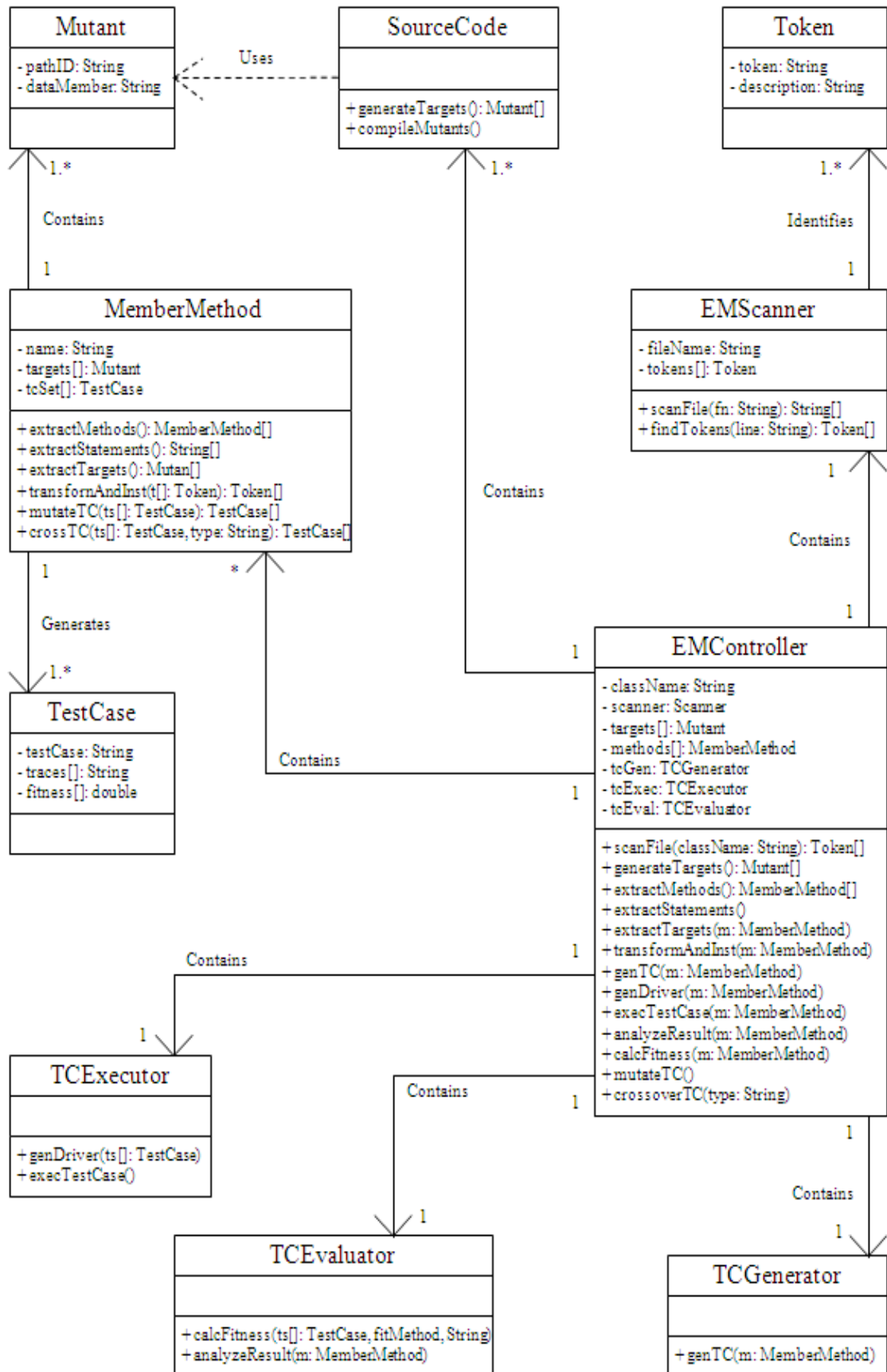


FIGURE 5.9: eMuJava Design Class Diagram

then the generated integer will be between "-100" and "100". Same range is used to generate fractional values where required.

- *Character Range*: The character range is used to generate ASCII characters at runtime. For a single instance only one character is generated randomly. To generate strings, eMuJava first picks a length and then characters to satisfy that length are generated randomly.
- *GA Timeout*: During test case execution, eMuJava generates driver classes. After issuing compile command, eMuJava waits for the .class files to be generated but in case if driver class or the instrumented programs have some syntax error, no .class files will be generated. In this situation the tool may went on waiting for indefinite amount of time. To handle this issue, this parameter provides a timeout limit in milliseconds. Once it exhausts, tool proceeds further and continues execution.
- *GA Max Threads*: eMuJava is multi-threaded tool that supports running 'n' instances of a given test case generation technique simultaneously. By default eMuJava executes single thread but on more powerful machines to get full advantage of powerful resources more than just one thread can be run. Using one thread on an Intel Core 2 Duo machine, eMuJava can successfully generate, execute, and analyze 150-200 test cases.
- *Old Population Rate*: eMuJava uses tournament selection method to pick the fitter test cases to perform crossover. This parameter provides the amount of fitter test cases to be selected from whole population. Similarly this parameter is used to pick the amount of test cases for biological mutation.
- *Method Class Sequence Count*: Every test case needs to invoke certain number of methods before calling the method under test. eMuJava selects the amount of methods to be added in the test case using this parameter. eMuJava generates a random number between 0 and the provided value for this parameter and then adds equal number of method calls in the test case.

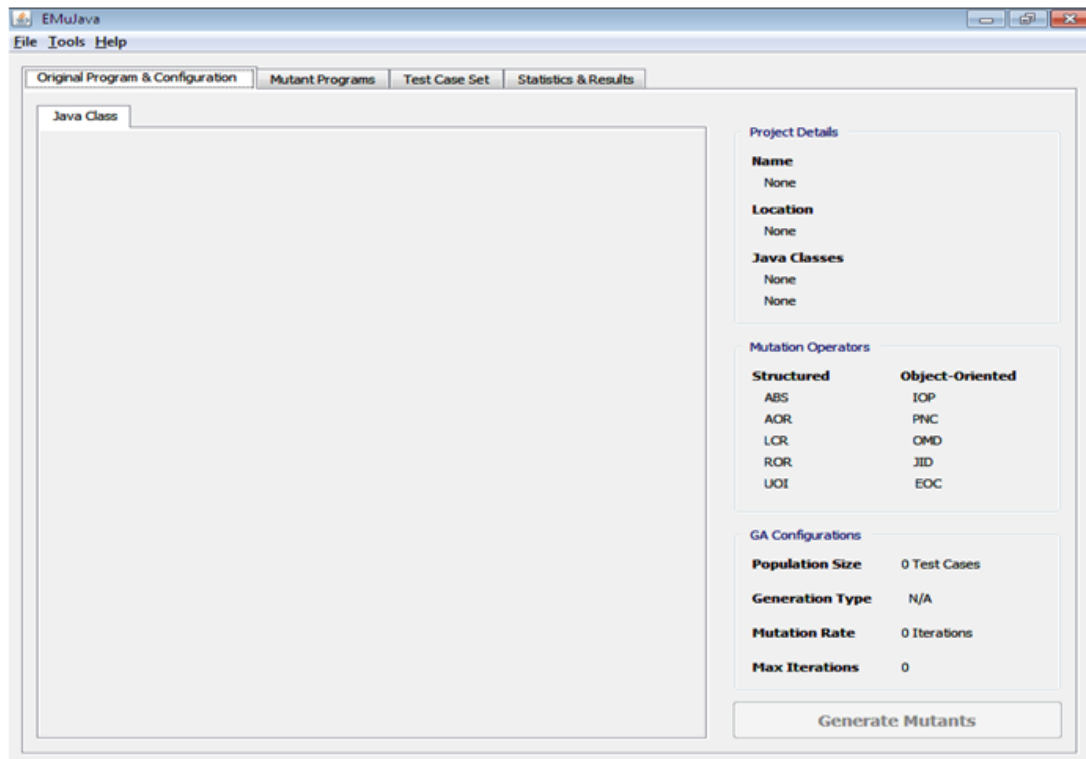


FIGURE 5.10: Main GUI - eMuJava Tool

5.6 Screenshots of eMuJava Tool

This section provides some insight on graphical user interface of the tool. eMuJava is a GUI based tool that provides user interactions and outputs on its interfaces. There are four main components of the eMuJava tool including 'Source Code & Configuration', 'Mutants Viewer', Test Case Viewer, and 'Statistics & Results'. The main interface of eMuJava tool is presented in figure 5.10.

5.6.1 Source Code & Configuration

The first component of the eMuJava tool helps the user to load the source code of class under test. User can create new project in eMuJava to open new project wizard and provide various types of inputs and configuration details to the tool. The wizard receives inputs in three steps. In the first step the wizard asks for project name, project location, Java source code file(s) as input as shown in figure 5.11 below;

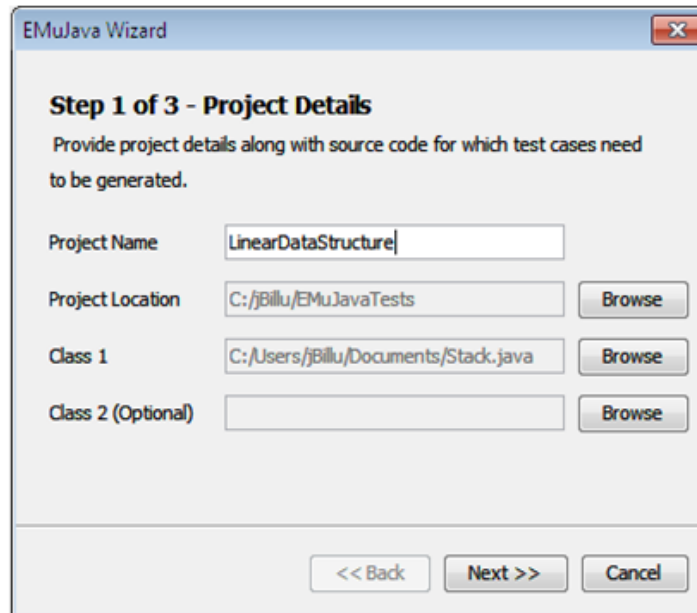


FIGURE 5.11: eMuJava Wizard Step 1

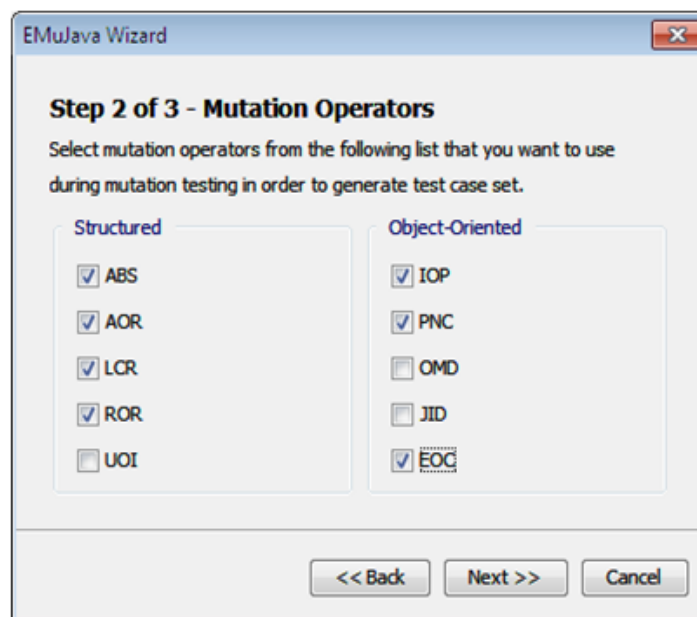


FIGURE 5.12: eMuJava Wizard Step 2

The second step of eMuJava wizard allows user to select the mutation operators that he wants to apply on the source code under test to generation mutants. eMuJava offers ten mutation operators; five from structured paradigm and five from object-oriented paradigm. The figure 5.12 shows the screenshot of the mutation operators selection step of the wizard;

In the third step the user can provide test configuration to the tool including

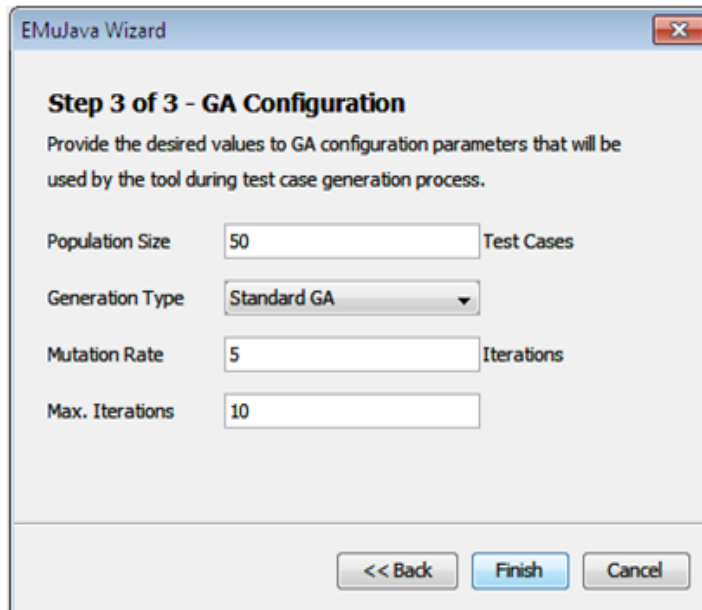


FIGURE 5.13: eMuJava Wizard Step 3

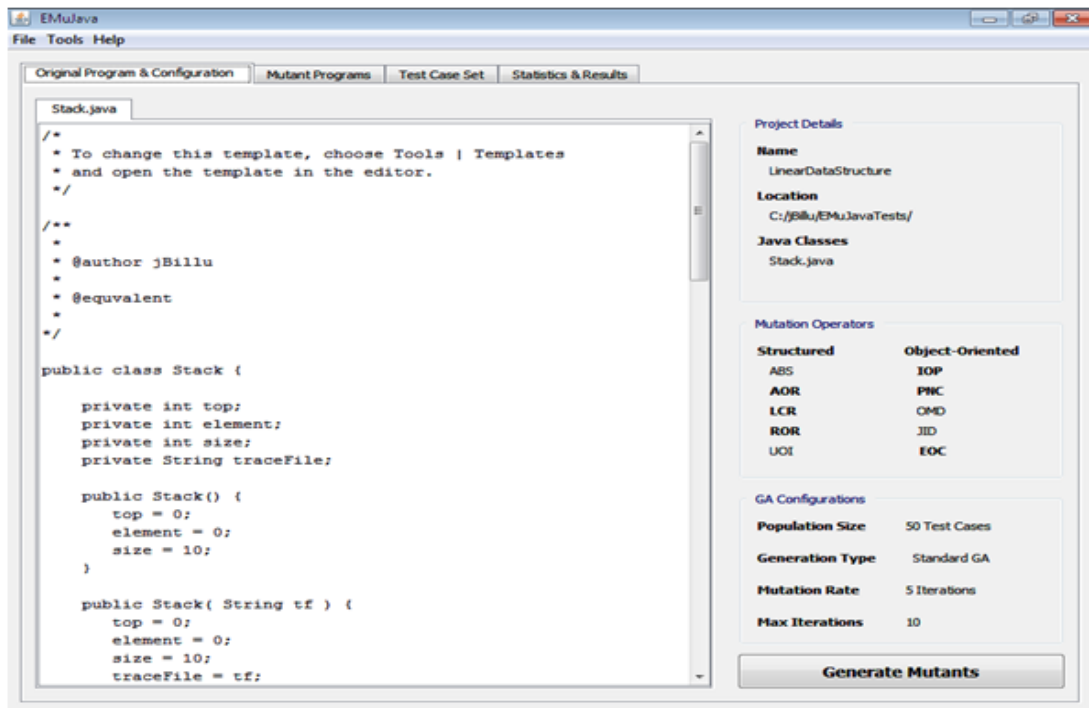


FIGURE 5.14: eMuJava Source Code & Configuration

population size, test case generation technique, mutation rate, and maximum iterations. The figure 5.13 shows the screenshot of third and final step and figure 5.14 shows the loaded code and configuration;

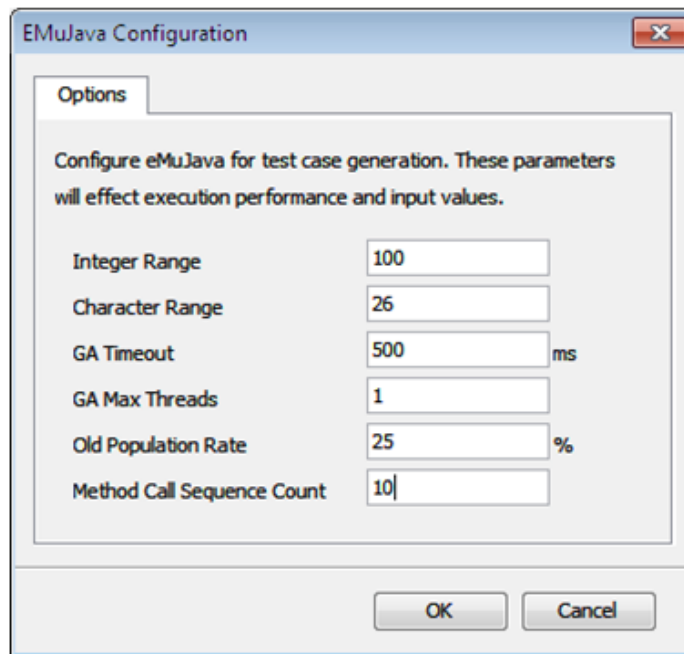


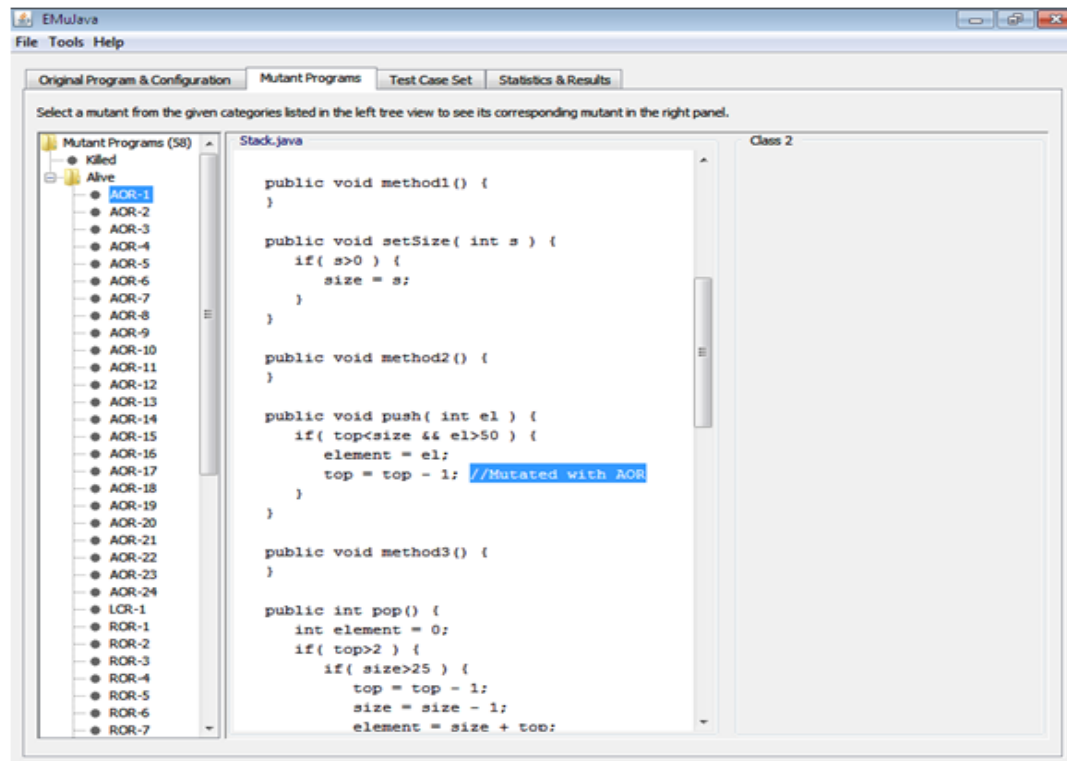
FIGURE 5.15: eMuJava Configuration Editor

5.6.2 Configuration Editor

The configuration editor allows the user to view current configurations and enables him to change them as per needs. The available options of configuration are presented in Section 5.5. Figure 5.15 presents the screenshot of configuration editor.

5.6.3 Mutants Viewer

This section allows the user to view all the mutants generated by eMuJava and it also shows the mutated statement in the code through a single-line comment as shown in the figure 5.16. This section divides the mutants in three categories; killed, alive, and suspicious. After the test case generation process completes, the tool updates this section to classify all the mutants and to show which mutant is declared as suspicious or killed, and which one remains as alive.



