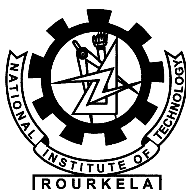
Computing MC/DC Criterion for Object-Oriented Systems

Arpita Dutta



Department of Computer Science and Engineering
National Institute of Technology Rourkela

Computing MC/DC Criterion for Object-Oriented Systems

Dissertation submitted in partial fulfillment

of the requirements of the degree of

Master of Technology

in

Computer Science and Engineering
(*Specialization*: *Computer Science*)

by

Arpita Dutta

(Roll Number: 215CS1067)

based on research carried out

under the supervision of

Prof. Durga Prasad Mohapatra

May, 2017

Department of Computer Science and Engineering
National Institute of Technology Rourkela

Department of Computer Science and Engineering
National Institute of Technology Rourkela

Prof. Durga Prasad Mohapatra
Associate Professor

May 23, 2017

Supervisor's Certificate

This is to certify that the work presented in the dissertation entitled *Computing MC/DC Criterion for Object-Oriented Systems* submitted by *Arpita Dutta*, Roll Number 215CS1067, is a record of original research carried out by her under my supervision and guidance in partial fulfillment of the requirements of the degree of *Master of Technology* in *Computer Science and Engineering*. Neither this dissertation nor any part of it has been submitted earlier for any degree or diploma to any institute or university in India or abroad.

Durga Prasad Mohapatra

Dedication

Dedicated to..........

My Loving Parents

Signature

Declaration of Originality

I, *Arpita Dutta*, Roll Number *215CS1067* hereby declare that this dissertation entitled *Computing MC/DC Criterion for Object-Oriented Systems* presents my original work carried out as a postgraduate student of NIT Rourkela and, to the best of my knowledge, contains no material previously published or written by another person, nor any material presented by me for the award of any degree or diploma of NIT Rourkela or any other institution. Any contribution made to this research by others, with whom I have worked at NIT Rourkela or elsewhere, is explicitly acknowledged in the dissertation. Works of other authors cited in this dissertation have been duly acknowledged under the sections "Reference" or "Bibliography". I have also submitted my original research records to the scrutiny committee for evaluation of my dissertation.

I am fully aware that in case of any non-compliance detected in future, the Senate of NIT Rourkela may withdraw the degree awarded to me on the basis of the present dissertation.

May 23, 2017
NIT Rourkela

Arpita Dutta

Acknowledgment

I owe deep gratitude to the ones who have contributed greatly in completion of this thesis.

Foremost, I would like to express my sincere gratitude to my supervisor, Dr. Durga Prasad Mohapatra for providing me with a platform to work in the field of Software Testing. He has always supported and guided on challenging areas of Modified Condition/ Decision Coverage and Concolic Testing. His profound insights and attention to details have been true inspirations to my research.

I am very much thankful to my Ph.D. senior Mr. Sanghartana Godboley. He has the one who suggested me to work in this area. He has always supported and guided me to the correct path of research. My research work and thesis is really impossible without his contribution and support.

I am very much indebted to Prof. Pabitra Mohan Khilar, Prof. Bibhudatta Sahoo, Prof. Ashok Kumar Turuk, and Prof. Ruchira Naskar for their encouragement and insightful comments at different stages of the thesis that were indeed thought provoking.

I express my gratitude to Prof. Rajib Mall of IIT Kharagpur for providing the necessary inputs and guidance at different stages of my work.

Most importantly, none of this would have been possible without the love of Mr. Arup Dutta (Baba), Mrs. Sangeeta Dutta (Maa), and Amrita Dutta (Sister). My family to whom this dissertation is dedicated to, has been a constant source of love, concern, support and strength all these years. I would like to express my heartfelt gratitude to them.

I also like to thank Dr. Subhrakanta Panda, and Dr. Jagannath Singh for their unique ideas and help whenever required.

I would like to thank all my friends and lab-mates (Bhagyashree Besra, Satya Manikyam Perabhatula,Anshu Katiyar, Jitendra Kumar, P. Shruthi, Srijan Das, Saurav Sharma and Mohammad Ashraf Gardizy) for their encouragement and understanding. Their help can never be penned with words.

May 23, 2017 — *Arpita Dutta*

NIT Rourkela — Roll Number: 215cs1067

Abstract

Modified Condition / Decision Coverage (MC/DC) is the second strongest criterion in coverage based white-box testing. According to RTCA DO-178B and DO-178C standards, it is mandatory for the safety critical systems to satisfy MC/DC criterion in order to ensure adequate testing. This work presents two different methodologies to calculate MC/DC% of a system. First, we compute MC/DC% of a software only after the completion of coding phase. There are so many techniques present to generate the test cases for a system. But, those test cases are not powerful enough to cover all the possible conditions present in the program. So, we propose a hybrid technique for MC/DC test data generation. We combine feedback-directed test case generation with concolic testing to form Java-Hybrid Concolic Testing (Java-HCT). Java-HCT generates more number of test cases since it combines the features of both. Hence, through Java-HCT we achieve high MC/DC. Combination of two approaches handles/ overcomes different tradeoffs of completeness and scalability. We develop Java-HCT using RANDOOP, jCUTE (Java Concolic Unit Testing Engine), and COPECA (COverage PErcentage CAlculator). Combination of RANDOOP and jCUTE creates more number of test cases. COPECA is used to measure MC/DC% taking the generated test cases as input. Our experiment with forty Java programs shows that Java-HCT produces better MC/DC% than individual testing techniques (feedback-directed random testing and concolic testing). We have improved MC/DC by $\times 1.62$ and by $\times 1.26$ in comparison to feedback-directed random testing and concolic testing, respectively.

In our second work, we compute MC/DC% of the given system, using model based approach. We have proposed a novel technique for MC/DC computation during design phase, using UML Sequence diagram. Sequence diagram presents the interactions among a set of collaborating objects. The sequences of synchronized and asynchronized messages in the sequence diagrams are used to define the code coverage goals. First, we design an UML Sequence Diagram and generate an XMI code from it. Next, JAXB converts the XMI code into Java code. After that, we supply Java code to jCUTE to generate concolic test cases. These test cases and the Java code are supplied to our tool COPECA to measure MC/DC%. We experimented with five case studies and worked on twenty seven sequence diagrams and on an average, we achieved 55.29% MC/DC.

Keywords: ***Feedback Directed Random Testing***; ***Concolic Testing***; ***UML Sequence Diagram***; ***MC/DC.***

Contents

Supervisor's Certificate **ii**

Dedication **iii**

Declaration of Originality **iv**

Acknowledgment **v**

Abstract **vi**

List of Figures **x**

List of Tables **xii**

List of Algorithms **xiii**

1 Introduction **1**

1.1 Motivation 4

1.2 Objectives 4

1.3 Thesis Organization 5

2 Basic Concepts **6**

2.1 Some Relevant Definitions 6

2.2 UML Diagrams 11

2.3 Summary 12

3 Literature Survey **14**

3.1 Test Data Generation 14

3.1.1 Random Testing 14

3.1.2 Symbolic Testing 15

3.1.3 Concolic Testing 15

3.1.4 Hybrid Concolic Testing 19

3.1.5 Other Related Works 20

3.2 MC/DC (Modified Condition/ Decision Coverage) Testing 21

3.3 Testing and coverage analysis using UML diagrams 22
3.4 Summary 23

4 Java-HCT: An approach to increase MC/DC using Hybrid Concolic Testing 24
4.1 Overview of proposed framework 24
4.2 Description in detail 26
4.2.1 Syntax Converter 26
4.2.2 RANDOOP 27
4.2.3 jCUTE 27
4.2.4 TCs Extractor 27
4.2.5 Test Cases Combiner 28
4.2.6 COPECA 28
4.2.7 TCs Minimizer 28
4.3 Algorithmic Description 29
4.4 Experimental Study 31
4.4.1 Experimental Setup 31
4.4.2 Assumptions 32
4.4.3 Implementation 32
4.4.4 Result Analysis 35
4.5 Threats to validity 45
4.6 Comparison with related works 45
4.7 Summary 50

5 Measuring MC/DC at Design Phase using UML Sequence Diagram 52
5.1 Overview of proposed framework 52
5.2 Description in detail 53
5.2.1 ArgoUML 53
5.2.2 JAXB 54
5.2.3 jCUTE 55
5.2.4 COPECA 55
5.3 Algorithmic Description 56
5.4 Experimental Study 56
5.4.1 Experimental Setup 57
5.4.2 Assumptions 57
5.4.3 Implementation 57
5.4.4 Result 58
5.5 Comparison with related work 64
5.6 Threats to Validity 68
5.7 Summary 68

6 Conclusions and Future Work **69**
6.1 Contributions 69
6.1.1 Java-HCT 69
6.1.2 MAUSD 69
6.2 Future Work 70

Dissemination **71**

List of Figures

1.1 Software testing technique classification 2

2.1 Euclid's GCD computation program 7
2.2 An example "if" structure to show MC/DC testing 8
2.3 Sample program for Concolic testing 10
2.4 UML diagrams showing the different views of a system 12
2.5 UML Sequence Diagram for Book Renewal Scenario 13

4.1 Schematic representation of Java-HCT 25
4.2 Benefit of Hybrid concolic testing 25
4.3 Original Java program 34
4.4 Graphical User Interface of Syntax_Converter 35
4.5 Java program in jCUTE executable format 36
4.6 Java program in RANDOOP executable format 37
4.7 Test data generation from RANDOOP framework 38
4.8 Successful execution of total test cases 38
4.9 RandoopTest.java program contains information about all the generated test case files 39
4.10 Test Cases present in a single test data file 39
4.11 Test data generation from jCUTE tool 40
4.12 Different parameter computation using jCUTE 40
4.13 Test Cases generated by jCUTE 41
4.14 Graphical User Interface of Java-HCT 42
4.15 Graphical User Interface of COPECA (Coverage Percentage Calculator) . . 43
4.16 Graphical User Interface of Minimizer 43
4.17 Total number of Test Cases generated 50
4.18 Computed MC/DC percentages 51
4.19 Increase in MC/DC percentages 51

5.1 Schematic representation of MAUSD 53
5.2 Fundamental working of JAXB 55
5.3 Sequence diagram for Job searching 58

5.4 XML code generated for the Sequence diagram shown in Figure 5.3 59
5.5 Java code generated from JAXB for the XML code shown in Figure 5.4 . . 60
5.6 Compilation on jCUTE 60
5.7 One complete execution on jCUTE 61
5.8 Results obtained by using jCUTE 61
5.9 Test Cases generated by jCUTE 62
5.10 MC/DC analysis using COPECA 62
5.11 Number of generated test cases vs. the number of conditions present in the Sequence Diagrams. 63
5.12 Number of independently affected conditions vs. number of simple conditions. 63
5.13 Branch Coverage percentage vs. Modified Condition/ Decision Coverage percentage 64

List of Tables

2.1 Extended Truth Table for MC/DC analysis 8
2.2 Associativity and precedence of logic gates 9

3.1 Summary of concolic testers with their properties. 18
3.2 Summary of different work on concolic testing. 19
3.3 Characteristics of different approaches on concolic testing. 19

4.1 Characteristics of different target programs 46
4.2 Statistics of results on execution of RANDOOP 47
4.3 Statistics of results on execution of jCUTE 48
4.4 Results on execution of COPECA 49

5.1 Characteristics of case studies 65
5.2 Results analysis of jCUTE 66
5.3 Result analysis of COPECA 67

List of Algorithms

1 Java-HCT 29
2 COPECA 30
3 MAUSD 56

Chapter 1

Introduction

Software Development Life Cycle (SDLC) has one important and expensive activity called software testing. It deals with the quality and reliability of the product developed. It includes driving test cases along with a developed program and computing the response. It targets to detects all bugs present in a software with the help of test cases vector. Test case design is the most important phase in software testing life cycle. The software testing goals are mainly classified into three categories:

1. Immediate goals or Short-term: It consists of Bug prevention, and Bug discovery.
2. Long-term Goals: It consists of Reliability, Customer satisfaction, Risk management, and Quality.
3. Post-implementation Goals: It consists of Reduced maintenance cost, and Improved testing process.

Software testing can be done in two ways- Manually and Automated. For practical usage, manual testing is not advisable. Because manual testing leaves an ample scope for un-catched errors. So, we have moved towards automated testing. Now-a-days a large number of automated software testing tools are available such as Concolic testers (CUTE, jCUTE, SCORE, CREST), random testers (RANDOOP) etc. Concolic testing is the combination of Concrete and Symbolic execution. It is a systematic technique that performs symbolic execution but uses randomly-generated test inputs to initialize the search and to allow the tool to execute programs when symbolic execution fails.

Software testing is also classified as white box testing and black box testing. In white box testing, we have the knowledge of internal structure of the software whereas in black box testing, we know only the functionality of the software. We don't have the knowledge of internal structure (code). We are focusing towards white box testing. White box testing a can be further classified as Fault based testing and Coverage based testing. Fault based testing targets to detect certain type of faults present in the program. Mutation testing is an example of a fault-based test strategy. Coverage based testing targets to cover certain specific elements of a program. Following are different coverage based testing techniques.

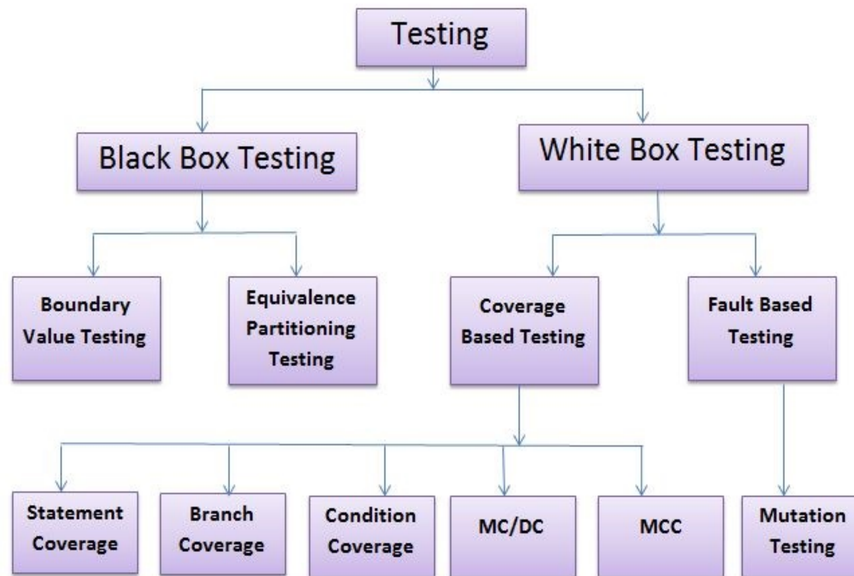


Figure 1.1: Software testing technique classification

- **Statement Coverage:** It aims to design test cases in order to execute all the statements present in the program.
- **Branch Coverage or Decision Coverage:** It aims to design test cases which invokes each decision present in the program for true and false result each, at-least for once.
- **Condition Coverage:** It aims to design test cases which invokes each condition of a decision present in the program for true and false result each, at-least for once.
- **Modified Condition/ Decision Coverage:** It aims to design test cases which is able to show the independent affect of each condition present in a predicate.
- **Multiple Condition Coverage:** Multiple Condition Coverage: This is the strongest code coverage criterion, which finds all possible combinations of condition outputs present in a predicate in a program. It invokes all entry and exit points at least once.
- **Path Coverage**: It tries to design test cases which covers all possible linear independent paths present in the program.

Figure 1.1 shows the classification of software testing techniques. In coverage based testing techniques, Multiple Condition Coverage (MCC) is the strongest one [1]. it subsumes all other coverage based testing criterion. But for our research we have chosen Modified Condition/Decision Coverage (MC/DC) criterion, which is the second strongest coverage criterion. The reason behind the selection of second strongest criterion is completely based upon the test case generation requirement. In order to test an n-condition predicate, in MCC we require 2^n number of test cases whereas in MC/DC it requires minimum (n+1)

and maximum (2*n) test cases. The 2^n number of test cases are redundant and it creates an combinatorial explosion problem of test data generation. So, in the practical usage and software application testing it is not advisable to adhere with Multiple Condition Coverage. In MC/DC, the required (n+1) number of test cases are unique, non-redundant and capable of invoking each and every atomic condition of a predicate.

According to guidelines provided by RTCA[1]/DO[2]-178B [60] and DO-178C [25] standards, it is mandatory to achieve MC/DC for Level A safety critical software applications. Effective test data generation for MC/DC coverage is a critical issue. The short circuit evaluation done by the compiler on logical operators makes it difficult to reach each and every atomic condition present in a predicate. Compiler simplifies a predicate and generates the equivalent code using only the basic "if" structure which is free from AND and OR operators. Further it uses code optimization techniques to simply the code. In this phase, based upon the short-circuit evaluation, it eliminates many simple condition which are very much essential for the MC/DC evaluation. There are many test data generation techniques available. But alone these techniques are not sufficient to achieve high MC/DC coverage because they are not capable of generating all possible useful test data for Modified Condition/Decision Coverage. To overcome the problem of less coverage attainment, we have proposed a hybridized method for test data generation. This technique combines the test cases generated by feedback directed random testing and concolic testing and tries to discover more number of independently affecting conditions. We have develop a tool called Java-HCT (Java-Hybrid Concolic Tester) using RANDOOP [3], jCUTE [4] and COPECA.

Test cases are usually designed to satisfy the requirements which are actually coded and presented through a software or program. It makes test case generation process very complex for cluster levels. Further this approach may be inefficient at component-based software development, where testers may not have actual source code. Hence, it is useful to generate test cases at the software design level instead from the source code. Testing at design phase is very advantageous for SDLC. Using this testing at design phase deals with the compliance of the implementation with the design documentation, which is missing is source code based testing. Also, in this case the generated test data is independent of any specific implementation of the design. So, we have proposed an automated technique of test data generation from UML Sequence Digram. The process of generating test cases from design documents is known as Grey box testing because it is combination of both black box and white box testing strategy. We have developed and implemented a tool called MAUSD (MC/DC Analyser for UML Sequence Diagram) for measuring MC/DC percentage using

[1] Radio Technical Commission for Aeronautics.
[2] Document
[3] https://github.com/randoop/randoop-eclipse-plugin
[4] http://osl.cs.illinois.edu/software/jcute/

ArgoUML [5], JAXB [6], jCUTE [7], and COPECA.

1.1 Motivation

This section presents the motivation behind the developing the techniques basically related with the Modified Condition/Decision Coverage.

- Limited work is done in the area of automated testing methods that support MC/DC using concolic testing.
- Automated tool for MC/DC test case generation
  - Improves software quality
  - Performs exhaustive testing of a software.
  - Reduces software testing time.
- RTCA/ DO-178B and DO-178C standardization: Safety critical systems strictly require the satisfaction of MC/DC for Level A certification of a software systems. So, it is desirable to develop an automated tool to compute the MC/DC percentage of given system.

While computing coverage, we have been motivated towards model based testing. In model based testing, we can start testing from design phase onwards. We don't have to wait for the completion of coding phase. UML diagrams represent various perspectives of the project under development. UML sequence diagram contains complex interactions among sets of collaborating objects from different classes. It shows the behavioral aspects of the system. so, we want to develop an approach to compute the MC/DC percentage of a project at design phase using UML sequence diagram.

1.2 Objectives

We set the following objectives for our research work based on the motivations outlined in the previous section.

- To develop a hybridized technique (combination of Feedback Directed Random Testing and Concolic Testing) for MC/DC test data generation in order to achieve high MC/DC coverage for Java programs.
- To compute MC/DC% of object-oriented systems using UML Sequence Diagram.

[5] http://argouml.tigris.org/
[6] https://jaxb.java.net/
[7] http://osl.cs.illinois.edu/software/jcute/

1.3 Thesis Organization

The rest of the thesis is organized as follows:

Chapter 2 presents the background knowledge required to understand the whole thesis. It contains the definition of *Condition*, *Predicate*, *Branch Coverage*, *Modified Condition/ Decision Coverage Primary Gates*, *Concolic testing*, *Feedback-Directed Random TestingUML diagrams* and specifically *UML sequence diagram* etc. We explain all the concepts with help of examples.

Chapter 3 provides an overview of the related work done in field of various test data generation strategies, MC/DC testing criterion and test and coverage analysis of UML diagrams. We have mainly focused on random testing, symbolic testing, concolic testing and hybrid concolic testing strategies of test data generation. In this chapter, we also discussed the work on testing of object-oriented software using UML sequence diagrams.

Chapter 4 presents a hybrid concolic testing (HCT) technique to improve the Modified Condition/ Decision Coverage for input Java programs. We present the schematic representation of proposed approach followed by detail description and algorithmic description. Also, we explain the proposed technique with the help of an example Java program. Subsequently, we present the experimental results and threats to validity of proposed approach. We also present comparison with related work.

Chapter 5 presents a technique for Modified Condition/ Decision Coverage measurement using UML sequence diagram. We present the description of each module used in framework followed by the Schematic representation of proposed technique. We present the algorithm used and a example to discuss the complete flow of approach. Subsequently, we present the result analysis of proposed technique and threats to validity of our approach. We have also compared our proposed technique with existing one.

Chapter 6 concludes the thesis with a summary of our contributions. We also give a brief idea towards the possible future extension of our work.

Chapter 2

Basic Concepts

In this chapter we discuss some important background concepts which will help to understand the further chapters.

2.1 Some Relevant Definitions

Definition 2.1 ***Condition***: *Booleans expression without and logical operator such as* ***AND*** *(&&) and* ***OR*** *(||) operator. Conditions are also known as clauses.*

For example, (var1<var2) is a condition, where var1 and var2 are variables. Conditions consists of relational operators such as <, >, >=, =< etc. Conditions are known as clauses.

Definition 2.2 ***Predicate***: *It is a group of one or more conditions connected with logical operators such as* ***AND*** *(&&) and* ***OR*** *(||) operator.*

For example, (var1<var2) && (var3>=80) is a predicate, (var1<var2) and (var3>=80) are two conditions connected with the help of **AND** operator.

Definition 2.3 ***Branch Coverage***: *Branch coverage is a method of testing which aims to ensure that all possible outcomes of a decision point has to be executed at least once. With the help of above statement, it ensures that all reachable code is executed at least once [14].*

The possible outcome for a branch coverage statement is either *true* or *false*. The branch coverage criterion tries to make sure that none of the branch statement present in the program leads to an abnormal behavior of the application. The branch coverage percentage of a program is computed with help of formula given in Equation 2.1.

$$Branch\ Coverage\ Percentage = \frac{No.\ of\ decision\ outcomes\ tested}{Total\ number\ of\ decisions\ present} \tag{2.1}$$

Let us explain the working of branch coverage using an example program. Figure 2.1 shows a sample program to evaluate Greatest Common Divisor (GCD) between two numbers. To achieve 100% branch coverage, a suitable test suite is presented below:

```
1. int GreatestCommonDivisor(int var1x, int var2y){
2.    while(var1x!=var2y){
3.     if(var1x>var2y){
4.       var1x=var1x-var2y;
5.         }
6.     else{
7.      var2y=var2y-var1x;
8.     }
9.   }
10. return var1x;
11. }
```

Figure 2.1: Euclid's GCD computation program

Test Suite {(var1x=3, var2y=3), (var1x=3, var2y=2), (var1x=4, var2y=3), (var1x=3, var2y=4)}.

Definition 2.4 ***Modified Condition / Decision Coverage*:** *According to DO178C standard, the essential requirements of MC/DC are follows [6, 7]:*

- All statement in the program has been invoked at least once.
- All point of entry and exit in the program has been invoked at least once.
- All control statement (i.e., branch point) in the program has taken all possible outcomes (i.e., branches) at least once.
- All non-constant Boolean expression in the program has evaluated to both a true and a false result.
- All non-constant condition in a Boolean expression in the program has evaluated to both a true and a false result.
- All non-constant condition in a Boolean expression in the program has been shown to independently affect that expression's outcome. Two different approaches to confirm that the minimum tests are achieved are the unique-cause approach and the masking approach.
  - For unique-cause MC/DC, a condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions.
  - For masking MC/DC, a condition is shown to independently affect a decision's outcome by applying principles of Boolean logic to assure that no other condition influences the outcome (even though more than one condition in the decision may change the value).

```
1. if(A||B){
2.  // Do-Something
3.   }
```

Figure 2.2: An example "if" structure to show MC/DC testing

Table 2.1: Extended Truth Table for MC/DC analysis

TCs No.	M	N	result	M	N
1	True	True	True	3	2
2	True	False	False		1
3	False	True	False	1	
4	False	False	False		

The concept of independence of a clause is mathematically explained below:

Let $P = (c_1, c_2, ..., c_i, ...c_{n-1}, c_n)$ be a predicate consisting of n clauses and Bool_Res(P) be a user defined mathematical function returns the boolean decision of predicate P. Independence of clause c_i is denoted as $\frac{\phi P(c)}{c_i}$. Mathematically,

$$\frac{\phi P(c)}{c_i} = Bool_Res(c_1, c_2, .., c_i, ..c_{n-1}, c_n) \oplus Bool_Res(c_1, c_2, .., \neg c_i, ..c_{n-1}, c_n) \quad (2.2)$$

where, $\oplus$ is an exclusive-or operation.

If $\frac{\phi P(c)}{c_i} = 1$, then c_i is an independent clause, otherwise not.

Let's take the example "if" structure in Figure 2.2. The MC/DC test cases are generated using the steps given below:

- Prepare a truth table for the predicate.
- Now, develop an Extended Truth Table (ETT) so that it indicates an atomic condition as independently influenced atomic conditions.
- Please, show the pairing of test cases. Here, independence of M shows to take 1+3, and independence of N shows to take 1+2.
- At last, maintain the test cases as 1+2+3, viz i.e.(True,True)+(True,False)+(False,True)

MC/DC subsumes the criteria of decision, condition and condition/decision coverage (C/DC). If MC/DC coverage is 100%, then all these structural coverages will be 100%. Equation 2.3 shows the subsumption relationship of all the coverages. Leftmost coverage criterion is the strongest one.

$$MC/DC \Rightarrow CDC \Rightarrow CC \Rightarrow DC \quad (2.3)$$

Where, MC/DC stands for Modified condition/ decision coverage.
CDC stands for Condition Decision Coverage.
CC stands for Condition Coverage.
DC stands for Decision Coverage.

