Improved Modified Condition/ Decision Coverage using Code Transformation Techniques

Sangharatna Godboley

(Roll no.: 211CS3293)

Department of Computer Science and Engineering

National Institute of Technology, Rourkela

Odisha - 769 008, India

Improved Modified Condition/ Decision Coverage using Code Transformation Techniques

Thesis submitted in partial fulfillment
of the requirements for the degree
of

Master of Technology

by

Sangharatna Godboley

Roll no-211CS3293

under the guidance of

Prof. Durga Prasad Mohapatra and Prof. Banshidhar Majhi

Department of Computer Science and Engineering
National Institute of Technology, Rourkela
Odisha,769 008, India

May 2013

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Odisha, India.

June 2, 2013

Certificate

This is to certify that the work in the thesis entitled ***"Improved Modified Condition/ Decision Coverage using Code Transformation Techniques"*** by ***Sangharatna Godboley*** is a record of an original research work carried out under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of Master of Technology in Computer Science and Engineering. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Prof. Banshidhar Majhi	**Prof. Durga Prasad Mohapatra**
Professor	**Associate Professor**

Acknowledgement

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 supervisors, Prof. Durga Prasad Mohaptra and Prof. Banshidhar Majhi for providing me with a platform to work on challenging areas of Modified Condition/ Decision Coverage and Concolic Testing. Their profound insights and attention to details have been true inspirations to my research.

I am very much indebted to Prof. Shantanu Kumar Rath, Prof. Ashok Kumar Turuk, Prof. Pabitra Mohan Khilar, and Prof. Bibhudatta Sahoo, for their encouragement and insightful comments at different stages of 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.

I am also indebited to Mr. Avijit Das of Advanced Systems Laboratory, Defence R & D Organisation, Hyderabad and Mr. Prasad Bokil of TRDDC, Pune for suggestions and guidance of my thesis work.

Most importantly, none of this would have been possible without the love of Mr. J. R. Godboley(Papa), Mrs. Rama Devi Godboley(Mumy), Er. Vivekratna Godboley(Brother), Prof. Buddharatna Godboley(Brother), and Er. Rajaratna Godboley(Brother). 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 appreciate the company of Mr. Yogendra Soni, Mr. Subhrakanta Panda, Mr. Yeresime Suresh and Mr. Vijay Sarthi for their unique ideas and help whenever required.

I would like to thank all my friends and lab-mates(Sanjaya Panda, Srinivasulu Dasari, Dinesh Kottha Reddy, Adepu Sridhar, Pratik Agarwal, Suchitra Kumari Mishra, Alina Mishra, and Swagatika Swain) for their encouragement and understanding. Their help can never be penned with words.

Sangharatna Godboley

Abstract

Modified Condition / Decision Coverage (MC / DC) is a white box testing criteria aiming to prove that all conditions involved in a predicate can influence the predicate value in the desired way. In regulated domains such as aerospace and safety critical domains, software quality assurance is subjected to strict regulations such as the DO-178B standard. Though MC/DC is a standard coverage criterion, existing automated test data generation approaches like CONCOLIC testing do not support MC/DC. To address this issue we present an automated approach to generate test data that helps to achieve an increase in MC/DC coverage of a program under test. We use code transformation techniques for transforming program. This transformed program is inserted into the CREST TOOL. It drives CREST TOOL to generate test suite and increase the MC/DC coverage. Our technique helps to achieve a significant increase in MC/DC coverage as compared to traditional CONCOLIC testings.

Our experimental results show that the proposed approach helps to achieve on the average approximately 20.194 % for Program Code Transformer(PCT) and 25.447 % for Exclusive-Nor Code Transformer. The average time taken for seventeen programs is 6.89950 seconds.

Keywords: CONCOLIC testing, Code transformation techniques, MC/DC, Coverage Analyser.