5.6.4 Test Case Viewer

The test case viewer shows the test cases generated during the iterations performed by eMuJava tool to achieve targets (kill mutants). There are two sections of this component; first shows the test cases being generated in different iterations along with their fitness and second section shows the test cases that are able to achieve the targets. This section is updated by the tool when an iteration completes and after each target is achieved during test case generation process so user sees live updates coming to it. Figure 5.17 shows its screenshot;

5.6.5 Statistics & Results

The figure 5.18 presents its screenshot; The last section statistics and results presents figures and graphs about test case generation process and it also gets updated live during test case generation process. There are two sub-sections of this section. The first sub-section shows the information about program under

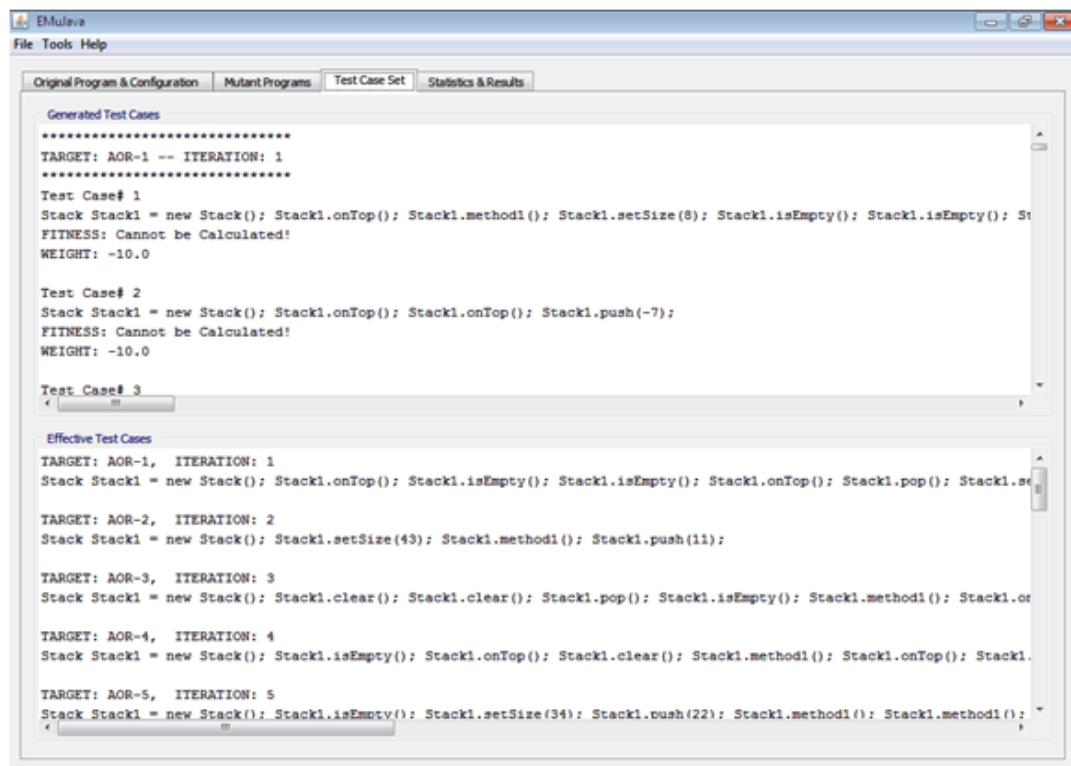


FIGURE 5.17: eMuJava Test Case Viewer



FIGURE 5.18: eMuJava Statistics & Results

test, number of mutants, alive, suspicious, killed and mutation score. The pie-chart is used to show the distribution of mutants in all categories. The second sub-section shows information about effort consumed by the process during test case generation including iterations performed, number of test cases generated, effective test cases, and total time elapsed.

Chapter 6

Experiments and Results Analysis

In this chapter, we present evaluation of the proposed fitness function, two-way crossover, and adaptable mutation methods. We have used our implemented tool eMuJava (Chapter 5) to perform experiments and empirical evaluation. We have used twenty programs to perform experiments for validation. We have carried out experiments in two phases and have compared multiple test case generation approaches. Initially we have used random testing and genetic algorithm with standard fitness function for test case generation and have compared the results with our proposed state-based & control-oriented fitness function. Later we have carried out experiments with our improved genetic algorithm (state-based & control-oriented fitness function, two-way crossover, and adaptable mutation) and compared the results with all three approaches we use in first phase. Also we have compared our improved genetic algorithm with an existing tool EvoSuite [34]. Towards the end we present statistical analysis on the results obtained from experiments.

6.1 Test Environment

Before we discuss details of the experiments that we have performed, first we present some information about test environment that we have prepared and have

used to conduct them using eMuJava tool.

The experiments have been performed on Intel 32-bit machine having Core 2 Duo Centrino processor. The computer has 2GB of memory (RAM) and enough hard disk to store the execution traces generated by the tool. The machine is running Windows 7 operating system supporting 32-bit architecture. The tool has been run using Java virtual machine (JVM) version 8 with update 45 released by Oracle for Windows platform. Every time a new experiment is conducted the tool has been rerun to ensure the resources are completely released by JVM and new experiments are not affected by any means. We have also made sure that source code under test is syntax error free and the experiments are not affected by any runtime exception generated by the tool.

We have placed source code of our tool and programs for researchers and students. The URL is <https://github.com/bilalbezar/eMuJava>. The programs are available for download from the aforementioned URL and everyone is free to modify and use it for experiments. We have chosen programs (containing one or more Java classes) from different domains for experiments and all of these programs are of different nature. Some of the programs are data structures (like `Stack` and `HashTable`), whereas others are popular general purpose programs (`Calculator` and `TemperatureConverter`) that we find in literature. Also we have chosen programs that help automating home appliances (like `ElectricHeather` and `AutoDoor`). Besides these programs we have also used 10 open source programs of Apache project available online at <http://commons.apache.org/>. These programs include `CLI`, `Collections`, `Compress`, `Crypto`, `CSV`, `JCS`, `Lang`, `Logging`, `Math`, and `Text`. Since eMuJava is a prototype tool and limited in syntax support (Appendix C) so we had to make some changes in the programs to make them compatible with our implementation. Table 6.1 presents list of all the programs that we have used for experiments along with related information about the programs including their names, number of classes, number of methods, lines of code, and total number of mutants generated from them. All the classes in programs exhibit variety of behaviors (through public methods) and in order to test those

TABLE 6.1: List of Programs and Details

Programs	Classes	Methods	Lines of Code	Mutants
AutoDoor	1	9	112	111
BankAccount	2	14	116	180
BinarySearchTree	1	6	119	189
Calculator	1	7	60	97
CGPACalc	1	4	105	111
CLI	2	6	160	24
Collections	3	16	386	131
Compress	3	11	291	87
Crypto	1	2	81	120
CSV	2	4	129	145
ElectricHeater	1	9	120	146
HashTable	1	8	98	103
JCS	1	5	135	74
Lang	2	12	405	51
Logging	2	11	133	95
Math	1	4	109	103
Stack	2	13	144	107
TempConverter	1	8	60	106
Text	1	3	143	97
Triangle	1	5	99	147
Total	30	157	3005	2224

behaviors, some of the classes require their objects to be in specific state (for example `AutoDoor`, `Stack`, `ElectricHeater` and so on). On the other hand some of the methods in classes contain simple arithmetic expressions (having standard arithmetic operations) and conditional statements hence they do not have any serious requirement for objects to be in some specific state.

Out of these twenty programs, some (`BankAccount`, `Stack`, `Collections` and

so on) are based on more than one class. The classes in these programs form inheritance relationship and the sub-classes override some of the methods from the base class. This relationship helps applying mutation operators that cannot be applied on a single class. eMuJava tool supports two of such mutation operators including overriding method deletion (IOD) and new method call with child class type (PNC).

6.2 Initial Experiments and Results

In this section, we present initial experiments and their results that we have performed using eMuJava tool to validate the effectiveness of the state-based & control-oriented fitness function [13]. The experiments are carried out on all ten applications (see Section 6.1 for details) that we selected from different domains. The experiment results are quite interesting and have helped us identifying a problem in our proposal that we have discussed later in this section.

Table 6.2 presents the results of initial experiments. eMuJava has been used to perform experiments with three approaches; random testing, genetic algorithm with standard fitness function, and genetic algorithm with state-based & control-oriented fitness function. All of these approaches generate test cases that try to satisfy three conditions to kill a mutant; reachability, necessity, and sufficiency. Table 6.2 presents the results of the experiments in terms of average mutation score of multiple runs (ranges from 5 to 10) by each approach against all the programs. In order to check if our proposed fitness function guides the search process well and if it helps achieving high mutation score in less number of iterations, we did not allow the tool to run for indefinite amount of time. Instead we let the tool to execute ten iterations per target to verify if it gains high mutation score with our proposed approach.

The results of initial experiments have proven to be quite interesting. In figure 6.1, we have plotted the results with the help of column chart for better understanding. In the chart, x-axis plots the tested programs (from Table 6.2) and y-axis plots

TABLE 6.2: Results of Initial Experiments with Fixed Number of Iterations

Programs	Loc	M	Mutation Score (%)		
			Random Testing	GA with Standard Fitness Function	GA with Proposed Fitness Function
AutoDoor	112	111	77	77	80
BankAccount	116	180	100	100	100
BinarySearchTree	119	189	87	85	83
Calculator	60	97	95	78	83
CGPACalc	105	111	96	84	84
CLI	160	24	78	73	82
Collections	386	131	70	73	78
Compress	291	87	71	78	82
Crypto	81	120	48	53	61
CSV	129	145	78	68	72
ElectricHeater	120	146	87	84	87
HashTable	98	103	68	65	74
JCS	135	74	50	48	55
Lang	405	51	83	85	83
Logging	133	95	49	58	56
Math	109	103	100	89	85
Stack	144	107	69	71	88
TempConverter	60	106	100	100	100
Text	143	97	82	86	92
Triangle	99	147	79	80	83
Total	3005	2224	78.3	76.7	80.4

mutation score. Results achieved by approaches are represented with a different shade (random testing red, genetic algorithm with standard fitness function blue, genetic algorithm with proposed fitness function - black) to distinguish among them. All the approaches have given good results in general but the proposed approach [13] has produced a touch better set of test cases than the other two.

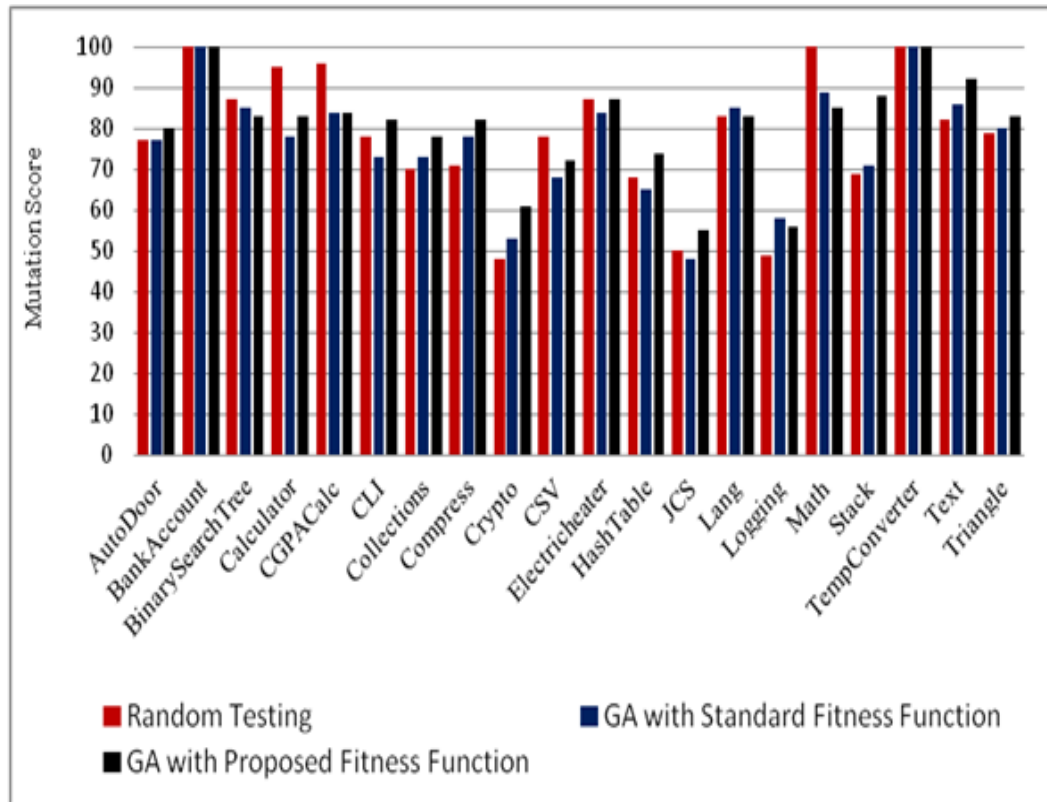


FIGURE 6.1: Comparison of Experiment Results among Random Testing, GA with Standard Fitness Function, and GA with Proposed Fitness Function

The proposed fitness function seems to obtain high mutation score for the programs that have state requirement (`AutoDoor`, `Collections`, `HashTable`, `Stack`, and `Triangle`) but the difference in mutation score is not very significant. When we analyze the results, we discover that although the proposed fitness function provides information about the weakness in a test case (whether the object is in desired state through state fitness) yet the next phase of genetic algorithm (crossover) fails to utilize it due to its inherent nature. Crossover does not consider object's state fitness to produce offsprings rather it randomly picks up a point in parents and crosses them over to form the offsprings. So the search process has to wait until biological mutation to use state fitness to form new method call sequence, which may help object in gaining desired state

Besides that for `ElectricHeater` our proposed fitness function manages equal mutation score and for `CGPACalc` it underperforms whereas both of these approaches have state requirement. Actually `ElectricHeater` and `CGPACalc` require values

for state variables that do not have any particular pattern. Such values can be generated quickly with random generation rather than crossover of test cases. Similarly random testing produces better mutation scores for `BinarySearchTree`, `Calculator`, `CSV`, and `Math` as compared to genetic algorithm. These programs do not have state requirement but some predicates in the methods of these programs have strict argument requirement (for example `a==50`). In this scenario crossover cannot be useful and search process has to wait until biological mutation comes into play and produces required method arguments to satisfy the conditions in predicates.

If we can enable crossover to use the object's state fitness to produce offspring, we may be able to push mutation score further up in limited number of iterations. For this our proposed two-way crossover method (Section 4.4) comes into play. Besides that if a program does not have state requirement and it relies on the method arguments, performing crossover for high number of iterations does not help. In this situation if we can dynamically increase rate of biological mutation, the search can produce required values quickly to kill the mutant in lesser number of iterations (Section 4.5 - adaptable mutation method).

6.3 Detailed Experiments and Results

This section provides results of extensive experiments that are carried out to validate the proposed two-way crossover and adaptable mutation methods. We have performed experiments using eMuJava on twenty applications using four supported approaches by the tool (Section 5.3). All four approaches generate test cases to kill the mutants and attempt to raise the mutation score. This section is further divided into three sub-sections and each section covers an aspect regarding experiments. We have performed experiments to show that our proposals help in generating a test case in less number of iterations, which can kill a mutant that may or may not require an object to be in a certain state. Secondly, we have

compared the results with another tool EvoSuite. We have also shown with experiments that detecting suspicious mutants can help identifying a bug (logical programming mistake). By removing such logical bug in the program, apparently looking equivalent mutant becomes non-equivalent that can then be killed to raise mutation score.

6.3.1 Less Iterations, Higher Mutation Score

First we present and discuss results of experiments that we have performed to validate if our proposed fitness function, two-way crossover, and adaptable mutation help the search process to converge towards target in less number of iterations. The experiments have been performed using same test configuration that includes number of iterations, number of test cases in a single iteration, crossover, and initial biological mutation rate.

We have presented results of experiments in Table 6.3 and they show that our proposed approach has performed well. For every approach we have used same test configuration and have not allowed the tool to run for indefinite amount of time. To kill a mutant, eMuJava runs 10 iterations each carrying population of 50 test cases. After every iteration fitter test cases are crossed over and after fifth iteration, test cases go through biological mutation. Random testing does not use crossover or biological mutation whereas our improved genetic algorithm uses two-way crossover and adaptable mutation rate. On completing 10 iterations, if mutant remains alive, tool leaves it and picks the next mutant.

If we carefully analyze the results, we notice that random testing and genetic algorithm with standard fitness function have not performed well on most of the programs specially those who have strict state requirement including `AutoDoor`, `Collections`, `HashTable`, `Stack` and so on. We have presented the comparison of these two approaches with genetic algorithm having state-based & control-oriented fitness function earlier (Section 6.2). Now we compare our improved genetic algorithm with random testing and standard genetic algorithm. For better

TABLE 6.3: Results of Experiments Performed using eMuJava

Programs	Loc	M	Mutation Score (%)			
			Random Testing	GA with Standard Fitness Function	GA with State-based & Control-oriented Fitness Function	Improved Genetic Algorithm
AutoDoor	112	111	77	77	80	88
BankAccount	116	180	100	100	100	100
BinarySearchTree	119	189	87	85	83	88
Calculator	60	97	95	78	83	93
CGPACalc	105	111	96	84	84	100
CLI	160	24	78	73	82	86
Collections	386	131	70	73	78	85
Compress	291	87	71	78	82	87
Crypto	81	120	48	53	61	65
CSV	129	145	78	68	72	79
ElectricHeater	120	146	87	84	87	90
HashTable	98	103	68	65	74	89
JCS	135	74	50	48	55	55
Lang	405	51	83	85	83	92
Logging	133	95	49	58	56	63
Math	109	103	100	89	85	95
Stack	144	107	69	71	88	98
TempConverter	60	106	100	100	100	100
Text	143	97	82	86	92	94
Triangle	99	147	79	80	83	89
Total	3005	2224	78.3	76.7	80.4	86.8

comparison we have plotted the results of these three approaches using a column chart in figure 6.2. All three are represented with distinguished colors (random testing - red, genetic algorithm with standard fitness function - blue, and improved genetic algorithm - black). The x-axis represents programs under test and y-axis represents achieved mutation score.

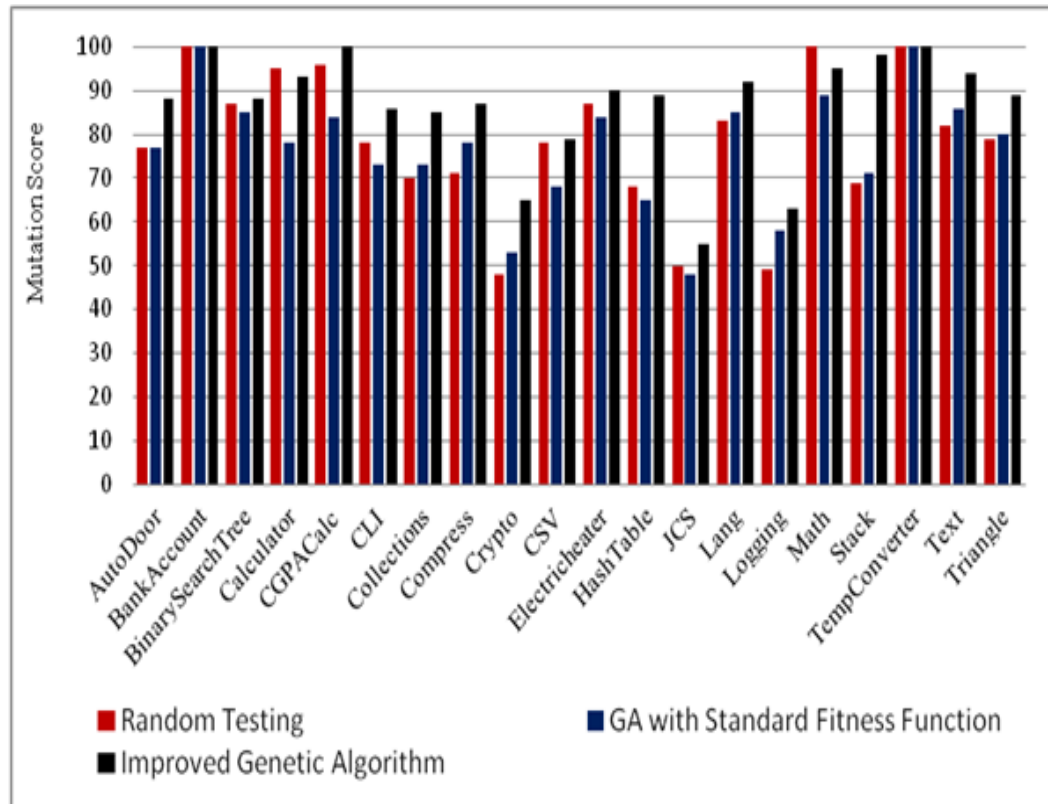


FIGURE 6.2: Comparison of Experiment Results among Random Testing, GA with Standard Fitness Function, and Improved Genetic Algorithm

The results clearly show that our improved genetic algorithm has a significant impact on mutation score and it has increased for all those programs (AutoDoor, CGPACalc, Collections, ElectricHeater, HashTable, Stack, and Triangle) that have strict state requirement. It has obtained equal or touch higher mutation score for other programs (BankAccount and TempConverter) that do not have any state requirement. With the help of adaptable mutation, our proposed genetic algorithm has managed to complete random for programs (BinarySearchTree, Calculator, CSV, and Math) that require specific range of values. Besides that our improved genetic algorithm clearly outperforms remaining rest of the programs (CLI, Compress, Crypto, JCS, Lang, Logging, Test, and Triangle). The difference in average mutation score obtained by our proposed approach from other approaches ranges approximately from 6% to 10%, which is promising.

We have analyzed our improved genetic algorithm with a different angle as well.

Earlier we have presented comparison of mutation scores obtained by all the approaches in a fixed number of iterations. Now we shall see how many iterations all the approaches require to achieving 100% mutation score (after filtering out all the equivalent mutants). We have again performed experiments with eMuJava tool and allow the tool to run until all the mutants get killed. The results are presented in Table 6.4 in terms of number of iterations (I) consumed by each approach as well as number of test case (TC) generated by all of the four approaches. In figure 6.3 we present average executed test cases by all the compared approaches using bar chart. The chart shows our improved genetic algorithm reduces computational effort in mutation testing to a certain degree.

We have plotted the results with the help of line chart in figure 6.4. Figure 6.4 presents 20 line charts, one for each program, which has been tested. All four approaches are represented with four different colors (random testing - blue, genetic algorithm with standard fitness function - red, genetic algorithm with state-based & control-oriented fitness function - green, and our improved genetic algorithm - black). In these charts x-axis represents number of iterations whereas y-axis represents mutation score. The values used to generate these line charts are provided in Appendix D for reference.