Definition 2.5 ***Primary logic gates/elements*****:** *There are three primary logic gates viz. AND (&&), OR (||) and NOT (!). Remaining all gates are derived by using these primary gates. These are used to join clauses present in the predicates. The associativity and precedence of these gates are shown in Table 2.2.*

Table 2.2: Associativity and precedence of logic gates

Logic Gate	Associativity	Precedence
$AND(\&\&)$	left to right	2
$OR(\|\|)$	left to right	3
$NOT(!)$	right to left	1

AND (&&) **Gate:** In a predicate, if all clauses are joined by using "&&", then MC/DC must satisfy the following criteria:

- There must be at least one test case that makes all clauses true in the same time. i.e. Bool_Res(P) = 1.
- Secondly, there must be at least n test cases (n is the number of clauses present in the predicate), that set each clause as false one by one by keeping all other clauses true. i.e. Bool_Res(P)=0.

OR (||) **Gate:** In a predicate, if all clauses are joined by using "||", then MC/DC must satisfy the following criteria:

- There must be at least one test case that makes all clauses false in the same time. i.e. Bool_Res(P)=0.
- Secondly, there must be at least n test cases (n is the number of clauses present in the predicate), that set each clause as true one by one by keeping all other clauses as false. i.e. Bool_Res(P)=1.

In conclusion, we can say that for conjunctive and disjunctive expressions we can easily find (n+1) test cases and using these two basic gates idea, we get (n+1) test cases required for any other predicate.

Definition 2.6 ***Concolic testing*****:** *Concolic testing explores all the execution paths and ensures that all the reachable paths are executed[3, 10]. Unit testing is of two types. 1)*

Concrete Execution, where the potential inputs are chosen randomly, so that the probability of reaching errors present in the program is astronomically less. 2) Symbolic Execution: It takes symbolic values for the variables present in the program and symbolically runs the program. It collects the symbolic path constraints with the help of theorem prover. It detects whether the branch will be taken or not. It is not scalable for large programs. Concolic testing is the combination of both concrete and symbolic execution for unit testing.

CONCrete + SymbOLIC= CONCOLIC testing uses concrete execution over a concrete input to guide symbolic execution[40]. In the first run, it takes the random value and covers a path, afterward with negating the conditions in the path covered and simplifying complex and unmanageable symbolic expressions with the help of constraint solvers like Z3 solver, lp-Solver, etc, it traces a new unexplored path. In this way, it reaches to all the possible paths present in the binary program computation tree. It achieves high scalability and branch coverage than symbolic or concrete execution.

Let us understand concolic testing with an example program shown in Figure 2.3.

```
1. struct node{
2.      int data;
3.      struct node *link;
4.       };
5. int doubly(int a)
6.      { return(2*a+1);}
7. int concolic_test(node *head,int val){
8.     if(val>0){
9.       if(head != NULL){
10.         if(doubly(val)==node->data){
11.           abort();
12.                  }
13.               }
14.             }
15.       return 0;
16. }
```

Figure 2.3: Sample program for Concolic testing

Initially the random test driver generates values (head=NULL, val=236). But, by using these values the probability of reaching to the abort statement is very less. But, as per the first time execution rule, concolic tester takes these random concrete values and set the concrete state as (head=NULL; val=236), symbolic state as $(head = head_0; val = val_0)$ with "NULL" constraints. When, the control reaches to the statement 8, it gets a constraint $(val > 0)$ and this constraint get resolved because (val=236) and then the control reaches to statement 9. The constraint at this point is not resolved because (head=NULL). Therefore the concolic tester will take such new values that can resolve both the constraints (val $>$ 0) && (head !=NULL). So, in the next run it takes concrete state as (val=236; head →data=634; head →link=NULL) and symbolic state as $(head = head_0; val = val_0$; head

$\rightarrow data = data_0$; head $\rightarrow link = n_0$). By, these values the control successfully reaches up to statement 9, but at the statement 10, it gets a new constraint ($(2 * val + 1)$=head →data) which is not solved by the taken values. So, concolic tester take such values that satisfies all the constraints that are $(val > 0)$ && $(head! = NULL)$ && (2*val+1=head →data). The concolic tester takes concrete value as (val=1; head →data=3; head →link=NULL) and symbolic state as $head = head_0; val = val_0$; head $\rightarrow data = data_0$; head $\rightarrow link = n_0$ by using these values the control reaches to the statement 11 and detects the error. This concolic testers are saving concrete and symbolic values to reaches all the possible paths of the program.

Definition 2.7 ***Feedback-Directed Random Testing***: *"It is a combination of random and systematic approach that results a test suite consisting of unit tests for the classes under test. Systematic approach deals with Feedback-Directed, i.e as soon as an input value is built, it is executed and checked against a set of contacts and filters. The result of the execution determines whether the input is redundant, illegal or useful for generation of more input [38]."*

We have an open source avalibale tool for Feedback-Directed Random Testing, it's called as Random Tester for object-Oriented Programs (RANDOOP)[1]. It is 100% automated testing tool and may not expect any input from the user, also scaled to realize the application with huge number of classes such as almost more than 100 classes

2.2 UML Diagrams

UML stands for Unified Modeling Language. It is basically used in the designing phase of object-oriented software systems. It models the software application in many different perspectives. There are nine different types of UML diagrams which present five different views of a system [4, 14].

The UML diagrams can capture the following five views of a system as shown in Figure 2.4.

- User's view
- Structural view
- Behavioral view
- Implementation view
- Environmental view

[1]https://github.com/randoop/randoop-eclipse-plugin

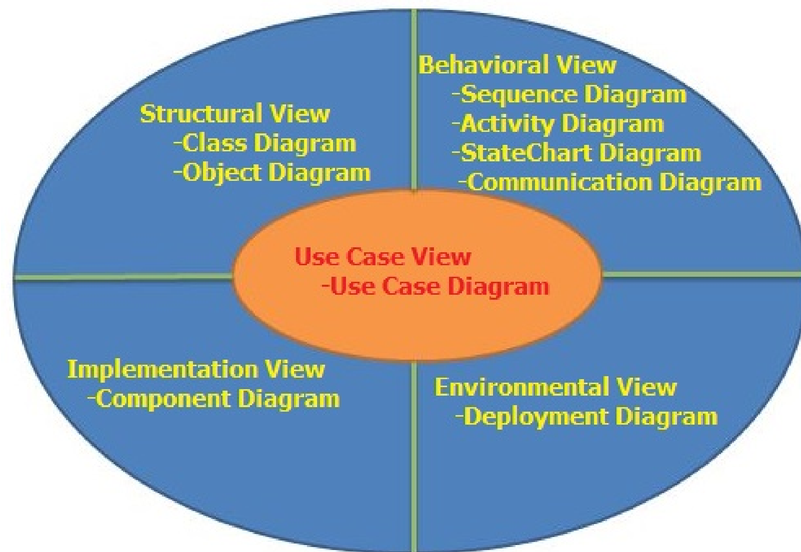


Figure 2.4: UML diagrams showing the different views of a system

UML Sequence diagram

UML Sequence digram represents the behavioral view of the system. It shows the interaction between different objects of a system in two dimensional chart format. The two dimensional chart is read from top to bottom. The objects involved in the scenario are present at the top of the chart as boxes attached to a vertical dashed line. Sequence digram also presents the timing sequence of the different activities involved for that particular scenario [11, 14]. An example sequence diagram for book renewal scenario is shown in Figure 2.5.

2.3 Summary

We have discussed all the basic definitions used in our proposed approach. We have explained Condition, Predicate, Branch Coverage, Modified Condition/ Decision Coverage Primary Gates, Concolic testing, Feedback-Directed Random Testing, UML diagrams and specifically UML sequence diagram etc. with help of example.

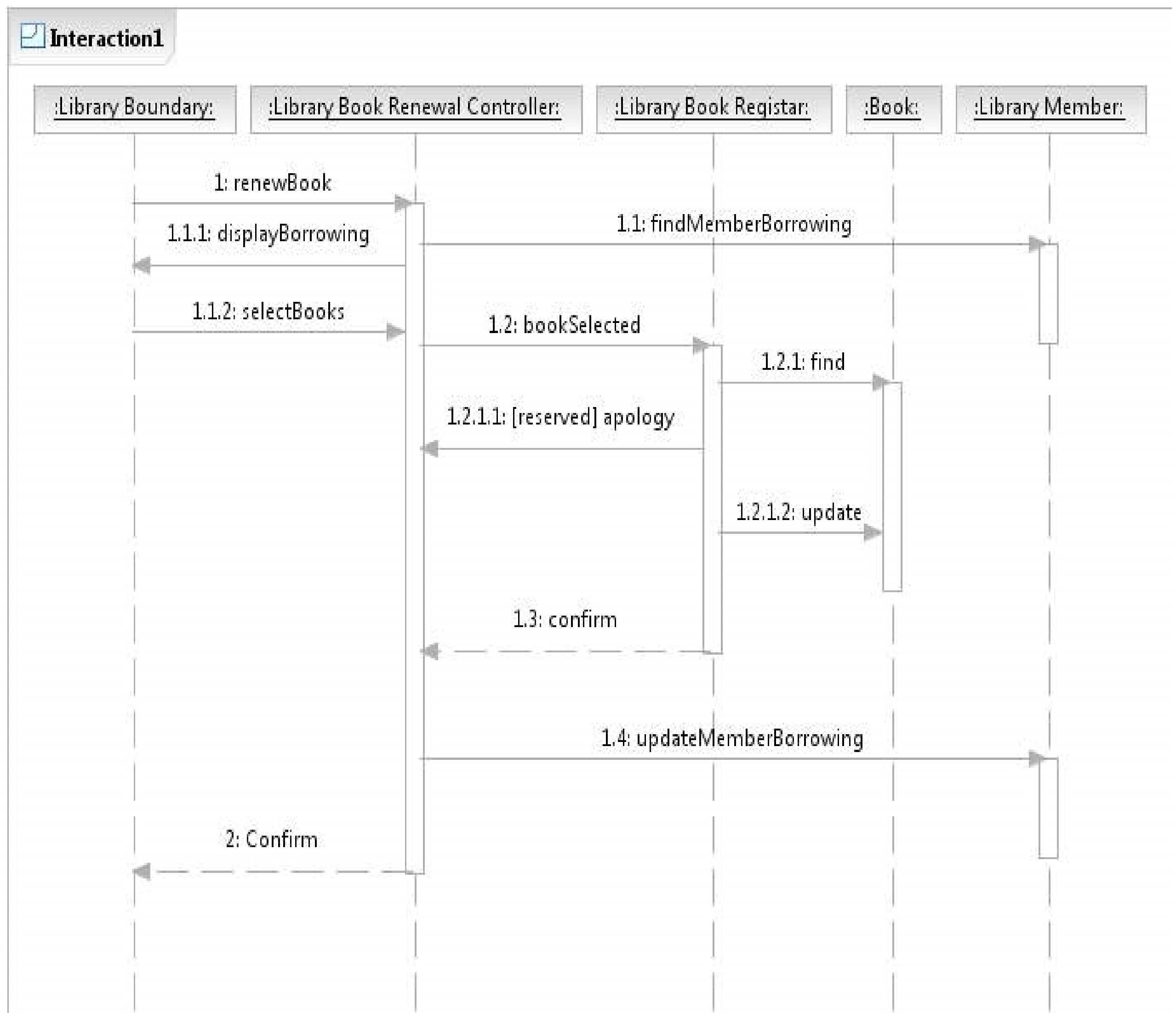


Figure 2.5: UML Sequence Diagram for Book Renewal Scenario

Chapter 3

Literature Survey

3.1 Test Data Generation

In this chapter, first we discuss some available related work carried out by different researchers in the area of random testing, symbolic testing, and concolic testing. Then, we discuss the available work on MC/DC testing. Finally, we describe some of the existing work related with testing of coverage analysis of UML diagrams.

3.1.1 Random Testing

Lei et al. [47] described a novel framework for randomized unit testing. They have proved the empirical importance of the randomly generated unit test data. They have minimized the test data failures, which increases significant benefits. Randomized test case generation techniques allows high coverage in less amount of time[8]. Sometimes, test cases are failed because of very long sequence of method calls. Lei et al. [47] has used the Zeller and Hildebrandt's test case minimization algorithm in order to reduce the long method call sequences. They had tested their proposed framework on lab-built data structures and open source data structures.

Bird et al. [61] proposed a random test generation technique for software systems. The nature of the test cases in such systems are designed to explore all possible branches of the execution tree. They have also predicated the total case generation time. The test cases are processing a self-checking property. The implementation of this technique is tested on various IBM programs such as sort/merge programs, /I language processors, and graphics support and Graphical Data Display Manager alphanumeric.

Quick Check is a random testing tool. It is developed for the Haskell programs. It is used for testing the properties of the program [24]. Haskell functions are used to describe the properties of the program. They are automatically tested with the help of random inputs. Random testing techniques are highly suitable for the function programs because, the properties of a program are stated in a very fine grain. When the function is developed from different tested components, th random testing provides good coverage of the definition

under test.

3.1.2 Symbolic Testing

In order to remove the limitations and inadequacy of concrete testing symbolic testing approaches are developed, In symbolic testing, we execute a program using symbolic variables instead of concrete one.

King [63] describes the symbolic execution of the programs. In symbolic execution, instead of supplying normal inputs (e.g. numbers) into the programs, symbolic variables are substituted. The flow of execution proceeds in the same way as it in concrete execution except the values that present in the form of symbolic formulas. The problem arise when the control flow reaches to a branch statement. They have also described a system called EFFIGY. EFFIGY provides a symbolic execution platform for program debugging and testing.

Clarke [62] describes a system which generates symbolic test cases for ANSI Fortran programs. For a given path, system creates a set of constraints on its symbolic execution by using the program's input values. If the constraint is linear, then linear programming techniques are used to generate the solutions. The solution consists of test data which will helps to drive the execution down to the given path. If the constraint is non-linear and inconsistent, then the given path is display as non-executable. In order to increase the detection rate of program errors artificial constraints are developed. Artificial constraints are used to simulate the error conditions and also try to solve the each set of augmented constraints. The system also provides the facility to represent the output variables in terms of the program input variables. The variables helps in the error detection and also in the automatic program documentation and assertion generation.

Visser et al. [39] shows how the model checking and symbolic execution is used to generate the test data in order to achieve the structural code coverage that manipulates a complex data structure. They have mainly focused branch coverage. They worked on the red-black tree of the Java TreeMap library with the help of JavaPathFinder as a Model Checker. They have introduced and compared three types test case generation techniques. The techniques are Black-Box model checking, Straight model checking and White-Box model checking. The main contribution of Visser et al. [39] is to show how efficient white-box test input generation can be done for code manipulating complex data, taking into account complex method preconditions.

3.1.3 Concolic Testing

Godfroid et al. [44] developed a tool and named it as DART (Directed Automated Random Testing) for automated software testing. It combines three main techniques: (i) automatic extraction of program interface with the external environment by using a static-code parsing

technique. (ii) Automated generation of a program test driver for the derived interface which will perform random testing in order to simulate the regular environment on which the program operate. (iii) Program analysis under the random input and automatic generation of new input values to explore the other possible paths present in program. The main strength of DART is that, it made the testing activity fully automated. Now there is zero requirement of writing any harness code or test driver. During execution, DART detect many standard errors. For example assertion violation, program crashes, and non-termination etc. DART is pioneer of concolic testing technique.

Sen at al. [45] proposed and developed a tool called CUTE (Concolic Unit Test Engine) for C program. This tool address the problem of automated unit testing with memory graphs as input. The approach used for memory graph is based on combination of Concolic and symbolic testing techniques. They have used an efficient constraint solver i.e. lp_Solver. lp_Solver has the following important properties which improves the strength of CUTE. It will do fast unsatisfiability check, common sub-constraint elimination, and also incremental checking. CUTE tries to cover all feasible paths present in the program in a similar way to systematic testing.

Sen et al. [40] developed another tool i.e jCUTE (*Java Concolic Unit Test Engine*) which is a concolic test generator for Java programs. It is an open source tool available on Internet [1]. It generates test cases for both simple and multi-threaded Java programs. It also supports the concurrent programs. Concolic testing combines the concrete and symbolic testing technique with using a powerful constraint solver. It discovers the deadlock and race conditions using schematic schedule explorations. jCUTE is using vectorized clock to generate large number of test cases and to support the concurrent programs. It creates execution tree for the program and tries to reach all the leaf nodes of the tree. jCUTE supports three different types of search strategies i.e. Random Search strategy, Depth First Search strategy and iii) Quick Search strategy. In the Depth First Search strategy we have to mention the maximum depth and in Quick Search strategy we have to mention the threshold value. The first value chosen jCUTE is a Random Number. Mostly the value is taken from one the largest number supported by the variable data type for jCUTE. It maintains log files and maintain traces for each run. The search optimality is based upon the path coverage and branch coverage. For our experimental study, we have used jCUTE as a concolic tester.

Kohkonen et al. [26] developed another concolic tester i.e. LCT [2] (Lime Concolic Tester) for sequential Java programs. Lime Concolic Tester instruments the byte-code of the Java program under test in order to enable the symbolic execution and then it collects the constraints generated on the input values which is further used to guide the tester to find the unexplored paths. LCT supports the distributed architecture. In the distributed environment, clients are generating the test input values for the program under test. On the

[1] http://osl.cs.illinois.edu/software/jcute/

[2] http://www.tcs.hut.fi/Software/lime/userguide.pdf

other side, Server node is is monitoring and collecting the generated test cases. LCT used bit vector SMT solver Boolector. Boolector helps to generate more precise integer values. It also allows to generate test data values for a given rang of integer data.

Cadar et al. [33] developed another family of concolic testing tools consists of EXE [33] (Execution generated executions), KLEE [51] and Rwset [34](Read Write Set). KLEE is the extended version of EXE. EXE is a bug finding tool for real time code based on concrete and symbolic execution. KLEE is also a bug finding tool, along with it also generates high line coverage on complex and environment intensive programs. Rwset uses an efficient technique to prune the redundant program paths by tracking the memory access (Read/Write) of program variables and based upon this information they limited the redundant, unimportant paths. We have used jCUTE as a concolic tester, and it supports most of the frequently used data types.

Jayraman et al. [30] developed a tool jFUZZ. It is a Concolic white box fuzzer for Java programs that built on the top of NASA tool Java Path Finder (JPF). It took a set of values from user and derived a fuzzy set of values base upon them. It helps to exercise new control path in the program. Tillman et al. [20] developed a tool Pex which is used for the test case generation of .NET based framework. It extended the concept of dynamic symbolic execution. We have developed the tool for Java programs. It takes Java program as an input and generates test cases as an output. We have developed a code transformation technique to increase the generation of test cases by jCUTE.

In Table 3.1, we have compared concolic testing tools. In Table 3.1, Column 2 shows the programing language supported by the tool. Column 3 and Column 4 represents the platform supported and Constraint solver used respectively. Column 5 shows the support for Float and Double data types are available or not. Column 6 presents the support for pointer variables is available or not. Similarly, Column 7 presents the support for native call. Column 8 and 9 shows the support for non-linear arithmetic expression and bitwise operator respectively. Column 10 and 11 tells about array offsets and function pointers. The abbreviations used in Table 3.1 are the following:

- “Y”means the tool supports the feature.
- “N”means the tool does not support the feature.
- “P”means the tool can partially support the feature.
- “NA”means unknown.

Table 3.2 shows different techniques developed using concolic testing. Column 3 and Column 4 presents the testing type and framework developed. Column 5 and Column 6 represents the input and output for the proposed approach. Similarly Table 3.3 shows characteristics of different techniques terms of test case generation, coverage percentage measurement, determination of time constraints and speed computation.

Table 3.1: Summary of concolic testers with their properties.

Tool Name	**Supporting Language**	**Supporting Platform**	**Support Constraints Solver**	**Support for float/double**	**Support for pointer**	**Support for native call**	**Support for non-linear op.**	**Support for bitwise op.**	**Support for offset**	**Support for function pointer**
DART	C	NA	LP_SOLVER	N	N	N	NA	NA	N	N
SMART	C	LINUX	LP_SOLVER	N	N	N	NA	NA	N	N
CUTE	C	LINUX	LP_SOLVER	N	Y	N	NA	NA	N	N
jCUTE	**JAVA**	**LINUX/WINDOWS**	**NA**	**N**	**-**	**N**	**NA**	**NA**	**N**	**N**
CREST	C	LINUX	YICES	N	N	N	P	P	N	N
EXE	C	LINUX	STP	N	Y	N	Y	Y	Y	N
KLEE	C	LINUX	STP	N	Y	P	Y	Y	Y	NA
RWSET	C	LINUX	STP	N	Y	N	Y	Y	Y	NA
jFUZZ	JAVA	LINUX	BUILT ON JPF	N	NA	N	N	N	NA	NA
PATH CRAWLER	C	NA	NA	NA	NA A	N	NA	NA	NA	NA
PEX	.NET	WINDOWS	Z3	N	NA	N	NA	NA	NA	NA
SAGE	MACHINE CODE	WINDOWS	DISOLVER	NA	N	Y	NA	NA	NA	NA
APOLLO	PHP	WINDOWS	CHOCO	NA	NA	N	NA	N	NA	NA
SCORE	C	LINIX	Z3 SMT Solver	Y	N	N	Y	N	NA	NA

Table 3.2: Summary of different work on concolic testing.

S.No	Authors	Testing Type	FrameWork Type	Input Type	Output Type
1	Das et al. [15, 22]	Concolic Testing, MC/DC	BCT,CREST, CA	C-Program	MC/DC%
2	Bokil et al. [29]	SC, DC, BC,MC/DC	AutoGen	C-Program	Test data, Time
3	Majumdar et al. [37]	HCT, BC	CUTE	Editor in C-Language	Test Cases
4	Burnim et al. [35]	Heuristics Concolic Testing, BC	CREST	Software Application in C	Branch Covered
5	Kim et al. [21]	Distributed Concolic Testing	SCORE	Embedded C Program	BC%, Effectiveness
6	Sen et al. [40]	Concolic Testing, BC	CUTE, JCUTE	C and Java Programs	Test Cases, BC%, Time

Table 3.3: Characteristics of different approaches on concolic testing.

Sl.No	Authors	Generated Test Cases	Measuring Coverage%	Determined Time Constraints	Computed Speed
1	Das et al. [15, 22]	✓	✓	X	X
2	Bokil et al. [29]	✓	X	✓	X
3	Majumdar et al. [37]	✓	X	X	X
4	Burnim et al. [35]	✓	X	X	X
5	Kim et al. [21]	✓	✓	X	✓
6	Sen et al. [40]	✓	✓	✓	X

3.1.4 Hybrid Concolic Testing

Majumdar et al. [37] presented a hybrid concolic testing for C programs. They have proposed an algorithm that interleaves random testing with concolic testing to achieve both a deep and a wide exploration of program state space. They have implemented their algorithm on top of CUTE tool and applied it to achieve better branch coverage for two large C based applications. For the same testing budget, almost they obtain 4× branch coverage and 2× branch coverage of random testing and concolic testing respectively. We are inspired from Majumdar et al.'s [37] core idea and proposed a new technique called Java-Hybrid Concolic Testing, which is implemented in Java language.

3.1.5 Other Related Works

Ganai et al. [58] and Ho et al. [54] proposed a techniques of VLSI design validation where a combination of formal (symbolic execution or BDD based reachability) and random simulation engines are combined to improve design coverage for big scale designs. Our proposed approach combines the Feedback-Directed Random Testing and Java Concolic Testing for Java programs to obtain better MC/DC.

Pacheco et al. [38, 41, 43] presented a technique that improves random test generation by incorporating feedback obtained from executing test cases as they are created. Their proposed approach results a test suite consisting of Java unit tests for the classes to be tested. Their experimental study shows that, use of feedback-directed random test generation is far better than systematic and undirected random test generation in term of coverage and error detection. In our approach, we used this improved random testing with the combination of Java concolic testing to obtain high MC/DC.

Ferguson et al. [59] proposed an input generation technique that is initiated by executing the program with a random input, and systematically creates the input values so that it follows the different path. In our proposed approach, we combine the generated test cases through two different testing methods to achieve high MC/DC.

Csallner et al. [36, 48] developed a tool called JCrasher. JCrasher is an independent implementation of undirected random test generation whose goal is to uncover exceptional behavior that points to a bug. It generates test data randomly, and then removes tests that throw exceptions not considered by JCrasher to potentially reveal the faults. JCrasher takes a list of classes to be tested and a 'depth' parameter that limits the number of method callers it chains together as input. JCrasher created much redundant and illegal input that could be detected using feedback-directed heuristic. Our proposed approach is based on Java Hybrid concolic testing and target to measure MC/DC percentage.

Bush et al. [56] developed a testing tool called PREfix. PREfix typically finds more defects than the software engineer. PREfix defects are not necessarily directly comparable to defect counts, because PREfix sometimes reports several defects for a single underlying cause. Bush et al. [56] reported the results for statement coverage, predicate coverage, branch coverage, and path coverage.

Li et al. [12] implemented a prototype tool named as XPTester(Xacml Policy Tester) and conducted extensive experiments upon real world policies to demonstrate the scalability, efficiency, and effectiveness. Li et al. [12] proposed an automatic XACML requests generation for testing access control policies by employing symbolic execution techniques.