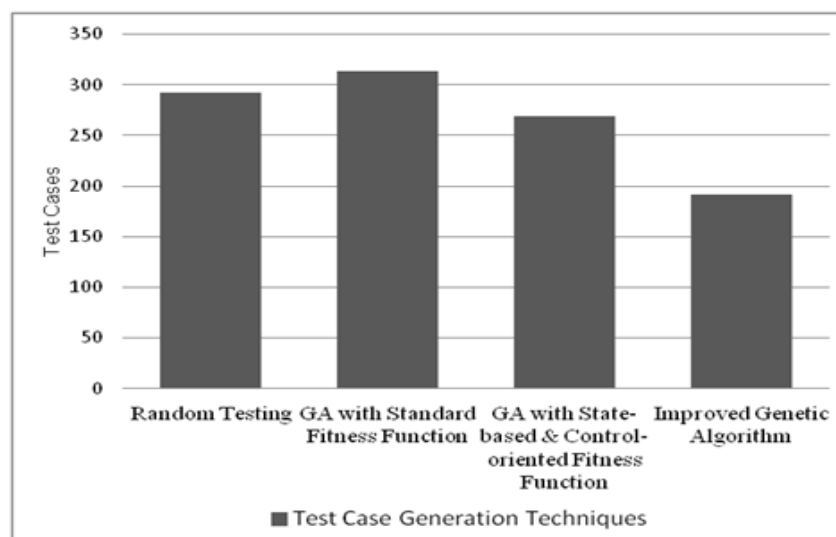
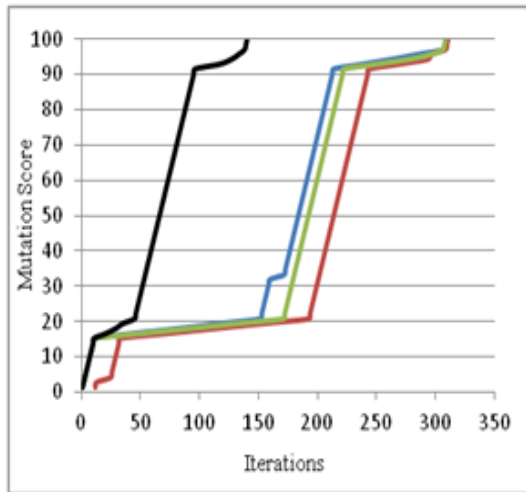


FIGURE 6.3: Comparison among Average Executed Test Cases by Test Generation Techniques

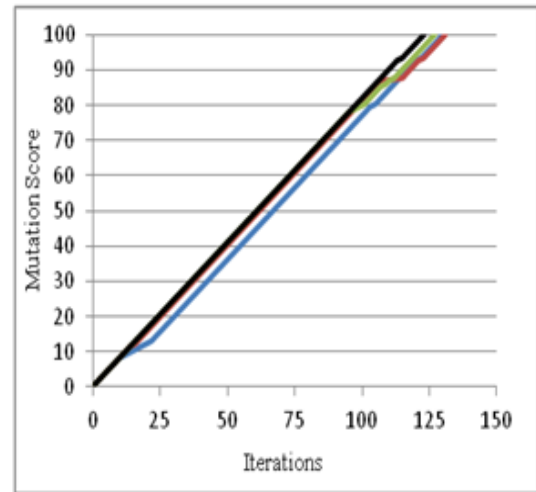
TABLE 6.4: Number of Iterations and Executed Test Cases by All Four Approaches

Programs	Random Testing		GA with Standard Fitness Function		GA with State-based & Control-oriented Fitness Function		Improved Genetic Algorithm	
	I	TC	I	TC	I	TC	I	TC
AutoDoor	310	7750	311	7775	308	7700	140	3500
BankAccount	130	3250	131	3275	127	3175	123	3075
BinarySearchTree	263	6575	263	6575	294	7350	255	6375
Calculator	130	3250	280	7000	207	5175	157	3925
CGPACalc	93	2325	229	5725	195	4875	74	1850
CLI	95	2375	110	2750	85	2125	79	1975
Collections	242	6050	229	5725	206	5150	188	4700
Compress	253	6325	219	5475	183	4575	165	4125
Crypto	270	6750	239	5975	217	5425	204	5100
CSV	409	10225	525	13125	474	11850	382	9550
ElectricHeater	380	9500	559	13975	589	14725	375	9375
HashTable	615	15375	569	14225	411	10275	289	7225
JCS	203	5075	210	5250	175	4375	162	4050
Lang	107	2675	105	2625	122	3050	90	2250
Logging	430	10750	370	9250	390	9750	276	6900
Math	104	2600	211	5275	184	4600	99	2475
Stack	1052	26300	942	23550	450	11250	208	5200
TempConverter	91	2275	93	2325	87	2175	83	2075
Text	246	6150	192	4800	155	3875	110	2750
Triangle	432	10800	476	11900	519	12975	373	9325
Total	5855	146375	6263	156575	5378	134450	3832	95800

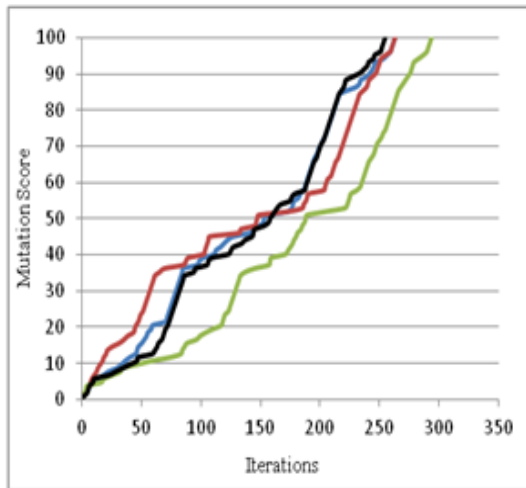
The line trace in these charts show the progress of all four approaches while they kill the mutants and achieve 100% mutation score. Our improved genetic algorithm seems to have clear advantage on all the other three approaches especially in the programs that have state requirement. For rest of the programs it performs equally well to the other approaches specially in comparison to random testing.



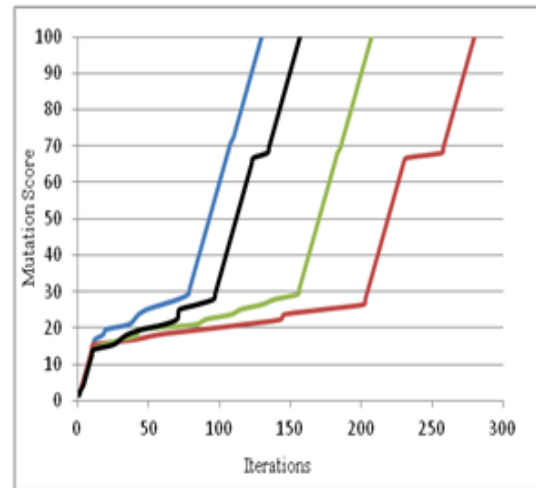
(a) AutoDoor



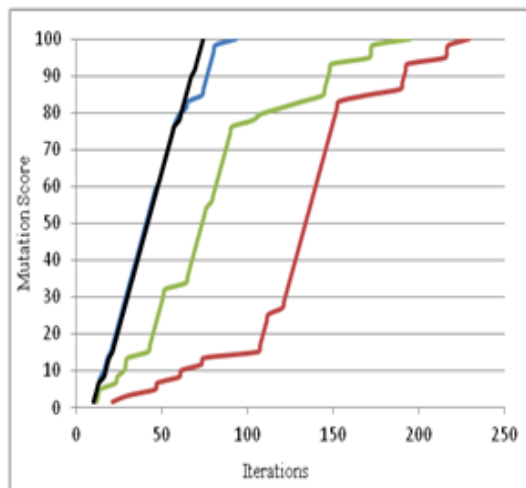
(b) BankAccount



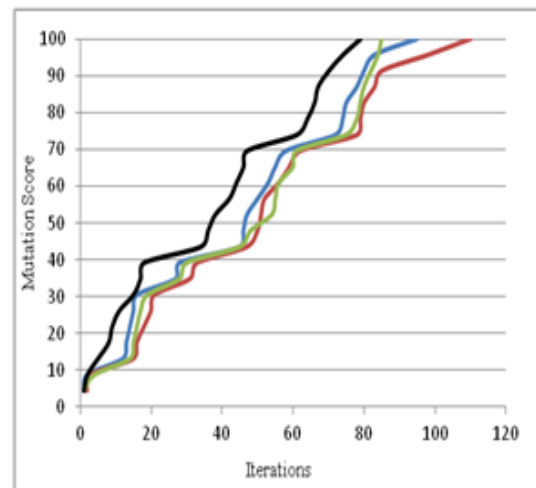
(c) BinarySearchTree



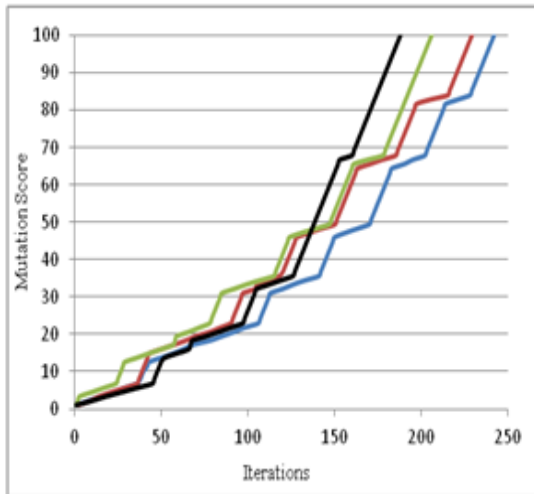
(d) Calculator



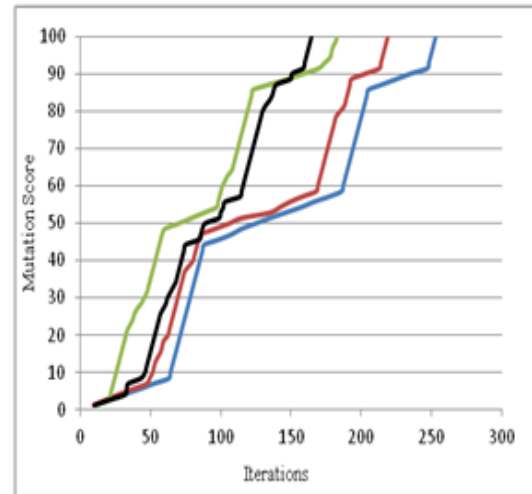
(e) CGPACalc



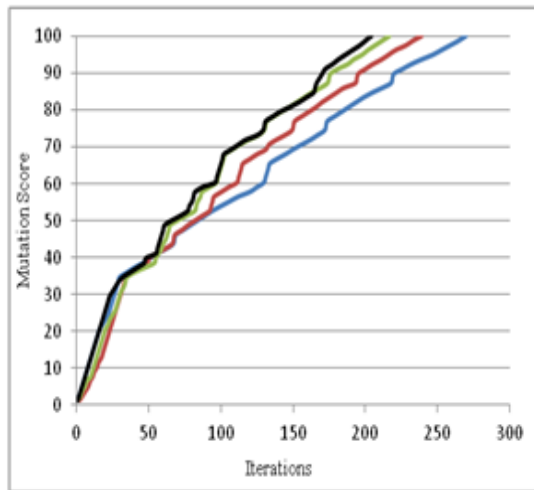
(f) CLI



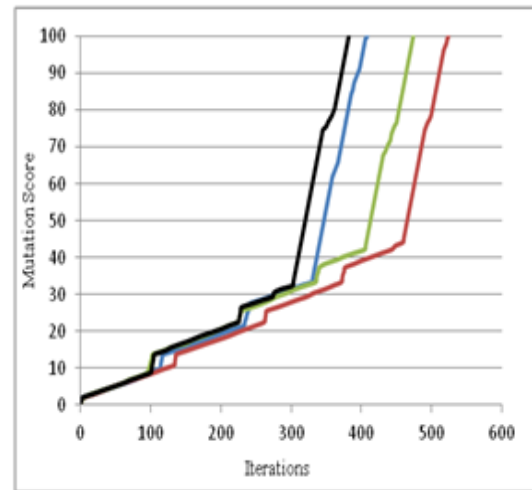
(g) Collections



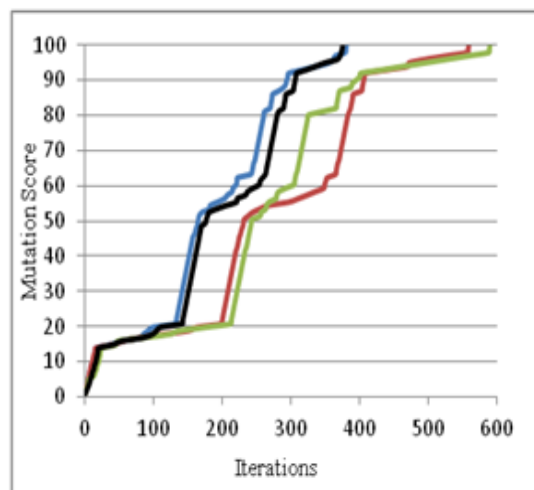
(h) Compress



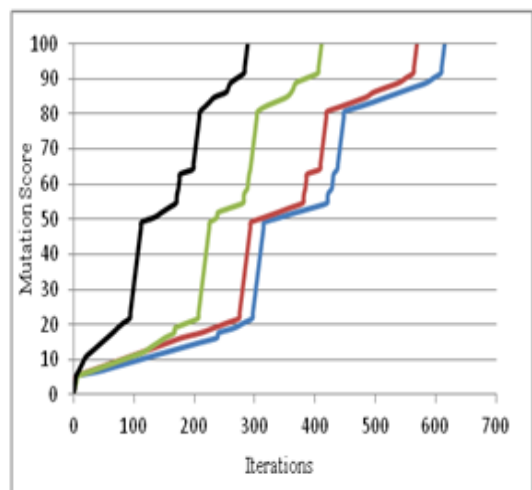
(i) Crypto



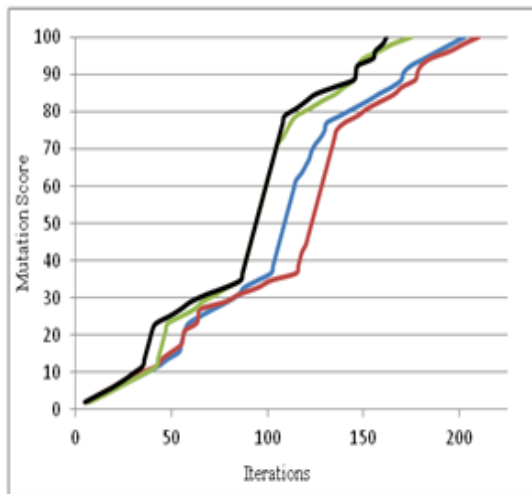
(j) CSV



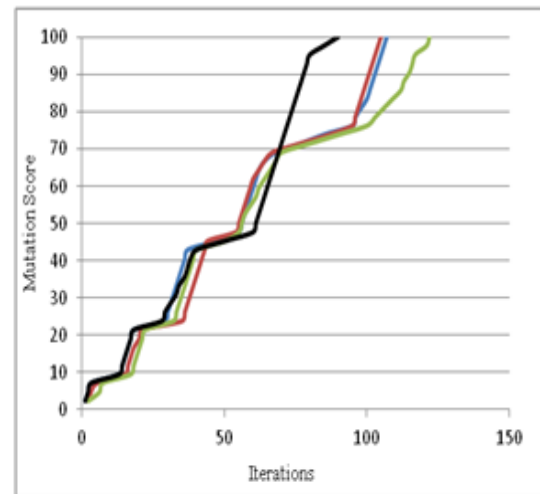
(k) ElectricHeater



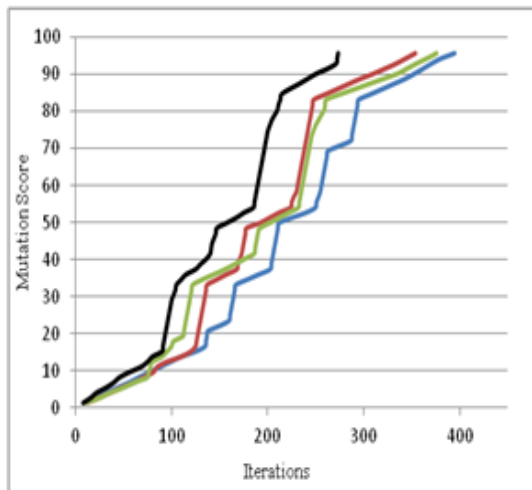
(l) HashTable



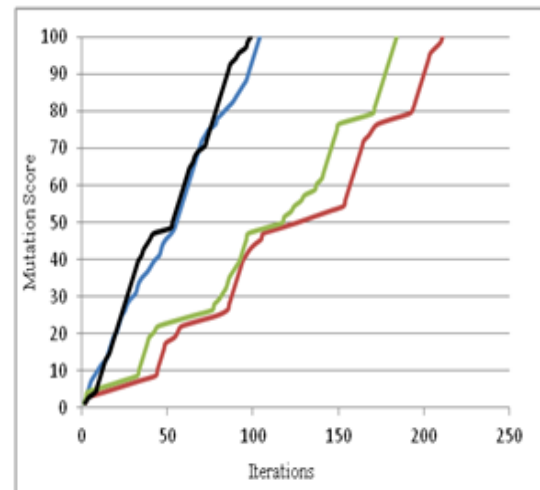
(m) JCS



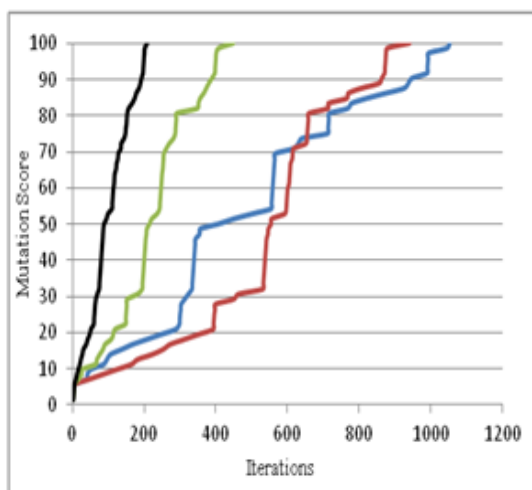
(n) Lang



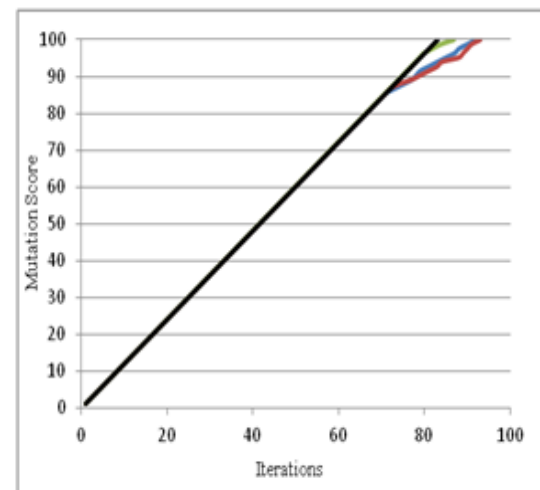
(o) Logging



(p) Math



(q) Stack



(r) TempConverter

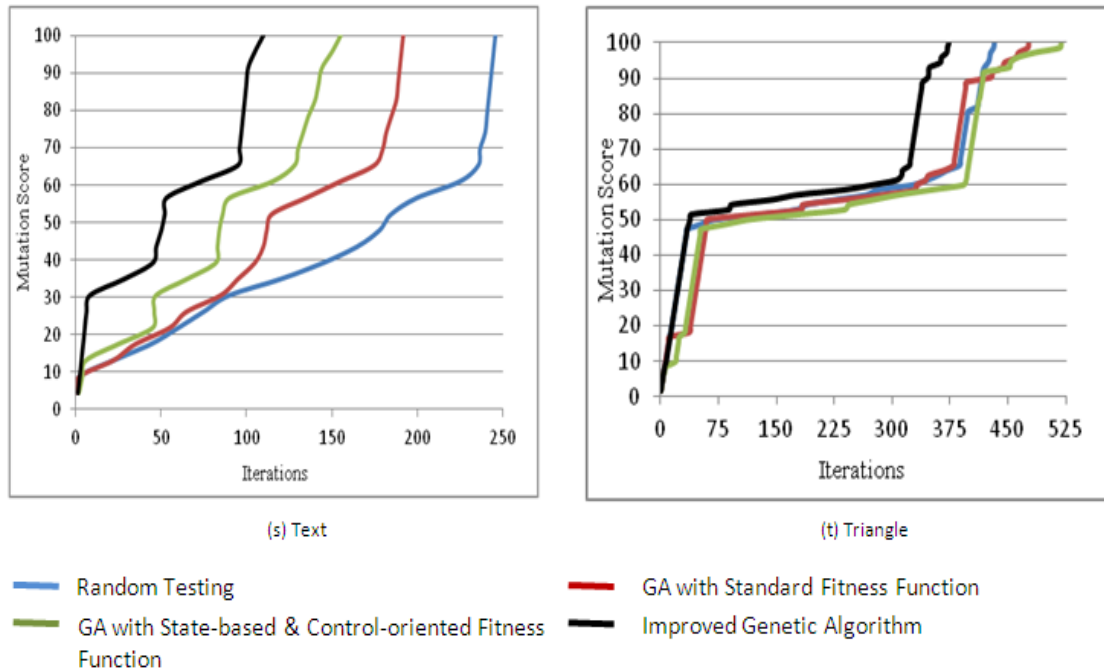


FIGURE 6.4: Comparison to Show Progress among All Approaches while Achieving 100% Mutation Score

The results of these experiments prove our claim that with the help of state-based & control-oriented fitness function, the search process gets better guidance. Through two-way crossover method test cases gain the desired state quickly. Adaptable mutation guides the genetic algorithm to increase the mutation frequency for programs that need specific values to satisfy conditions. Eventually improved genetic algorithm converges towards the targets in less number of iterations and helps in reducing required testing effort.

6.3.2 Comparison with EvoSuite

Now we present our experiment results in comparison to EvoSuite [34], which is the only available tool. EvoSuite can test Java based programs and can generate tests for branch coverage and mutation based coverage. It only supports unit level testing so EvoSuite cannot apply mutation operators that involve more than one class (mutation operators for inheritance or polymorphism). In general EvoSuite does not support any object-oriented feature and supports limited set of conventional mutation operators from research of Andrews et al. [46] and Schuler and Zeller

[47]. Due to this reason although EvoSuite can test object-oriented programs in general but it can apply only structured mutation operators. Besides these problems, EvoSuite seems to have performed well as the results of experiments we find in study [10] so that makes it as best choice for validation and comparison with our implementation.

There are many issues that we have discovered while experimenting with EvoSuite. EvoSuite does not provide information about the mutants it generates, how many mutants remain alive, how does it handle equivalent mutants, and if it includes all the mutants in calculating mutation score. Also the experiments we have performed earlier to compare our improved genetic algorithm with existing approaches, we restricted the iterations to see the performance of our approach but in case of EvoSuite it is not possible by any mean to limit the number of iterations to see how many targets it is able to achieve. Our implemented tool eMuJava differs a lot from EvoSuite and it is difficult to give an accurate comparison between them due to hidden information in EvoSuite. By making some adjustments and selecting the same mutation operators, we are able to produce a fair comparison between both approaches. We have presented the results of experiments in Table 6.5.

We have used strong mutation testing parameter with EvoSuite to perform the experiments. For better understanding, we have plotted the experiment results presented in Table 6.5 using column chart in figure 6.5. The x-axis represents the case studies we have used for experiments and y-axis shows average mutation score achieved after multiple runs. The grey bars represent mutation score achieved by EvoSuite whereas bars in black color represent mutation score achieved by eMuJava (our improved genetic algorithm). The difference between mutation scores shows that our improved genetic algorithm has performed well as compare to EvoSuite for all the case studies used. eMuJava is able to kill more mutants in less number of iterations hence it has produced better mutation score.

TABLE 6.5: Experiment Results

Programs	Loc	M	Mutation Score (%)	
			EvoSuite	Improved Genetic Algorithm
AutoDoor	112	78	44	73
BankAccount	116	120	47	73
BinarySearchTree	119	102	38	60
Calculator	60	67	55	64
CGPACalc	105	59	59	58
CLI	160	18	61	79
Collections	386	70	71	84
Compress	291	56	75	87
Crypto	81	61	51	51
CSV	129	85	64	78
ElectricHeater	120	103	29	80
HashTable	98	71	44	57
JCS	135	43	32	39
Lang	405	34	76	91
Logging	133	54	33	48
Math	109	51	80	98
Stack	144	81	35	67
TempConverter	60	76	53	77
Text	143	58	89	96
Triangle	99	72	50	41
Total	3005	1359	54.3	77.8

6.3.3 Detecting Suspicious Mutants to Raise Mutation Score

Sometimes, it has been noticed that a small programming mistake (logical) affects mutation score badly. Due to the logical error in program, the mutant becomes equivalent, which not only damages mutation score but it also wastes a lot of time during test case generation. In our earlier work [1], we coined an idea of using

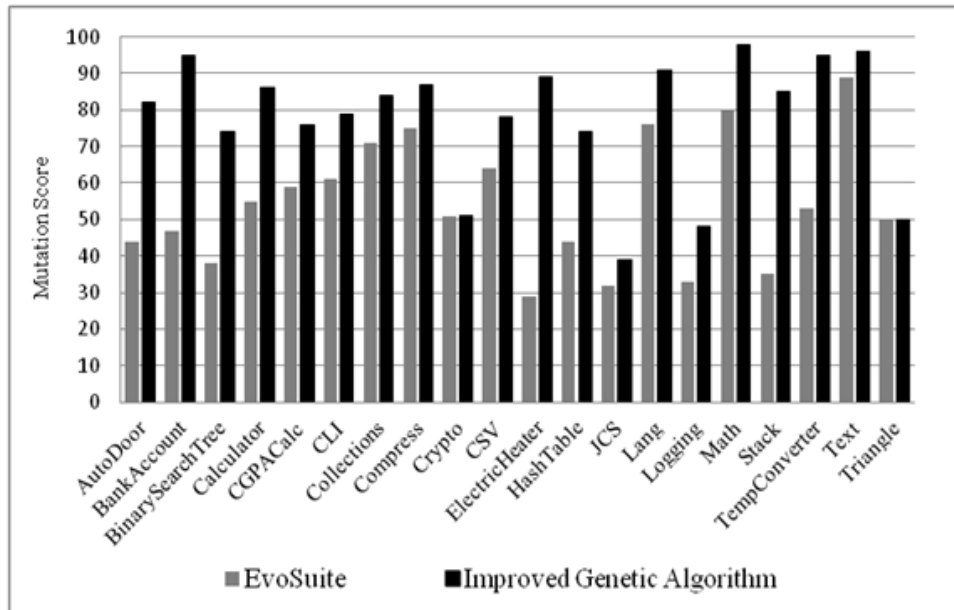


FIGURE 6.5: Comparison of Experiment Results Obtained from EvoSuite and eMuJava (Improved Genetic Algorithm)

control-flow information besides output of the program to see how mutant behaves as compare to original program. This idea leads us to very interesting results and we noticed that if program has a logical error, although original and modified programs produce same output yet they exercise different execution path because sometimes the logical error masks the change introduced by mutation hence the mutant remains alive. We call such a mutant as **suspicious mutant**. So what our approach does is that, if a mutant remains alive but for a given test case, it exercises different execution path as compared to original program, we declare it suspicious and encourages manual checking of the program to ensure a logical programming error is not causing this problem. The figure 6.6 contains a code snippet of `CGPACalc` program that explains the concept of suspicious mutant.

The incorrect string literal at line 16 will always cause the method to return "Pass" even if the value of variable `cgpa` is less than 2.5. Hence all the mutants generated from this method will remain alive. By using control-flow information this error can be highlighted and corrected, which results in raising the mutation score. We have used the same idea in our earlier work [13] in which we present a fitness function for evolutionary mutation testing of object-oriented programs (Section 4.3). In that work, we use control flow information as part of test case

```
...
public String calculateResult() {
    ...
    //Mutated statement
    ...
14. if( cgpa>=2.5 ) {
15.     return "Pass";
    } else {
16.     return "Pass";           //bug
    } //END if-else STATEMENT
} /END calculateResult() METHOD
```

FIGURE 6.6: CGPACalc Mutant with Logical Bug (Suspicious Mutant)

fitness. Now in this section, we have presented results of experiments that prove its effectiveness.

None of the approaches including genetic algorithm, random testing, and EvoSuite uses control flow information as part of test case fitness, hence they suffer from such a situation where a logical error in the program causes the mutants to remain alive. To show the difference in mutation scores between our proposed work and others (genetic algorithm and EvoSuite), we have performed experiments using CGPACalc program by introducing the same error we have presented in figure 6.6. The figure 6.7 plots the results of experiments and proves the effectiveness of using control flow information to evaluate a test case.

6.4 Statistical Analysis

In this section we present statistical analysis that we have performed to analyze the results to find out their significance. We have performed analysis on the results we obtained initially with respect to state-based & control-oriented fitness function and later the results produced in validation of improved genetic algorithm. We have used *MannWhitney U-test* and *Vargha and Delaneys A measure* to perform statistical analysis. We have used a popular statistical tool, R [48] to perform all the tests.

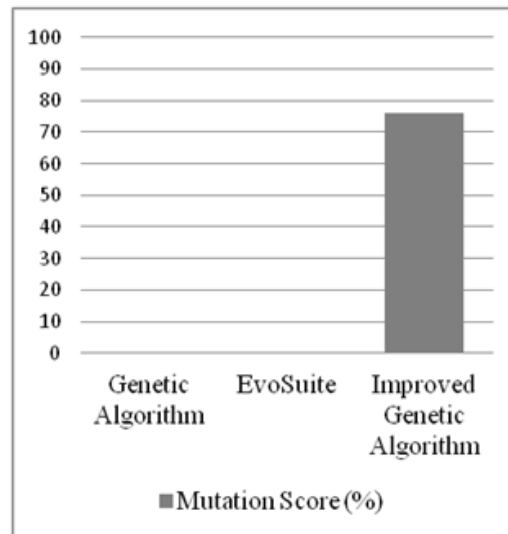


FIGURE 6.7: Comparison of Mutation Scores among Genetic Algorithm, EvoSuite, and Improved Genetic Algorithm

6.4.1 *MannWhitney U-test*

The *MannWhitney U-test* is also called *MannWhitneyWilcoxon (MWW)* or *Wilcoxon rank-sum test*. It is a statistical procedure, which statisticians use to check if a randomly selected value from one sample will be less than or greater than a randomly selected value from the other sample. This method is useful to make comparison of test case generation techniques to see if the proposed technique has made any significant change in the mutation score. We have chosen this method to compare our experiments' results because it is easy to apply and it provides meaningful results in no time.

The *U-test* is also known as non-parametric test and unlike *t-test* it does not require the population of data to be normally distributed. In most of the cases the data produced by randomized algorithms do not exhibit normality [49]. Literature also shows that data samples that do not variate a lot as we have in this case the mutation scores (1-100), researchers can use *t-test* but this phenomenon has been mostly found in natural and social sciences [49]. The best solution in this case is to test the normality of the data and then on the basis of results make the decision about statistical test.

6.4.2 Normality Test

We want to statistically analyze the mutation scores presented in table 6.4. Before deciding on what test should we use, first we have tested the data for normality using *Shapiro Wilk test* using `R-tool` [48]. We have used "0.05" for *p-value* to decide if a given data samples are normally distributed or not. Data that will produce lesser *p-value* will be considered as non-normal and data producing higher *p-value* will be considered as normal. Results of experiments are presented below.

Null Hypothesis: The mutation scores are normally distributed.

1. Random Testing

- The computed value of W is 0.92328.
- The *p-value* is 0.1146.
- **Conclusion:** We accept the null hypothesis.

2. GA with Standard Fitness Function

- The computed value of W is 0.96319.
- The *p-value* is 0.6093.
- **Conclusion:** We accept the null hypothesis.

3. GA with State-based & Control-oriented Fitness Function

- The computed value of W is 0.90434.
- The *p-value* is 0.04976.
- **Conclusion:** We reject the null hypothesis.

4. Improved Genetic Algorithm

- The computed value of W is 0.83566.
- The *p-value* is 0.003094.
- **Conclusion:** We reject the null hypothesis.

5. EvoSuite

- The computed value of W is 0.96154.
- The p -value is 0.5751.
- **Conclusion:** We accept the null hypothesis.

The results show that data samples 3 and 4 are non-normal but normality tests are influenced by sample size so it is advised that we should use visual representations like Q - Q Plot for verification. The Q - Q Plot proved that both data samples 3 and 4 are non-normal. In this situation parametric tests like t -test are not suitable [49]. So based on the normality test results, we decide to use U -test to compare mutation scores.

6.4.3 Analysis of Initial Experiments and Results

First we have performed U -test on the results of initial experiments presented in Section 6.2. Here we compare state-based & control-oriented fitness function with random testing and GA with standard fitness function. Before we present the results of U -test, we set null and alternate hypothesis;

Null Hypothesis: The state-based & control-oriented fitness function produces same mutation score.

Alternate Hypothesis: The state-based & control-oriented fitness function produces higher mutation score.

1. Random Testing

- The mean and standard deviation of mutation scores are 78.35 and 15.94 respectively for random testing.
- The computed value of W is 177.
- The p -value is 0.5418.
- **Result:** The location shift is not equal to 0.

- **Conclusion:** We reject the null hypothesis and accept the alternate hypothesis.

2. GA with Standard Fitness Function

- The mean and standard deviation of mutation scores are 76.75 and 13.39 respectively for GA with standard fitness function.
- The computed value of W is 167.5.
- The *p-value* is 0.3859.
- **Result:** The location shift is not equal to 0.
- **Conclusion:** We reject the null hypothesis and accept the alternate hypothesis.

6.4.4 Analysis of Detailed Experiments and Results

We have also performed U-test to compare improved genetic algorithm with random testing, GA with standard fitness function, and EvoSuite. For this we have used the results presented in Section 6.3. Before we present the results of U-test, we set null and alternate hypothesis;

Null Hypothesis: The improved genetic algorithm produces same mutation score.

Alternate Hypothesis: The improved genetic algorithm produces higher mutation score.

1. Random Testing

- The mean and standard deviation of mutation scores are 78.35 and 15.94 respectively for random testing.
- The computed value of W is 132.5.
- The *p-value* is 0.06935.
- **Result:** The location shift is not equal to 0.

- **Conclusion:** We reject the null hypothesis and accept the alternate hypothesis.

2. GA with Standard Fitness Function

- The mean and standard deviation of mutation scores are 76.75 and 13.39 respectively for GA with standard fitness function.
- The computed value of W is 98.
- The p -value is 0.00596.
- **Result:** The location shift is not equal to 0.
- **Conclusion:** We reject the null hypothesis and accept the alternate hypothesis.

3. EvoSuite

- The mean and standard deviation of mutation scores are 54.3 and 16.92 respectively for EvoSuite.
- The computed value of W is 69.
- The p -value is 0.0004137.
- **Result:** The location shift is not equal to 0.
- **Conclusion:** We reject the null hypothesis and accept the alternate hypothesis.

To summarize we can say that the state-based & control-oriented fitness function produced briefly better results than random testing and GA with standard fitness function. On the other hand our improved genetic algorithm has produced fairly better results as compared to all other approaches and U-test proves this claim.

6.4.5 Effect Size Measure

The U-test has proven that our improved genetic algorithm has produced better mutation scores but it is equally important to check the magnitude of the improvement. For this we have chosen *Vargha and Delaney's A measure*. The reason

of selection is our data is non-normal that we want to compare and for situations like these *A measure* is the best choice [49]. We have compared our improved genetic algorithm with random testing, GA with Standard Fitness Function, GA with State-based & Control-oriented Fitness Function, and EvoSuite. The threshold value is "0.5" and as far as the value goes beyond the threshold, it indicates magnitude of improvement.

- **Random Testing:** The effect size of our proposed approach with respect of random testing is 0.67.
- **GA with Standard Fitness Function:** The effect size of our proposed approach with respect to GA with standard fitness function is 0.76.
- **GA with State-based & Control-oriented Fitness Function:** The effect size of our proposed fitness function with respect to state-based & control-oriented fitness function is 0.72.
- **EvoSuite:** The effect size of our proposed approach with respect to EvoSuite is 0.83.

The results of *A measure* indicates fair amount of improvement in the mutation scores produced by our improved genetic algorithm.

6.5 Test Set Evaluation

In this section we present evaluation of the test case set generated by all the testing techniques implemented in eMuJava. For evaluation, we have used the test data generated by the tool when number of iterations were fixed (section 6.3.1). We want to asses quality of the generated test cases through injecting multiple faults in the programs and running the test cases again on them. We then analyze the execution traces to see which mutated statements are executed. We also compared the results with oracle to determine how many mutations are caught by the test case set.

We have chosen some new mutation operators from structured paradigm list. The operators have been selected from the work of King and offutt [28]. Authors have designed mutation operators for FORTRAN programming language but they are applicable on Java programs too. We want to see if test cases generated for a certain mutant can catch a different mutation also. The selected operators are listed below.

- **CRP:** Constant replacement
- **CSR:** Constant for scalar variable replacement
- **SCR:** Scalar for constant replacement
- **SVR:** Scalar variable replacement

We have seeded multiple faults using above mentioned mutation operators in the program under test. We have introduced one fault per line of code. Table 6.6 presents the results of test set evaluation. The table presents number of total faults injected in second column and then presents number of faults detected by each technique. If a give test case is able to execute a statement having a mutation and mutant program produces different output then we declare that mutation as detected. Since eMuJava does not support above mentioned mutation operators and does not support injection of multiple faults, we had to produce the mutant programs manually. After execution of the test cases, we manually examine the code to determine how many mutations have been detected by a given test case.

These results of experiments revealed interesting information. We have noticed that multiple mutations can cause the program to behave in different ways. For instance a predicate $x > y$ can be changed like $x > N$ where N is any random number. This simple mutation can cause the mutant to become extremely hard to kill and mutant may become equivalent in some cases. Similarly it can cause the mutant to become extremely easy to kill also. We have made sure that the mutations that we have applied do not cause the mutant to become equivalent or raise a runtime exception. The results indicate that our improved genetic algorithm has

TABLE 6.6: Results of Test Set Evaluation

Programs	Faults	Number of Detected Faults			
		Random Testing	GA with Standard Fitness Function	GA with State-based & Control-oriented Fitness Function	Improved Genetic Algorithm
AutoDoor	29	29	29	29	28
BankAccount	26	26	23	26	23
BinarySearchTree	36	29	29	30	30
Calculator	17	17	17	17	17
CGPACalc	19	15	17	15	18
CLI	11	10	10	10	10
Collections	27	14	17	17	17
Compress	12	8	8	12	12
Crypto	10	10	10	10	10
CSV	10	8	0	1	8
ElectricHeater	39	33	35	35	35
HashTable	23	19	19	19	18
JCS	12	10	8	10	10
Lang	33	26	25	27	25
Logging	20	18	17	17	15
Math	23	16	16	23	23
Stack	22	20	20	18	20
TempConverter	16	16	16	16	16
Text	17	17	17	17	17
Triangle	24	20	19	19	19
Total	426	361	352	368	371

been able to catch more faults. The reason being our proposed approach was able to achieve high mutation score in limited number of iterations (section 6.3.1) that also means that the generated test cases are better. So some hard to kill mutants were detected with those test cases.

Chapter 7

Conclusion and Future Work

Evolutionary mutation testing is a merger of two disciplines; evolutionary algorithms and mutation testing. This discipline provides basis to automatically generate test cases for mutation testing using evolutionary approaches like genetic algorithm that can help in reducing the testing effort. Object's state is an important aspect of object-oriented programs and it plays a significant role in testing because objects behave differently in different states. So to test an object's behavior, the object may need to be in a specific state. Besides that, the control flow information of a program provides useful information about program's behavior. If we use that information carefully, that can lead to produce interesting results during software testing. In this thesis, we have proposed some extensions to properly evaluate a test case, to perform crossover on test cases to enhance its strength, and to biological mutation to generate specific inputs quickly. We have improved genetic algorithm with these proposals and have validated their effectiveness through experiments using our own implementation (eMuJava).

Mutation testing techniques that we find in the literature (discussed in Appendix A) compare the output of original and mutant programs to decide if a test case is able to catch the injected fault or not. But the behavior of a program is defined by output as well as flow of control. If we ignore control flow of a program while analyzing its behavior on a test case, we may end up losing important information. So when a test case is executed, besides the output, the control flow information

should also be compared. We propose and present a new technique for mutation testing (see Section 4.2 for details) that considers both of these behavioral elements (output and control flow information) to decide if a mutant is killed or alive. With this modification in mutation testing, a comprehensive comparison of both the programs can be made. Sometimes on a test case, original and mutant programs produce same output but mutant program suspiciously exercises different execution path. This may happen due to the presence of some logical mistake (potential bug) in the code, which is usually made by the programmer.

Using control-flow information and object's state as part of test case' fitness, we have proposed a new fitness function for evolutionary mutation testing (Section 4.3). Our proposed fitness function evaluates three costs against three conditions that a test case tries to satisfy to kill a mutant and as used by Bottaci [18]. We have extended those conditions used by Botacci [18] and have incorporated control-flow information (Section 4.1) and object's state as part of fitness in them. The idea of using object's state as part of test case fitness is taken from our previous work [2]. Our proposed fitness function suits well for object-oriented testing because we propose treating state of the object as a fitness component that helps in evaluating a test case properly, which can help the search process to converge to the target in less number of iterations. Besides that other than comparing outputs, we also suggest to compare flow of control of original and mutant programs to decide on mutant's status. With the help of control flow information, we can identify suspicious piece of code and highlight suspicious behavior of the program. This helps in improving the overall quality of the program as well.

As mentioned above, we introduce two new values in a test case fitness including object's state and control flow information. Both of these values are calculated from execution traces of program under test and are represented as separate costs in fitness. With initial experiments and analysis of our approach we discover that although the proposed fitness function provides information about the weakness in a test case (whether object in the test case has gained desired state through state fitness) yet the next phase of genetic algorithm (crossover) fails to utilize it due to its inherent nature. Whether genetic algorithm uses one-point crossover

or two-point crossover, it does not consider object's state fitness to produce offsprings (new test cases for next iteration). The process has to wait until biological mutation uses state fitness to form new method call sequence in a test case, which may help object in gaining desired state. To overcome this limitation, we have proposed a new two-way crossover method (Section 4.4) that uses object's state fitness to decide if standard crossover method shall be applied to generate two offsprings or our proposed two-way crossover be applied to generate 4 new offsprings. The two-way crossover method covers the input domain well and does not let an opportunity go that can converge the process towards the target (required test case that kills a mutant).

Biological mutation is an important activity in genetic algorithm that prevents the search process to get stuck in local optima. Too low mutation rate can cause the search process to keep on looking in a certain region of input domain and too high mutation rate can cause the process to become completely random. In some situations, method under test needs specific set of input values that crossover cannot generate and performing repeated crossovers on test cases waste testing effort. In such a situation biological mutation can do the job and even increasing its frequency can save some effort. All the existing evolutionary mutation testing techniques use fix mutation rate. We propose adjusting the mutation rate dynamically during execution of genetic algorithm and call it as adaptable mutation method (Section 4.5). The mutation rate is adjusted by checking object's state fitness after each iteration. None of the existing evolutionary mutation testing approaches use this type of flexible mutation that can adapt to the situation intelligently. Experiments have proven that adaptable mutation saves effort as it helps in generating the required input parameters in less number of iterations.

We have implemented our proposals in a tool called eMuJava (Chapter 5), which stands for **e**volutionary **M**utation testing of **J**ava programs. This tool is efficient and is able to evaluate the fitness of test cases by considering the object's state fitness. eMuJava can classify mutants as suspicious or normal on the basis of control-flow information obtained through execution traces. Besides that this tool

performs two-way crossover and dynamically adjust mutation rate using state-based fitness of a test case. eMuJava also supports multiple methods for test case generation including random generation of test cases, genetic algorithm with standard fitness function, genetic algorithm with state-based & control-oriented fitness function, and genetic algorithm with state-based & control-oriented fitness function, two-way crossover, and adaptable mutation methods.