Saito et al. [13] proposed an approach to generate test data for knowledge based approach to generate test scenarios for Web Applications. Their approach can generate two types of test data: Constraints-based test data and database-based test data. In our approach is based on Java language and able to process Java programs. Our tool generate test cases to compute

MC/DC%.

Godboley et al. [9, 16–18] proposed a number of code transformation technique in order to automate concolic testing for modified condition/ decision coverage test data generation. Their pre-processing techniques are helpful in solving the short-circuiting effect of logical expression which leads to a less number of test data generation. By using this code transformation technique complier will generate more number of test data by traversing all possible paths present in the execution tree.

3.2 MC/DC (Modified Condition/ Decision Coverage) Testing

Awedikian et al. [31] have proposed a new approach of automatic test cases generation for achieving MC/DC coverage. They have developed a new fitness function for the genetic algorithm. The limitation of their work is that most of the time their hill climbing strategy is used to get stuck in local minima instead of reaching to the global minima. We have developed a code transformation technique to explore some uncovered paths present in the program. The transformed program generates a number of test cases, with the help of which we can achieve higher MC/DC.

Hayhurst et al. [53] have explained the building blocks of MC/DC and interpreted this testing as a logic gate testing. They have used some boolean logic gate simplification to reduce the number of gates required to get a boolean expression. We have used pattern based short circuit methodology to generate the minimal *(n+1)* test cases required for MC/DC of *n*-conditional predicate.

Kuhn [57] has proposed the boolean difference approach to generate MC/DC specific test cases. He has also defined about the faults that may come while MC/DC test case generation. In our proposed approach, we have also tried to give a reflection of required test cases according to the MC/DC code coverage. This will help the concolic testers to generate a number of useful test cases.

Bokil et al. [29] have proposed and developed a tool called as Autogen, which generates non-redundant MC/DC test cases. This tool works only for C-programs. They have used the assert insertion methodology to create the test cases. They have found that there technique of MC/DC test case generation takes 1/3 of time than manual test case generation. Our proposed and developed tool is working for Java programs. We have used code transformation technique to increase the number of test case generate to achieve higher MC/DC.

Godboley et al. [5] have developed a new approach to distributed concolic testing (DCT) to enhance the MC/DC coverage. They have named their tool as SMDCT (Scaling MC/DC percentage using DCT). In this technique, they have used EX-NOR code transformation[32, 50] and SCORE tool for reliable and scalable concolic testing. We have used jCUTE as a

concolic test case generating tool. We have not applied any distributed environment for our proposed technique.

3.3 Testing and coverage analysis using UML diagrams

Swain et al. [28] developed a test case generation technique using UML use case and sequence diagram. Their proposed technique is used for system and integration testing. They have generated Use case dependency graph (UDG) using use-case diagram and using sequence diagram. They generated concurrent control flow graph (CCFG). By using these two graphs they have generated test cases. Their testing strategy was based upon predicate coverage.

Fraikin et al. [52] developed the concept for automated testing of OO (object oriented)-programs and also developed a tool called SeDiTeC. It uses sequence diagrams, which are complemented by test case data sets consisting of various parameters and return values for the method calls. It supports specification of various test case data sets for each and every sequence diagram. They have also introduced the concept of combined sequence diagram to reduce the number of sequence diagrams.

Rountev et al. [46] defined various coverage criteria based on control flow. This testing was for interactions among a set of collaborating objects. They proposed their technique for UML Sequence Diagrams. The coverage criteria were based on Sequence Diagram that were reverse-engineered from the program code. Their results compared different techniques for testing of object interactions and provided insights for testers and for builders of test coverage tools.

Nayak et al. [27] proposed a technique for synthesizing test data from the information embedded in model elements such as class diagrams, sequence diagrams, and OCL constraints. The test effectiveness of the system was dependent on the selection of different tests. In this regard, selecting the test cases and identifying test cases boundary was an important task. The final result of test data synthesis denoted a feasible domain which was a sub-domain of the initial domain for the selected scenarios.

Abdurazik et al. [49] proposed an approach on the fault revealing capabilities of test sets. The test cases were generated from UML statecharts and sequence diagrams. Their experimental analysis concluded that the UML diagram can be use to produce test cases. Also, they concluded that different UML diagrams play different roles in testing. Abdurazik et al. [49] considered both State chart and sequence diagrams which is a merit to their proposed work.

Chartchai et al. [23] proposed a technique which generated test data during the design phase of a software under development. They proposed a genetic algorithm (GA) technique for searching quality test data. Finally, they used these generated test data along with classes created to generate JUnit test cases.

Vadakkumacheril et al. [19] proposed a technique for implementation of sequence diagram to generate Java code with the help of XMI representation. They mainly focused on Sequence Diagrams as the model. The transformation of UML to XMI was done with help of BOUML[3] tool. They produced Java files according to the sequence diagram. They had not proposed any technique to generate test cases, they have not computed any code coverages.

3.4 Summary

In this chapter, we have thoroughly discussed the related work done in the area of test data generation. We mainly focused on random, symbolic, concolic and hybrid concolic test data generation techniques. We also presented few other related work on test data generation. We discussed about the work related to modified condition/ decision coverage testing technique. Subsequently, we have explained various works related to test data generation using UML diagrams. Out of the nine important UML diagrams, we have focused on sequence diagram.

[3] http://www.bouml.fr/index.html

Chapter 4

Java-HCT: An approach to increase MC/DC using Hybrid Concolic Testing

In this chapter we discuss a hybridized technique of MC/DC test data generation. We present the algorithmic description of proposed approach followed by detail description. We also discuss the assumptions taken and experimental analysis of the proposed technique.

Java-Hybrid Concolic Testing is the best combination of Feedback-Directed Random testing and Java Concolic Testing to achieve better MC/DC. We have inspired from the core-idea proposed by Majumdar et al.[37]. They proposed a Hybrid Concolic Testing algorithm, that interleaves random testing with concolic execution to obtain both a deep and a wide exploration of program state space. They have implemented their algorithm on top of concolic tester (CUTE) and experimented to obtain high branch coverage for two large programs; *VIM 5.7* and *Red black tree*. Their results show that hybrid concolic testing obtains almost 4× than random testing and almost 2× than concolic testing. We extend Majumdar et al.'s [37] work for measuring MC/DC and that too for Java programs. Majumdar et al. [37] implemented their algorithm using undirected random testing and concolic testing, whereas we proposed an efficient technique i.e. Feedback-Directed Random testing with concolic testing to obtain high MC/DC.

4.1 Overview of proposed framework

Our proposed technique Java-HCT consists of seven modules. These are i) Syntax_Converter, ii) RANDOOP, iii) jCUTE, iv) TCs Extractor, v) TCs Combiner, vi) TCs Minimizer, and vii) COPECA. These modules are shown in Figure 4.1. Java-HCT accepts a Java program and produces MC/DC%. Basically Java-HCT is the combination of RANDOOP and jCUTE that produces test cases which are plugged into our developed tool COPECA (Coverage Percentage Calculator) so that, the hybrid tool is capable of computing MC/DC%. Java-HCT deals with hybrid concolic testing of Java programs by combining Feedback-Directed Random Testing and concolic testing. Our proposed technique provides deep as well as wide exploration of concolic execution, which is represented in Figure 4.2.

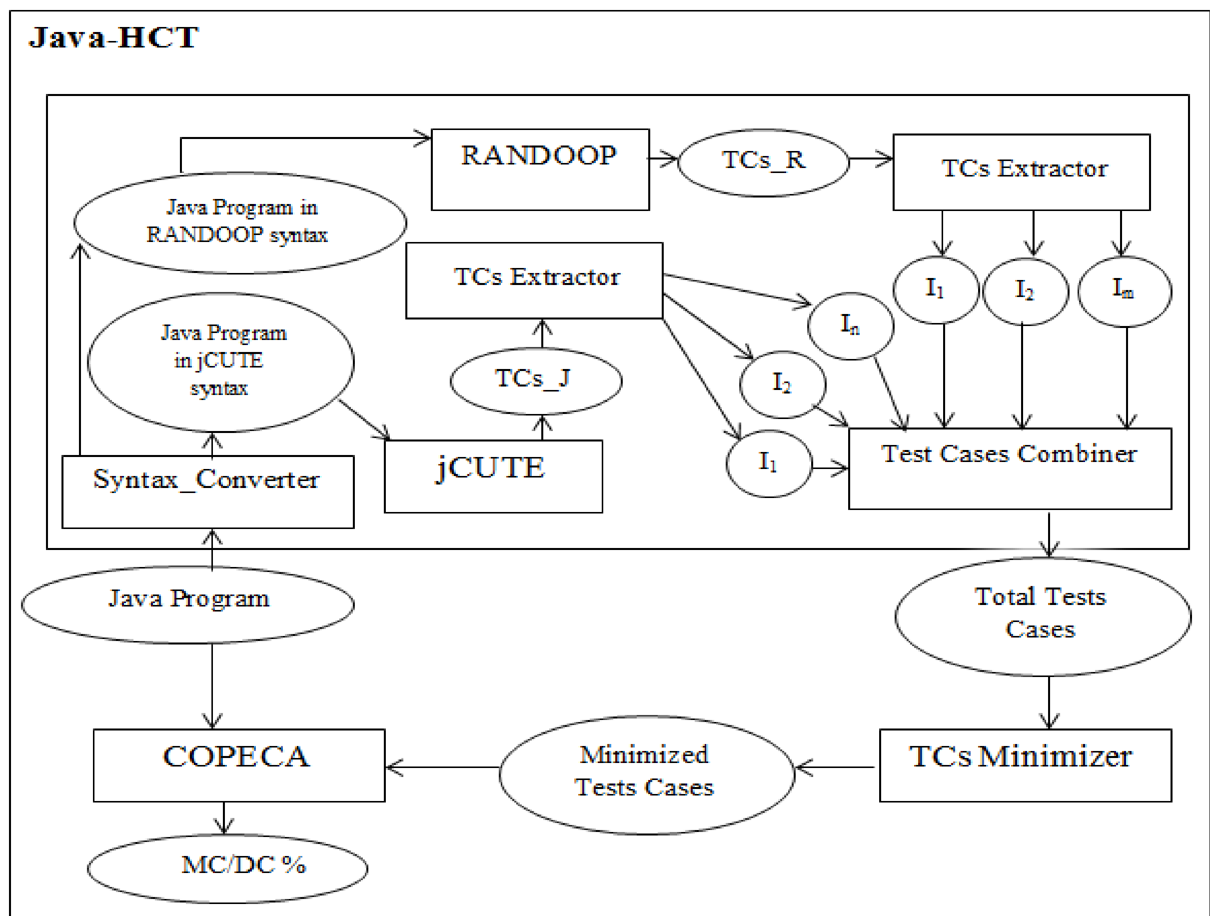


Figure 4.1: Schematic representation of Java-HCT

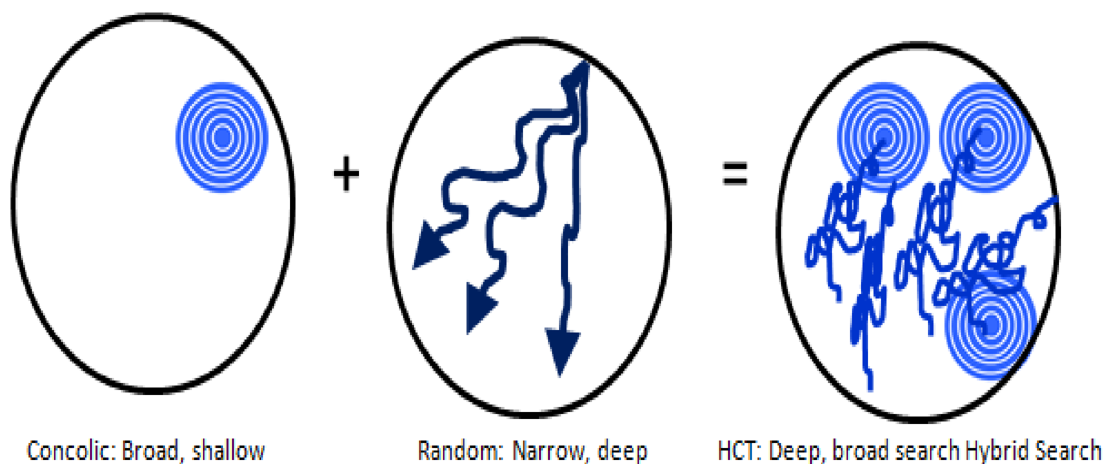


Figure 4.2: Benefit of Hybrid concolic testing

4.2 Description in detail

Figure 4.1 shows the tool for Java-Hybrid Concolic Testing (Java-HCT). Java-HCT is developed by integrating seven modules. The process starts by supplying a Java program. From Figure 4.1 we can observe that, this Java program is converted into two different syntaxes using *Syntax_converter*. Since, we supply this Java program into both RANDOOP and jCUTE, it is essential to convert the original Java program into respective tool syntaxes. Now, the Java program in RANDOOP syntax is supplied to *Random tester for Object-Oriented Programs (RANDOOP)* to generate *TCs_R* automatically. Similarly Java program in jCUTE syntax is supplied into *Java Concolic Unit Testing Engine (jCUTE)* to generate *TCs_J* automatically. Unfortunately, TCs_R and TCs_J are not in same syntax. Therefore, *TCs Extractor* module is used for both test suites to extract the input values those are present in TCs_R and TCs_J as described in Figure 4.1. Then all the extracted input values are supplied into *TCs Combiner* to produce Total test cases. Since, these test cases may be redundant and useless for MC/DC, therefore we have developed a module named *TCs Minimizer* that accepts all the input values and checks which are essential to compute MC/DC percentage and removes rest of those non-essential test cases. Now, the minimized test cases are supplied into *COverage PErcenatge CAlculator (COPECA)*. Since, we focus on increasing MC/DC percentage, so we have developed this COPECA to measaure MC/DC percentage. COPECA accepts the minimized test cases along with the original Java program as input and produces MC/DC%, as output.

Below, we discuss in detail about each module used in Java-Hybrid Concolic testing.

4.2.1 Syntax Converter

Syntax Converter is developed by us. The original Java program is not executable in both jCUTE and RANDOOP tester. So, syntax convertor converts the Java program into an appropriate formate of jCUTE and RANDOOP. In RANDOOP, we have add a new user defined function e.g. **function** which takes the input data values from the RANDOOP tester instead of the user. For RANDOOP, Syntax converter replaces all the statements inside the main function body to an another function and passed the variables as function parameter for which the RANDOOP drive test values. In RANDOOP, there is no need to add any external import package in the Java program. RANDOOP supports all the Java library files. Whereas, for jCUTE, syntax converter has to add a new package i.e. “import cute.Cute;”. Also, wherever the variables are scanning values from user r any pre-specifiewd file we have to replace it with a particular syntax of jCUTE according to the datatype of the variable.

4.2.2 RANDOOP

RANDOOP is an open source tool which generates feedback-directed random test cases for unit testing. It is a plugin for Eclipse or NetBeans IDE. We have used it with Eclipse. Radoop generates test cases randomly, but smartly. It generates sequences of constructor invocations or method invocations for the classes under test. It executes the created test sequences, and using the result generated from the execution it creates assertions, which helps to capture the behavior of the program. It generates the test cases from assertions and code sequences. RANDOOP is mainly used for two purposes: i) To detect the bugs present in the program. ii) To create test cases for regression testing. It is very powerful tool. It detects mainly previously undetected errors in IBM's JDKs, SUN's JDKs and Core.Net etc.

4.2.3 jCUTE

It stands for *Java Concolic Unit Test Engine*. It is an open source tool available on Internet [1]. It is an automated concolic testing tool which generates test cases for both simple and multi-threaded Java programs. It also supports the concurrent programs. Concolic testing combines the concrete and symbolic testing technique with using a powerful constraint solver. It discovers the deadlock and race conditions using schematic schedule explorations. jCUTE is using vectorized clock to generate large number of test cases and to support the concurrent programs. It creates execution tree for the program and tries to reach all the leaf nodes of the tree. jCUTE supports three different types of search strategies i.e. Random Search strategy, Depth First Search strategy and iii) Quick Search strategy. In the Depth First Search strategy we have to mention the maximum depth and in Quick Search strategy we have to mention the threshold value. The first value chosen jCUTE is a **Random Number**. Mostly the value is taken from one the largest number supported by the variable data type for jCUTE. It maintains log files and maintain traces for each run. The search optimality is based upon the path coverage and branch coverage.

4.2.4 TCs Extractor

It is developed by us. TCs Extractor is used for the test cases generated by RANDOOP. Actually, RANDOOP generates **'n'** number of test data files along with one extra file which contains the information about the total number of generated files. In each generates test data file there are **'m'** number of test cases are present. So, TCs Extractor extracts all the **'m'** test cases from each **'n'** number of test data files and keep it in a single file which contains total **'n*m'** test cases. This module is developed in Java language.

[1] http://osl.cs.illinois.edu/software/jcute/

4.2.5 Test Cases Combiner

Test Cases Combiner is also developed by us. It combines the test cases generated from jCUTE and RANDOOP. jCUTE generates a single test data file. So, we can directly supply it to the Test Cases Combiner, whereas RANDOOP generates a number of test data file. So, we use TCs Extractor to store all Test cases generated by RANDOOP in a single file. Therefore, the file generated by TCs Extractor is passed to the Test Cases Combiner. The output of the Test Cases Combiner is a single test data file which contains both jCUTE and RANDOOP generated test cases. The module is also developed in Java.

4.2.6 COPECA

It stands for COverage PErcentage CAlculator. COPECA is developed by us. It measures the MC/DC coverage of the given Java program using the test cases generated from jCUTE. We have developed COPECA in Java. The working principle of COPECA is based upon ETT(Extended Truth Table) creation. For each of the predicate present in the program COPECA creates Truth Table and Extended Truth Table with the help of test cases generates by the jCUTE tool. Using the extended truth table, it detects the number of independent clauses present for that particular predicate. MC/DC % is computed with the help of the following formula:

$$MC/DC\% = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m} I_j}{\sum_{i=1}^{n}\sum_{j=1}^{m} C_j} * 100 \qquad \forall_j \, C_j = 1 \tag{4.1}$$

Where, n is the total number of predicates present the program and for each predicate m number of conditions present in the predicate i. The value of m is varying from one predicate to another predicate. The value of $I_j = 1$, if I_j is an independent clause otherwise $I_j = 0$. COPECA is a very robust tool. It can handle java program of any size. The Graphical User Interface of the developed COPECA is shown in Figure 4.15.

4.2.7 TCs Minimizer

This module is also developed by us. The TCs Minimizer is used to eliminate all the redundant and non-essential test cases present in the combine test data suite. The working principle of TCs Minimizer is based upon the generated Extended Truth Table (ETT) for a predicate. It detects first essential test case pair in order to prove a simple condition as an independently affecting condition. It stores all the first time detected essential pair of test cases for each independent clauses present in the program in a unique test case set. Finally, it eliminates all the non-essential test cases present from the test suite and keeps only the essential one. This working principle of TCs Minimizer ensures that, it will not reduce the MC/DC% of the program which is computed with the set of original combined test data suite.

The Graphical User Interface of the developed TCs Minimizer is shown in Figure 4.16.

4.3 Algorithmic Description

In this section we present the algorithms used in ourproposed technique.

Algorithm 1 deals with the pseudocode of Java-Hybrid Concolic Testing. We can observe from this algorithm that we supply a Java program into Java-HCT tool to produce MC/DC%. Algorithm 1 shows the control flow of the overall procedure of our proposed approach.

Algorithm 1 Java-HCT

Input: J (Java Program)
Output: MC/DC%

1: $J_R, J_J \leftarrow Syntax_Converter(J)$
2: $TCs_R \leftarrow RANDOOP(J_R)$
3: $TCs_J \leftarrow jCUTE(J_J)$
4: $Input_values \leftarrow TCs_Extractor(TCs_R, TCs_J)$
5: $Total_TCs \leftarrow TCs_Combiner(Input_values)$
6: $Minimized_TCs \leftarrow TCs_Minimizer(Total_TCs)$
7: $MC/DC\% \leftarrow COPECA(J, Minimized_TCs)$
8: **return** $MC/DC\%$

Line 1 shows the execution of Syntax_Converter using Java program as input and produces Java program in RANDOOP syntax (J_R) and Java program in jCUTE syntax (J_J) as two outputs. Line 2 shows the execution of RANDOOP tool by supplying J_R as input to generate test cases from RANDOOP tool (TCs_R). Line 3 presents the execution of jCUTE tool by supplying J_J as input to generate test cases from jCUTE tool (TCs_J). Now, these two generated test case sets (TCs_R, TCs_J) are forwarded to Test Cases Extractor (TCs Extractor) modules to separate each input values after extracting from these two sets as presented in Line 4.

Line 5 shows the execution of Test cases Combiner (TCs Combiner). This Combiner module collects all the input values created from TCs Extractor and gathers in single set called Total Test Cases (Total TCs). Line 6 shows the minimization process of total test cases generated through Test Cases Minimizer (TCs Minimizer). This module produces Minimized Test Cases (Minimized TCs).

Line 7 deals with the computation of MC/DC% through COPECA after supplying the original Java program along with the Minimized TCs as input. Line 8 returns the final MC/DC% as output.

Algorithm 2 describes the process of (COverage PErcentage CAlculator(COPECA)). COPECA accepts Java program along with Test cases. COPECA produces MC/DC%, Time, Predicates, Clauses(C), Variables, and Independently affected Conditions (I). Line 1 starts recording execution time. Lines 2 to 4 scan all the statements in program and identify

predicates present in program. Lines 5 to 7 identify total clauses and variables present in the program. Lines 8 to 9 perform test cases file separator. Lines 10 to 13 use all test cases and drive our predicates to generate Extended Truth Table (ETT). ETT refers the concept of MC/DC. Through ETT we detect the total number of independently affected conditions after following MC/DC rules as discussed in Basic Concept. Line 17 uses a mathematical equation to finally compute the MC/DC%. Line 18 stops recording the execution time i.e. Last time-stamp. Line 19 calculates the difference of both first and last recorded time stamps to measure total execution time of COPECA.

Algorithm 2 COPECA

Input: Java Program (J), TC
Output: MC/DC%, Time, Predicate, Causes(C), Variable, Independently affected conditions(I)

1: First Time-stamp ← Start time of recording
2: **for** <$each\ statement\ s\ \epsilon\ J$> **do**
3:     **if** $s\ contains$ && or || **then**
4:         $P \leftarrow P \bigcup \{s\}$
5: **for** <$each\ predicate\ p\ \epsilon\ P$> **do**
6:     $C \leftarrow C \bigcup \{c\}$;Identify clause c present in predicate p
7:     $V \leftarrow V \bigcup \{v\}$;Identify variable v present in clause c
8: **for** <$each\ testcase\ t_c\ \epsilon\ TC$> **do**
9:     Create separate file that contains selected input values
10: **for** <$each\ predicate\ p\ \epsilon\ P$> **do**
11:     **for** <$each\ testcase\ t_c\ \epsilon\ TC$> **do**
12:         Assign selected input values from t_c into corresponding variable present in predicate p.
13:         Generate Extended-Truth Table
14: **for** <$each\ clause\ c\ \epsilon\ C$> **do**
15:     **if** $c\ is\ independently\ affected\ clause$ **then**
16:         $I \leftarrow I \bigcup \{c\}$
17:     $MCDC\% = \frac{|I|}{|C|} * 100$
18: Last Time-stamp ← Stop time of recording
19: Total execution time ← Last Time-stamp - First Time-stamp

---

Explanation 1: Why this TCs Extractor is needed?

Justification: TCs Extractor is required because the test data generated from RANDOOP are present in different test data files and each of the test data file contains many test cases. So, these two generated test case sets (Test cases generated from jCUTE and RANDOOP) are in different formats. TCs_R, and TCs_J consist of other information also. But to measure MC/DC%, we require only input values those are automatically selected for declared variables. Therefore, this extractor retrieves the only useful input values from both the sets and saves in different files which are compatible to COPECA.

Explanation 2: Why this TCs Minimizer is needed?
Justification: TCs Minimizer is required to remove non-essential test cases [42]. Non-essential test cases are test cases those are not required to support the set of conditions to be included in independently affected conditions set according to the definition of MC/DC. We observed that RANDOOP generates a large number of redundant test cases, so our minimization is helpful to remove such redundant test cases. Since, we merge two test case sets i.e. (TCs_R, TCs_J), so there may exist duplicate test cases, therefore we use our minimizer technique to remove such duplicate test cases.

4.4 Experimental Study

In this section we discuss our experimental study by explaining the experimental setup, result analysis, and threats to validity.

4.4.1 Experimental Setup

The experimental programs are ran on a computer system that has 4GB of memory (RAM) Intel(R) Core(TM)i5 CPU 650 @ 3.20 GHz 3.19 GHz and 32-bit operating system.

Our tool Java-HCT consists of mainly seven modules as shown in Figure 4.1. We have developed Syntax_Converter that converts the original Java program into two syntactically different programs according to RANDOOP and jCUTE syntax. We have used Feedback-Directed Random Testing tool (RANDOOP) developed by Pacheco et al. [43] RANDOOP uses random testing and systematic testing (Feedback-Directed) in such a way that it generates test cases that achieve better code coverage as compared to existing random testing techniques. For more information regarding RANDOOP tool, the readers are advised to refer Pacheco et al. [43]. We have used Java Concolic Unit Testing Engine(jCUTE) developed by Sen et al. [40]. CUTE uses concrete and symbolic testing simultaneously to generate test cases automatically. For more details on jCUTE please refer to [40] We have developed TCs Extractor that accepts two different test case sets (TCs_R, TCs_J) individually and produces different input values. We have developed Test Cases Combiner (TCs Combiner) that collects all input values extracted from TCs_R & TCs_J, and combines them into a single test case set. We have developed COverage PErcentage CAlculator (COPECA) based on Extended Truth Table (ETT) concept. COPECA identifies total independently affected conditions according to MC/DC criterion. COPECA receives the original Java program along with the Minimized test cases as input and to produces MC/DC% as output. Integrating all these seven modules forms the Java-HCT. Hence, we propose a new hybrid concolic testing for Java programs which is the combination of Feedback-Directed Random Testing and Java Concolic Testing. We have carried out experimental analysis of our proposed approach with forty Java programs, selected from

various sources. Some of the programs such as StringBuffer, SwitchTest etc are taken from Open Systems Laboratory Repository [2]. Some of the programs are considered from programming sites [3] and rest of the other programs are considered from the student assignments submitted by PG (Post Graduate) students of Software Engineering course at NIT Rourkela.

4.4.2 Assumptions

- The experimental program must contain at least one predicate with minimum two clauses.
- The clauses must in form of strings with pattern "$[(]^n[A-Za-z0-9_]^+[)]^n[<=,<,==,!=,>,>=][(]^n[A-Za-z0-9_]^+[)]^n$". where n ⊂ N
- To identify independently affected conditions through effect analyzer, it is essential to supply at least two test cases according to the definition of MC/DC.

4.4.3 Implementation

In this section, we present the working of our proposed technique with the help of a sample Java program.

Figure 4.3, shows a sample Java program i.e. *GradeCalculation.java*. This Java program calculates the grade of students based upon their marks. There is only one single integer variable present in this program i.e. *marks*. To execute the Java program in jCUTE or RANDOOP, we have to convert it into their respective formats. First, we supply *GradeCalculation.java* to Syntax_Converter as shown in Figure 4.4. The output of Syntax_Converter is two Java files. One is executable on jCUTE and other is on RANDOOP tester.

Figure 4.5 shows the jCUTE executable Java code. In this the header part is replaced by the jCUTE related package. We have added "import cute.Cute;" statement. Also, it has changed the variable definition syntax as per the rule of jCUTE. The integer variable *marks* is definition as "marks= Cute.input.Integer();". Remaining other statements are written same as its is. Similarly, for RANDOOP executable Java code it has made few changes. The RANDOOP executable Java code is shown in Figure 4.6. In this program all the scanning variables must be passed through as function parameter. So, we have added a new function called *function* in the program and substituted whole logic code inside it. we have passed the *marks* variable through the function as per the RANDOOP syntax. Unlike jCUTE, there is no requirement of any specific library file.

[2] https://github.com/osl/jcute/tree/master/src/tests
[3] http://www.programmingsimplied.com/java-source-codes

Figure 4.7 shows the execution of Java program on RANDOOP and successful test case generation. There are total seventeen files of test cases are generated. Each of these seventeen files contains variable number of test cases. Figure 4.8 shows the total one hundred seventy seven test cases are generated. Figure 4.9 shows the RandoopTest.java file which contains the information about all other generated test data files. It doesn't contain any test case value. It shows that by combining all the test data files, it creates a test suite. Figure 4.10 shows a test data file. A single test test data file contains a large number of test cases. All the test data values are present as the function parameter.

Figure 4.11 shows the test case generation from jCUTE tool. There are total five test cases are generated from jCUTE. The test data generate rate of jCUTE is very much lesser than the test data generate rate of RANDOOP. But, the data values generated by jCUTE is very much effective even though it is much less. Figure 4.12 shows the other parameters computed by jCUTE. We can observe that the number of functions invoked is one, total number of branches covered is eight, branch coverage percentage achieved 25% and the total number of paths traversed is seventy four out of five hundred within 21466 milliseconds $\equiv$ 21 seconds. Figure 4.13 shows the test cases generated by jCUTE.

After the successful execution of Java program, we have to execute the TCs_Executor which will merge the total 177 test cases generated by RANDOOP in a single test case file. After that the single test data file generated for the RANDOOP is combined with the total test cases generated from jCUTE. This process of test cases combination is done by an another module that is Test Cases Combiner. We have developed a tool and named it as Java-HCT. This Java-HCT contains this four modules which are TCs_Executor, Test Cases Combiner, TCs Minimizer and COPECA (Coverage Percentage Calculator). Figure 4.14 shows the graphical user interface of Java-HCT. For the given Java program, we have analyzed the Modified Condition/ Decision Coverage percentage using our developed tool COPECA. We achieved 41.667% of MC/DC with Independent Clause is equal to 5 and Simple Clause is equal to 12, when analyzed with only jCUTE generated test cases. Similarly, we have checked for RANDOOP generated test cases and we have achieved 66.667% of MC/DC. The value of Independent clause is equal to 8 and Total simple conditions is equal to 12. Then, we have combined the test cases from jCUTE (Test cases generated is equal to 6) and RANDOOP (Test cases generated is equal to 177)and with the help of combined test cases (Now, the test cases is equal to 182 (177+6)), we have computed the MC/DC%. The MC/DC percentage achieved is 83.33% which is higher than the previously computed two M<C/DC percentages. The total detected independent clauses are ten.

We can observe that there are number of entries in the extended truth table for a single condition. To prove a condition as independent a single pair of test case is also sufficient. So, we have developed a test case minimization approach based upon the redundant and non-essential test data removal strategy. The graphical user interface developed of minimization is shown in Figure 4.16. We can observe that, for the given program the

```
package hybrid_testing1;
import java.util.Scanner;
public class GradeCalculation {
public static void main(String[] args) {
Scanner in = new Scanner(System.in);
int marks = in.nextInt();
try{
System.out.print("Please enter your Marks (between 0 to 100) >> ");
if(marks<0)
{System.out.println("Marks can not be negative: Your entry= "+ marks );}
else if(marks==0)
{System.out.println("You got Zero Marks: Go to ZOO");}
else if (marks>100)
{System.out.println("Marks can not be more than 100:"+ marks );}
else if ((marks>0) && (marks<35))
{System.out.println("Failed");}
else if ((marks>=35) && (marks <50))
{System.out.println("YYour grade is C");}
else if ((marks>=50) && (marks <60))
{System.out.println("Your grade is C+");
}else if ((marks>=60) && (marks <70))
{System.out.println("Your grade is B");}
else if ((marks>=70) && (marks <80))
{System.out.println("Your grade is B+");}
else if((marks>=80) && (marks <90))
{System.out.println("Your grade is A ");}
else if (marks>=90){System.out.println("Your grade is A+");}
}catch (Exception e){
System.out.println("Invalid entry for marks:" );}
}}
```

Figure 4.3: Original Java program

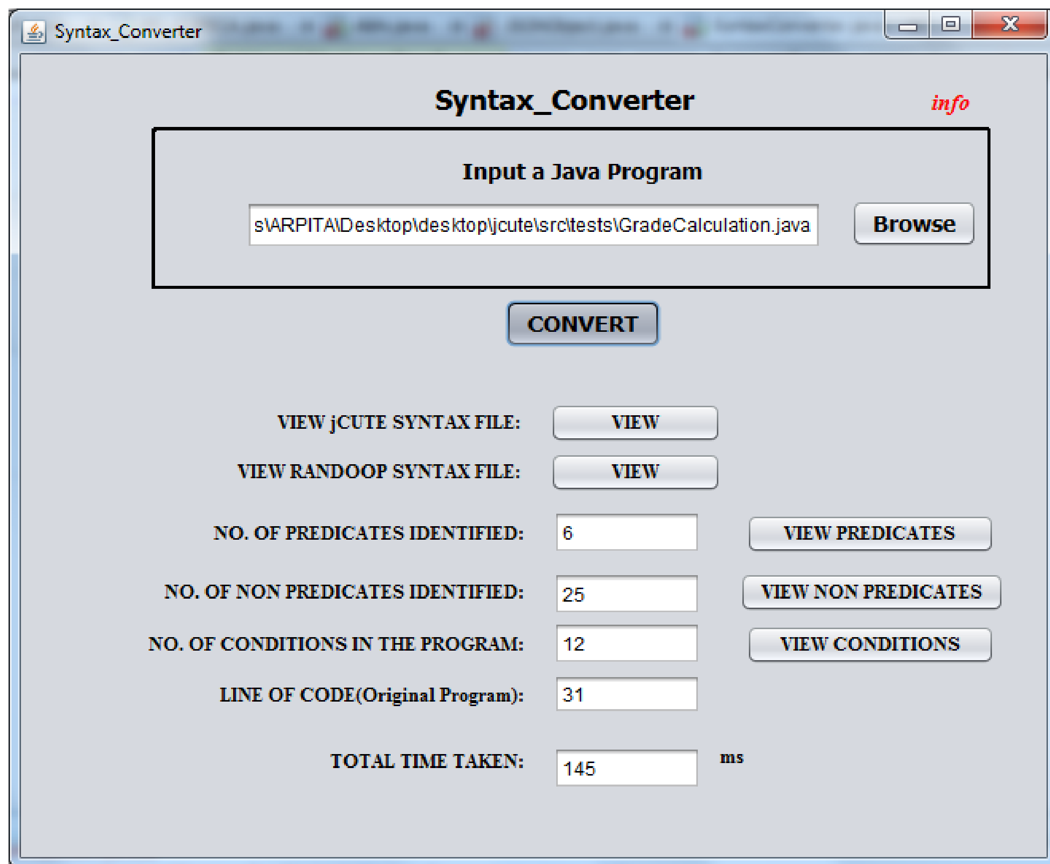


Figure 4.4: Graphical User Interface of Syntax_Converter

total number of combined test cases is 182. After applying the minimization technique, the test cases are reduced upto a very large extent. Now, the number of essential condition is only eighteen and other remaining test cases are non-essential. We have gain verified the MC/DC% with the help of these minimized test cases. We have achieved same MC/DC value i.e. 83.33%. There is no change in MC/DC. The graphical interface developed for COPECA is shown in Figure 4.15.

4.4.4 Result Analysis

Table 4.1 deals with the characteristics of different target Java programs. Column 3 shows the size of programs in Lines of codes (LOCs). Columns 4,5,6, show the Predicates, Conditions, and Variables respectively.

Table 4.2 shows the statistics of results on execution of RANDOOP and jCUTE tool. Column 3 deals with the total test cases generated. Column 4 presents the reduced number of test cases. Column 5 presents the total execution time of RANDOOP (Time_1) in seconds. Here, we have set the time 100 sec for all programs.

Table 4.3 shows the statistics of results on execution of jCUTE. Column 3 detects the total number of branches covered. jCUTE explores the unexplored paths. Column 4 shows the total number of paths covered. jCUTE finds the errors after executing the Java program.

```
package tests;
import cute.Cute;
public class GradeCalculation {
public static void main(String[] args) {
int marks;
marks= Cute.input.Integer();
try{
System.out.print("Please enter your Marks (between 0 to 100) >> ");
if(marks<0)
{System.out.println("Marks can not be negative: Your entry= "+ marks );}
else if(marks==0)
{System.out.println("You got Zero Marks: Go to ZOO");}
else if (marks>100)
{System.out.println("Marks can not be more than 100:"+ marks );}
else if ((marks>0) && (marks<35))
{System.out.println("Failed");}
else if ((marks>=35) && (marks <50))
{System.out.println("YYour grade is C");}
else if ((marks>=50) && (marks <60))
{System.out.println("Your grade is C+");
}else if ((marks>=60) && (marks <70))
{System.out.println("Your grade is B");}
else if ((marks>=70) && (marks <80))
{System.out.println("Your grade is B+");}
else if((marks>=80) && (marks <90))
{System.out.println("Your grade is A ");}
else if (marks>=90){System.out.println("Your grade is A+");}
}catch (Exception e){
System.out.println("Invalid entry for marks:" );}
}}
```

Figure 4.5: Java program in jCUTE executable format

```
package hybrid_testing;
import java.util.Scanner;
public class GradeCalculation {
public void function(int marks){
try{
System.out.print("Please enter your Marks (between 0 to 100) >> ");
if(marks<0)
{System.out.println("Marks can not be negative: Your entry= "+ marks );}
else if(marks==0)
{System.out.println("You got Zero Marks: Go to ZOO");}
else if (marks>100)
{System.out.println("Marks can not be more than 100:"+ marks );}
else if ((marks>0) && (marks<35))
{System.out.println("Failed");}
else if ((marks>=35) && (marks <50))
{System.out.println("YYour grade is C");}
else if ((marks>=50) && (marks <60))
{System.out.println("Your grade is C+");
}else if ((marks>=60) && (marks <70))
{System.out.println("Your grade is B");}
else if ((marks>=70) && (marks <80))
{System.out.println("Your grade is B+");}
else if((marks>=80) && (marks <90))
{System.out.println("Your grade is A ");}
else if (marks>=90){System.out.println("Your grade is A+");}}
catch (Exception e)
{System.out.println("Invalid entry for marks:" );}}
public static void main(String[] args) {
Scanner in = new Scanner(System.in);
int marks = in.nextInt();
GradeCalculation gc =new GradeCalculation();
gc.function(marks);
}}
```

Figure 4.6: Java program in RANDOOP executable format

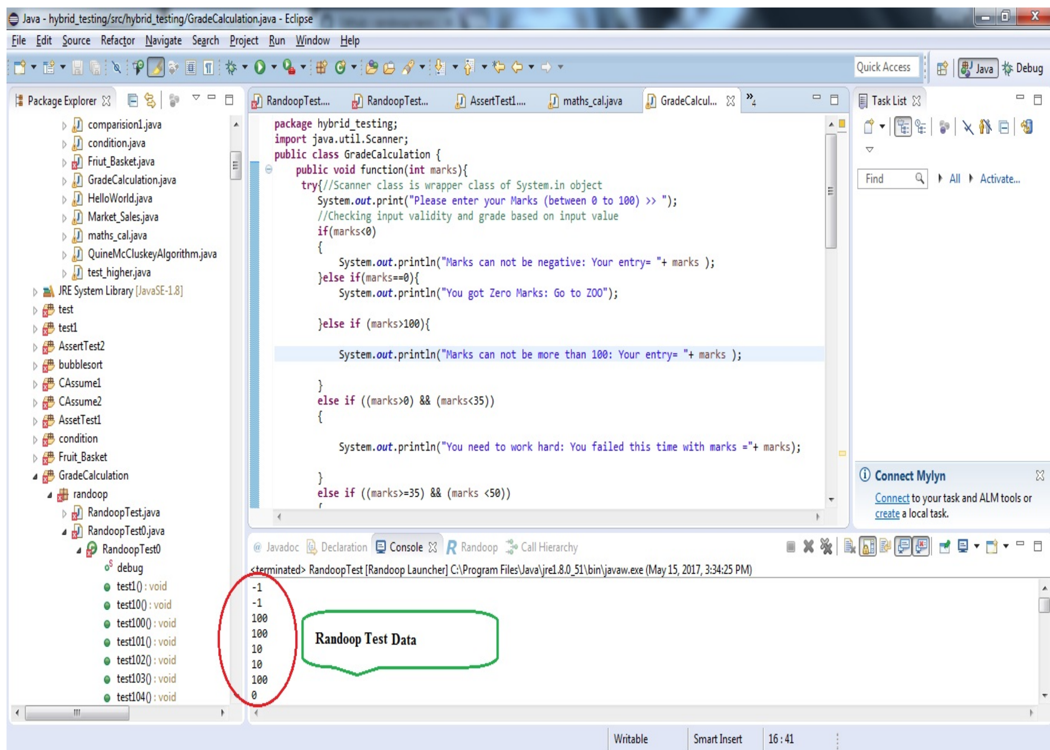


Figure 4.7: Test data generation from RANDOOP framework

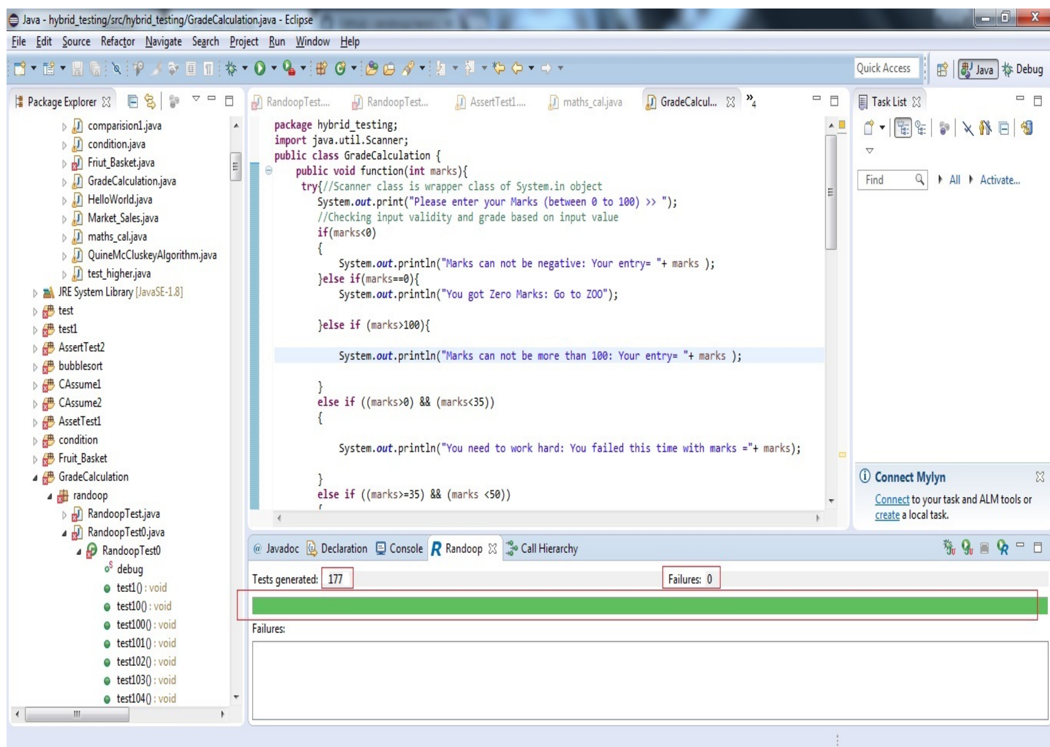


Figure 4.8: Successful execution of total test cases

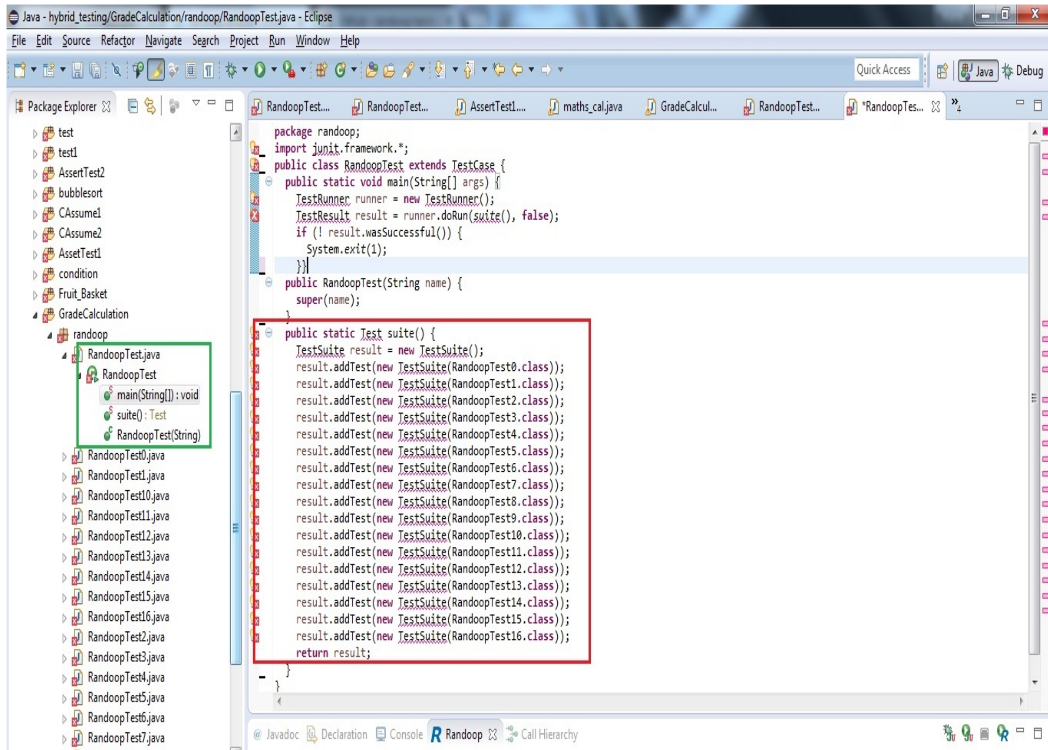


Figure 4.9: RandoopTest.java program contains information about all the generated test case files

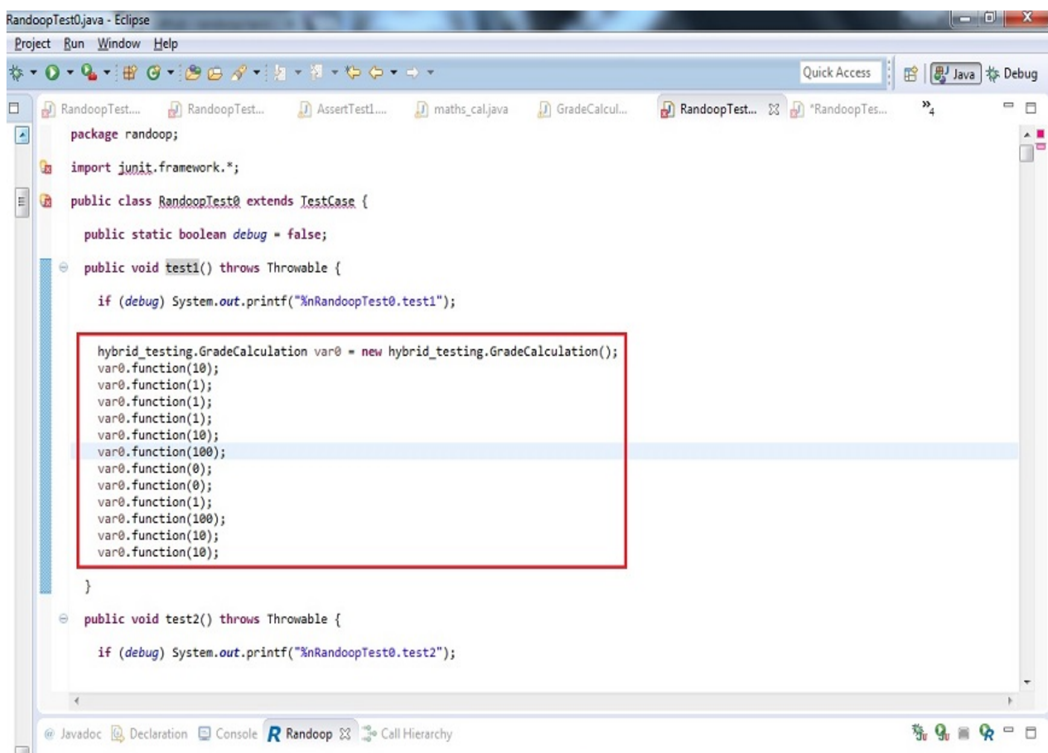


Figure 4.10: Test Cases present in a single test data file

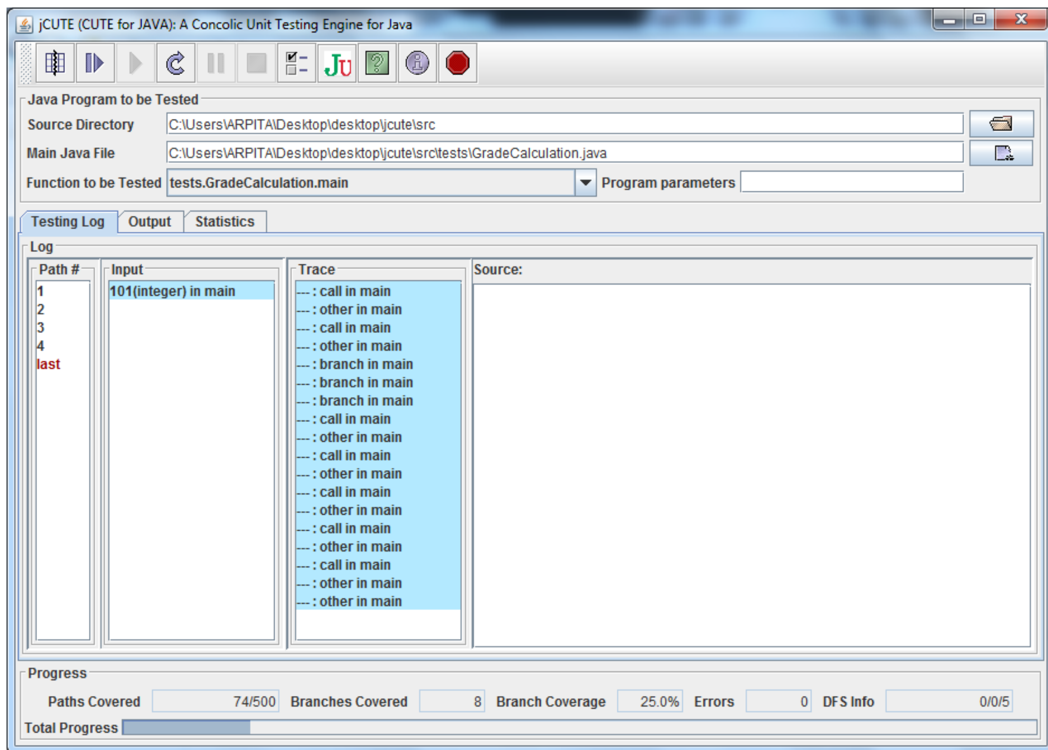


Figure 4.11: Test data generation from jCUTE tool

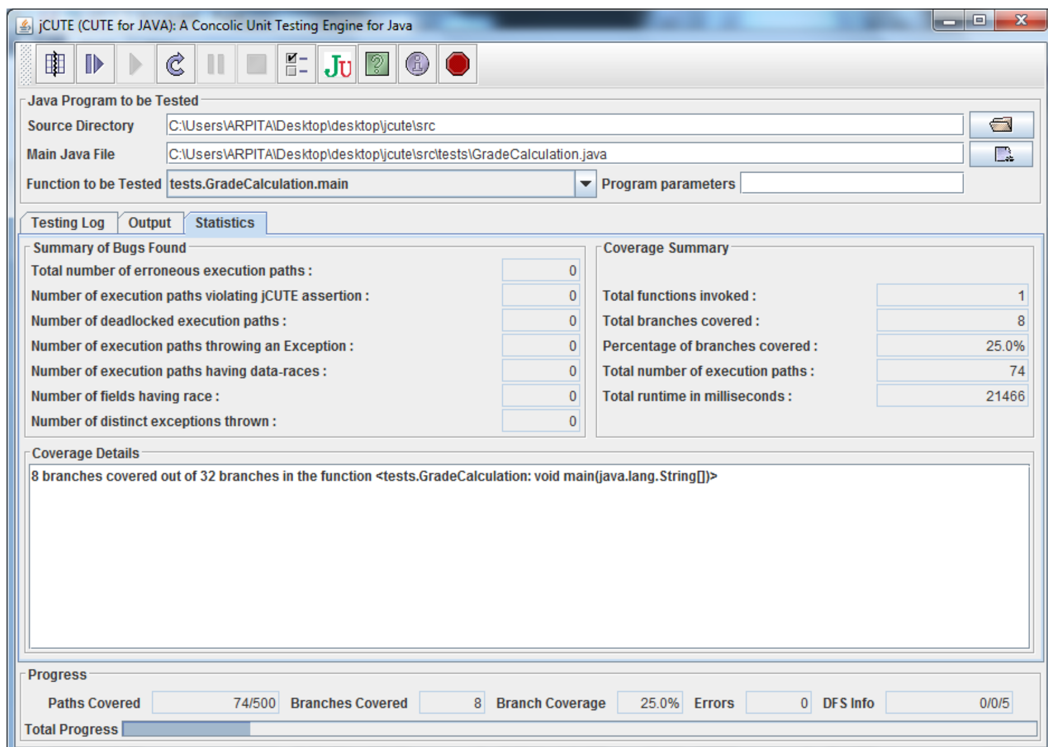


Figure 4.12: Different parameter computation using jCUTE

```
    public void test1(){
        i=0;
        input = new Object[1];
        input[i++] = new Integer(577319795);
        i=0;
        cute.Cute.input = this;
        tests.GradeCalculation.main(null);
    }

    public void test2(){
        i=0;
        input = new Object[1];
        input[i++] = new Integer(-1);
        i=0;
        cute.Cute.input = this;
        tests.GradeCalculation.main(null);
    }

    public void test3(){
        i=0;
        input = new Object[1];
        input[i++] = new Integer(0);
        i=0;
        cute.Cute.input = this;
        tests.GradeCalculation.main(null);
    }

    public void test4(){
        i=0;
        input = new Object[1];
        input[i++] = new Integer(1);
        i=0;
        cute.Cute.input = this;
        tests.GradeCalculation.main(null);
    }

}
```