We have performed detailed experiments using eMuJava tool (Chapter 6) to validate our proposed improvements in genetic algorithm and have compared the results with other technique. The experiments have given positive results and we are in the position to say that with the usage of object's state fitness, control flow information, two-way crossover, and adaptable mutation, the evolutionary mutation testing process improves. The improved genetic algorithm is capable of reducing mutation testing cost in terms of time and effort. The testing process becomes effective and efficient because of the appropriate guidance it gets, mutation scores are increased, and logical programming errors are identified.

7.1 Future Work

This section presents the future directions for researchers;

- Extend the implementation of eMuJava tool to support complete Java language syntax and support for additional object-oriented mutation operators.
- Further experiments can be conducted to compare the effectiveness of our proposed approach with other search based techniques like particle swarm optimization (PSO) and artificial immune system (AIS).
- The proposed modifications for test case evaluation (state-based & control-oriented fitness function) can be applied with other evolutionary approaches to see if they work equally good as they do with the genetic algorithm.
- Extension of eMuJava tool to support multiple evolutionary approaches to perform experiments and evaluation.

Bibliography

- [1] M. B. Bashir and A. Nadeem, “Control oriented mutation testing for detection of potential software bugs,” in *10th International Conference on Frontiers of Information Technology*. IEEE, 2012, pp. 35–40.
- [2] M. B. Bashir and A. Nadeem, “A state based fitness function for evolutionary testing of object-oriented programs,” in *7th ACIS International Conference on Software Engineering Research, Management and Applications*, vol. 253. Springer, 2009, pp. 83–94.
- [3] B. Jones, H. Sthamer, and D. Eyres, “Automatic structural testing using genetic algorithms,” *Software Engineering Journal*, vol. 11, no. 5, pp. 299–306, 1996.
- [4] P. Tonella, “Evolutionary testing of classes,” in *ACM SIGSOFT International Symposium of Software Testing and Analysis*. ACP, 2004, pp. 119–128.
- [5] Y. Cheon and M. Kim, “A specification-based fitness function for evolutionary testing of object-oriented programs,” in *8th Annual Conference on Genetic and Evolutionary Computation*. ACM, 2006, pp. 1953–1954.
- [6] G. Fraser and A. Zeller, “Mutation-driven generation of unit tests and oracles,” *IEEE Transactions on Software*, vol. 38, no. 2, pp. 278–292, 2011.
- [7] P. May, J. Timmis, and K. Mander, “Immune and evolutionary approaches to software mutation testing,” in *6th International Conference on Artificial Immune Systems*. Springer, 2007.

-
- [8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.
- [9] K. K. Mishra, S. Tiwari, A. Kumar, and A. Misra, "An approach for mutation testing using elitist genetic algorithm," in *3rd IEEE International Conference on Computer Science and Information Technology*. IEEE, 2010, pp. 426–429.
- [10] G. Fraser and A. Arcuri, "Achieving scalable mutation-based generation of whole test suites," *Empirical Software Engineering*, vol. 20, no. 3, pp. 783–812, 2015.
- [11] R. Silva, S. Rocio, S. Souza, and P. Srgio, "A systematic review on search based mutation testing," *Information and Software Technology*, vol. 81, pp. 19–35, 2017.
- [12] M. B. Bashir and A. Nadeem, "Object oriented mutation testing: A survey," in *8th International Conference on Emerging Technologies*. IEEE, 2015, pp. 1–6.
- [13] M. B. Bashir and A. Nadeem, "A fitness function for the evolutionary mutation testing of object-oriented programs," in *9th International Conference on Emerging Technolgoies*. IEEE, 2013, pp. 1–6.
- [14] M. B. Bashir and A. Nadeem, "Improved genetic algorithm to reduce mutation testing cost," *IEEE Access*, vol. 5, pp. 3657–3674, 2017.
- [15] "Software testing," in <http://standards.ieee.org/findstds/standard/29119-1-2013.html> (accessed on September 16, 2017). IEEE, 1998.
- [16] R. Just, D. Jalali, L. Inozemtseva, M. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *The 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. ACM, 2014.
- [17] J. Domnguez, A. Estero, A. Garcia, and I. Medina, "Evolutionary mutation testing," *Information and Software Technology*, vol. 53, pp. 1108–1123, 2011.

-
- [18] L. Bottaci, “A genetic algorithm fitness function for mutation testing,” in *International Workshop on Software Engineering using Metaheuristic Inovative Algorithms*. ACM, 2001, pp. 3–7.
- [19] M. Masud, A. Nayak, M. Zaman, and N. Bansal, “A strategy for mutation testing using genetic algorithms,” in *Canadian Conference on Electrical and Computer Engineering*. IEEE, 2005, pp. 1049–1052.
- [20] M. Bybro, “A mutation testing tool for java programs,” in *A Mutation Testing Tool for Java Programs*. Department of Numerical Analysis and Computer Science, 2003.
- [21] “Junit,” in <http://junit.org/junit5/> (accessed September 16, 2017). GitHub, 2000.
- [22] M. Papadakis and N. Malevris, “Automatic mutation based test data generation,” in *13th Annual Conference Companion on Genetic and Evolutionary Computation*. ACM, 2011, pp. 247–248.
- [23] J. Wegener, A. Baresel, and H. Sthamer, “Evolutionary test environment for automatic structural testing,” *Information and Software Technology*, vol. 43, no. 14, pp. 841–854, 2001.
- [24] S. Subramanian and R. Natarajan, “A tool for generation and minimization of test suite by mutant gene algorithm,” *Journal of Computer Science*, vol. 7, no. 10, pp. 1581–1589, 2011.
- [25] J. Louzada, C. Camilo, A. Vincenzi, and C. Rodrigues, “An elitist evolutionary algorithm for automatically generating test data,” in *IEEE Congress on Evolutionary Computation*. IEEE, 2012, pp. 1–8.
- [26] Y. Ali and F. Benmaiza, “Generating test case for object-oriented software using genetic algorithm and mutation testing method,” *International Journal of Applied Metaheuristic Computing*, vol. 3, p. 1, 2012.
- [27] M. Rad and S. Bahrekazemi, “Applying genetic evolutionary, bacteriological and quantum evolutionary algorithm for improving performance optimization

- segment of test data sets in mutation testing method,” *International Journal of Soft Computing and Software Engineering*, vol. 4, no. 1, pp. 167–186, 2014.
- [28] K. King and J. Offutt, “A fortran language system for mutation-based software testing,” *Software: Practice and Experience*, vol. 21, no. 7, pp. 685–718, 1991.
- [29] “Matlab,” in <http://www.mathworks.com/products/matlab/> (accessed on September 24, 2017). MathWorks, 1984.
- [30] “Evosuite,” in <http://www.evosuite.org/downloads/> (accessed September 16, 2017). EvoSuite, 2011.
- [31] J. Miguel, M. Vivanti, A. Arcuri, and G. Fraser, “A detailed investigation of the effectiveness of whole test suite generation,” *Empirical Software Engineering*, vol. 22, no. 2, pp. 852–893, 2016.
- [32] N. Jatana, B. Suri, S. Misra, P. Kumar, and A. R. Choudhury, “Particle swarm based evolution and generation of test data using mutation testing,” in *International Conference on Computational Science and Its Applications*. Springer, 2016, pp. 585–594.
- [33] P. Delgado, I. Medina, S. Segura, A. Garca, and J. Jos, “Gigan: Evolutionary mutation testing for c++ object-oriented systems,” in *Symposium on Applied Computing*. ACM, 2017, pp. 1387–1392.
- [34] G. Fraser and A. Arcuri, “Evosuite: Automatic test suite generation for object-oriented software,” in *ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 416–419.
- [35] “Eclipse,” in <https://www.eclipse.org/downloads/> (accessed September 16, 2017). Eclipse, 2001.
- [36] P. McMinn and M. Holcombe, “The state problem for evolutionary testing,” in *Genetic and Evolutionary Computation Conference*, vol. 2724. Springer, 2003, pp. 2488–2497.

- [37] J. Offutt, A. Lee, G. Rothermel, R. Untch, and Z. C., “An experimental determination of sufficient mutant operators,” *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 2, pp. 99–118, 1996.
- [38] E. Barbosa, J. Maldonado, and A. Vincenzi, “Toward the determination of sufficient mutant operators for c,” *Software Testing, Verification and Reliability*, vol. 11, no. 2, pp. 113–136, 2001.
- [39] J. Offutt, Y. S. Ma, and Y. R. Kwon, “The class-level mutants of mujava,” in *2006 International Workshop on Automation of Software Test*. ACM, 2006, pp. 78–84.
- [40] B. Baudry, F. Fleurey, J. M. Jezequel, and Y. L. Traon, “Automatic test case optimization: A bacteriologic algorithm,” *IEEE Software*, vol. 22, no. 2, pp. 76–82, 2005.
- [41] X. Xie, B. Xu, C. Nie, L. Shi, and L. Xu, “A dynamic optimization strategy for evolutionary testing,” in *29th Annual International Computer Software and Applications Conference*. IEEE, 2005.
- [42] C. S. S. Dharsana and A. Askarunisha, “Java based test case generation and optimization using evolutionary testing,” in *International Conference on Computational Intelligence and Multimedia Applications*, vol. 4. IEEE, 2007, pp. 44–49.
- [43] I. Alsmadi, “Using genetic algorithms for test case generation and selection optimization,” in *23rd Canadian Conference on Electrical and Computer Engineering*. IEEE, 2010.
- [44] M. Wang, B. Li, Z. Wang, and X. Xie, “An optimization strategy for evolutionary testing based on cataclysm,” in *34th Annual Computer Software and Applications Conference Workshops*. IEEE, 2010, pp. 359–364.
- [45] W. E. Wong and A. Mathur, “Reducing the cost of mutation testing: An empirical study,” *The Journal of Systems and Software*, vol. 31, no. 3, pp. 185–196, 1995.

- [46] J. Andrews, L. Briand, and Y. Labiche, “Is mutation an appropriate tool for testing experiments?” in *27th International Conference on Software Engineering*. ACM, 2005, pp. 402–411.
- [47] D. Schuler and A. Zeller, “(un-)covering equivalent mutants,” in *3rd International Conference on Software Testing Verification and Validation*. IEEE, 2010, pp. 45–54.
- [48] R. D. C. Team, “R: A language and environment for statistical computing,” in <https://www.r-project.org/> (accessed on November 30, 2017). R Foundation 1288 for Statistical Computing, 2008.
- [49] A. Arcuri and L. Briand, “A hitchhiker’s guide to statistical tests for assessing randomized algorithms in software engineering,” *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.
- [50] S. Kim, J. Clark, and J. McDermid, “The rigorous generation of java mutation operators using hazop,” in *12th International Conference Software & Systems Engineering and their Applications*. University of York, 1999.
- [51] T. Kletz, “Hazop and hazan: Identifying and assessing process industry hazards,” in *HAZOP and HAZAN: Identifying and Assessing Process Industry Hazards*. Hemisphere Publishers, 1992.
- [52] S. Kim, J. Clark, and J. McDermid, “Class mutation: Mutation testing for object-oriented programs,” in *Net.ObjectDays Conference on Object-Oriented Software Systems*. Object-Oriented Software Systems, 2000.
- [53] P. Chevalley, “Applying mutation analysis for object oriented programs using a reflective approach,” in *8th Asia-Pacific Software Engineering Conference*. IEEE, 2001, pp. 267–270.
- [54] Y. S. Ma, Y. R. Kwon, and J. Offutt, “Inter-class mutation operators for java,” in *13th IEEE International Symposium on Software Reliability Engineering*. IEEE, 2002, pp. 352–363.

- [55] L. Gallagher, J. Offutt, and A. Cincotta, "Integration testing of object-oriented components using finite state machines: Research articles," *Software Testing, Verification & Reliability*, vol. 16, no. 4, pp. 215–266, 2006.
- [56] P. Chevalley and P. Thvenod-Fosse, "A mutation analysis tool for java programs," in *LAAS Report No 01356*, vol. 5. Springer, 2001, pp. 90–103.
- [57] R. Alexander, J. Bieman, S. Ghosh, and J. Bixia, "Mutation of java objects," in *13th International Symposium on Software Reliability Engineering*. IEEE, 2003.
- [58] J. Offutt, Y. S. Ma, and Y. R. Kwon, "An experimental mutation system for java," in *ACM SIGSOFT Software Engineering Notes*, vol. 29. ACP, 2004, pp. 1–4.
- [59] A. Derezińska and A. Szustek, "Object-oriented testing capabilities and performance evaluation of the c# mutation system," in *4th IFIP TC 2 Central and East European conference on Advances in Software Engineering Techniques*. Springer, 2009, pp. 229–242.
- [60] Y. S. Ma, M. Harrold, and Y. R. Kwon, "Evaluation of mutation testing for object-oriented programs," in *28th International Conference on Software Engineering*. ACM, 2006, pp. 869–872.
- [61] I. Moore, "Jester," in <http://jester.sourceforge.net/> (accessed September 24, 2017). SourceForge, 2001.
- [62] Y. S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system: Research articles," *Software Testing, Verification and Reliability*, vol. 15, no. 2, p. 2, 2005.
- [63] J. Bradbury, J. Cordy, and J. Dingel, "Exman: A generic and customizable framework for experimental mutation analysis," in *Second Workshop on Mutation Analysis*. IEEE, 2006.
- [64] "Jumble," in <http://jumble.sourceforge.net/> (accessed on September 24, 2017). SourceForge, 2007.

-
- [65] B. Grun, D. Schuler, and A. Zeller, “The impact of equivalent mutants,” in *International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 2009, pp. 192–199.
- [66] L. Madeyski and N. Radyk, “Judya mutation testing tool for java,” *IET Software*, vol. 4, no. 1, pp. 32–42, 2010.
- [67] J. Wegener, K. Buhr, and H. Pohlheim, “Automatic test data generation for structural testing of embedded software systems by evolutionary testing,” in *Genetic and Evolutionary Computation Conference*. ACM, 2002, pp. 1233–1240.
- [68] S. Wappler and F. Lammermann, “Using evolutionary algorithms for the unit testing of object-oriented software,” in *Genetic and Evolutionary Computation Conference*. ACM, 2005, pp. 1053–1060.
- [69] Y. Cheon, M. Kim, and A. Perumandla, “A complete automation of unit testing for java programs,” in *International Conference on Software Engineering Research and Practice*. UTEP, 2005.
- [70] A. Seesing and H. Gross, “A genetic programming approach to automated test generation for object-oriented software,” *International Transactions on Systems Science and Applications*, vol. 1, no. 2, pp. 127–134, 2006.
- [71] R. Pargas, M. Harrold, and R. Peck, “Test-data generation using genetic algorithms,” *Software Testing, Verification & Reliability*, vol. 9, no. 4, pp. 263–282, 1999.
- [72] P. McMinn, “Search-based software test data generation: a survey,” *Journal of Software Testing, Verifications, and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.
- [73] K. Liaskos, M. Roper, and M. Wood, “Investigating data-flow coverage of classes using evolutionary algorithms,” in *9th Annual Conference on Genetic and Evolutionary Computation*. ACM, 2007, pp. 1140–1140.

-
- [74] M. B. Bashir and A. Nadeem, “A state-based fitness function for the integration testing of object-oriented programs,” in *10th International Conference on Emerging Technologies*. IEEE, 2014, pp. 24–29.
- [75] “Geatbx,” in <http://www.geatbx.com/> (accessed September 24, 2017). GEATbx.com, 1995.

Appendix A

Literature Survey - Mutation Testing

In this appendix, we describe set of evaluation parameters, details of existing mutation testing techniques for object-oriented program, and present a brief analysis of those techniques. Also in the end we present some information about the automated solutions that exist in the literature.

A.1 Object-Oriented Mutation Testing

The object-oriented way of programming offers variety of features that makes it different from the structured paradigm. Some of them are listed below;

- It allows modeling of objects from real world to digital world using classes
- It classifies the data of a class into groups and restricts the access for safety (encapsulation)
- It encourages to reuse the code to reduce effort (inheritance) and
- It makes maintenance easy by allowing to add functionality without having to change the design of the software (polymorphism).

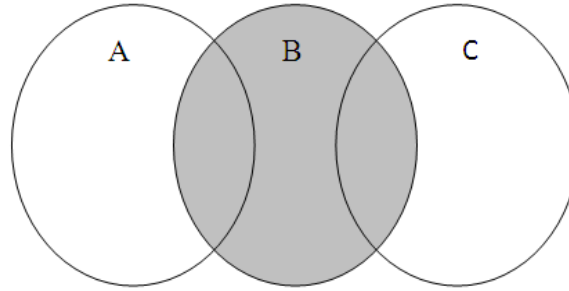


FIGURE A.1: Mutation Operators Set [1]

The above mentioned features present object-oriented paradigm as the most suitable for real world projects but testing object-oriented programs also becomes challenging due to them. The structured mutation testing techniques are not adequate to test object-oriented programs because their mutation operators are insufficient and cannot support object-oriented features. So besides using mutation operators of structured paradigm we need to introduce new operators so they can cover all the object-oriented features.

The overall steps and information used by object-oriented mutation testing is same with structured paradigm but there exists some difference. The inputs of the process include program under test and mutation operators. Using mutation operators, all possible mutants are generated. To kill those mutants, test cases are written and revised until all non-equivalent mutants are killed. The effective test cases and obtained mutation score are the outputs of the process. The difference exists in the format of test case and type of mutation operators. The test case contains more than just input values including call to the constructor for object creation, certain number of invocations to the different functions of the class and input values that are required for the function invocations. On the other hand mutation operators set is also larger than the structured paradigm's set to cover all the object-oriented features as shown in figure A.1;

The figure A.1 models the mutation operators set for structured and object-oriented paradigm. The figure tries to explain how for object-oriented paradigm the mutation operators set is larger than the set of structured paradigm. In general we know that almost all the features of structured programming language are supported by an object-oriented programming language including variables, data

types, operators (arithmetic, relational, conditional and so on), functions, and many other features. But object-oriented features are not supported by structured languages. Now if we take a look at figure A.1, there are there ovals there. The oval A represents mutation operators to cover the features of structured paradigm. The oval C represents those features that are specific to a given programming language like Java supports extensive file handling, exception handling, and so on. The oval B is the one that contains mutation operators for object-oriented features. It also overlaps to oval A and oval C that means it utilizes operators from the domain A and C. Hence the oval B contains larger mutation operators set.

A.2 Evaluation Criteria

In this section we present the list of parameters that we have devised to evaluate mutation testing techniques for object-oriented paradigm.

A.2.1 Cost Effective

The more a technique helps to reduce computational cost in mutation testing, the more it is suitable for practical use. For example a good technique may help to reduce execution of large number of mutants, which can be achieved by reducing mutation operators. We present evaluation criteria for this parameter in table A.1 below;

TABLE A.1: Evaluation Criteria for Cost Effective

Value	Criteria
Yes	If a technique offers reduced set of mutation operators, we assign value Yes to that technique.
No	If a technique does not offer reduced set of mutation operators, we assign value No to that technique.

A.2.2 Equivalent Mutant Detection

Equivalent mutants are semantically similar to the original program so they cannot be killed. The mutation testing technique should offer mutation operators that produce less number of equivalent mutants or at least the technique should be able to catch equivalent mutants to save the effort. The evaluation criterion is presented in Table A.2;

TABLE A.2: Evaluation Criteria for Equivalent Mutant Detection

Value	Criteria
Yes	If a technique can avoid generation of or can detect equivalent mutants, it gets Yes value for this parameter.
No	If a technique cannot avoid generation of or cannot detect equivalent mutants, it gets No value for this parameter.

A.2.3 OO Feature Coverage

An object-oriented mutation testing technique should provide full support to cover all the object-oriented features. In literature there is no metrics available to calculate or judge the amount of coverage a given technique provides so we will have to take some existing study as a benchmark. We find technique of Offutt et al. [39] most suitable for this purpose because it provides maximum support for object-oriented features. We present the evaluation criterion in Table A.3.

TABLE A.3: Evaluation Criteria for OO Feature Coverage

Value	Criteria
Full	We assign value Full to a technique that provides equal support for object-oriented features as compared to our benchmark study.
Partial	We assign value Partial to a technique that supports less amount of object-oriented feature as compared to our benchmark study.

A.2.4 Level of Testing

Some object-oriented features involve interaction of more than one class so a technique that operates at unit level cannot cover them. Mutation testing technique should provide inter-class testing support. We present the criterion for this parameter in Table A.4;

TABLE A.4: Evaluation Criteria for Level of Testing

Value	Criteria
AC	If a mutation testing technique operates at unit (class) level, we assign value AC to it.
IC	If a mutation testing technique supports and operates at inter-class level, we assign value IC to it.

A.2.5 State Mutation Support

Object posses data and states and a technique should be able to test all possible states of an object. This cannot be achieved without having to mutate object's state during execution. Most of the techniques support mutation at compile time but for a technique to be able to mutate object at runtime is also important. The criterion for this parameter is presented in Table A.5;

TABLE A.5: Evaluation Criteria for State Mutation Support

Value	Criteria
Yes	We assign value Yes to a technique that has the capability of mutating state of an object during testing.
No	We assign value No to a technique that does not have the capability of mutating state of an object.

A.2.6 Potential Mutation Operators

To make mutation testing more useful a set of reduced mutation operators is required that can ensure maximum feature coverage as well as got the potential to catch real faults. We present the criterion for this parameter in Table A.6;

TABLE A.6: Evaluation Criteria for Potential Mutation Operators

Value	Criteria
Yes	We assign value Yes to a technique if it lists of potential mutation operators.
No	We assign value No to a technique if it does not list of potential mutation operators.

A.2.7 Tool Support

A mutation testing technique is considered practical and usable if it provides automated solution in the form of a tool for the testers. The evaluation criterion is presented in Table A.7 for this parameter;

TABLE A.7: Evaluation Criteria for Tool Support

Value	Criteria
Yes	If a testing technique provides full tool support for performing mutation testing, we assign value Yes to that.
No	If a testing technique does not provide full tool support for performing mutation testing, we assign value No to that.