Figure 4.13: Test Cases generated by jCUTE

Figure 4.14: Graphical User Interface of Java-HCT

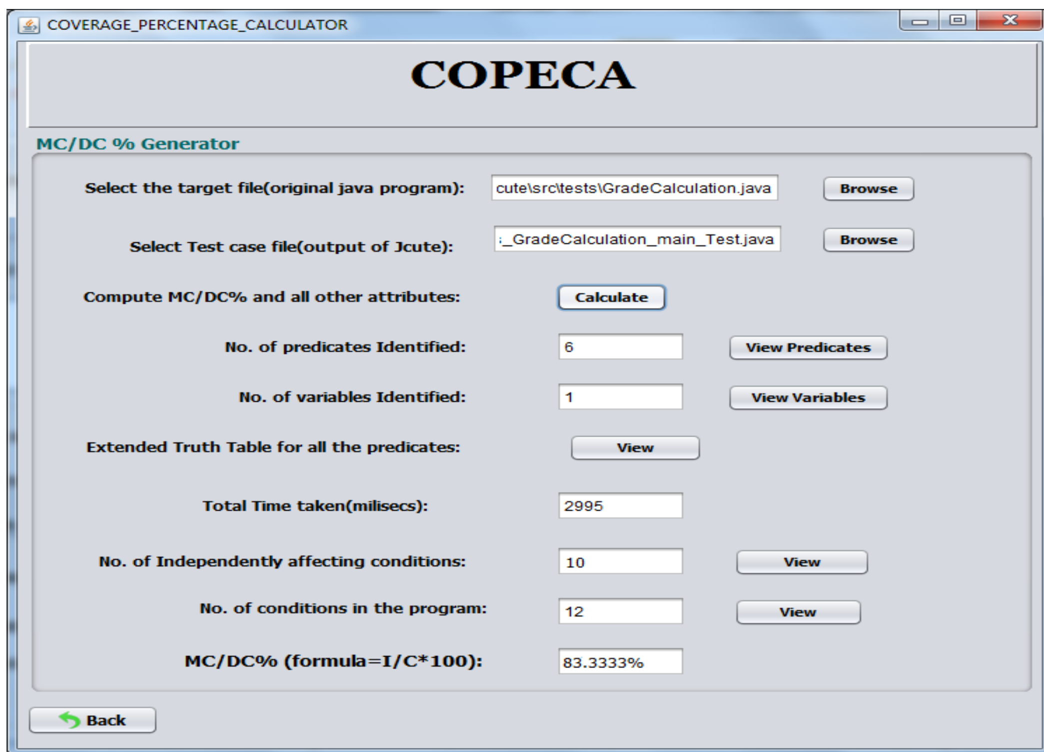


Figure 4.15: Graphical User Interface of COPECA (Coverage Percentage Calculator)

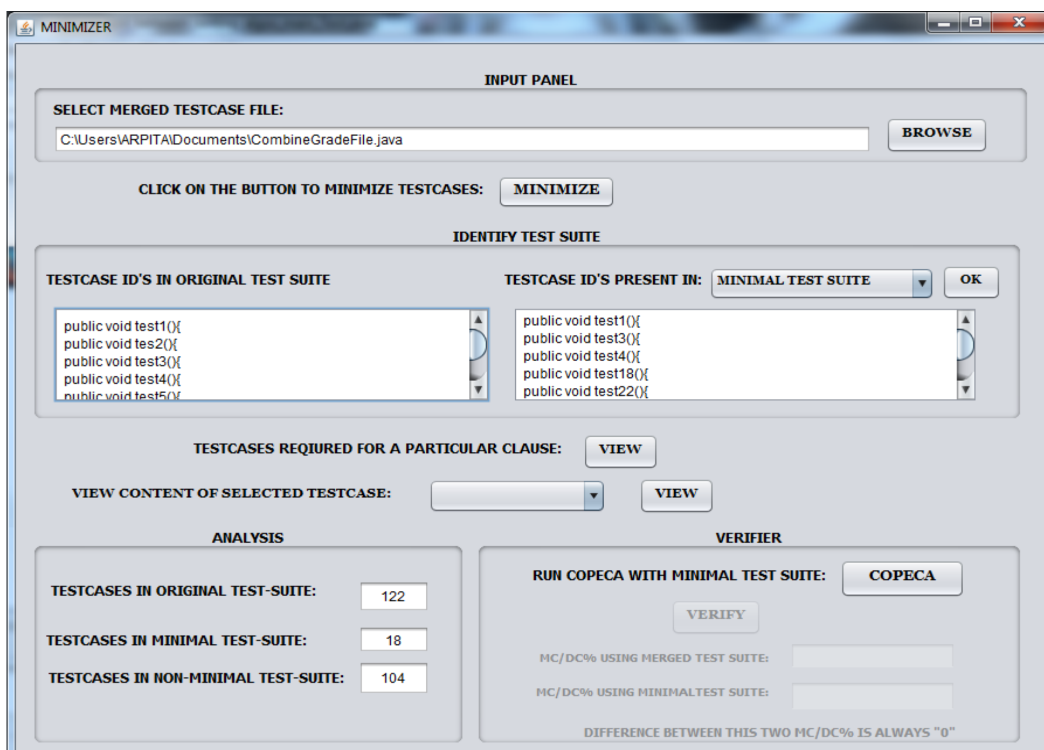


Figure 4.16: Graphical User Interface of Minimizer

Column 5 deals with the total errors detected. jCUTE produces test cases automatically. Column 6 presents total number of test cases generated. Column 7 presents the total execution time of jCUTE (Time_2).

Table 4.4 presents the generated test cases and MC/DC% for RANDOOP, jCUTE, and Java-HCT. Figure 4.17 shows the comparison of generated Test Cases. In Figure 4.17, X-axis shows the Sl. No. of programs and Y-axis shows the total number of test cases. Column 3 shows the test cases generated by Feedback-Directed Random Tetsing. RANDOOP is the tool that generates these test cases. Column 4 presents the test cases generated by Java Concolic Unit Testing Engine (jCUTE). Column 5 shows the total number of test cases of RANDOOP and jCUTE. TCs Minimizer accepts these total test cases and only selects the essential test cases according to MC/DC criterion. Column 6 presents the minimized test cases. Columns 7,8,9 deal with the MC/DC percentages according to RANDOOP, jCUTE, and Java-HCT. Figure 4.18 shows the comparison of all three MC/DC percentages. In Figure 4.18, X-axis shows the programs and Y-axis shows the computed MC/DC percentage. These percentages are defined below:

Definition 4.1 ***MC/DC_1%:*** *This MC/DC percentage is computed through RANDOOP and COPECA.*

Definition 4.2 ***MC/DC_2%:*** *This MC/DC percentage is computed through jCUTE and COPECA.*

Definition 4.3 ***MC/DC_3%:*** *This MC/DC percentage is computed through RANDOOP, jCUTE and COPECA or Java-HCT.*

Column 10 and 11 deal with the increase in MC/DC. Column 10 is named as Inc_1 and shows the difference between MC/DC_1% and MC/DC_3% as shows in Eq.4.2, whereas Column 11 named as Inc_2 shows the difference between MC/DC_2% and MC/DC_3% as shown in Eq.4.3.

$$\boxed{\text{Inc_1 = MC/DC_3\% - MC/DC_1\%}} \tag{4.2}$$

$$\boxed{\text{Inc_2 = MC/DC_3\% - MC/DC_2\%}} \tag{4.3}$$

We have experimented for forty Java programs. On an average we computed the values of Inc_1 and Inc_2 which are 29.91% and 16.26% respectively. According to the observation of our experimental study, Java-HCT achieved better MC/DC by $\times$ 1.62 as compared to RANDOOP and by $\times$ 1.26 as compared to jCUTE. Figure 4.19 shows the line graph of Increase in MC/DC for forty Java programs. In Figure 4.19, X-axis shows the programs and

Y-axis shows the differences of MC/DC percentages i.e. Increase in MC/DC.

4.5 Threats to validity

- Since, we compute MC/DC percentage, therefore programs without predicates are not useful to our experimental study.
- We combine two different testing techniques, so there may be chance of duplications, that we resolved by using TCs. Minimizer.
- The third threat to validity concerns with string value operations used by the target programs, that are not supported by our developed tool COPECA.

4.6 Comparison with related works

In this section, we present the comparison of proposed technique with some of the existing similar approaches.

Godefroid et al. [44] proposed an improved random testing technique by providing Directed fashion (Systematic way) combined with symbolic execution to generate test input values. They have merged the improved concrete and static symbolic testing and developed a new test test data generated strategy and named it as DART (Directed Automated Random Testing.) In our proposed work, we used feedback-directed random testing instead of only directed because feedback-directed provides better code coverage. We have combined feedback-directed random testing with concolic testing in order to explore more number of paths. According to Pacheco et al. [38] RANDOOP is better in completeness and scalability. So, we have combine jCUTE [45] and RANDOOP to implement the hybridization approach.

Majumdar et al. [37] presented a hybrid concolic testing for C programs, whereas we have implemented hybrid concolic testing for Java programs. They have implemented their algorithm on top of CUTE tool and random tester. We have implemented the technique with jCUTE and RANDOOP testing tools. They have worked for branch coverage, whereas we have targeted for MC/DC. They obtained $\times 4$ branch coverage and $\times 2$ branch coverage using hybrid technique over random testing and concolic testing respectively. We have improved MC/DC by $\times 1.62$ and by $\times 1.26$ for feedback-directed random testing and concolic testing respectively using hybridization.

Table 4.1: Characteristics of different target programs

Sl. No.	Program Name	LOC	# of Predicates	# of Conditions	# of Variables
1	SwitchTest	84	1	2	2
2	StringBuffer	1369	5	10	3
3	ScopeCheck	148	8	18	8
4	MyQuickSort	87	1	2	3
5	MathCall1	190	13	26	4
6	MyInsertionSort	70	2	6	4
7	Condition	60	4	9	3
8	FruitSales	267	23	69	4
9	InsertionSort	163	7	14	6
10	Comparison1	128	17	43	4
11	DSort1	136	10	20	2
12	GradeCalculation	103	6	12	1
13	MarketSales1	179	8	17	4
14	FruitBasket1	209	12	38	2
15	BSTree	307	6	13	3
16	SwitchTest2	104	6	16	5
17	AssertTest	75	3	7	3
18	BubbleSort	142	6	14	7
19	DSort_BST	305	3	7	3
20	CAssume	63	3	7	3
21	Demo1	76	3	8	2
22	MarketSales2	230	24	49	7
23	MathCall2	160	7	14	4
24	Selection_Sort	163	7	14	6
25	Sorting_algo	336	25	50	9
26	SwitchTest3	80	2	2	1
27	StringBuffer1	485	5	15	4
28	StudentGrades	67	5	10	1
29	Testy	53	3	6	1
30	Weight	39	1	3	3
31	Weight_Exp1	114	10	22	3
32	Weight_Exp2	77	5	13	3
33	Wildlife1	17	9	28	3
34	Wildlife2	199	13	40	3
35	Zodiac	104	18	84	10
36	WBS	321	5	10	3
37	AssertTest2	91	7	21	7
38	HelloWorld	44	2	4	2
39	IFExample	82	2	4	2
40	IFSample	95	6	12	3

Table 4.2: Statistics of results on execution of RANDOOP

Sl. No.	Program Name	Test Cases	Reduced TCs (TCs_R)	Time_R (Sec)
1	SwitchTest	31173	20	100
2	StringBuffer	12866	12	100
3	ScopeCheck	56174	20	100
4	MyQuickSort	4112	5	100
5	MathCall1	32731	70	100
6	MyInsertionSort	9008	13	100
7	Condition	12013	26	100
8	FruitSales	131201	114	100
9	InsertionSort	28011	28	100
10	Comparison1	47813	93	100
11	DSort1	30132	39	100
12	GradeCalculation	27613	23	100
13	MarketSales1	31031	43	100
14	FruitBasket1	53131	271	100
15	BSTree	21941	86	100
16	SwitchTest2	31217	29	100
17	AssertTest	9017	31	100
18	BubbleSort	18106	43	100
19	DSort_BST	2017	36	100
20	CAssume	7362	73	100
21	Demo1	9894	44	100
22	MarketSales2	50136	313	100
23	MathCall2	31014	38	100
24	Selection_Sort	35814	53	100
25	Sorting_algo	71313	343	100
26	SwitchTest3	2015	5	100
27	StringBuffer1	30131	15	100
28	StudentGrades	17012	103	100
29	Testy	17134	19	100
30	Weight	6893	10	100
31	Weight_Exp1	2013	25	100
32	Weight_Exp2	33134	26	100
33	Wildlife1	51013	40	100
34	Wildlife2	46813	50	100
35	Zodiac	96001	190	100
36	WBS	12813	20	100
37	AssertTest2	19315	42	100
38	HelloWorld	6814	10	100
39	IFExample	7969	12	100
40	IFSample	8981	24	100

Table 4.3: Statistics of results on execution of jCUTE

Sl. No.	Program Name	Branches Covered	Paths Covered	Errors Found	Test Cases (TCs_J)	Time_J (milli sec)
1	SwitchTest	18	101	0	8	20640
2	StringBuffer	32	131	42	9	28895
3	ScopeCheck	53	346	0	22	77311
4	MyQuickSort	15	1	0	5	957
5	MathCall1	41	313	0	10	90911
6	MyInsertionSort	20	32	0	6	12366
7	Condition	27	83	0	7	23546
8	FruitSales	81	470	75	12	140154
9	InsertionSort	45	225	0	10	21900
10	Comparison1	91	1000	0	27	237689
11	DSort1	52	250	3	4	68551
12	GradeCalculation	8	66	0	5	17484
13	MarketSales1	35	139	0	8	39810
14	FruitBasket1	44	150	0	8	326767
15	BSTree	41	6	1	5	2580
16	SwitchTest2	42	684	0	14	159575
17	AssertTest	14	11	11	7	15174
18	BubbleSort	39	132	0	8	218288
19	DSort_BST	19	12	3	8	2986
20	CAssume	10	181	0	6	49683
21	Demo1	9	73	0	4	26663
22	MarketSales2	71	38	0	11	12664
23	MathCall2	32	159	0	11	45596
24	Selection_Sort	45	114	0	9	38443
25	Sorting_algo	114	224	0	9	122434
26	SwitchTest3	234	75	0	11	18350
27	StringBuffer1	41	7	0	7	4321
28	StudentGrades	18	41	1	8	6894
29	Testy	4	13	0	3	514
30	Weight	39	35	0	4	1041
31	Weight_Exp1	114	142	0	10	6692
32	Weight_Exp2	77	133	0	8	5318
33	Wildlife1	176	173	0	6	2070
34	Wildlife2	199	234	0	10	4227
35	Zodiac	104	25	0	63	14028
36	WBS	321	63	0	7	23634
37	AssertTest2	91	100	120	13	170047
38	HelloWorld	44	121	0	5	1937
39	IFExample	82	7	1	7	3174
40	IFSample	95	11	6	13	13519

Table 4.4: Results on execution of COPECA

Sl. No.	Program Name	RANDOOP TCs	jCUTE TCs	Total TCs TCs	Minimized	MC/DC_1%	MC/DC_2%	MC/DC_3%	Inc_1	Inc_2
1	SwitchTest	20	8	28	4	50	50	1000	50	50
2	StringBuffer	12	9	21	20	40	50	80	40	30
3	ScopeCheck	20	25	45	30	77.77	83.33	100	23.23	16.67
4	MyQuickSort	5	5	10	3	100	100	100	0	0
5	MathCall1	70	10	80	45	16.66	46.15	69.23	52.57	23.08
6	MyInsertionSort	13	6	19	11	0	50	83.33	83.33	33.33
7	Condition	26	7	33	15	44.44	66.67	88.88	44.44	22.21
8	FruitSales	114	12	126	112	31.88	42.02	56.52	24.64	14.5
9	InsertionSort	28	10	38	20	71.42	78.57	85.71	14.29	7.14
10	Comparison1	93	27	120	81	27.90	41.86	58.13	30.23	16.27
11	DSort1	39	4	43	28	50	75	85	35	10
12	GradeCalculation	23	5	28	18	33.33	50	75	16.7	25
13	MarketSales1	43	8	51	23	52.94	64.70	88.23	35.29	23.53
14	FruitBasket1	271	8	279	59	39.47	50	60.52	21.05	10.52
15	BSTree	86	5	91	24	23.07	69.23	84.61	61.54	15.38
16	SwitchTest2	29	14	42	30	12.5	18.75	31.25	18.75	12.5
17	AssertTest	31	7	38	9	57.14	57.14	100	42.86	42.86
18	BubbleSort	43	8	56	21	35.71	42.85	64.28	28.57	21.43
19	DSort_BST	36	8	44	9	42.85	28.57	57.14	14.29	28.57
20	CAssume	73	6	79	10	71.42	85.71	100	28.58	14.29
21	Demo1	44	4	48	12	62.5	75	87.5	25	12.5
22	MarketSales2	313	11	324	78	69.38	73.46	73.46	4.08	0
23	MathCall2	38	11	49	20	57.14	64.28	71.42	14.28	7.14
24	Selection_Sort	53	9	62	18	35.71	42.85	50	14.29	7.15
25	Sorting_algo	343	9	352	73	28	50	70	42	20
26	SwitchTest3	5	11	16	4	50	100	100	50	0
27	StringBuffer1	15	7	22	23	86.66	86.66	100	13.34	13.34
28	StudentGrades	103	8	111	20	30	50	80	50	30
29	Testy	19	3	22	11	50	66.66	83.33	33.33	16.67
30	Weight	10	4	14	5	33.33	33.33	66.66	33.33	33.33
31	Weight_Exp1	25	10	35	33	95.45	95.45	95.45	0	0
32	Weight_Exp2	26	8	34	18	100	100	100	0	0
33	Wildlife1	40	6	46	32	7.14	17.85	53.57	46.43	35.72
34	Wildlife2	50	10	60	59	10	40	50	40	10
35	Zodiac	190	63	253	131	5.95	16.66	27.86	21.91	11.2
36	WBS	20	7	27	18	0	20	30	30	10
37	AssertTest2	42	13	55	40	38.09	66.67	76.19	38.1	9.52
38	HelloWorld	10	5	15	5	100	100	100	0	0
39	IFExample	12	7	19	7	50	100	100	50	0
40	IFSample	24	13	37	5	75	83.33	100	25	16.67

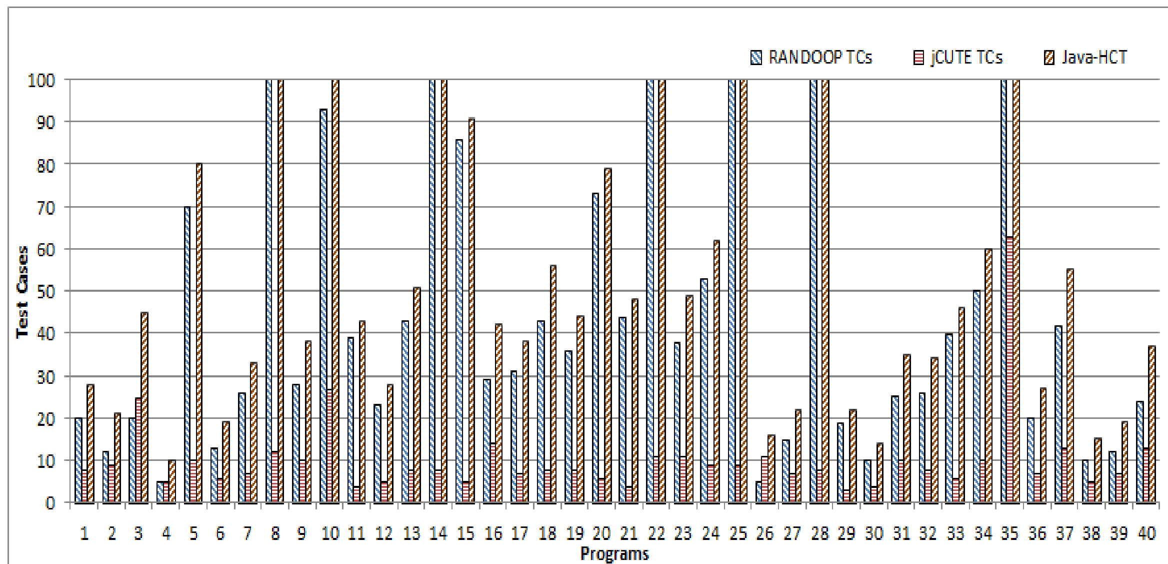


Figure 4.17: Total number of Test Cases generated

4.7 Summary

We have proposed a hybrid technique of feedback-directed random testing and concolic testing to improve the MC/DC% of input Java programs. We have explained the proposed technique in detail with its schematic representation. We have shown the implementation of the proposed algorithm with the help of an example Java program. We have stated the assumptions taken. We have experimented with forty Java programs and found on an average increase of 29.91% and 16.26%, when compared to feedback-directed random testing and concolic testing respectively. We have improved MC/DC by $\times 1.62$ and by $\times 1.26$ in comparison to feedback-directed random testing and concolic testing respectively. In the next chapter, we propose an approach to compute MC/DC% at design phase using UML sequence diagram.

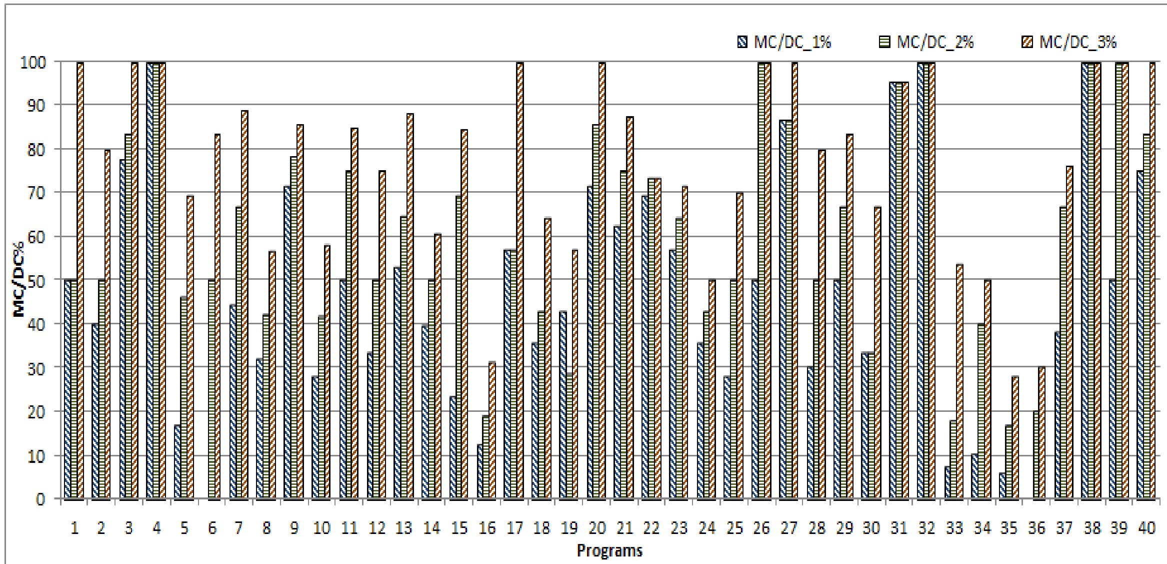


Figure 4.18: Computed MC/DC percentages

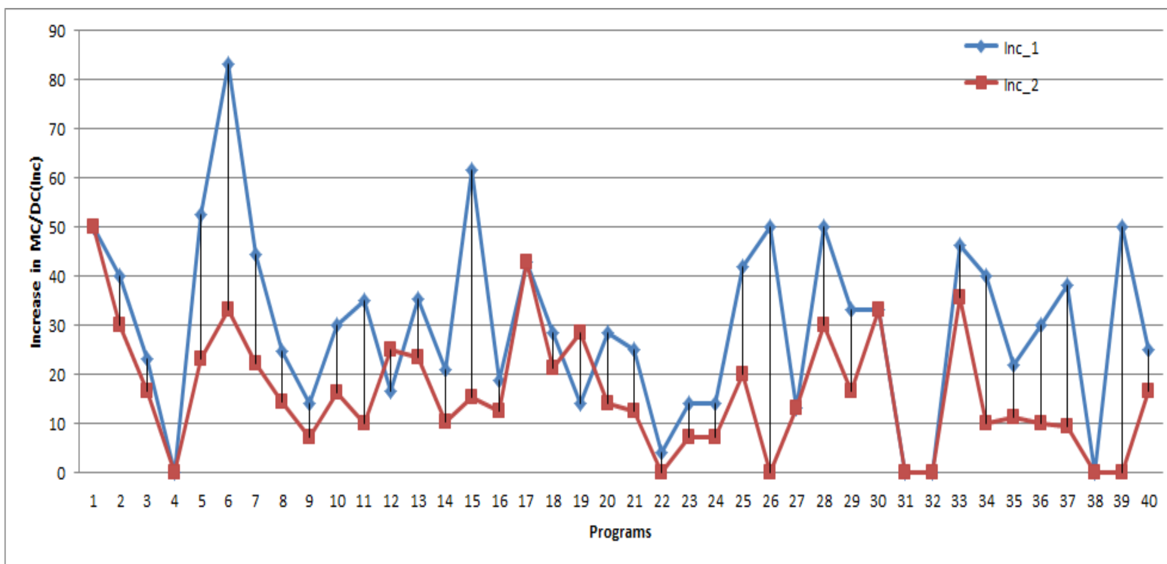


Figure 4.19: Increase in MC/DC percentages

Chapter 5

Measuring MC/DC at Design Phase using UML Sequence Diagram

In this chapter, we present our proposed approach fo measuring MC/DC at design phase using UML sequence diagrams. We describe the proposed approach in detail with the algorithmic description and implementation using a sample sequence diagram. We also present our experimental analysis of the proposed approach.

Now-a-days, testing starts from requirements gathering onwards. As early we start the testing of the given system, the chances of failure reduces. We know that, Object-oriented programs are much more difficult and require more efficient techniques for designing and testing as compared to procedural programs. For procedural programs, we use to draw data flow diagrams (DFD). But, DFDs have manly limitations. They do not depict the control flow information and many other relevant information. So, in order to resolve those limitations, for object-oriented programs, UML diagrams are introduced. UML diagrams contain different views of the software system with the help of different diagrams. We have developed a technique to test UML sequence diagram. Testing at design phase helps to plan a better program structure, increases the software reliability and also reduces the overall testing cost. We measure MC/DC percentage of the given system using UML sequence diagram. It also helps to understand the complexity of the given system.

5.1 Overview of proposed framework

Figure 5.1 shows the schematic representation of MC/DC Analyzer for UML Sequence Diagram (MAUSD). MAUSD consists of four modules, which are as follows: ArgoUML [1], JAXB, jCUTE [2], and COPECA. ArgoUML, JAXB, and jCUTE are open source modules, whereas COPECA is our developed module. ArgoUML produces XML after designing UML Sequence Diagram. Java Architecture for XML Binding (JAXB) converts the XML code into Java code. Now, this converted Java code is prepared manually according to

[1] http://argouml.tigris.org/
[2] http://osl.cs.illinois.edu/software/jcute/

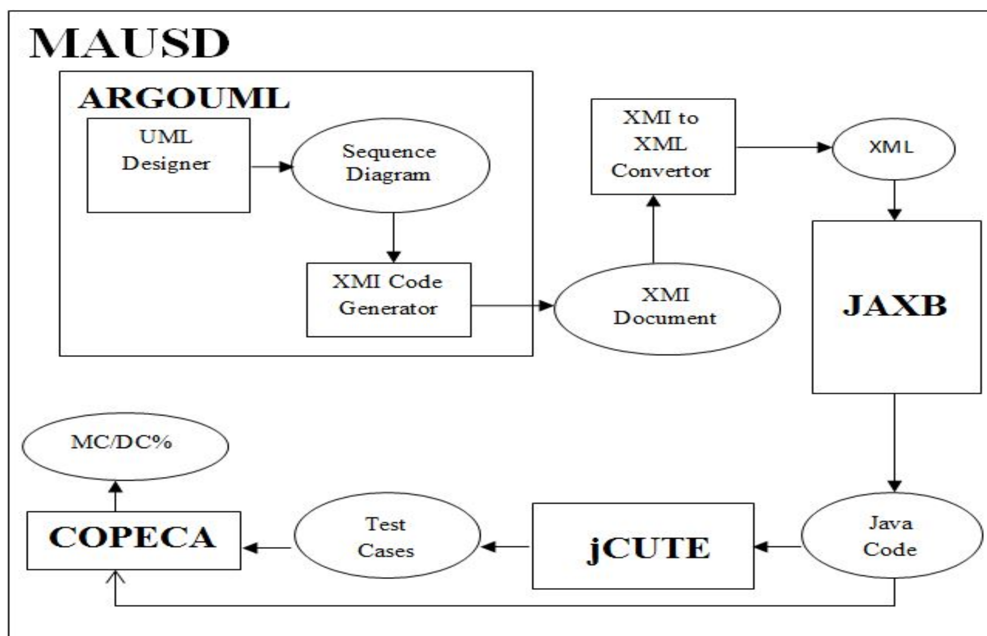


Figure 5.1: Schematic representation of MAUSD

the template of jCUTE tool. jCUTE accepts this Java program and generates test cases automatically. Now, these test cases are supplied along with the Java program to compute MC/DC%.

5.2 Description in detail

In this section, we discuss the flow of schematic representation of MAUSD. Also, we describe each component of MAUSD.

5.2.1 ArgoUML

ArgoUML is an open source UML (Unified Modeling Language) tool with BSD license. It supports all standard UML 1.4 diagrams. UML 1.4 diagrams are Class diagram, Activity diagram (including Swimlanes), Statechart diagram, Use Case diagram, Deployment diagram (includes Object and Component diagram in one), Collaboration diagram. AlgoUML is running on Java platform. It is available in 10 different languages. It also provides code generation for C++, Java, PHP4 and PHP5. Argo UML also supports the reverse engineering from the Java source code (i.e. diagram generation from the source code). Documentation of ArgoUML is available [3]. ArgoUML also provides platform to generate the database schema and it also allows to do code in other languages such as Delphi or Ruby. ArgoUML generates .xmi file for each of the behavioral diagram. For our proposed work of measuring MC/DC% of UML Sequence Diagram, we have used

[3]http://argouml.tigris.org/documentation/index.html

AgroUML to design the Sequence diagram. We have generated the .xmi file from the sequence diagram using the same tool only. The .xmi file contains all the information present in the sequence diagram. For each of the component present in the sequence diagram, it generates an unique xmi:id. The main components of the .xmi file are as follows:

- Connector
- Message
- Lifeline
- Attribute

The .xmi file is a type of .xml file with few more information. In order to generate the Java code we require XML file. So, we have used the .xmi code to generate the .xml file. For this transformation, it requires only to change the extension of the file format.

5.2.2 JAXB

It stands for *Java Architecture for XML Binding*. It is an open source tool [4]. It converts the Java objects into XML and XML into Java Objects. It provides the facility to change the Java objects into xml, the conversion process of xml to Java objects is known as **marshalling** (write). Similarly, it also converts the XML into Java objects. The vice-versa conversion process is known as **unmarshalling**(read). The basic operation of JAXB is shown in Figure 5.2. Features of JAXB are as follows:

1. Annotation support
2. Additional Validation Capabilities
3. Small Runtime Library
4. Additional Validation Capabilities
5. Reduction of generated schema-derived classes

In the proposed approach, we have used JAXB to generate Java Objects from XML file generated from the sequence diagram. The Java objects are present in a .java file. The .java file is having structure of normal Java with some additional packages. The additional packages are annotation packages. Such as javax.xml.bind.annotation.XmlAttribute; javax.xml.bind.annotation.Xml Element; javax.xml.bind.annotation.XmlRootElement; etc. The process of to Java object conversion is as follows:

[4]http://www.java2s.com/Code/Jar/j/Downloadjaxbapi22jar.htm

Figure 5.2: Fundamental working of JAXB

- **Step 1:** Create POJO or bind the schema and generate the classes
- **Step 2:** Create the JAXBContext object
- **Step 3:** Create the Unmarshaller objects
- **Step 4:** Call the unmarshal method
- **Step 5:** Use getter methods of POJO to access the data

5.2.3 jCUTE

It stands for *Java Concolic Unit Test Engine*. It is an open source tool available on Internet [5]. It is an automated concolic testing tool which generates test cases for both simple and multi-threaded Java programs. It also supports concurrent programs. Concolic testing combines the concrete and symbolic testing techniques using a powerful constraint solver. It discovers the deadlock and race conditions using schematic schedule explorations. jCUTE uses vectorized clock to generate large number of test cases and to support the concurrent programs. It creates execution tree for the program and tries to reach all the leaf nodes of the tree. jCUTE supports three different types of search strategies i.e. Random Search strategy, Depth First Search strategy and Quick Search strategy. In the Depth First Search strategy we have to mention the maximum depth and in Quick Search strategy we have to mention the threshold value. The first value chosen by jCUTE is a **Random Number**. Mostly the value is taken from one the largest number supported by the variable data type for jCUTE. It maintains log files and traces for each run. The search optimality is based upon the path coverage and branch coverage.

5.2.4 COPECA

It stands for COverage PErcentage CAlculator. COPECA is developed by us. It measures the MC/DC coverage of the given Java program using the test cases generated from jCUTE. We have developed COPECA in Java. The working principle of COPECA is based upon ETT(Extended Truth Table) creation. For each of the predicate present in the program, COPECA creates the Truth Table and Extended Truth Table with the help of test

[5]http://osl.cs.illinois.edu/software/jcute/

cases generated by jCUTE tool. Using the extended truth table, it detects the number of independent clauses present in that particular predicate. MC/DC % is computed using the following formula:

$$MC/DC\% = \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} I_j}{\sum_{i=1}^{n} \sum_{j=1}^{m} C_j} * 100 \qquad \forall_j \; C_j = 1 \tag{5.1}$$

where, n is the total number of predicates present the program and for each predicate m number of conditions present in the predicate i. The value of m varies clause, otherwise $I_j = 0$. COPECA is a very robust tool. It can handle Java programs of any size.

5.3 Algorithmic Description

Algorithm 3 MAUSD

Input: UML Sequence Diagram

Output: MC/DC%

1: Design a Sequence diagram for the given use cases using ArgoUML.
2: Generate the XMI code from the designed UML Sequence diagram using ArgoUML.
3: Execute JAXB using XMI code to produce an executable Java code.
4: Generate test cases for the executable Java code using jCUTE.
5: Taking the test cases and Java code as input, Compute MC/DC% using COPECA and the formula given below:

$$MC/DC\% = (I/C) \times 100 \tag{5.2}$$

where, the number of independently affected conditions is denoted by I and the total number of simple conditions is denoted by C.

Algorithm 3 deals with the function of MC/DC Analyzer for UML Sequence Diagram (MAUSD). Line 1 of Algorithm 3 shows the use of ArgoUML to design the UML Sequence Diagram after understanding the concept of given use cases. Line 2 shows code-generation of UML Sequence Diagram in the form of XMI using ArgoUML. Line 3 presents the code conversion from XMI to executable Java code using JAXB. Line 4 shows the use of jCUTE (Java concolic tester) to accepts the Java code to generate test cases. Line 5 deals with COPECA execution. COPECA takes the test cases and Java code as input and computes the MC/DC percentage as output.

5.4 Experimental Study

Here we discuss the details of tools, experimental setup required results of our experiment.

5.4.1 Experimental Setup

Our computer is configured by using dual core processor of Intel(R) Core(TM) i5 version. The processing speed of CPU is 3.20 GHz. RAM is installed of 4 GB and operating system is of 32 bit. We have used Windows 7 Professional as experimental platform. We have used three open source tools and one is our developed tool. The open source tools are ArgoUML (Argo Unified Modeling Language), JAXB (Java Architecture for XML Binding), jCUTE (Java Concolic Unit Test Engine), and our developed tool is COPECA (Coverage Percentage Calculator). JAXB is available as a plug-in for Eclipse. Remaining ArgoUML and jCUTE are individual tools easily available on Internet.

5.4.2 Assumptions

In this section, we present the assumptions taken for the proposed technique.

- All the sequence diagrams are designed based upon the given requirement and designer understandability. It can be varied from person to person.
- The program should contain at-least one predicate (two clauses). If value of Clauses is “0” then MC/DC percentage will be undefined.
- We require at least two test cases to prove a condition as independent condition.

5.4.3 Implementation

In this section, we explain the proposed approach of MC/DC analysis at design level using an example Sequence diagram.

Figure 5.3 shows Sequence diagram for a scenario where a job seeker is searching for the information related to jobs. In this diagram the number of actors involved is 1 and the number of objects involved is 2. total synchronous and asynchronous messages present is 3 and 3 respectively. The number of loops involved is 1. Figure 5.4 shows the .xml code generated for the sequence diagram. After getting the .xml code, we supplied it to JAXB(Java Architecture for XML Binding) to generate the compatible Java objects. The Java code obtained by JAXB is shown in Figure 5.5. But, the Java file generated from JAXB is not compatible with the concolic tester jCUTE. So, we have done some manual interpretation in the Java code obtained from JAXB to make it compatible with jCUTE. After that the compatible Java code is compiled and executed on Concolic tester jCUTE. Figure 5.6 shows the compilation of Java code on jCUTE. Similarly Figure 5.7, Figure 5.8and Figure 5.9 show the execution of jCUTE and the results obtained. The total number of branches covered, paths covered are 65 and 10 respectively. The branch coverage percentage is the 62.5% in the total execution time of 2069 ms. At last, we supplied the compatible Java code and the test cases obtained by jCUTE to the COPECA (Coverage Percentage Calculator) and

computed the MC/DC percentage. The MC/DC percentage obtained for the Java program is 100%. Developed Graphical User Interface (GUI) for COPECA is shown in Figure 5.10.

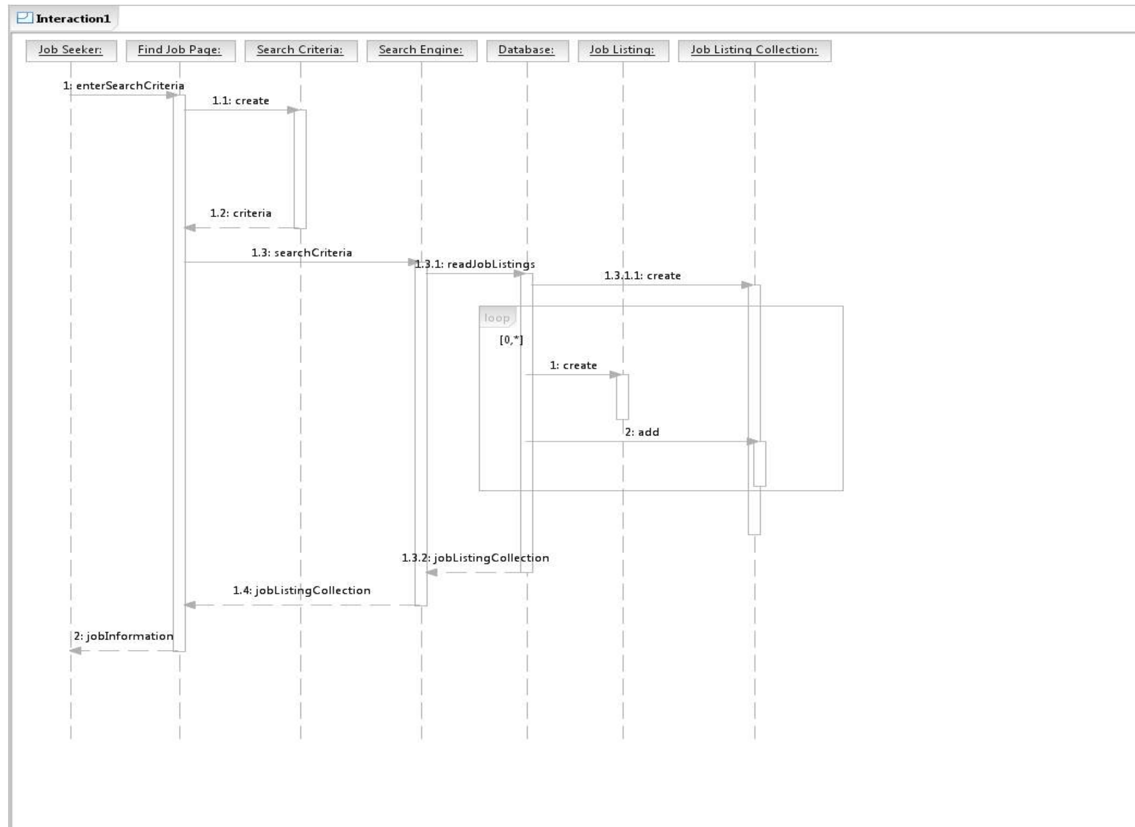


Figure 5.3: Sequence diagram for Job searching

5.4.4 Result

Table 5.1 shows the experimental characteristics of the case studies under taken for our experiment. We experimented with five case studies. Individual case study consists of different number of use cases. So, according to the use cases, there exists different number of corresponding sequence diagrams. In Table 5.1, Columns 2 and 3 show the case study names and scenarios related to the use cases, respectively. Column 4 shows the total number of actors involved. Column 5 presents the total number of objects created for each sequence diagram. Columns 6 and 7 show the total number of synchronous messages and asynchronous messages respectively. Column 8 shows the number of loop combine fragments in the sequence diagram. Column 9 shows the number of Alternative combine fragments. Column 10 shows the total number of conditions present.

Table 5.2 shows the results obtained on the execution of jCUTE (Java Concolic Test Engine) tool for the Java program obtained from JAXB after some manual changes. Column 4 and 5 present the number of branches covered and number of paths covered. Column 6

```xml
<message xmi:type="uml:Message" xmi:id="_2gWseDaDEee_DJV0-7utXQ" name="criteria" messageSort="reply" receiveEvent="_2gWsVzaDEee_DJV0-7utXQ" sendEvent=
"_2gWsVjaDEee_DJV0-7utXQ" connector="_2gWsNzaDEee_DJV0-7utXQ"/>
<message xmi:type="uml:Message" xmi:id="_2gWseTaDEee_DJV0-7utXQ" name="searchCriteria" receiveEvent="_2gWsWTaDEee_DJV0-7utXQ" sendEvent=
"_2gWsWDaDEee_DJV0-7utXQ" connector="_2gWsPTaDEee_DJV0-7utXQ"/>
<message xmi:type="uml:Message" xmi:id="_2gWsejaDEee_DJV0-7utXQ" name="jobListingCollection" messageSort="reply" receiveEvent="_2gWscjaDEee_DJV0-7utXQ"
sendEvent="_2gWscTaDEee_DJV0-7utXQ" connector="_2gWsPTaDEee_DJV0-7utXQ"/>
<message xmi:type="uml:Message" xmi:id="_2gWsezaDEee_DJV0-7utXQ" name="readJobListings" receiveEvent="_2gWsXDaDEee_DJV0-7utXQ" sendEvent=
"_2gWsWzaDEee_DJV0-7utXQ" connector="_2gWsQDaDEee_DJV0-7utXQ"/>
<message xmi:type="uml:Message" xmi:id="_2gWsfDaDEee_DJV0-7utXQ" name="jobListingCollection" messageSort="reply" receiveEvent="_2gWscDaDEee_DJV0-7utXQ"
sendEvent="_2gWsbzaDEee_DJV0-7utXQ" connector="_2gWsQDaDEee_DJV0-7utXQ"/>
<message xmi:type="uml:Message" xmi:id="_2gWsfTaDEee_DJV0-7utXQ" name="create" receiveEvent="_2gWsXzaDEee_DJV0-7utXQ" sendEvent=
"_2gWsXjaDEee_DJV0-7utXQ" connector="_2gWsQzaDEee_DJV0-7utXQ"/>
<message xmi:type="uml:Message" xmi:id="_2gWsfjaDEee_DJV0-7utXQ" name="create" receiveEvent="_2gWsZzaDEee_DJV0-7utXQ" sendEvent=
"_2gWsZjaDEee_DJV0-7utXQ" connector="_2gWsRjaDEee_DJV0-7utXQ"/>
<message xmi:type="uml:Message" xmi:id="_2gWsfzaDEee_DJV0-7utXQ" name="add" receiveEvent="_2gWsazaDEee_DJV0-7utXQ" sendEvent="_2gWsajaDEee_DJV0-7utXQ"
connector="_2gWsQzaDEee_DJV0-7utXQ"/>
</ownedBehavior>
<ownedAttribute xmi:type="uml:Property" xmi:id="_2gWsgDaDEee_DJV0-7utXQ" name="Job Seeker" end="_2gWsNTaDEee_DJV0-7utXQ"/>
<ownedAttribute xmi:type="uml:Property" xmi:id="_2gWsgTaDEee_DJV0-7utXQ" name="Find Job Page" end="_2gWsNjaDEee_DJV0-7utXQ _2gWsODaDEee_DJV0-7utXQ
_2gWsPjaDEee_DJV0-7utXQ"/>
<ownedAttribute xmi:type="uml:Property" xmi:id="_2gWsgjaDEee_DJV0-7utXQ" name="Search Criteria" end="_2gWsOTaDEee_DJV0-7utXQ _2gWsOzaDEee_DJV0-7utXQ
_2gWsPDaDEee_DJV0-7utXQ"/>
<ownedAttribute xmi:type="uml:Property" xmi:id="_2gWsgzaDEee_DJV0-7utXQ" name="Search Engine" end="_2gWsPzaDEee_DJV0-7utXQ _2gWsQTaDEee_DJV0-7utXQ"/>
<ownedAttribute xmi:type="uml:Property" xmi:id="_2gWshDaDEee_DJV0-7utXQ" name="Database" end="_2gWsQjaDEee_DJV0-7utXQ _2gWsRDaDEee_DJV0-7utXQ
_2gWsRzaDEee_DJV0-7utXQ"/>
<ownedAttribute xmi:type="uml:Property" xmi:id="_2gWshTaDEee_DJV0-7utXQ" name="Job Listing" end="_2gWsSDaDEee_DJV0-7utXQ"/>
<ownedAttribute xmi:type="uml:Property" xmi:id="_2gWshjaDEee_DJV0-7utXQ" name="Job Listing Collection" end="_2gWsRTaDEee_DJV0-7utXQ"/>
</packagedElement>
<packagedElement xmi:type="uml:SendOperationEvent" xmi:id="_2gWshzaDEee_DJV0-7utXQ" name="SendOperationEvent1"/>
<packagedElement xmi:type="uml:ReceiveOperationEvent" xmi:id="_2gWsiDaDEee_DJV0-7utXQ" name="ReceiveOperationEvent1"/>
<profileApplication xmi:type="uml:ProfileApplication" xmi:id="_2gWsiTaDEee_DJV0-7utXQ">
<xmi:Extension extender="http://www.eclipse.org/emf/2002/Ecore">
<eAnnotations xmi:type="ecore:EAnnotation" xmi:id="_2gWsijaDEee_DJV0-7utXQ" source="http://www.eclipse.org/uml2/2.0.0/UML">
<references xmi:type="ecore:EPackage" href="http://schema.omg.org/spec/UML/2.1.1/StandardProfileL2.xmi#_yzU58YinEdqtvbnfB2L_5w"/>
```


Figure 5.4: XML code generated for the Sequence diagram shown in Figure 5.3

shows the achieved branch coverage percentage. Column 7 shows the total execution time (in milliseconds) of jCUTE. jCUTE also generates the test cases. but we have kept the test data information in an another table because of its comparison with MC/DC related parameters. On an average for the five case study projects, we obtained 11.59 number of branches covered, 8.11 number of paths covered, 73.12% percentage of branch coverage in 6095.22 ms of execution time.

Table 5.3 shows the result analysis of our experiment. Column 4 shows the total number of test cases generated through jCUTE tool. Column 5 shows the total number of independently affected conditions present in the Java code of each sequence diagram. Column 6 shows the total number of simple conditions present in the Java code. The entries in Columns 5 and 6 are computed using COPECA (Coverage percentage Calculator). COPECA also computes MC/DC% which is presented in Column 7. Column 7 presents the execution time of COPECA. On an average we have obtained $(1.7 \equiv 2)$ number of independent clause and $(55.29 \equiv 55)$% MC/DC for five case study projects with $(3.85 \equiv 4)$ number of average test cases. The average execution time of COPECA is $(59.25 \equiv 59)$ ms.