A.3 Surveyed Techniques

We need a specialized set of mutation operators for testing object-oriented programs with mutation testing because of the salient features of object-oriented

paradigm including encapsulation, inheritance, and polymorphism. Although survey shows a significant amount of research has already been done but still some areas need improvement. Besides mutation testing techniques, literature survey shows automated tools are also developed.

S. Kim, J. Clark, and J. McDermid [50]

Kim et al. [50] propose a novel approach that uses a concept of HAZOP [51] from another engineering discipline. HAZOP stands for Hazard and Operability Study. Authors have applied their approach on Java programming language specification. First they list out all the constructs of Java and then identify their attributes. Using a list of 10 guide words, authors find all the possible deviations in a Java element and the consequence of that deviation. When a meaningful deviation is found, it is listed as possible mutation operator that can be applied on a Java program. This way authors cover whole Java specification and as a result a comprehensive set of mutation operators is found by the authors. This is a generalized method and can be applied to any programming language to build its complete mutation operators set. But this is also the limitation of this approach because it finds a huge set of mutation operators and applying all of them on a program increases the cost of mutation testing.

S. Kim, J. Clark, and J. McDermid [52]

Kim et al. [52] propose an approach for object-oriented mutation testing. The name of their proposed approach is Class Mutation. This approach supports integration level mutation testing of object-oriented programs. This approach allows the tester to choose mutation operators as per requirement that makes it less expensive in nature. Despite the fact this approach supports inter-class level mutation operators but complete object-oriented feature coverage is not guaranteed. There are no experimental results provided by the authors in the published work though they claim that they have applied Class Mutation on an IBM application Product Starter Kit (PSK). This approach is generalized in nature because it can be merged with languages that have both non-object oriented features as well as object oriented features like C++ and Java.

P. Chevalley [53]

Chevalley [53] proposes an extension to the existing work of Kim et al. [52]. He adds 5 mutation operators for mutation testing of object-oriented programs. The new additions deal with inter-class level mutations. One noticeable contribution of this research is that author has tried to reduce the mutation operators set without compromising their impact on the testing process. Author has also done a survey on existing approaches and tools in his work. He has developed an automated tool with the name JavaMut that implements his approach. The tool has a graphical user interface that makes it easy to use for the testers. JavaMut supports 26 mutation operators covering both structured and object-oriented features. Out of 26, fifteen operators are taken from the work of Kim et al. [52], five are newly added operators, and six are selective mutation operators from structured paradigm. JavaMut has been used to perform experiments on a Flight Guidance System (FGS) and the results of the experiments are presented by the author. The new operators have been introduced based on the past experience of the author instead of using some rigorous method so this approach is not covering all the object-oriented features.

Y.-S. Ma, Y.-R. Kwon, and J. Offutt [54]

Ma et al. [54] propose a novel approach for integration level mutation testing of object oriented programs. Their proposal is based on an existing fault model of Gallagher et al. [55]. The fault model lists all possible faults that can appear while collaboration of more than one class. One major contribution of this work is the analysis of existing mutation testing approaches of Kim et al. [52], Chevalley, and Thvenod-Fosse [56], and Chevalley [53]. Authors state in their study that the work of aforementioned researchers is not generalized rather it is either application or problem specific. In this work, authors have provided formula to calculate number of mutants that a particular mutation operator can generate. Authors have developed and presented an automated solution that is based on a reflective system that operates at compile time.

R. Alexander, J. Bieman, S. Ghosh, and J. Bixia [57]

Alexander et al. [57] propose an approach that deals with mutation testing of Java objects. Authors state that mutation operators are designed to inject faults in the code that are syntactically correct as well as these operators are used to attain code coverage criterion like branch coverage. User defined objects are application specific so it is difficult to know their purpose without having knowledge about the reason of their creation and due to this reason it is difficult to design mutation operators for changing user defined object's state. But in general, some operators can be designed for Java packages because a package usually contains similar classes in it. This is one of the contributions of authors in this research. They also have built a Mutation Engine that generates mutants for each type of mutation. This is a unique work of authors because no other research in literature deals directly with objects though some experiments need to be performed to test its applicability on real world projects.

Offutt, Ma, and Kwon [39]

Offutt et al. [39] have extended their previous work [58] in this research. They have made some modifications in the mutation operators set and devised a new set by introducing 3 new operators, splitting 3 existing mutation operators, and merging 2 mutation operators into 1. The split operators are as follows; first is "super keyword deletion" (ISK), "static modifier change" (JSC), and "Member variable initialization deletion" (JTD). The six new operators that originate from these three include "super keyword deletion" (ISD), "super keyword insertion" (ISI), "static modifier insertion" (JSI), "static modifier deletion" (JSD), "this keyword deletion" (JTD), and "this keyword insertion" (JTI). Authors have also proposed new operators that include "Type cast operator insertion" (PCI), "Type cast operator deletion" (PCD), and "Cast type change" (PCC). Authors have also combined two existing operators to form a new one. The name of new operator is "Arguments of overloading method call change" (OAC), which is originated from "Argument order change" (OAD) and "Argument number change"

(OAN). Another contribution of authors in this research is that they present some general rules that can be followed in order to avoid generating equivalent mutants.

A. Derezińska and A. Szustek [59]

Derezińska and Szustek [59] present an approach on mutation testing of C# programs. They make some enhancements in the object-oriented mutation operators set to test C# programs comprehensively. The two major contributions that authors make includes; one is new object-oriented mutation operators for C# and to avoid generating invalid mutants and partially equivalent mutants, second they speed up the mutation testing process with the help of a new parser and reflection method as well as they reduce the space to store mutant updates. The authors have also discussed mutation testing of six open source projects that are commonly used. They have checked the quality of the test case set that is provided with these open source projects and have done performance evaluation too.

A.4 Analysis of Surveyed Techniques

Here in this section, we provide detailed analysis of mutation testing techniques that we have surveyed. We have presented seven mutation testing techniques in section A.3 and in section A.4 we have presented our benchmark parameters that we have used to perform the analysis. In table A.8, we present the analysis of surveyed techniques in row-column format. We have shown it in table for clarity and for better understanding of the reader. One side of the table presents surveyed techniques and in front of them we assign them a value for each of the parameter from evaluation criteria that we have devised for their evaluation.

TABLE A.8: Analysis Table of Mutation Testing Techniques for Object-Oriented Programs

Techniques	Evaluation Parameters						
	Cost Ef- fective	Equivalent Mutant Detection	OO Fea- ture Cov- erage	Level of Testing	State Mu- tation Support	Potential Mutation Operators	Tool Sup- port
Kim et al. [50]	No	No	Partial	IC	No	No	No
Kim et al. [52]	No	No	Partial	IC	No	No	No
Chevalley [53]	No	Yes	Partial	IC	No	No	Yes
Ma et al. [54]	No	Yes	Partial	IC	No	Yes	Yes
Alexander et al. [57]	No	No	Partial	IC	Yes	No	No
Offutt et al. [39]	No	Yes	Full	IC	No	No	Yes
Derezinska and Szustek [59]	No	No	Full	IC	No	No	No

Mutation testing is computationally expensive, which is why it has not gained wide range of acceptance in the software industry for testing real world projects. One way of reducing the high computational cost is to reduce execution of mutants, which can be done by reducing mutation operators set. Although we find reduced set of operators for structured paradigm but still this has to be done for object-oriented mutation operators. Some studies like Kim et al. [50, 52], Chevalley and Thvenod-Fosse [56], Ma et al. [60], and Offutt et al. [39] do discuss this issue and give their analysis on mutation operators that can be more useful in testing and operators that mostly generate equivalent mutants but still more work has to be done to find out reduced set of mutation operators. The study also needs to be backed up with large amount of experiments to prove the significance of selected operators set.

It is not possible to kill equivalent mutants because they are semantically similar to original program. Due to their presence the testing cost and effort increases for mutation testing. Detection and identification of equivalent mutants cannot be automated because it is an undecidable problem so no algorithm can be devised. Usually detection of equivalent mutants requires human (tester) inspection of the code. But we can still take some measures that can at least avoid generation of equivalent mutants. For this we need to know the mutation operators that have the tendency of generating equivalent mutants. Some existing research studies including Chevalley [53] and Ma et al. [54] have done some progress on this and they have shown with the help of experiments, which operators produce most of the equivalent mutants. Examples of such mutation operators include "Access Modifier Change" (AMC), "static Modifier Change" (SMC), "Instance Variable Declaration with Parent Class Type" (PMD), "Parent Variable Declaration with Child Class Type" (PPD), and "Hiding Variable Insertion" (IHI). These studies can be made as benchmark to convince testers not to use them during testing. In the study of Offutt et al. [39] we find 'equivalency conditions' that provide us situations that result in generation of equivalent mutants. If we avoid those equivalency conditions, we can avoid generation of equivalent mutants to a great deal.

The mutation testing technique should support mutation operators that can cover all the object-oriented features. Unless and until all the object-oriented features including encapsulation, inheritance, and polymorphism are covered, thoroughness of testing cannot be ensured. To do this we need a comprehensive and rigorous method to generate all possible mutation operators so they can ensure complete coverage of features. In literature, we find mutation operators set designed and proposed by Offutt et al. [39] that provide maximum coverage if we compare it with other related approaches.

A mutation testing technique should support testing of interaction among classes because two of object-oriented features including inheritance and polymorphism operate on more than one class. So if there is a technique that supports unit level of testing, it cannot test these object-oriented features. The techniques we find in the literature (Section A.3) for object-oriented paradigm do support inter-class level of testing.

An object contains data and states and during runtime the object can shift from one state to another as some operation is performed on it and when its data is changed. The mutation testing technique should be able to alter the states of an object at runtime forcefully to see if the test cases can distinguish between the states of objects involved in original and modified version of the programs. This concept is called state based testing and a very limited amount of work has been done so far in this domain. We find in the literature that Alexander et al. [57] proposed a method that changes the states of objects at runtime by manipulating their values. They apply their proposal on three Java APIs to demonstrate how it works.

Mutation testing is fault based testing technique so besides ensuring the object-oriented feature coverage, we need to design a technique that injects real faults in the programs. This can be achieved by studying the type of faults an object-oriented program can have. In the research study of Ma et al. [54], we find them using a fault model that presents object-oriented faults and they design mutation

operators using them. None of the other techniques that we find in the literature uses this method to generate mutants and perform mutation testing.

One way of reducing required effort to perform mutation testing is to use automated tools. Researcher usually developer prototype tools for their approaches to provide a proof that their approaches are practical and automate-able. In the literature we find the techniques of Chevalley and Thvenod-Fosse [56], Chevalley [53], Ma et al. [54], and Offutt et al. [39] are fully supported by a complete automated tool. Alexander et al. [57] also implement and present a tool in their work but that is just a prototype tool whereas Kim et al. [50, 52] provide no details of any tool in their research. Most of the surveyed techniques have also presented experiment results but in the work of Kim et al. [50, 52] and Alexander et al. [57] although we find discussion on experiments but their results are not presented by the researchers.

A.5 Mutation Testing Tools

In this subsection, we present the automated solutions that researchers have developed to aid testers.

JavaMut [56]

Chevalley and Thvenod-Fosse [56] have built a tool called JavaMut, which is a mutation testing tool for object-oriented programs. JavaMut uses a compile time reflection system known as OpenJava at back-end to obtain required information about Java class under test. In this tool the tester can view the mutant program using its graphical user interface. JavaMut supports up to 26 mutation operators from which 6 are selective operators from the work of Offutt et al. [37] and Barbosa et al. [38], fifteen are obtained from the work of Kim et al. [52], whereas five new operators in the tool have been introduced by the authors. The tool analyzes mutants before execution and considers only those, which do not have any syntax error and secondly, their byte-code differs from the byte-code of original program.

Jester [61]

Moore [61] has built an automated solution for mutation testing of object-oriented programs. This automation can test Java-based programs only and it has given a name, Jester. Jester can perform mutation testing at unit level because it can test only one Java class at a time. Therefore Jester has got support for only those mutation operators that can be applied to one class. Jester relies on JUnit [21] for test case generation. One limitation of Jester is that it does not have support for mutation operators that are designed for inheritance, association, and polymorphism.

Object Mutation Engine [57]

Alexander et al. [57] have developed a mutation testing tool to test Java APIs. The name of the tool is Object Mutation Engine (OME). OME comprises of seven different components that make it complex in nature. All these components perform different responsibilities; the overall execution of tool is controlled by OME Executor, Test Manager manages the execution of test job, State Inspector and Results Collection keeps an eye on the state of an object and its values, results are rendered in the required format by Result Rendering, Test Drivers prepare the environment, Mutation Operators contain one type of operators for mutation, and Test Evaluation to evaluate the mutation testing results.

MuJava [62]

Offutt et al. [58], and Ma et al. [62] have developed an automated tool called MuJava, which stands for Mutation System for Java. The tool performs mutation testing on Java programs through an easy to use user interface. MuJava performs all the mutation testing tasks like mutant generation and their execution. The earlier release of MuJava supports 29 mutation operators from which five are selective mutation operators [37] and rest are object-oriented operators [54]. The tool comprises of three components responsible for performing different tasks; Mutants Generator generates mutants, Mutants Executor executes them, and Mutants Viewer presents the results to tester on the tool interface. MuJava has been

used to perform extensive experimental evaluations on various Java programs and the results have been published [60] for the researchers. The later release of MuJava [39] tool has some modifications in the object-oriented mutation operators set. The current release of MuJava [39] tool now supports 29 object-oriented mutation operators. These mutation operators have been designed by making some omissions and additions to the previous set of operators.

ExMan [63]

Bradbury et al. [63] propose an approach for mutation analysis, which is generalized in nature. According to their proposal, the artifacts and components can be interchanged to compare quality assurance tools. While using this tool, they start through a setup phase in which they create a profile to tell command-line usage and purpose of using this application. Then they select a project that we want to run on it. After that they provide the tool with original program so it can generate mutants and can compile original and mutant programs. In the last phase they provide benchmarks to ExMan for the comparison of results or assertions. These benchmarks are used by the ExMan tool to perform analysis and to produce results.

Jumble [64]

Jumble [64] is similar in nature with Jester [61]. Jumble is also a mutation testing tool for Java-based programs that can test one class at a time only hence, it supports unit level mutation testing. Jumble uses JUnit [21] for generating test cases for the class under test. Just like Jester, the limitation of Jumble is that it does not have support for mutation operators that are designed for inheritance, association, and polymorphism.

JavaLanche [65]

Grun et al. [65] propose a framework and they implement it in a tool they name as JavaLanche. They want to determine the impact of equivalent mutants. JavaLanche supports only selective mutation operators. The tool runs only those

test cases that can execute mutated statement and to find out such a test case, the tool uses coverage data about the test case.

Judy [66]

Madeyski and Radyk [66] have developed a tool for mutation testing of Java programs. They have implemented a novel approach in the tool to reduce the required effort for mutation testing in terms of time. For this they use a concept of point-cut and advice, which is used in Aspect-oriented Paradigm to create collection of mutants. The collection of mutants help avoiding multiple compilations of mutant programs and it eventually reduces total time required for mutation testing. They have performed experiments using Judy and have compared them with an existing tool MuJava [58, 62].

Appendix B

Literature Survey Evolutionary Testing

In this appendix, we present details of existing evolutionary testing techniques for object-oriented program, and we provide some information about the automated solutions that exist in the literature.

B.1 Evolutionary Testing Techniques

We have considered those approaches for our survey that use Genetic Algorithm for evolutionary testing of object-oriented programs. Researchers have been working on devising an optimal fitness function that can evaluate a test case accurately as well as can guide the search.

P. McMinn and M. Holcombe [36]

McMinn and Holcombe [36] use an Ant Colony System in their proposed technique in order to problem of states in the program. The proposed technique uses constants (static) variables that do not lose their values even after the execution of a method. First they provide an example to describe the problem and then propose possible solutions in detail along with their limitations. They use data

dependence analysis and optimization at two levels to solve the state problem of programs. Their approach requires constructing a directed graph of complete program where the statements become nodes. Initially a test case is evaluated by path-oriented fitness function [23, 67] and the best among those are selected for ET-state algorithm. After that evaluation on state is performed, more nodes can be added to improve state if they are required. Their approach looks quite good but directed graph of complete program, path-oriented fitness function and two-level optimization adds up complexity. They have shown its application on programs written using structured paradigm and they have tested it on a small scale. Also their approach seems to be quite in-appropriate and time consuming when applied to object-oriented programs where several classes can be involved even in testing a single class and each class may have several methods.

P. Tonella [4]

Tonella [4] presents a technique that tests a class as a single unit and in isolation. His proposed approach supports unit testing using genetic algorithm. Tonella proposes a new template to write test cases for object-oriented programs and his approach supports mainly branch coverage. His technique uses four types of operations to repair a test case. These operations are mutating the input data, replacing constructors, removing and adding new method calls and one point crossover. His proposes to evaluate a test case on the basis of its potential of getting close to the respective target (approximation level). Test cases should be in the format of JUnit [21] finally in order to execute on the class under test. This technique requires insertion of assertions into driver class manually in order to execute the test cases, which is a limitation of this approach. Tonella has also developed a tool called eToc (Evolutionary Testing of Classes) to perform experiments. The results of Tonella's experiments look promising when using branch converge for testing a program.

S. Wappler and F. Lammermann [68]

Wappler and Lammermann [68] propose an approach that supports white-box testing of object-oriented programs using genetic algorithm. They use universal

evolutionary algorithms provided by popular toolboxes and these algorithms are application independent. This is also a major difference from other approaches, authors have presented till now. We can say that we do not need to build an approach for evolutionary testing from scratch rather we can use the generalized genetic algorithms by molding them according to the requirement. Authors provide proper encoding and decoding strategy in order to represent a test case. The test case consists of constructor invocation, method calls with their required parameters and the input required to be passed as the method parameters. To evaluate a test case, authors use different type of information in this technique. Their technique uses number of errors, constructor distance, and dynamic error evaluation. Also a tool has been developed in Matlab [29] for experimentation.

Y. Cheon, M. Kim, A. Perumandla [69]

Cheon et al. [69] propose a white-box testing technique for object-oriented programs. Their approach supports programs written in Java programming language. The main idea behind this work is to automate the tasks performed in testing phase. This will not only speed up the testing process but will also save some resources in the terms of time and money. According to their approach assertions are used to build an assertion tree. This tree can help identifying feasible method call sequences and evaluating object's state. To apply their technique on a Java class, the code should first be annotated using Java Modeling Language (JML) because the annotation is further used to detect assertions and to build the assertion tree. Sometimes specification of a program is not available and it may not have been written in some cases, therefore, for that program it will not be possible to perform annotation hence this technique will not be applicable on such a program, which is a huge limitation of this proposed work.

Y. Cheon and M. Kim [5]

Cheon and Kim [5] present a technique to apply white-box testing on object-oriented programs written in Java language. This particular approach is specification based written in Java Modeling Language (JML). Their proposed approach

can work in scenarios where source code is not available but it requires JML specification which is also a limitation of this approach. If a state variable is used in predicate, authors suggest changing the predicate condition for that variable. This technique looks simple and the experiments show that it has the potential but more experiments need to be performed on real world projects to check its effectiveness and applicability.

A. Seesing and H. Gross [70]

Seesing and Gross [70] use genetic algorithms to test object-oriented programs written in Java language. First they provide brief introduction of evolutionary testing and comparison of testing procedural and object-oriented programs. After that they present some related work and application of genetic algorithms to test object-oriented programs. They have proposed an approach to generate test software for testing object-oriented programs. For evaluation of a test case, the fitness function their approach uses is similar to the one proposed by Jones et al. [3] and Pargas et al. [71]. They have evaluated their approach on a very small scale but the results show promising results. Since this approach inherits fitness function from Jones et al. [3] and Pargas et al. [71], hence it also inherits the problem of guidance for search process, which is also discussed by McMinn [72]

K. Liaskos, M. Roper, and M. Wood [73]

Liaskos et al. [73] present their work which is in fact an extension of Tonellas [4] work. Their main focus is on the data-flow coverage while using genetic algorithm for evolutionary testing. They first describe the challenges of object-oriented evolutionary testing and then present brief description on data-flow coverage. They present the testing framework for data-flow coverage afterwards. Authors then discuss three different levels of data-flow coverage of methods in unit testing of classes. Their approach requires approximation level to evaluate a test case, which indicates how close a test case was able to reach to the target. They have applied their proposed solution on six classes of Java library and have compared the results of data-flow coverage with the results presented in the work of Tonella [4].

M. B. Bashir and A. Nadeem [2]

Bashir and Nadeem [2] present a novel fitness function for performing evolutionary testing on object-oriented programs. Their basic aim of this proposal is to solve the object's state problem and for this they propose to isolate object's state variables fitness from the fitness of local variables. When object's state fitness is isolated, the search process gets better guidance that helps to achieve the targets quickly. As a proof of concept they have also developed a prototype tool called SOFT (State Oriented Fitness evaluation of Test-cases) and performed some experiments using it. The results obtained from experiments are interesting and indicate the usefulness of their proposal but large scale experiments are required to further validation. Bashir and Nadeem [74] extended this work proposed a novel approach for integration level testing of object-oriented programs. This technique also uses state-based fitness function for the evaluation of test cases generated to test interactions between two objects. The idea works well at the integration level as the technique evaluates a test case on the basis of its ability to gain the desired state of objects involved in the interaction.

B.2 Evolutionary Testing Tools

We have covered the proposed approaches by researchers in the area of evolutionary testing of object-oriented programs and next we present the available automated solutions for the aid of testers.

eToc [4]

Tonella [4] has developed a tool called eToc (Evolutionary Testing of Classes) for his proposed technique and has presented the results of testing Java standard classes using this tool. The experiments are performed using six standard API classes of Java. The eToc tool supports two white-box coverage criterion including branch coverage and data-flow coverage. Tonella has performed experiments with eToc to test its ability to attain high branch coverage. The results of Tonella's

experiments look promising when using branch converge for testing a program. eToc is a command based tool and it is available for download and experimentation [4].