We can also compute the test case generation speed of Concolic tester jCUTE using Equation 5.3

$$Test\ Case\ Generation\ Speed = \frac{Number\ of\ Test\ Cases}{Test\ Case\ Generation\ Time} \tag{5.3}$$

Fig. 5.11 shows the total number of test cases generated for the number of conditions present in the sequence diagram. Number of test cases generates is always greater than or equal to the number of conditions present the program. Fig. 5.12 presents the comparison

```
 2⊕// This file was generated by the JavaTM Architecture for XML Binding(JAXB) Reference Implement
 7 package xmljava;
 8⊖import java.io.Serializable;
 9 import java.util.ArrayList;
10 import java.util.List;
11 import javax.xml.bind.JAXBElement;
12 import javax.xml.bind.annotation.XmlAccessType;
13 import javax.xml.bind.annotation.XmlAccessorType;
14 import javax.xml.bind.annotation.XmlAttribute;
15 import javax.xml.bind.annotation.XmlElement;
16 import javax.xml.bind.annotation.XmlElementRef;
17 import javax.xml.bind.annotation.XmlElements;
18 import javax.xml.bind.annotation.XmlMixed;
19 import javax.xml.bind.annotation.XmlRootElement;
20 import javax.xml.bind.annotation.XmlSchemaType;
21 import javax.xml.bind.annotation.XmlType;
22 import javax.xml.bind.annotation.XmlValue;
23 import javax.xml.namespace.QName;
24 @XmlAccessorType(XmlAccessType.FIELD)
25 @XmlType(name = "", propOrder = {
26     "_import",
27     "partnerLinks",
28     "variables",
29     "sequence"
30 })
31 @XmlRootElement(name = "process")
32 public class Process {
33
34⊖    @XmlElement(name = "import")
35     protected List<Process.Import> _import;
36⊖    @XmlElement(required = true)
37     protected Process.PartnerLinks partnerLinks;
38⊖    @XmlElement(required = true)
39     protected Process.Variables variables;
```

Figure 5.5: Java code generated from JAXB for the XML code shown in Figure 5.4

```
jCUTE (CUTE for JAVA): A Concolic Unit Testing Engine for Java

Java Program to be Tested
Source Directory      C:\Users\DPM\Documents\NetBeansProjects\PDL\jcute_for_after_transformation\jcute\src
Main Java File        C:\Users\DPM\Documents\NetBeansProjects\PDL\jcute_for_after_transformation\jcute\src\tests\xmljavaProcess.java
Function to be Tested tests.xmljavaProcess.main      Program parameters

Testing Log | Output | Statistics
Output
cd C:\Users\DPM\Documents\NetBeansProjects\PDL\jcute_for_after_transformation\jcute\tmpjcute
javac -d C:\Users\DPM\Documents\NetBeansProjects\PDL\jcute_for_after_transformation\jcute\tmpjcute\classes -sourcepath
C:\Users\DPM\Documents\NetBeansProjects\PDL\jcute_for_after_transformation\jcute\src
C:\Users\DPM\Documents\NetBeansProjects\PDL\jcute_for_after_transformation\jcute\src\tests\xmljavaProcess.java
Exit 0

cd C:\Users\DPM\Documents\NetBeansProjects\PDL\jcute_for_after_transformation\jcute\tmpjcute
java -Xmx512m -Xms512m -Dcute.sequential=false cute.instrument.CuteInstrumenter -keep-line-number -d
C:\Users\DPM\Documents\NetBeansProjects\PDL\jcute_for_after_transformation\jcute\tmpjcute\classes -x com.vladium -x cute -x lpsolve --app
tests.xmljavaProcess
Soot started on Fri May 12 09:48:49 IST 2017
Transforming tests.xmljavaProcess...
Sig:"<java.lang.Object: void <init>()>"
Sig:"<cute.Input: int Integer()>"
Sig:"<java.io.PrintStream: void print(java.lang.String)>"
Writing to C:\Users\DPM\Documents\NetBeansProjects\PDL\jcute_for_after_transformation\jcute\tmpjcute\classes\tests\xmljavaProcess.class
Soot finished on Fri May 12 09:48:49 IST 2017
Soot has run for 0 min. 0 sec.
Exit 0

Progress
Paths Covered 0   Branches Covered 0   Branch Coverage 0.0   Errors 0   DFS Info 0
Total Progress
```

Figure 5.6: Compilation on jCUTE

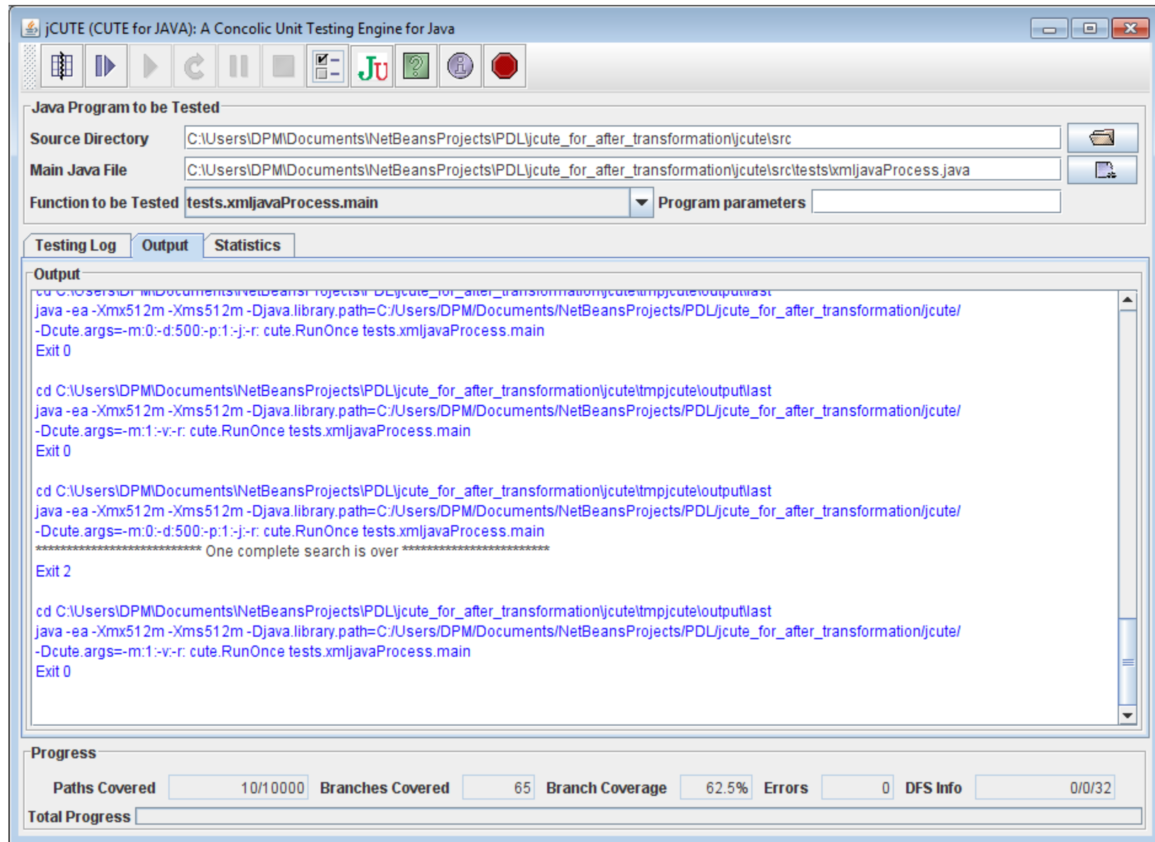


Figure 5.7: One complete execution on jCUTE

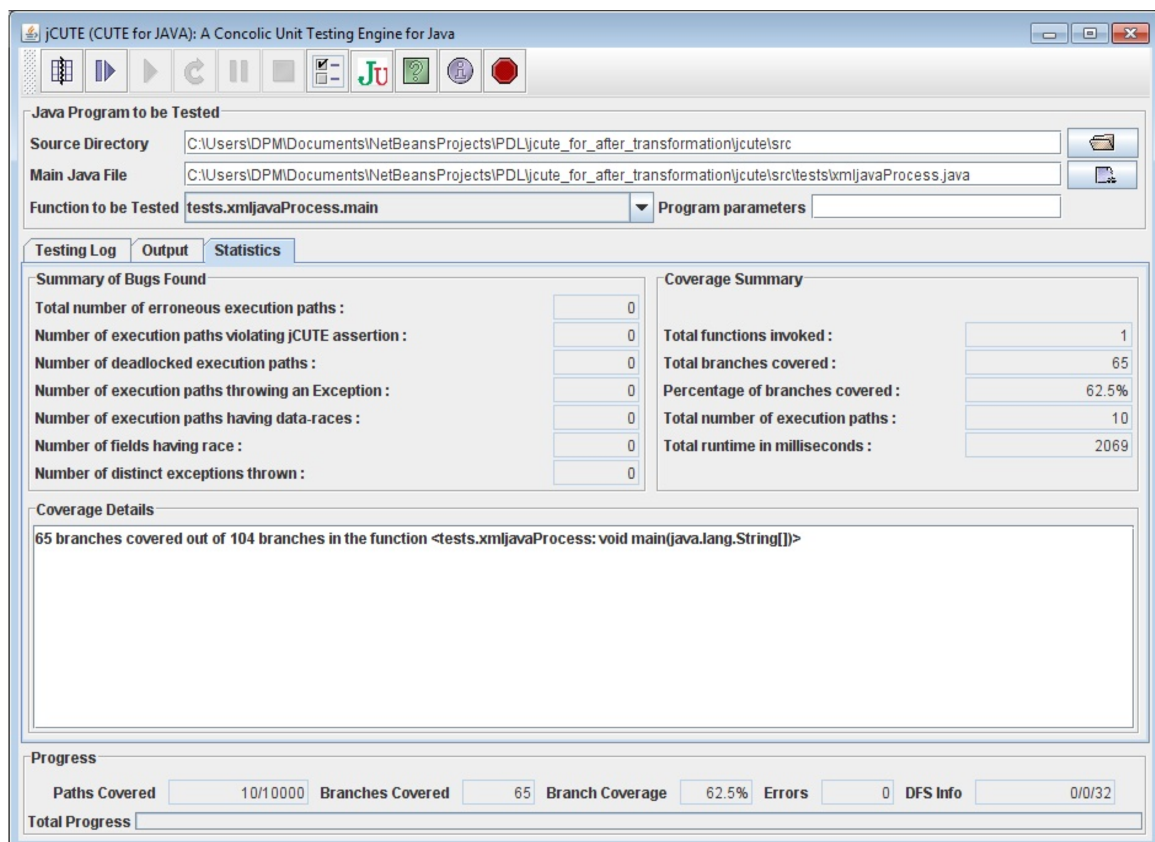


Figure 5.8: Results obtained by using jCUTE

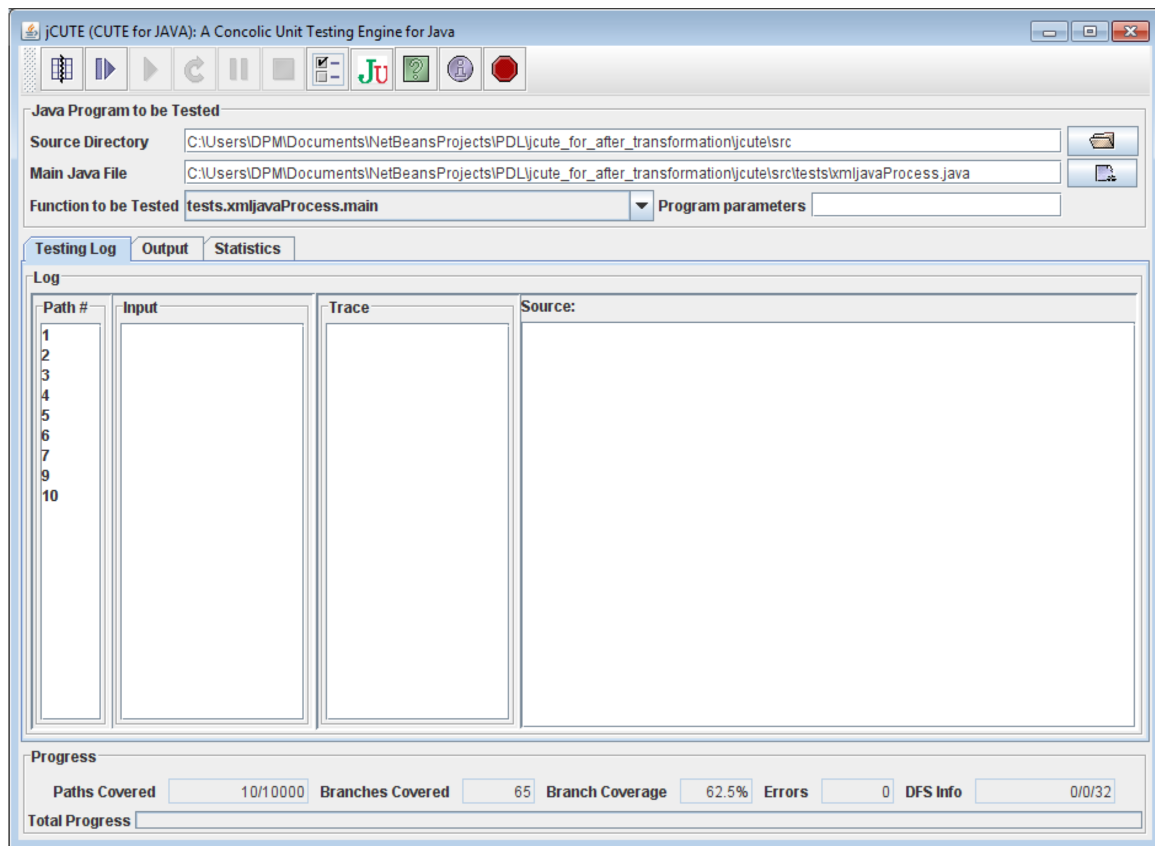


Figure 5.9: Test Cases generated by jCUTE

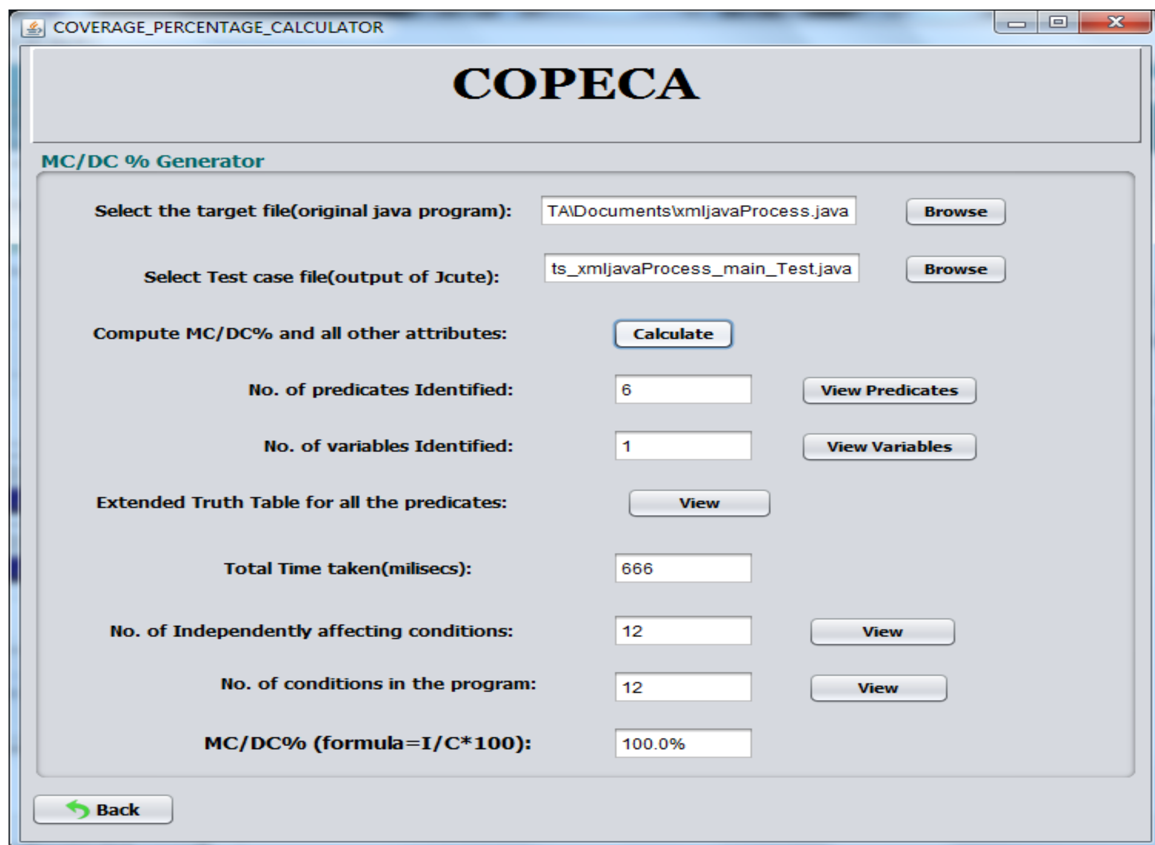


Figure 5.10: MC/DC analysis using COPECA

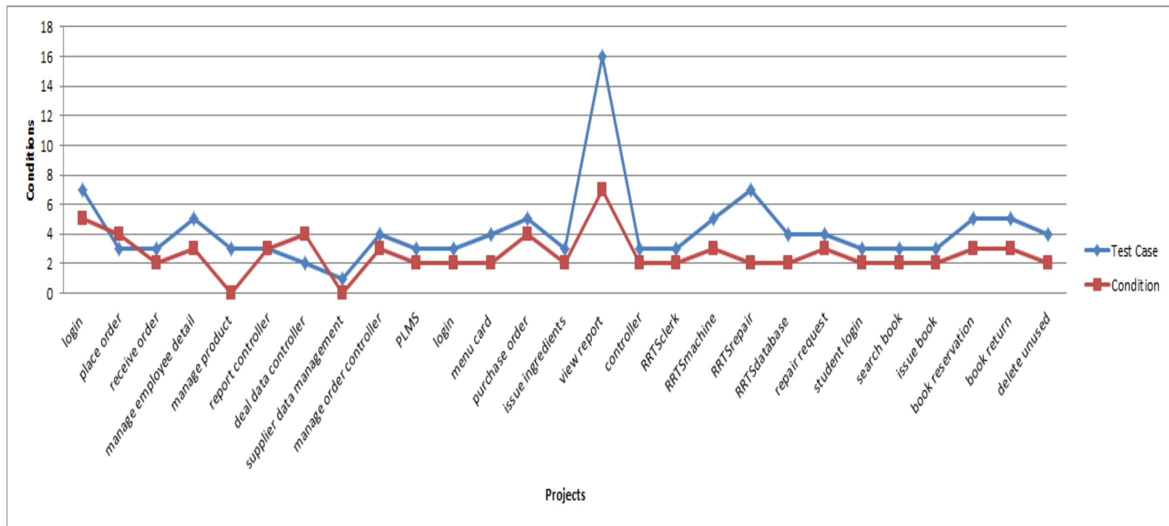


Figure 5.11: Number of generated test cases vs. the number of conditions present in the Sequence Diagrams.

between the total number of independently affected conditions and total number simple conditions. Similarly, Figure 5.13 shows the comparison between the branch coverage percentage and MC/DC percentage. We can observe that the MC/DC coverage is always less than the branch coverage. The reason behind the lesser amount of MC/DC than branch coverage is the subsumption relationship. MC/DC is a stronger coverage criteria than branch coverage. In branch coverage, we have to satisfy the each atomic condition present in the predicate but, in case of MC/DC coverage we have to satisfy the whole composite condition relation in order to prove the independence of the unique clause.

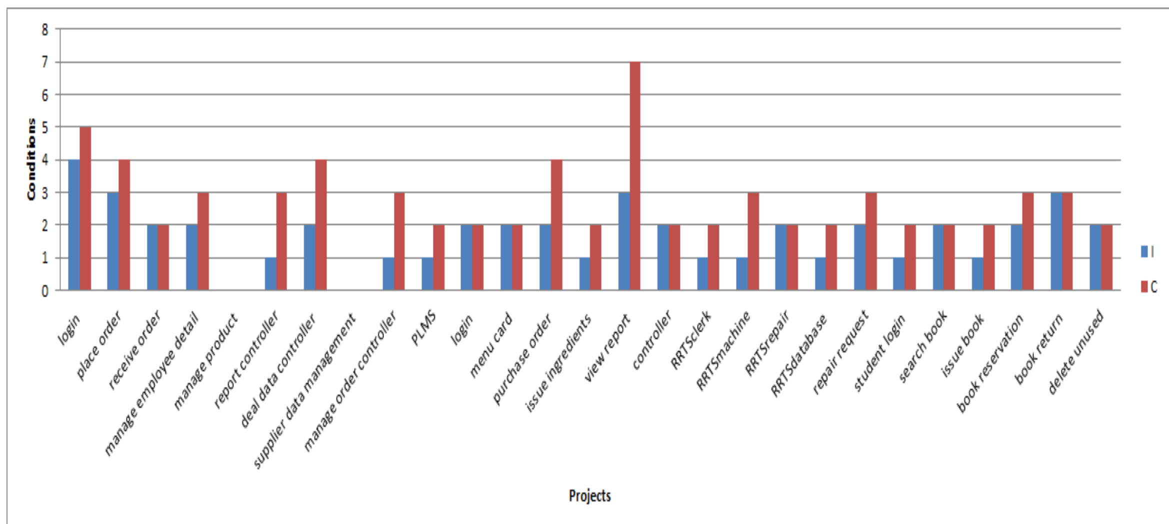


Figure 5.12: Number of independently affected conditions vs. number of simple conditions.

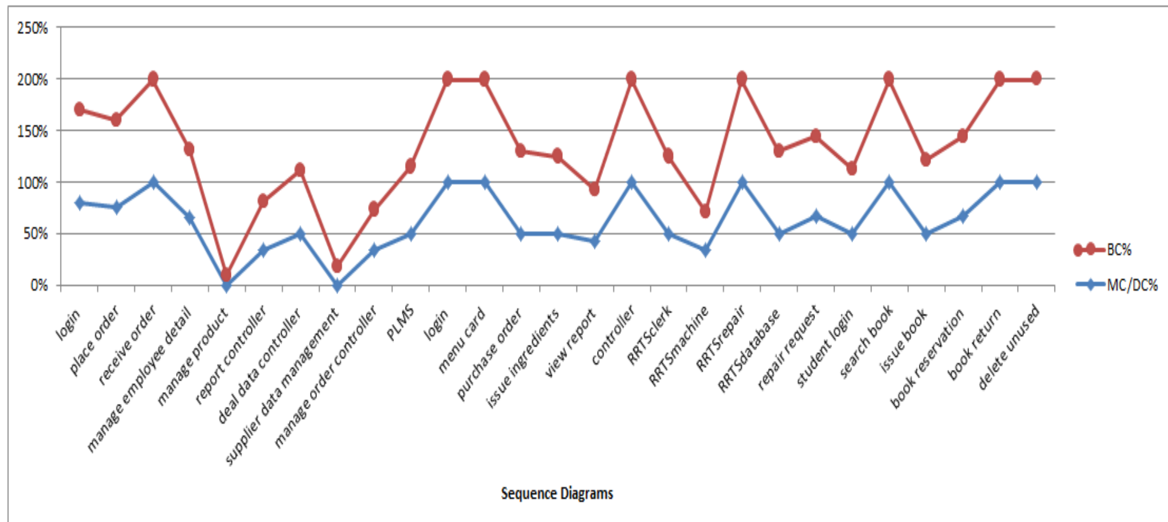


Figure 5.13: Branch Coverage percentage vs. Modified Condition/ Decision Coverage percentage

5.5 Comparison with related work

Abdurazik et al. [49, 55] proposed an approach on the fault revealing capabilities of test sets generated from StateChart and Sequence Diagram. In our proposed work we only considered sequence diagram, in our future work we may consider both the diagram to achieve high coverage.

Vadakkumacheril et al. [19] proposed a technique to generate Java code from XMI representation of sequence diagram. They have used BOUML tool. In our proposed work, we also generate Java code with the help of XMI representation through JAXB tool and ArgoUML tool. Our work is more advanced than the work proposed by the Vadakkumacheril et al. [19], since we generate test cases and measure code coverage for the produced Java code.

Eriksson et al. [2] has developed a novel technique to measure logical coverage design diagrams. They have chosen class diagram, first they have generated the structural code from the diagram and then measured the logical coverage percentage. In our proposed technique, we have taken sequence diagram to generate the code. They have worked in C++ language whereas we have implemented our proposed approach using Java language.