Implementation by Wappler and Lammermann [68]

Another implementation that we have found in the literature is done by Wappler and Lammermann [68]. They have implemented their approach in Matlab [29] using genetic algorithm toolbox GEATbx [75]. The implementation demonstrates their approach and the purpose of this implementation is to perform experiments for empirical evaluation. It requires test cases to be in the format supported by JUnit [21] since it uses that format for the execution of test cases.

TCGOJ [42]

Dharsana and Askarunisha [42] have implemented an evolutionary testing tool for Java programs. The name of that tool is TCGOJ (Test Case Generation and Optimization for Java Programs). The tool can generate optimized set of test cases for any Java program using Genetic Algorithm that it implements. TCGOJ uses crossover operation on chromosomes as well as does biological mutation to gain maximum coverage.

SOFT [2]

Bashir and Nadeem [2] have developed a tool for evolutionary testing of Java programs. They name the tool as SOFT that stands for State Oriented Fitness evaluation of Test-cases. SOFT tool performs testing of a Java programs at unit (class) level and generates test cases to achieve statement coverage. The tool implements Genetic Algorithm with their proposed state-based fitness function. At first the SOFT tool takes source code of Java class as input, performs evolutionary testing on the class under test, and eventually produces test case set for statement coverage. This is a prototype tool developed in Java language and it supports limited Java constructs.

Appendix C

Java Subset for eMuJava Tool

In this appendix we present the details of Java language subset we have chosen and assumptions that we have made for the implementation of eMuJava tool.

C.1 Java Language Subset for eMuJava

For implementation of eMuJava tool, we have selected a subset of Java language and cut down the complete set of keywords and syntax to a small set. This section provides details about this.

C.1.1 Context Free Grammar (CFG)

We have chosen a subset of Java programming language to implement the tool and we have defined a context free grammar for it. The CFG is presented in figure C.1.

C.1.2 Keywords

We have chosen a subset of Java keywords that are supported in the eMuJava tool. Table C.1 shows the list of supported keywords in eMuJava tool.

```

S → class
class → class_signature { data_declaration constructor method }
class_signature → public class identifier
data_declaration → access_modifier static data_type identifier; data_declaration | ε
access_modifier → public | private
static → static | ε
data_type → byte | short | int | long
identifier → alphabet alpha_numeric
alpha_numeric → alphabet digit alpha_numeric | digit alphabet alpha_numeric | ε
digit → 0|1|2|...|9
integer → digit | digit integer | ε
alphabet → A|B|C|...|X|Y|Z|a|b|c|...|x|y|z
constructor → constructor_signature { statements }
constructor_signature → access_modifier identifier( parameter_list )
method → method_signature { statements return_statement }
method_signature → access_modifier return_type identifier( parameter_list )
return_type → void | data_type
parameter_list → parameter | parameter, parameter_list | ε
parameter → data_type identifier
return_statement → return identifier | return integer | ε
statements → assignment statements | function_call statements | conditions statements | ε
assignment → identifier = expression; | this.identifier = expression; |
                local_declaration = expression;
expression → identifier | digit | expression arith_operator expression
arith_operator → + | - | * | / | %
local_declaration → data_type identifier
function_call → identifier( argument_list ); | this.identifier( argument_list );
argument_list → identifier | identifier, argument_list | ε
conditions → if_condition | while_condition
if_condition → if( predicate ) { statements }
while_condition → while( predicate ) { statements }
predicate → identifier rel_operator identifier | identifier rel_operator digit
rel_operator → < | <= | > | >= | != | ==

```

FIGURE C.1: Context Free Grammar for eMuJava Tool

TABLE C.1: Supported Keywords by eMuJava

boolean	byte	char	class
double	else	extends	float
if	int	long	private
protected	public	return	short
static	this	void	while

C.1.3 Data Types

We have considered the following primitive data types of Java shown in Table C.2.

TABLE C.2: List of Primitive Data Types

byte	short	int	long
float	double	char	String

C.1.4 Special Symbols

We have considered the following special symbols of Java. Table C.3 shows the list of special symbols supported by eMuJava tool.

TABLE C.3: List of Special Symbols

Special Symbols	() { } ; : .
-----------------	---------------

C.1.5 Operators

We have considered the following operators of Java, which is shown in Table C.5.

TABLE C.4: List of Operators

Relational Operators	< <= > >= == !=
Conditional Operators	&&
Arithmetic Operators	+ - / * %
Assignment Operator	=

C.1.6 Tool Assumptions

We have chosen a limited set of Java programming language and implemented it in our tool. There are certain assumptions regarding the source code which we have mentioned below. The subset of Java programming language that we have chosen can demonstrate our proposed technique and due to this reason we have not implemented complete Java programming language syntax. Due to the limited number of control constructs there are few assumptions that we have made and they are categorized into three groups which we have explained below:

1. General

- There should be no syntax error in the source code.
- Java standard libraries are not supported so there should be no import statement in the code.

2. Declarations

- Only primitive data types and String class is supported.
- Arrays are not supported.

3. Methods and Statements

- Methods can only be non-static.
- Class under test should not have `main()` method.
- Only `if`, `if-else`, and `while` conditions should be used in the code, nesting is allowed though.
- There ! logical operator should not be there in the source code.
- Ternary operator is not considered.

Appendix D

Additional Experiment Results

In this appendix we present the results of experiments that are used to generate line-charts of figure 6.4 in chapter 6. These are results of 20 case studies that are generated using eMuJava (see Chapter 5) tool.

We will use short names in the table headers to save some space. The list is given below and Table D.1 lists all the case studies in experiments.

M#: Mutant Number

RT: Random Testing

SGA: GA with Standard Fitness Function

PGA: GA with State-based & Control-oriented Fitness Function

IGA: Improved Genetic Algorithm

TABLE D.1: List of Case Studies

No.	Case Study	No.	Case Study	No.	Case Study	No.	Case Study
1	AutoDoor	6	JCS	11	CSV	16	Math
2	Calculator	7	CGPACalc	12	ElectricHeater	17	Stack
3	BankAccount	8	Collections	13	HashTable	18	TempConverter
4	CLI	9	Compress	14	Lang	19	Text
5	BinarySearchTree	10	Crypto	15	Logging	20	Triangle

TABLE D.2: Iterations Used while Test Case Generation for AutoDoor

M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA
1	1	12	1	1	25	173	203	181	55	49	197	227	205	79
2	2	13	2	2	26	174	204	182	56	50	198	228	206	80
3	3	25	3	3	27	175	205	183	57	51	199	229	207	81
4	4	26	4	4	28	176	206	184	58	52	200	230	208	82
5	5	27	5	5	29	177	207	185	59	53	201	231	209	83
6	6	28	6	6	30	178	208	186	60	54	202	232	210	84
7	7	29	7	7	31	179	209	187	61	55	203	233	211	85
8	8	30	8	8	32	180	210	188	62	56	204	234	212	86
9	9	31	9	9	33	181	211	189	63	57	205	235	213	87
10	10	32	10	10	34	182	212	190	64	58	206	236	214	88
11	11	33	11	11	35	183	213	191	65	59	207	237	215	89
12	44	73	51	21	36	184	214	192	66	60	208	238	216	90
13	82	113	91	29	37	185	215	193	67	61	209	239	217	91
14	117	153	131	35	38	186	216	194	68	62	210	240	218	92
15	152	193	171	45	39	187	217	195	69	63	211	241	219	93
16	153	194	172	46	40	188	218	196	70	64	212	242	220	94
17	154	195	173	47	41	189	219	197	71	65	213	243	221	95
18	155	196	174	48	42	190	220	198	72	66	214	244	222	96
19	156	197	175	49	43	191	221	199	73	67	239	269	250	117
20	157	198	176	50	44	192	222	200	74	68	264	294	278	127
21	158	199	177	51	45	193	223	201	75	69	286	297	296	133
22	159	200	178	52	46	194	224	202	76	70	308	309	306	138
23	160	201	179	53	47	195	225	203	77	71	309	310	307	139
24	172	202	180	54	48	196	226	204	78	72	310	311	308	140

TABLE D.3: Iterations Used while Test Case Generation for Calculator

M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA
1	1	1	1	1	25	82	208	159	101	49	106	257	183	134
2	2	2	2	2	26	83	209	160	102	50	107	258	185	135
3	3	3	4	4	27	84	210	161	103	51	108	259	186	136
4	4	4	5	5	28	85	211	162	104	52	110	260	187	137
5	5	5	6	6	29	86	212	163	105	53	111	261	188	138
6	6	6	7	7	30	87	213	164	106	54	112	262	189	139
7	7	7	8	8	31	88	214	165	107	55	113	263	190	140
8	8	8	9	9	32	89	215	166	108	56	114	264	191	141
9	9	9	10	10	33	90	216	167	109	57	115	265	192	142
10	10	10	11	11	34	91	217	168	110	58	116	266	193	143
11	11	11	18	25	35	92	218	169	111	59	117	267	194	144
12	12	40	30	30	36	93	219	170	112	60	118	268	195	145
13	18	57	42	35	37	94	220	171	113	61	119	269	196	146
14	20	87	43	45	38	95	221	172	114	62	120	270	197	147
15	36	117	83	61	39	96	222	173	115	63	121	271	198	148
16	40	143	90	70	40	97	223	174	116	64	122	272	199	149
17	43	146	108	71	41	98	224	175	117	65	123	273	200	150
18	49	174	115	72	42	99	225	176	118	66	124	274	201	151
19	60	202	130	85	43	100	226	177	119	67	125	275	202	152
20	71	203	139	96	44	101	227	178	120	68	126	276	203	153
21	78	204	155	97	45	102	228	179	121	69	127	277	204	154
22	79	205	156	98	46	103	229	180	122	70	128	278	205	155
23	80	206	157	99	47	104	230	181	123	71	129	279	206	156
24	81	207	158	100	48	105	232	182	124	72	130	280	207	157

TABLE D.4: Iterations Used while Test Case Generation for BankAccount

M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA
1	1	1	1	1	42	48	43	42	42	83	89	84	83	83
2	2	2	2	2	43	49	44	43	43	84	90	85	84	84
3	3	3	3	3	44	50	45	44	44	85	91	86	85	85
4	4	4	4	4	45	51	46	45	45	86	92	87	86	86
5	5	5	5	5	46	52	47	46	46	87	93	88	87	87
6	6	6	6	6	47	53	48	47	47	88	94	89	88	88
7	7	7	7	7	48	54	49	48	48	89	95	90	89	89
8	8	8	8	8	49	55	50	49	49	90	96	91	90	90
9	9	9	9	9	50	56	51	50	50	91	97	92	91	91
10	10	10	10	10	51	57	52	51	51	92	98	93	92	92
11	12	11	11	11	52	58	53	52	52	93	99	94	93	93
12	14	12	12	12	53	59	54	53	53	94	100	95	94	94
13	16	13	13	13	54	60	55	54	54	95	101	96	95	95
14	18	14	14	14	55	61	56	55	55	96	102	97	97	96
15	20	16	15	15	56	62	57	56	56	97	103	98	99	97
16	22	17	16	16	57	63	58	57	57	98	105	100	101	98
17	23	18	17	17	58	64	59	58	58	99	106	101	102	99
18	24	19	18	18	59	65	60	59	59	100	107	102	103	100
19	25	20	19	19	60	66	61	60	60	101	108	103	104	101
20	26	21	20	20	61	67	62	61	61	102	109	104	105	102
21	27	22	21	21	62	68	63	62	62	103	110	105	106	103
22	28	23	22	22	63	69	64	63	63	104	111	106	107	104
23	29	24	23	23	64	70	65	64	64	105	112	107	109	105
24	30	25	24	24	65	71	66	65	65	106	113	108	110	106
25	31	26	25	25	66	72	67	66	66	107	114	115	112	107
26	32	27	26	26	67	73	68	67	67	108	115	116	113	108
27	33	28	27	27	68	74	69	68	68	109	116	117	114	109
28	34	29	28	28	69	75	70	69	69	110	117	118	115	110
29	35	30	29	29	70	76	71	70	70	111	118	119	116	111
30	36	31	30	30	71	77	72	71	71	112	119	120	117	112
31	37	32	31	31	72	78	73	72	72	113	120	121	118	113
32	38	33	32	32	73	79	74	73	73	114	122	123	119	115
33	39	34	33	33	74	80	75	74	74	115	123	124	120	116
34	40	35	34	34	75	81	76	75	75	116	124	125	121	117
35	41	36	35	35	76	82	77	76	76	117	125	126	122	118
36	42	37	36	36	77	83	78	77	77	118	126	127	123	119
37	43	38	37	37	78	84	79	78	78	119	127	128	124	120
38	44	39	38	38	79	85	80	79	79	120	128	129	125	121
39	45	40	39	39	80	86	81	80	80	121	129	130	126	122
40	46	41	40	40	81	87	82	81	81	122	130	131	127	123
41	47	42	41	41	82	88	83	82	82					

TABLE D.5: Iterations Used while Test Case Generation for CLI

M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA
1	1	2	1	1	9	28	33	30	18	17	72	78	75	61
2	2	3	4	2	10	45	47	45	34	18	74	79	78	64
3	12	15	14	5	11	46	50	48	36	19	75	80	79	66
4	13	16	15	8	12	47	51	54	38	20	78	83	80	67
5	14	18	16	9	13	50	52	55	42	21	80	85	82	70
6	15	20	17	11	14	53	56	56	44	22	83	98	84	74
7	16	21	19	15	15	55	59	60	46	23	95	110	85	79
8	27	31	28	17	16	58	62	61	47					

TABLE D.6: Iterations Used while Test Case Generation for BinarySearchTree

M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA
1	1	1	1	1	35	83	61	133	86	69	197	216	245	198
2	2	2	2	4	36	84	65	137	94	70	198	217	246	199
3	5	4	3	5	37	85	69	145	95	71	199	218	247	200
4	6	6	4	6	38	97	85	157	105	72	201	219	248	201
5	7	7	16	9	39	98	87	158	106	73	202	220	250	203
6	13	8	18	10	40	102	89	159	109	74	204	221	252	204
7	18	10	25	22	41	111	102	171	123	75	205	222	253	205
8	22	12	32	28	42	112	103	173	125	76	206	223	255	206
9	29	13	34	34	43	116	104	175	127	77	207	224	256	207
10	33	15	46	40	44	119	105	177	134	78	208	225	257	208
11	36	17	58	46	45	122	106	178	137	79	209	226	258	209
12	40	18	74	47	46	127	107	180	143	80	210	227	259	210
13	45	20	83	59	47	141	132	181	144	81	211	228	260	211
14	46	21	84	61	48	143	133	184	145	82	212	229	261	212
15	47	25	86	63	49	149	145	185	153	83	213	230	262	213
16	50	31	88	64	50	152	146	187	157	84	214	231	263	214
17	52	35	96	67	51	153	147	188	158	85	215	232	264	215
18	54	38	99	68	52	156	148	189	160	86	216	233	265	216
19	55	43	104	69	53	173	173	205	162	87	222	236	266	218
20	58	44	110	70	54	176	185	221	164	88	230	239	268	220
21	59	45	117	72	55	177	186	223	167	89	233	240	270	221
22	70	47	118	73	56	178	188	224	175	90	234	241	272	222
23	71	48	119	74	57	182	189	225	176	91	239	244	274	228
24	72	49	120	75	58	183	190	226	179	92	242	247	276	234
25	73	51	123	76	59	186	203	231	187	93	244	248	277	237
26	74	52	124	77	60	187	204	234	188	94	245	249	278	240
27	75	53	125	78	61	188	205	235	189	95	247	250	279	241
28	76	54	126	79	62	189	206	236	190	96	253	251	283	245
29	77	55	127	80	63	190	209	237	191	97	256	255	287	246
30	78	56	128	81	64	191	210	238	192	98	259	259	290	251
31	79	57	129	82	65	192	211	239	193	99	260	260	291	252
32	80	58	130	83	66	193	212	240	194	100	261	261	292	253
33	81	59	131	84	67	194	213	241	195	101	262	262	293	254
34	82	60	132	85	68	195	215	243	197	102	263	263	294	255

TABLE D.7: Iterations Used while Test Case Generation for JCS

M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA
1	7	6	7	5	19	102	115	87	87	37	125	134	105	105
2	14	12	15	12	20	103	116	88	88	38	128	135	108	106
3	21	19	22	19	21	104	117	89	89	39	130	136	110	107
4	28	25	29	25	22	105	118	90	90	40	131	140	112	108
5	36	33	36	30	23	106	120	91	91	41	138	147	115	109
6	43	42	42	35	24	107	121	92	92	42	145	152	122	115
7	48	45	43	36	25	108	122	93	93	43	152	159	128	120
8	54	50	44	37	26	109	123	94	94	44	158	166	135	125
9	55	55	45	38	27	110	124	95	95	45	165	170	140	135
10	56	56	46	39	28	111	125	96	96	46	170	177	145	145
11	57	57	47	40	29	112	126	97	97	47	171	178	146	146
12	59	63	48	42	30	113	127	98	98	48	175	180	147	147
13	65	64	55	49	31	114	128	99	99	49	182	185	150	155
14	72	65	62	55	32	115	129	100	100	50	189	195	158	156
15	79	78	66	60	33	118	130	101	101	51	196	202	165	160
16	85	85	73	68	34	120	131	102	102	52	203	210	175	162
17	88	95	79	78	35	122	132	103	103					
18	95	102	86	86	36	123	133	104	104					

TABLE D.8: Iterations Used while Test Case Generation for CGPACalc

M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA
1	10	21	12	10	21	32	125	65	33	41	53	145	87	53
2	11	30	13	11	22	33	126	66	34	42	54	146	88	54
3	12	46	14	12	23	34	127	67	35	43	55	147	89	55
4	13	47	23	13	24	35	128	68	36	44	56	148	90	56
5	14	60	24	16	25	36	129	69	37	45	57	149	91	57
6	16	61	28	17	26	37	130	70	38	46	58	150	103	60
7	17	73	29	18	27	38	131	71	39	47	60	151	108	61
8	18	74	30	19	28	39	132	72	40	48	64	152	120	62
9	20	106	42	21	29	40	133	73	41	49	65	153	132	63
10	21	107	43	22	30	41	134	74	42	50	73	169	144	64
11	22	108	44	23	31	42	135	75	43	51	74	189	145	65
12	23	109	45	24	32	43	136	76	44	52	75	190	146	66
13	24	110	46	25	33	44	137	79	45	53	76	191	147	67
14	25	111	47	26	34	45	138	80	46	54	77	192	148	69
15	26	112	48	27	35	46	139	81	47	55	78	193	149	70
16	27	120	49	28	36	48	140	82	48	56	79	215	171	71
17	28	121	50	29	37	49	141	83	49	57	80	216	172	72
18	29	122	51	30	38	50	142	84	50	58	81	217	173	73
19	30	123	52	31	39	51	143	85	51	59	93	229	195	74
20	31	124	64	32	40	52	144	86	52					

TABLE D.9: Iterations Used while Test Case Generation for Collections

M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA
1	1	2	1	1	30	133	112	107	119	59	202	185	178	160
2	8	9	2	10	31	141	118	115	126	60	203	186	179	161
3	15	14	3	18	32	142	120	116	127	61	204	187	180	162
4	22	21	10	27	33	143	121	117	128	62	205	188	181	163
5	29	29	17	36	34	144	122	118	129	63	206	189	182	164
6	36	36	24	45	35	145	123	119	130	64	207	190	183	165
7	38	37	25	46	36	146	124	120	131	65	208	191	184	166
8	39	38	26	47	37	147	125	121	132	66	209	192	185	167
9	40	39	27	48	38	148	126	122	133	67	210	193	186	168
10	42	40	28	49	39	149	127	123	134	68	211	194	187	169
11	43	41	29	50	40	150	128	124	135	69	212	195	188	170
12	50	42	36	52	41	156	135	131	136	70	213	196	189	171
13	57	43	43	59	42	163	142	139	137	71	214	197	190	172
14	63	50	50	66	43	170	150	147	138	72	221	205	191	173
15	68	57	57	67	44	171	151	148	139	73	228	215	192	174
16	78	64	58	68	45	172	152	149	140	74	229	216	193	175
17	85	70	59	76	46	173	153	150	141	75	230	217	194	176
18	92	78	66	83	47	174	154	151	142	76	231	218	195	177
19	98	84	72	90	48	175	155	152	143	77	232	219	196	178
20	106	90	78	97	49	176	156	153	144	78	233	220	197	179
21	107	91	79	98	50	177	157	154	145	79	234	221	198	180
22	108	92	80	99	51	178	158	155	146	80	235	222	199	181
23	109	93	81	100	52	179	159	156	147	81	236	223	200	182
24	110	94	82	101	53	180	160	157	148	82	237	224	201	183
25	111	95	83	102	54	181	161	158	149	83	238	225	202	184
26	112	96	84	103	55	182	162	159	150	84	239	226	203	185
27	113	97	85	104	56	183	163	160	151	85	240	227	204	186
28	120	104	92	105	57	190	170	161	152	86	241	228	205	187
29	126	105	99	112	58	195	177	169	153	87	242	229	206	188

TABLE D.10: Iterations Used while Test Case Generation for Compress

M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA
1	10	9	11	10	25	82	73	50	69	49	194	176	112	123
2	20	18	20	21	26	83	74	51	70	50	195	177	113	124
3	31	27	21	32	27	84	77	52	71	51	196	178	114	125
4	42	37	22	33	28	85	80	53	72	52	197	179	115	126
5	52	47	23	34	29	86	81	54	73	53	198	180	116	127
6	63	49	24	43	30	87	82	55	74	54	199	181	117	128
7	64	51	25	46	31	88	83	56	75	55	200	182	118	129
8	65	52	26	47	32	99	84	57	85	56	201	185	119	130
9	66	53	27	48	33	108	86	58	86	57	202	188	120	132
10	67	55	28	49	34	115	96	60	87	58	203	189	121	135
11	68	57	29	50	35	125	106	70	89	59	204	190	122	137
12	69	58	30	51	36	135	115	79	99	60	205	191	123	138
13	70	59	31	52	37	146	135	88	100	61	215	192	135	140
14	71	62	32	53	38	157	142	97	102	62	225	193	148	150
15	72	63	33	54	39	166	149	98	103	63	235	203	159	151
16	73	64	35	55	40	177	158	99	114	64	247	213	170	159
17	74	65	37	56	41	186	168	100	115	65	248	214	174	160
18	75	66	38	57	42	187	169	101	116	66	249	215	178	161
19	76	67	40	59	43	188	170	103	117	67	250	216	179	162
20	77	68	43	61	44	189	171	105	118	68	251	217	180	163
21	78	69	45	62	45	190	172	108	119	69	252	218	182	164
22	79	70	47	64	46	191	173	109	120	70	253	219	183	165
23	80	71	48	66	47	192	174	110	121					
24	81	72	49	68	48	193	175	111	122					