Table 5.1: Characteristics of case studies

Sl. No.	Case Studies	Scenarios	# of actors involved	# of objects involved	# of Synchronous messages	# of Asynchronous messages	# of Loops present	# of of Alternatives involved	# of Conditions involved
1	WBAMS	login	1	3	5	1	0	2	3
2	WBAMS	place order	2	4	5	1	0	0	0
3	WBAMS	receive order	2	4	2	3	0	1	2
4	WBAMS	manage employee detail	2	3	8	3	0	0	0
5	WBAMS	manage product	1	3	5	1	0	0	0
6	WBAMS	report controller	1	3	3	1	0	0	0
7	WBAMS	deal data controller	1	3	3	1	0	0	0
8	WBAMS	supplier data management	1	3	3	1	0	0	0
9	WBAMS	manage order controller	3	7	13	7	0	2	3
10	PLMS	PLMS	2	4	4	2	0	1	2
11	RAS	login	2	4	4	0	0	1	2
12	RAS	menu card	2	6	7	1	0	1	2
13	RAS	purchase order	3	6	14	2	0	2	4
14	RAS	issue ingredients	2	5	5	1	0	1	2
15	RAS	view report	2	3	5	0	1	3	7
16	RRS	controller	2	4	3	1	0	1	2
17	RRS	RRTSclerk	2	4	2	1	0	1	2
18	RRS	RRTSmachine	3	5	6	3	0	2	3
19	RRS	RRTSrepair	3	5	3	2	0	1	2
20	RRS	RRTSdatabase	2	3	2	1	0	1	2
21	RRS	repair request	3	5	4	1	0	2	3
22	LMS	student login	2	4	4	0	0	1	2
23	LMS	search book	1	3	2	0	0	1	2
24	LMS	issue book	1	3	2	0	0	1	2
25	LMS	book reservation	2	3	4	1	0	2	3
26	LMS	book return	2	4	4	1	0	3	3
27	LMS	delete unused	2	3	2	0	0	2	2

Table 5.2: Results analysis of jCUTE

Sl. No.	Case Studies	Sequence Diagram	# of Branches covered	# of Paths covered	Branch Coverage%	Time Taken (ms)
1	WBAMS	login	12	8	90	7863
2	WBAMS	place order	14	11	85	5855
3	WBAMS	receive order	15	13	100	3671
4	WBAMS	manage employee detail	8	6	66	4945
5	WBAMS	manage product	7	4	10	1010
6	WBAMS	report controller	9	6	48.44	4482
7	WBAMS	deal data controller	8	5	62	7687
8	WBAMS	supplier data management	5	2	18.32	3483
9	WBAMS	manage order controller	8	5	40.72	13019
10	PLMS	PLMS	10	7	65	4486
11	RAS	login	15	10	100	6898
12	RAS	menu card	13	11	100	4487
13	RAS	purchase order	13	8	80	7371
14	RAS	issue ingredients	17	8	75	10015
15	RAS	view report	12	6	50	17016
16	RRS	controller	14	11	100	4093
17	RRS	RRTSclerk	15	8	75	5072
18	RRS	RRTSmachine	10	4	38	7077
19	RRS	RRTSrepair	15	11	100	8469
20	RRS	RRTSdatabase	12	9	80	3033
21	RRS	repair request	16	7	77.77	5992
22	LMS	student login	12	6	63	3481
23	LMS	search book	11	10	100	2650
24	LMS	issue book	8	7	72	7615
25	LMS	book reservation	4	7	78	6528
26	LMS	book return	13	12	100	8127
27	LMS	delete unused	17	11	100	4146

Table 5.3: Result analysis of COPECA

Sl. No.	Case Studies	Sequence Diagram	# of Test Cases	Independent conditions (I)	Simple conditions (C)	MC/DC%	Execution Time (ms)
1	WBAMS	login	7	4	5	80%	54
2	WBAMS	place order	3	3	4	75%	38
3	WBAMS	receive order	3	2	2	100%	26
4	WBAMS	manage employee detail	5	2	3	66.66%	33
5	WBAMS	manage product	3	0	0	0%	19
6	WBAMS	report controller	3	1	3	33.33%	63
7	WBAMS	deal data controller	2	2	4	50%	123
8	WBAMS	supplier data management	1	0	0	0%	47
9	WBAMS	manage order controller	4	1	3	33.33%	23
10	PLMS	PLMS	3	1	2	50%	58
11	RAS	login	3	2	2	100%	75
12	RAS	menu card	4	2	2	100%	56
13	RAS	purchase order	5	2	4	50%	61
14	RAS	issue ingredients	3	1	2	50%	136
15	RAS	view report	16	3	7	42.85%	121
16	RRS	controller	3	2	2	100%	16
17	RRS	RRTSclerk	3	1	2	50%	54
18	RRS	RRTSmachine	5	1	3	33.33%	68
19	RRS	RRTSrepair	7	2	2	100%	92
20	RRS	RRTSdatabase	4	1	2	50%	43
21	RRS	repair request	4	2	3	66.66%	26
22	LMS	student login	3	1	2	50%	84
23	LMS	search book	3	2	2	100%	45
24	LMS	issue book	3	1	2	50%	33
25	LMS	book reservation	5	2	3	66.66%	72
26	LMS	book return	5	3	3	100%	121
27	LMS	delete unused	4	2	2	100%	43

5.6 Threats to Validity

- We have only considered, UML Sequence diagram using ArgoUMl. We can't assure for the higher version and other existing tools.
- We have manually change the Java Object code into jCUTE executable .java program.
- We are not able to compute the total execution time of MAUSD, because the execution time of ArgoUML depends on individual software designer. Also, we are not able to record the execution time of JAXB. But, we have computed the total time taken by jCUTE and COPECA.

5.7 Summary

We have developed an approach to measure the MC/DC% using sequence diagram. We explained the proposed technique in detail using block diagram. We have explained each module used in detail. We have explained the flow of execution using an example sequence diagram. We have experimented for five case studies. These case studies include twenty-seven sequence diagrams. On an average, we have achieved 55.29% of MC/DC in 59 ms.

Chapter 6

Conclusions and Future Work

This thesis is mainly focused on automating the MC/DC analysis of object oriented systems. We have developed techniques to measure MC/DC percentage both at design level and coding level.

In this chapter, we summarize the major contribution made in this thesis. Subsequently, we present some suggestions for the extension of proposed techniques.

6.1 Contributions

In this section we present the major contributions. There are two main contributions, Java-HCT, and MAUSD.

6.1.1 Java-HCT

To improve existing concolic testing and obtain high Modified Condition/Decision Coverage (MC/DC), we proposed a novel technique called Java-Hybrid Concolic Testing (Java- HCT). This technique is called hybrid because it is the combination of two testing techniques i.e. Feedback-directed Random Testing and Concolic Testing. We experimented Java- HCT for forty Java programs and found on an average increase of 29.91% and 16.26% MC/DC, when compared to feedback-directed random testing and concolic testing respectively. We have improved MC/DC by $\times 1.62$ and by $\times 1.26$ for feedback-directed random testing and concolic testing respectively

6.1.2 MAUSD

We proposed an automated technique to measure MC/DC percentage for UML Sequence Diagram using concolic testing. We experimented five case studies and worked on twenty seven Sequence diagrams. On an average, for the twenty-seven sequence diagrams, we achieved 55.29% MC/DC.

6.2 Future Work

In this section, we present some of the possible extensions to our proposed techniques.

- We can extend the hybrid concolic testing in distributed environment to enhance the useful test data generation in less amount of time.
- We will develop new code transformation techniques to increase the MC/DC% of Java programs.
- We can make more robust and develop a dynamic version of the present MC/DC analyzer i.e. COPECA.
- We will compute some more coverage metrics for some other UML behavioral diagrams such as Statechart diagram, Activity diagram etc.
- We will merge the test cases generated from various UML diagrams to test complete behavioral aspect of the systems.
- We will develop some efficient test case prioritization techniques for procedural and object-oriented software.

Dissemination

Internationally indexed journals (*Web of Science, SCI, Scopus, etc.*)[1]

1. **Arpita Dutta**, Sangharatna Godboley, and Durga Prasad Mohapatra, HiRSA: Computing Hit Ratio for SOA applications through Tcases, International Journal of Computational Systems Engineering (IJCSYSE), Inderscience. 2017. (Accepted)

2. Sangharatna Godboley, **Arpita Dutta**, Durga Prasad Mohapatra, and Rajib Mall. GECOJAP: A novel source-code preprocessing technique to improve code coverage. Computer Standards & Interfaces, Elsevier. 2017.(Accepted) (**SCI**)

3. Sangharatna Godboley, **Arpita Dutta**, Durga Prasad Mohapatra, and Rajib Mall. J^3 Model: A novel framework for Improved Modified Condition/Decision Coverage Analysis. Computer Standards & Interfaces, Elsevier, Volume 50, pages 1-17, 2016. (**SCI**)

4. Sangharatna Godboley, Subhrakanta Panda, **Arpita Dutta**, and Durga Prasad Mohapatra. An Automated Analysis of the Branch Coverage and Energy Consumption Using Concolic Testing, Arabian Journal for Science and Engineering. 2016. DOI:10.1007/s13369-016-2284-2. (**SCI**)

5. Sangharatna Godboley, **Arpita Dutta**, Avijit Das, Durga Prasad Mohapatra, and Rajib Mall.Making a concolic tester achieve increased MC/DC, Innovations Systems and Software Engineering, 12(4), 319-332, 2016. DOI:10.1007/s11334-016-0284-8.

6. Sangharatna Godboley, **Arpita Dutta**, and Durga Prasad Mohapatra. Reduced Energy Consumption for MC/DC Testing, International Journal of Business Information Systems (IJBIS), Inderscience. (In press)

7. Sangharatna Godboley, **Arpita Dutta**, Durga Prasad Mohapatra , and Rajib Mall. Green J3 Model: A novel approach to measure Energy Consumption of Modified Condition/ Decision Coverage using Concolic Testing, CSI Transactions on ICT, pages 1-17, Springer, 2016. DOI 10.1007/s40012-017-0157-9.

[1] Articles already published, in press, or formally accepted for publication.

8. Sangharatna Godboley, **Arpita Dutta**, and Durga Prasad Mohapatra. Green-DRCT:Measuring Energy Consumption of an enhanced Branch Coverage and Modified Condition/Decision Coverage Technique, IGI Global. (Accepted)

International Conferences

1. **Arpita Dutta**, Sangharatna Godboley and Durga Prasad Mohapatra, COLT: Extending CONCOLIC Testing to measure LCSAJ Coverage, 30th IEEE TENCON-16, Singapore, pp.373-378, 2016. DOI: 10.1109/TENCON.2016.7848024.

2. **Arpita Dutta**, Sangharatna Godboley and Durga Prasad Mohapatra, Measuring Branch Coverage for the SOA based Application using Concolic Testing, International Conference on Advances in Computing and Data Sciences (ICACDS-16), Springer, Krishna Engineering College, Ghaziabad (UP) India, 2016. (Presented)

3. **Arpita Dutta**, Sangharatna Godboley and Durga Prasad Mohapatra, Measuring Hit Ratio metric for SOA based Application using Black-box testing, 3rd International Conference on Computational Intelligence in Data Mining (ICCIDM-16), Springer, KIIT, Bhubaneswar, 2016. (Presented)

4. Durga Prasad Mohapatra, Sangharatna Godboley and **Arpita Dutta**, Measuring Hit ratio of Software Systems using UML Sequence Diagram, 58th Technical Annual Session by The Institute of Engineers (India), 2016. (**Received SANDEEP MOHAPATRA MEMORIAL MEDAL**)

5. Sangharatna Godboley, **Arpita Dutta**, Avijit Das, and Durga Prasad Mohapatra, Measuring MC/DC at Design Phase using UML Sequence Diagram and Concolic Testing, 13th International IEEE India Conference INDICON, IISC, Bengaluru, India, pp. 1-6, 2016. DOI: 10.1109/INDICON.2016.7839079.

6. Sangharatna Godboley, **Arpita Dutta**, and Durga Prasad Mohapatra. Java-HCT: An approach to increase MC/DC using Hybrid Concolic Testing for Java programs. In proceedings of the 15th Federated Conference on Computer Science and Information Systems (36th IEEE Software Engineering Workshop), Gdansk University of Technology, Gdansk, Poland,Annals of Computer Science and Information Systems, Volume 8, pages 1709-1713, 2016.

7. Sangharatna Godboley, **Arpita Dutta**, Bhagyashree Besra and Durga Prasad Mohapatra, Green-JEXJ: A new tool to measure energy consumption of improved concolic testing, 2015 International Conference on Green Computing and Internet of Things (ICGCIoT), Noida, pp. 36-41. 2015. DOI: 10.1109/ICGCIoT.2015.7380424. (**Best Paper Award**)

Article under preparation [2]

1. **Arpita Dutta**, Sangharatna Godboley, and Durga Prasad Mohapatra, Driving Tcases to compute Hit Ratio for UML 1.X Sequence Diagram, International Journal of System Assurance Engineering and Management, Springer **(Scopus)**. (Major Revision)

2. Sangharatna Godboley, **Arpita Dutta**, Durga Prasad Mohapatra, and Rajib Mall, DRCT: A New Transformation Technique to Achieve Increase in MC/DC, IET Software (**SCI**). (Under Revision)

3. Sangharatna Godboley, **Arpita Dutta**, Durga Prasad Mohapatra, Avijit Das, and Rajib Mall, Scaling Modified Condition / Decision Coverage using Distributed Concolic Testing for Java programs, Computer Standards & Interfaces, Elsevier (**SCI**). (Under Review)

4. Sangharatna Godboley, **Arpita Dutta**, Devang Swami, Durga Prasad Mohapatra, Towards Green Software Testing: A promising approach to compute CO2 Emission and Cost Analysis of Software Testing Tools, Sustainable Computing, Elsevier **(SCIE)**. (Under Revision)

[2] Articles under review, communicated, or to be communicated.

Bibliography

[1] Son, H.S., Park, Y.B. and Kim, R.Y.C., 2016. MCCFG: an MOF-based multiple condition control flow graph for automatic test case generation. Cluster Computing, Springer, pp.1-10.

[2] Eriksson, A. and Lindström, B., 2016. UML Associations: Reducing the gap in test coverage between model and code. In proceedings of 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), February 19-21, Rome, Italy, Vol. (1), pp. 589-599. SciTePress.

[3] Dhok, M., Ramanathan, M.K. and Sinha, N., 2016, May. Type-aware concolic testing of JavaScript programs. In proceedings of the 38th International Conference on Software Engineering,ACM. pp. 168-179.

[4] Han, D., Xing, J., Yang, Q., Wang, H. and Zhang, X., 2016. Formal Sequence: Extending UML Sequence Diagram for Behavior Description and Formal Verification. In proceedings of IEEE 40th Annual Computer Software and Applications Conference (COMPSAC), Vol. (2), pp. 474-481.

[5] Godboley, S., Mohapatra, D.P., Das, A., and Mall, R., 2016. An improved distributed concolic testing approach. *Software: Practice and Experience*,Wiley, DOI: 10.1002/spe.2405.

[6] Godboley, S., Dutta, A., Mohapatra, D.P., Das, A., and Mall, R., 2016. Making A Concolic Tester Achieve Increased MC/DC, *Innovations in systems and software engineering*, Springer, DOI:10.1007/s11334-016-0284-8.

[7] Godboley, S., Dutta, A., Mohapatra, D.P., and Mall, R., 2016. J3 Model: A novel framework for improved Modified Condition/Decision Coverage analysis,*Computer Standards and Interfaces*, Elsevier,Vol.(50), pp. 1-17, DOI: 10.1016/j.csi.2016.09.006.

[8] Chen, J., Kuo, F.C., Chen, T.Y., Towey, D., Su, C. and Huang, R., 2016. A Similarity Metric for the Inputs of OO Programs and Its Application in Adaptive Random Testing. IEEE Transactions on Reliability.

[9] Godboley, S., Panda, S., Dutta, A. and Mohapatra, D.P., 2016. An Automated Analysis of the Branch Coverage and Energy Consumption Using Concolic Testing. Arabian Journal for Science and Engineering, pp.1-19.

[10] Sen, K., Necula, G., Gong, L. and Choi, W., 2015, August. MultiSE: Multi-path symbolic execution using value summaries. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM., pp. 842-853.

[11] Noh, S. and Shortle, J.F., 2015. Sensitivity analysis of event sequence diagrams for aircraft accident scenarios. In proceedings of 34th Conference on Digital Avionics Systems (DASC), IEEE/AIAA, pp. 3E2-1.

[12] Li, Y., Li, You., Wang, L., and Chen, G., 2014. *Automatic XACML requests generation for testing access control policies*. In proceedings of SEKE-14, Hyatt Regency, Vancouver, Canada, pp. 217–222.

[13] Saito, H., Takada, S., Tanno, H., and Oinuma, M., 2014. *Test Data Generation for Web Applications: A Constraint and Knowledge-based Approach*. In proceedings of SEKE-14, Hyatt Regency, Vancouver, Canada. pp. 110–114.

[14] Mall, R., 2014. Fundamentals of software engineering. PHI Learning Pvt. Ltd.

[15] Das, A., and Mall, R., 2013. Automatic Generation of MC/DC Test Data. *International Journal of Software Engineering, Acta Press*, 2(1).

[16] Godboley, S., and Mohapatra, D.P., Time Analysis of Evaluating Coverage Percentage for C Program using Advanced Program Code Transformer *In proceedings 7th CSI International Conference on Software Engineering* Vol. 11, Chennai, India, pp. 91-97.

[17] Godboley, S., Prashanth, G.S., Mohapatra, D.P., and Majhi, B., 2013. Enhanced modified condition/decision coverage using exclusive-nor code transformer. *In proceedings of International Multi-Conference on Automation, Computing, Communication, Control and Compressed Sensing (iMac4s)*, IEEE, Kottayam, Kerala, India, pp. 524-531.

[18] Godboley, S., Prashanth, G.S., Mohapatra, D.P., and Majhi, B., Increase in Modified Condition/Decision Coverage using program code transformer. *In proceedings of IEEE 3rd International Advance Computing Conference (IACC)*, Ghaziabad, Indiapp, 1400-1407.

[19] Vadakkumcheril, T., Mythily, M., and Valarmathi, ML., 2013. A Simple Implementation of UML Sequence Diagram to Java Code Generation through XMI Representation. *International Journal of Emerging Technology and Advanced Engineering*, 2 (12).

[20] Tillman, N., Jamrozik, K., Fraser, G.,and Halleux, J., 2013. Generating test suites with augmented dynamic symbolic execution. *In proceedings of International Conference on Tests and Proofs*, Springer, Berlin, pp. 152-167.

[21] Kim, M., Kim, Y., and Rothermel, G., 2012. A Scalable Distributed Concolic Testing Approach: An Empirical Evaluation. *In proceedings of Fifth IEEE International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, Downtown Montreal Montreal, QC, Canada pp. 340-349.

[22] Das, A., 2012. *Automatic Generation of MC/DC Test Data*. Master Thesis, Computer Science & Engineering, Indian Institute of Technology, Kharagpur, India.

[23] Chartchai, D., 2011. Generation of Software Test Data from the Design Specification Using Heuristic Techniques. *Thesis (Ph.D.) Department of Computing University of Bradford.*

[24] Claessen, K., and Hughes, J., 2011. QuickCheck: a lightweight tool for random testing of Haskell programs. Acm sigplan notices, 46(4), pp. 53-64.

[25] RTCA Inc. 2011. DO-178C: Software Considerations in Airborne Systems and Equipment Certification, Washington, D.C.

[26] Kähkönen, K., Launiainen, T., Saarikivi, O., Kauttio, J., Heljanko, K., and Niemelä, I., 2011. LCT: An open source concolic testing tool for Java programs. In proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE), pp. 75-80.

[27] Nayak, A., and Samanta, D., 2010. Automatic Test Data Synthesis using UML Sequence Diagrams, *Journal of Object Technology*, 9(2), pp. 115–144.

[28] Swain, S.K., Mohapatra, D.P. and Mall, R., 2010. Test case generation based on use case and sequence diagram. International Journal of Software Engineering, 3(2), pp.21-52.

[29] Bokil, P., Darke, P., Shrotri, U., and Venkatesh, R., 2009. Automatic Test Data Generation for C Programs. *In proceedings of 3rd IEEE International Conference on Secure Software Integration and Reliability Improvement*, Washington, DC, USA, pp. 359-368.

[30] Jayaraman, K., Harvison, D., Ganesh, V., and Kiezun, A., 2009. jFuzz: A concolic whitebox fuzzer for java, *In proceedings of NASA Formal Methods, Springer*, pp. 121-125.

[31] Awedikian, Z., Ayari, K., and Antoniol, G., 2009. MC/DC automatic test input data generation. *In proceedings of Genetic and Evolutionary Computation Conference (GECCO)*, New York, USA, pp. 1657-1664.

[32] McMinn, P., Binkley, D. and Harman, M., 2009. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(3), pages 11.

[33] Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L. and Engler, D.R., 2008. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, pp. 322-335.

[34] Boonstoppel, P., Cadar, C. and Engler, D., 2008. RWset: Attacking path explosion in constraint-based test generation. *In International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, Berlin, Heidelberg, pp. 351-366.

[35] Burnim, J., and Sen, K., 2008. Heuristics for scalable dynamic test generation. *In proceedings of Automated Software Engineering (ASE)*, pages 443-446, Washington, D.C., USA.

[36] Csallner, C., Smaragdakis, Y., and Xie, T., 2008. Dsd-crasher: A hybrid analysis tool for bug finding, *ACM Transaction on Software Engineering and Methodology*, 17(2), pp. 8:1–8:37.

[37] Majumdar, R., and Sen, K., 2007. Hybrid concolic testing, In proceedings of 29th International Conference on *Software Engineering 2007*, pp. 416–426.

[38] Pacheco, C., Lahiri, S. K., Ernst, M. D., and Ball, T., 2007. Randoop: Feedback-directed random test generation, In proceedings of 29th International Conference on Software Engineering, 2007. ICSE 2007.

[39] Visser, W., Păsăreanu, C. S., and Pelánek, R., Test input generation for java containers using state matching, *In proceedings of the 2006 International Symposium on Software Testing and Analysis*, ser. ISSTA, New York, NY, USA: ACM, pp. 37–48.

[40] Sen, K., and Agha, G., 2006. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. *In International Conference on Computer Aided Verification*, Springer Berlin Heidelberg, pp. 419-423.

[41] Pacheco, C., Lahiri, S. K., Ernst, M. D., and Ball, T., 2006. Feedback-directed random test generation, In *Technical Report MSR-TR-2006-125, Microsoft Research,* pp. 75–84.

[42] Lei, Y., and Andrews, J. H., 2005. Minimization of randomized unit test cases. In proceedings of the 16th IEEE International Symposium on *Software Reliability Engineering, 2005. ISSRE 2005,* pp. 10 pp.–276.

[43] Pacheco, C. and Ernst, M. D., 2005. Eclat: Automatic Generation and Classification of Test Inputs.*In proceedings of ECOOP 2005 - Object-Oriented Programming: 19th European Conference, Glasgow, UK, July 25-29, 2005.* Heidelberg: Springer Berlin, pp. 504–527.

[44] Godefroid, P., Klarlund, N., and Sen, K., 2005. DART: Directed automated random testing. *In proceedings of Programming Language Design and Implementation (PLDI)*, New York, USA, pp. 75-84.

[45] Sen, K., Marinov, D., and Agha, G., 2005. CUTE: A concolic unit testing engine for C. *In proceedings of European Software Engineering Conference / Foundations of Software Engineering (ESEC/FSE)*, Lisbon, Portugal, pp. 263-272.

[46] Rountev, A., Kagan, S., and Sawin, J., 2005. Coverage criteria for testing of object interactions in sequence diagrams, *In Fundamental Approaches to Software Engineering, LNCS 3442*, pap. 282–297.

[47] Lei, Y., and Andrews, J. H., 2005. Minimization of randomized unit test cases. In proceedings of 16th IEEE International Symposium on Software Reliability Engineering, 2005. ISSRE 2005, pp. 10.

[48] Csallner, C., and Smaragdakis, Y., 2004. Jcrasher: an automatic robustness tester for java, *Software: Practice and Experience*, 34(11), pp. 1025–1050.

[49] Abdurazik, A., Offutt, J., and Baldini, A., 2004. A controlled experimental evaluation of test cases generated from UML diagrams. *Technical Report, ISE-TR-04-03. George Mason University*.

[50] Harman, M. , Hu, L., Hierons, R., Wegener, J., Sthamer, H., Baresel, A., Roper, M., 2004. Testability Transformation. *IEEE Transactions on Software Engineering* pp. 3-16.

[51] Cadar, C., Dunbar, D. and Engler, D.R., 2004. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *In proceedings of Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, pp. 209-224.

[52] Fraikin, F. and Leonhardt, T., 2002. SeDiTeC-testing based on sequence diagrams. In the proceedings of the 17th IEEE international conference on Automated Software Engineering (ASE) 2002. Washington, DC, USA, pp. 261-266.

[53] Kelly, H. J., Dan, V. S., John, C. J., and Leanna, R.K., 2001. A practical tutorial on modified condition/decision coverage, NASA Langley Technical Report .

[54] Ho, P.H., Shiple, T., Harer, K., Kukula, J., Damiano, R., Bertacco, V., Taylor, J. and Long, J., 2000. November. Smart simulation using collaborative formal and simulation engines. In proceedings of the 2000 IEEE/ACM international conference on Computer-aided design, pp. 120-126.

[55] Abdurazik, A., Offutt, J., 2000. Using UML collaboration diagrams for static checking and test generation. *In proceedings of International Conference on the Unified Modeling Language.* pp. 383-395.

[56] Bush, W.R., Pincus, J.D., and Sielaff, D.J., 2000. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7), pp. 775-802.

[57] Kuhn R., 1999. Fault classes and error detection capability of specification-based testing. *In proceedings of ACM Transactions on Software Engineering Methodology*, 8(4), New York, USA, pp. 411-424.

[58] Ganai, M.K. and Tech, B., 1998. Enhancing simulation with BDDs and ATPG (Master's thesis, University of Texas at Austin).

[59] Ferguson, R. and Korel, B., 1996. The chaining approach for software test data generation. ACM Transactions on Software Engineering and Methodology (TOSEM), 5(1), pp.63-86.

[60] RTCA, Inc., 1992. RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Certification, Washington, D.C.

[61] Bird, D.L. and Munoz, C.U., 1983. Automatic generation of random self-checking test cases. IBM systems journal, 22(3), pp.229-245.

[62] Clarke, L.A., 1976. A system to generate test data and symbolically execute programs. IEEE Transactions on software engineering, (3), pp.215-222.

[63] King, J.C., 1976. Symbolic execution and program testing. Communications of the ACM, 19(7), pp.385-394.