TABLE D.11: Iterations Used while Test Case Generation for Crypto

M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA
1	1	2	1	1	27	30	34	35	32	53	145	125	103	102
2	2	4	3	2	28	35	39	40	37	54	150	131	108	107
3	3	6	4	3	29	40	44	47	42	55	156	133	113	112
4	4	8	6	4	30	46	49	54	47	56	162	138	119	117
5	5	9	7	5	31	51	50	55	48	57	167	144	126	125
6	6	11	10	6	32	56	55	57	55	58	172	149	130	129
7	7	12	11	7	33	62	61	59	56	59	173	150	131	130
8	8	14	12	8	34	67	66	60	57	60	174	151	132	131
9	9	15	13	9	35	68	67	62	58	61	179	156	137	136
10	10	17	14	10	36	69	68	63	59	62	184	161	142	141
11	11	18	15	11	37	74	73	64	60	63	189	166	148	148
12	12	19	16	12	38	79	77	65	61	64	194	170	153	154
13	13	20	17	13	39	84	82	70	66	65	199	175	158	159
14	14	21	18	14	40	89	87	76	72	66	205	180	163	164
15	15	22	19	15	41	94	92	82	77	67	212	185	169	165
16	16	23	20	16	42	100	93	83	78	68	218	193	174	166
17	18	24	22	17	43	106	94	84	80	69	219	194	175	168
18	20	25	23	18	44	112	95	86	81	70	220	195	176	170
19	22	26	25	19	45	120	100	87	82	71	226	200	181	172
20	23	27	27	20	46	125	105	92	87	72	232	205	188	177
21	24	28	28	21	47	130	111	97	96	73	239	211	192	181
22	25	29	29	22	48	131	112	98	97	74	247	216	198	186
23	26	30	30	23	49	132	113	99	98	75	253	221	202	191
24	27	31	31	25	50	133	114	100	99	76	259	228	207	196
25	28	32	33	27	51	134	115	101	100	77	265	233	212	200
26	29	33	34	29	52	139	120	102	101	78	270	239	217	204

TABLE D.12: Iterations Used while Test Case Generation for CSV

M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA
1	1	1	2	1	35	331	373	336	304	69	369	484	431	338
2	2	3	4	2	36	332	374	337	305	70	370	485	434	339
3	17	18	15	16	37	333	376	339	306	71	371	486	436	340
4	32	32	30	30	38	334	377	340	307	72	372	487	438	341
5	47	45	46	45	39	335	390	348	308	73	373	488	441	342
6	62	60	60	60	40	336	402	363	309	74	374	489	442	343
7	77	75	70	72	41	337	416	375	310	75	375	490	443	344
8	87	90	85	87	42	338	430	390	311	76	376	491	445	345
9	97	105	96	100	43	339	444	405	312	77	377	493	447	350
10	113	118	97	101	44	340	449	406	313	78	378	495	450	352
11	114	133	99	102	45	341	460	407	314	79	379	498	451	355
12	115	134	100	103	46	342	461	408	315	80	380	500	452	358
13	116	135	102	104	47	343	462	409	316	81	381	501	453	360
14	118	136	103	105	48	344	463	410	317	82	382	502	454	362
15	133	150	118	120	49	345	464	411	318	83	383	503	455	363
16	148	165	130	130	50	346	465	412	319	84	384	504	456	364
17	162	180	150	144	51	347	466	413	320	85	385	505	457	365
18	177	195	160	158	52	348	467	414	321	86	386	506	458	366
19	190	210	175	170	53	349	468	415	322	87	388	507	459	367
20	204	222	188	185	54	350	469	416	323	88	389	508	460	368
21	219	236	200	200	55	351	470	417	324	89	390	509	461	369
22	233	250	212	212	56	352	471	418	325	90	392	510	462	370
23	234	262	225	225	57	353	472	419	326	91	394	511	463	371
24	236	263	226	226	58	354	473	420	327	92	396	512	464	372
25	237	264	228	227	59	355	474	421	328	93	398	513	465	373
26	239	265	229	228	60	356	475	422	329	94	399	514	466	374
27	240	280	244	229	61	357	476	423	330	95	400	515	467	375
28	241	294	258	244	62	358	477	424	331	96	401	516	468	376
29	256	308	270	259	63	359	478	425	332	97	402	517	469	377
30	271	323	280	274	64	361	479	426	333	98	403	518	470	378
31	286	333	294	275	65	363	480	427	334	99	404	520	471	379
32	300	348	305	285	66	365	481	428	335	100	405	522	472	380
33	314	360	320	302	67	367	482	429	336	101	406	523	473	381
34	330	372	335	303	68	368	483	430	337	102	409	525	474	382

TABLE D.13: Iterations Used while Test Case Generation for ElectricHeater

M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA
1	1	1	1	2	35	146	213	226	156	69	248	371	313	268
2	2	2	2	4	36	147	214	227	157	70	249	372	314	269
3	3	3	4	6	37	148	215	228	158	71	250	373	315	270
4	4	4	5	8	38	149	216	229	159	72	251	374	316	271
5	5	5	6	9	39	150	217	230	160	73	252	375	317	272
6	6	6	13	11	40	151	218	231	161	74	253	376	318	273
7	7	7	15	12	41	152	219	232	162	75	254	377	319	274
8	8	8	17	14	42	153	220	233	163	76	255	378	320	275
9	9	9	18	15	43	154	222	235	164	77	256	379	321	276
10	12	11	20	17	44	155	223	236	165	78	257	380	322	277
11	13	12	21	18	45	156	224	237	166	79	258	381	323	278
12	14	13	22	19	46	157	225	238	167	80	259	382	324	279
13	15	15	23	20	47	160	227	239	168	81	260	383	325	280
14	16	16	24	21	48	161	228	240	169	82	261	385	345	282
15	38	41	45	42	49	163	230	241	170	83	269	387	365	289
16	52	55	48	54	50	164	231	242	176	84	270	388	366	290
17	80	80	85	85	51	165	232	243	177	85	271	389	367	291
18	85	115	122	99	52	166	238	255	179	86	272	390	368	292
19	92	150	138	105	53	170	244	256	181	87	273	391	369	293
20	99	164	175	111	54	181	255	263	192	88	282	403	371	303
21	132	199	212	142	55	182	267	264	206	89	289	404	387	304
22	133	200	213	143	56	193	297	268	221	90	292	405	388	305
23	134	201	214	144	57	203	311	278	222	91	293	406	394	306
24	135	202	215	145	58	207	324	279	234	92	295	407	400	307
25	136	203	216	146	59	214	336	282	237	93	296	408	401	308
26	137	204	217	147	60	215	348	292	245	94	316	439	432	322
27	138	205	218	148	61	220	349	304	254	95	336	470	463	336
28	139	206	219	149	62	221	351	305	255	96	354	472	494	349
29	140	207	220	150	63	222	353	306	259	97	361	495	525	367
30	141	208	221	151	64	241	365	307	263	98	364	526	556	371
31	142	209	222	152	65	243	366	309	264	99	378	557	587	373
32	143	210	223	153	66	244	367	310	265	100	379	558	588	374
33	144	211	224	154	67	245	368	311	266	101	380	559	589	375
34	145	212	225	155	68	247	370	312	267					

TABLE D.14: Iterations Used while Test Case Generation for HashTable

M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA
1	1	1	1	1	26	306	284	216	103	51	441	412	297	202
2	2	2	3	2	27	307	285	217	104	52	442	413	298	203
3	3	3	5	3	28	308	286	218	105	53	443	414	299	204
4	4	4	6	4	29	309	287	219	106	54	444	415	300	205
5	42	26	28	8	30	310	288	220	107	55	445	416	301	206
6	70	48	50	12	31	311	289	221	108	56	446	417	302	207
7	98	70	72	16	32	312	290	222	109	57	447	418	303	208
8	126	92	94	20	33	313	291	223	110	58	448	419	304	209
9	154	114	116	29	34	314	292	224	111	59	449	420	305	210
10	182	136	128	38	35	315	293	225	112	60	473	442	317	218
11	210	158	140	47	36	316	294	226	113	61	497	464	335	226
12	238	180	152	56	37	342	316	238	134	62	519	485	353	234
13	240	211	167	65	38	368	338	239	146	63	541	497	361	253
14	266	232	168	73	39	396	359	260	158	64	563	518	365	257
15	281	253	187	82	40	420	380	281	170	65	585	539	369	261
16	296	274	206	93	41	421	381	282	171	66	597	551	387	272
17	297	275	207	94	42	422	382	283	172	67	609	563	405	283
18	298	276	208	95	43	428	384	288	174	68	610	564	406	284
19	299	277	209	96	44	429	385	289	175	69	611	565	407	285
20	300	278	210	97	45	430	386	290	176	70	612	566	408	286
21	301	279	211	98	46	431	387	291	177	71	613	567	409	287
22	302	280	212	99	47	437	408	293	198	72	614	568	410	288
23	303	281	213	100	48	438	409	294	199	73	615	569	411	289
24	304	282	214	101	49	439	410	295	200					
25	305	283	215	102	50	440	411	296	201					

TABLE D.15: Iterations Used while Test Case Generation for Lang

M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA
1	1	2	2	1	15	34	40	37	36	29	68	67	70	69
2	2	3	6	2	16	35	41	38	37	30	77	77	80	70
3	3	5	7	3	17	36	42	39	38	31	85	87	90	71
4	13	15	17	13	18	37	43	40	40	32	95	95	100	72
5	14	16	18	14	19	47	44	50	50	33	96	96	103	73
6	15	17	19	15	20	55	54	55	60	34	98	97	106	74
7	16	18	20	16	21	56	55	56	61	35	100	98	109	75
8	17	20	21	17	22	57	56	57	62	36	101	99	112	76
9	18	21	22	18	23	58	57	59	63	37	102	100	113	77
10	29	35	32	28	24	59	58	61	64	38	103	101	115	78
11	30	36	33	29	25	60	59	62	65	39	104	102	116	79
12	31	37	34	31	26	61	60	64	66	40	105	103	117	80
13	32	38	35	33	27	62	62	66	67	41	106	104	121	85
14	33	39	36	34	28	64	64	68	68	42	107	105	122	90

TABLE D.16: Iterations Used while Test Case Generation for Logging

M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA
1	10	12	10	8	25	179	147	132	110	49	262	237	242	195
2	22	24	25	16	26	191	157	144	115	50	263	238	243	196
3	32	36	36	22	27	203	168	156	125	51	275	239	244	197
4	44	48	50	32	28	204	169	166	130	52	287	240	245	198
5	56	61	61	40	29	205	171	176	136	53	288	241	246	199
6	66	71	75	46	30	206	173	186	140	54	289	242	248	200
7	77	81	76	56	31	207	174	187	141	55	290	243	250	202
8	88	85	78	68	32	208	175	188	142	56	291	244	253	204
9	99	95	80	75	33	209	176	189	144	57	292	245	256	207
10	110	110	90	80	34	210	177	190	146	58	293	246	259	210
11	125	120	95	90	35	211	178	192	147	59	294	247	260	211
12	135	125	100	91	36	212	192	202	157	60	295	248	261	213
13	136	126	102	92	37	225	202	212	167	61	307	260	275	214
14	137	127	112	93	38	240	212	222	175	62	319	272	290	222
15	138	128	113	94	39	250	224	232	185	63	331	284	305	232
16	150	129	114	95	40	251	225	233	186	64	343	296	320	241
17	160	130	115	96	41	253	227	234	187	65	353	310	335	250
18	161	131	116	97	42	255	230	235	188	66	362	322	345	262
19	162	132	117	98	43	256	231	236	189	67	371	334	355	271
20	163	133	118	99	44	257	232	237	190	68	381	344	366	272
21	164	134	119	100	45	258	233	238	191	69	395	354	376	273
22	165	135	120	102	46	259	234	239	192	70	405	366	388	274
23	166	136	121	104	47	260	235	240	193	71	416	367	389	275
24	167	137	122	105	48	261	236	241	194	72	430	370	390	276

TABLE D.17: Iterations Used while Test Case Generation for Math

M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA
1	1	2	1	2	24	35	91	86	30	47	68	163	145	68
2	2	3	2	4	25	38	92	88	31	48	69	164	146	72
3	3	13	3	8	26	40	93	90	32	49	70	165	147	73
4	4	23	13	9	27	42	94	92	33	50	72	168	148	74
5	5	33	23	10	28	45	96	93	35	51	74	170	149	75
6	7	43	32	11	29	46	98	94	36	52	78	173	150	76
7	9	44	33	12	30	47	101	95	38	53	79	182	160	77
8	11	45	34	13	31	49	105	96	40	54	82	192	170	78
9	14	46	35	14	32	52	106	97	42	55	85	194	171	79
10	15	47	36	16	33	54	116	107	52	56	88	195	172	80
11	16	48	37	17	34	55	126	117	53	57	90	196	173	81
12	17	49	38	18	35	56	135	118	54	58	92	197	174	82
13	18	54	39	19	36	57	144	122	55	59	94	198	175	83
14	20	56	42	20	37	58	153	124	56	60	96	199	176	84
15	21	58	44	21	38	59	154	128	57	61	97	200	177	85
16	22	68	54	22	39	60	155	130	58	62	98	201	178	86
17	23	79	65	23	40	61	156	136	59	63	99	202	179	87
18	25	85	76	24	41	62	157	137	60	64	100	203	180	90
19	26	86	77	25	42	63	158	140	61	65	101	204	181	92
20	28	87	80	26	43	64	159	141	62	66	102	207	182	96
21	31	88	82	27	44	65	160	142	63	67	103	210	183	97
22	32	89	84	28	45	66	161	143	65	68	104	211	184	99
23	33	90	85	29	46	67	162	144	66					

TABLE D.18: Iterations Used while Test Case Generation for Stack

M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA
1	1	1	1	1	25	335	534	196	75	49	564	614	254	126
2	2	2	2	2	26	336	535	197	76	50	565	615	255	127
3	3	3	3	3	27	337	536	198	77	51	617	618	262	134
4	4	4	4	4	28	338	537	199	78	52	630	652	270	135
5	39	44	14	8	29	339	538	200	79	53	640	655	282	143
6	41	84	24	11	30	340	539	201	80	54	712	656	287	147
7	51	124	25	15	31	341	540	202	81	55	713	657	288	148
8	86	164	63	19	32	342	541	203	82	56	714	658	289	150
9	96	182	67	22	33	343	542	204	83	57	715	659	290	151
10	106	223	75	26	34	355	546	206	84	58	716	660	291	152
11	137	253	84	29	35	356	547	207	85	59	768	714	349	156
12	168	273	90	37	36	408	555	215	87	60	778	715	352	166
13	208	313	107	41	37	450	556	221	94	61	821	767	357	173
14	248	353	113	47	38	502	593	236	98	62	873	770	367	177
15	288	393	118	50	39	554	597	243	109	63	925	807	374	185
16	298	394	147	58	40	555	598	244	110	64	937	857	381	189
17	299	395	148	59	41	556	599	245	111	65	947	864	390	192
18	300	396	149	60	42	557	600	246	112	66	989	872	398	196
19	302	398	150	61	43	558	601	247	113	67	990	873	399	197
20	303	399	151	62	44	559	605	248	115	68	991	874	400	198
21	313	450	152	64	45	560	606	249	116	69	992	875	401	199
22	323	464	181	68	46	561	607	250	117	70	993	876	402	200
23	333	532	194	73	47	562	608	252	121	71	1045	880	409	201
24	334	533	195	74	48	563	609	253	122	72	1052	942	450	208

TABLE D.19: Iterations Used while Test Case Generation for TempConverter

M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA
1	1	1	1	1	29	29	29	29	29	57	57	57	57	57
2	2	2	2	2	30	30	30	30	30	58	58	58	58	58
3	3	3	3	3	31	31	31	31	31	59	59	59	59	59
4	4	4	4	4	32	32	32	32	32	60	60	60	60	60
5	5	5	5	5	33	33	33	33	33	61	61	61	61	61
6	6	6	6	6	34	34	34	34	34	62	62	62	62	62
7	7	7	7	7	35	35	35	35	35	63	63	63	63	63
8	8	8	8	8	36	36	36	36	36	64	64	64	64	64
9	9	9	9	9	37	37	37	37	37	65	65	65	65	65
10	10	10	10	10	38	38	38	38	38	66	66	66	66	66
11	11	11	11	11	39	39	39	39	39	67	67	67	67	67
12	12	12	12	12	40	40	40	40	40	68	68	68	68	68
13	13	13	13	13	41	41	41	41	41	69	69	69	69	69
14	14	14	14	14	42	42	42	42	42	70	70	70	70	70
15	15	15	15	15	43	43	43	43	43	71	71	71	71	71
16	16	16	16	16	44	44	44	44	44	72	73	72	72	72
17	17	17	17	17	45	45	45	45	45	73	75	74	73	73
18	18	18	18	18	46	46	46	46	46	74	77	77	74	74
19	19	19	19	19	47	47	47	47	47	75	78	79	75	75
20	20	20	20	20	48	48	48	48	48	76	79	81	76	76
21	21	21	21	21	49	49	49	49	49	77	81	83	77	77
22	22	22	22	22	50	50	50	50	50	78	83	84	78	78
23	23	23	23	23	51	51	51	51	51	79	85	88	79	79
24	24	24	24	24	52	52	52	52	52	80	87	89	80	80
25	25	25	25	25	53	53	53	53	53	81	88	90	82	81
26	26	26	26	26	54	54	54	54	54	82	90	91	84	82
27	27	27	27	27	55	55	55	55	55	83	92	93	87	83
28	28	28	28	28	56	56	56	56	56					

TABLE D.20: Iterations Used while Test Case Generation for Text

M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA
1	1	1	1	1	26	90	85	54	42	51	220	135	111	78
2	2	2	2	2	27	92	88	55	43	52	221	140	115	79
3	3	4	3	3	28	96	89	60	44	53	222	142	120	85
4	5	9	4	5	29	99	90	62	45	54	223	144	122	86
5	9	10	5	6	30	100	92	64	46	55	224	145	124	88
6	15	15	6	7	31	120	95	65	47	56	225	149	126	90
7	18	17	7	8	32	125	105	82	48	57	226	150	129	94
8	22	22	8	9	33	135	108	85	50	58	227	155	130	96
9	25	25	15	11	34	136	109	86	53	59	228	158	133	97
10	28	28	20	15	35	137	110	87	55	60	229	160	134	98
11	35	29	21	16	36	140	112	88	56	61	230	162	135	99
12	36	30	22	17	37	155	113	89	57	62	231	163	136	100
13	39	34	23	20	38	165	114	90	58	63	232	164	137	101
14	44	35	25	22	39	178	115	91	59	64	233	165	138	102
15	45	36	30	25	40	180	116	92	61	65	234	175	139	103
16	46	40	32	26	41	181	120	93	63	66	235	180	140	104
17	47	42	36	28	42	182	122	94	65	67	236	182	141	105
18	55	48	40	30	43	183	123	95	66	68	237	184	142	106
19	60	55	45	35	44	184	124	96	67	69	238	185	143	107
20	75	65	46	36	45	185	125	97	68	70	239	188	144	108
21	76	68	47	37	46	190	126	98	69	71	240	189	145	109
22	79	69	50	38	47	200	129	99	70	72	244	190	148	110
23	84	70	51	39	48	205	130	100	72	73	245	191	150	111
24	85	75	52	40	49	208	131	105	75	74	246	192	155	112
25	86	79	53	41	50	212	134	110	77					

TABLE D.21: Iterations Used while Test Case Generation for Triangle

M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA	M#	RT	SGA	PGA	IGA
1	1	1	1	1	25	26	50	44	26	49	389	381	401	324
2	2	2	2	2	26	27	51	45	27	50	390	382	402	325
3	3	3	3	3	27	28	52	46	28	51	391	383	403	326
4	4	4	4	4	28	29	53	47	29	52	392	384	404	327
5	5	5	6	5	29	30	54	48	30	53	393	385	405	328
6	6	6	8	6	30	31	55	49	31	54	394	386	406	329
7	7	7	20	8	31	32	56	50	32	55	395	387	407	330
8	8	8	21	9	32	33	57	51	33	56	396	388	408	331
9	9	9	22	10	33	34	58	52	34	57	397	389	409	332
10	10	10	23	11	34	35	59	53	35	58	398	390	410	333
11	11	11	24	12	35	53	60	85	37	59	410	391	411	334
12	12	12	25	13	36	71	61	117	39	60	411	392	412	335
13	14	38	32	14	37	132	122	178	40	61	412	393	414	336
14	15	39	33	15	38	177	183	239	89	62	413	394	415	337
15	16	40	34	16	39	189	184	244	92	63	414	395	416	338
16	17	41	35	17	40	226	236	276	144	64	415	396	417	339
17	18	42	36	18	41	267	288	308	177	65	416	428	418	346
18	19	43	37	19	42	281	330	350	236	66	417	429	419	347
19	20	44	38	20	43	326	331	392	273	67	419	443	452	348
20	21	45	39	21	44	344	343	396	305	68	424	445	453	362
21	22	46	40	22	45	359	347	397	312	69	426	460	462	363
22	23	47	41	23	46	371	367	398	313	70	427	463	484	370
23	24	48	42	24	47	387	379	399	322	71	431	475	517	371
24	25	49	43	25	48	388	380	400	323	72	432	476	519	373