Contents

1 Introduction **2**

1.1 Software Testing . . . 2

1.1.1 Software Testing Goals . . . 2

1.1.2 Software Testing Life Cycle . . . 3

1.1.3 Software Testing Techniques . . . 3

1.1.4 Software Testing Strategies . . . 4

1.2 Problem Statement . . . 6

1.2.1 Automated Testing . . . 6

1.2.2 The objective of our approach . . . 6

1.3 Organization of the Thesis . . . 7

2 Basic Concepts **9**

2.1 Some Relevent Definitions . . . 9

2.2 Modified Condition/ Decision Coverage . . . 10

2.3 Determination of Predicates . . . 12

2.4 CONCOLIC Testing . . . 13

2.4.1 Definition . . . 14

2.4.2 Process . . . 15

2.5 Summary . . . 16

3 Review of Related Work **17**

3.1 Automated testing for branch coverage . . . 17

3.2 MC / DC Automatic Testing . . . 18

3.3 Other related works . . . 19

3.4 Summary . . . 20

4 Program Code Transformer Technique 21
4.1 Formal Definition 21
4.2 Our first proposed approach MC/DC Tester-I [MT-I] 22
4.3 Program Code Transformer 23
4.3.1 Description of Algorithm 1 24
4.3.2 Description of Algorithm 2 26
4.3.3 Description of Algorithm 3 27
4.3.4 Example of PCT 28
4.3.5 Complexity of PCT 28
4.4 CONCOLIC Testing 28
4.5 MC/DC Coverage Analyser 29
4.5.1 Description of Algorithm 4 30
4.6 Experimental Study 30
4.6.1 PCT details 30
4.6.2 CREST details 31
4.6.3 CA details 38
4.6.4 Experimental Reqiurements 38
4.6.5 Results 40
4.6.6 Time constraints 41
4.6.7 Comparison Between BCT and PCT 41
4.7 Limitation of PCT 43
4.8 Conclusion 43

5 Exclusive-Nor Code Transformer Technique 44
5.1 Our Proposed Approach MC/DC Tester-II [MT-II] 44
5.2 X-Nor Code Transformer (X-NCT) 45
5.2.1 Description of Algorithm 1 46
5.2.2 Description of Algorithm 2 47
5.3 Exclusive-NOR operation 48
5.4 Example for X-NCT 48
5.5 Complexity for X-NCT 51
5.6 CONCOLIC Tester 51
5.7 MC/DC Coverage Analyser 51
5.8 Experimental Study 51
5.8.1 X-NCT details 51

5.8.2 Experimental Reqiurements . . . 52
5.8.3 Results . . . 53
5.8.4 Comparison Between XCT and X-NCT . . . 54
5.8.5 Analysis of Result . . . 54
5.9 Conclusion . . . 57

6 Conclusions and Future Work **58**
6.1 Contributions . . . 58
6.1.1 Program Code Transformer . . . 58
6.1.2 Exclusive-Nor Code Transformer . . . 59
6.1.3 Coverage Analyser . . . 59
6.2 Future Work . . . 59

BIBLIOGRAPHY **61**

List of Figures

1.1 An example program for coverage criteria 5

2.1 Schematic representation for different gates 11
2.2 An example program to explain CONCOLIC testing 14

4.1 Schematic representation of our first approach [MT-I] 23
4.2 An example showing concept of PCT 27
4.3 Transformed form of Fig. 4.2 28
4.4 Transformed program from PCT 31
4.5 An Example program from chapter 4 33
4.6 Compilation of original program 33
4.7 Run execution of original program 34
4.8 Generated Test data in input file for original program 34
4.9 Automatically generated coverage file for original program 35
4.10 Transformed program of original program 35
4.11 Compilation of transformed program 36
4.12 Run execution of transformed program 36
4.13 Generated Test data in input file for transformed program 37
4.14 Automatically generated coverage file for transformed program 37
4.15 Screenshot of Coverage analyser 38
4.16 Time Constarint Graph 42
4.17 Comparison between transformation techniques BCT and PCT 43

5.1 Schematic representation of our second approach [MT-II] 45
5.2 An example for X-NOR operation 49
5.3 Transformed program for Fig. 5.2 49
5.4 Transformed program from X-NCT 52

5.5 Comparison between transformation techniques XCT and X-NCT 55
5.6 Analysis of all example program for Evaluated MC/DC coverage percentage 56
5.7 Increase in MC/DC percentage comaparision analysis for all example programs 57
5.8 Comparison analysis for PCT and X-NCT 57

List of Tables

2.1 Truth table for two variables 10
2.2 Truth table for three variables 10
2.3 MC/DC result for four variables 12

4.1 Summary of Characteristics of the program under test for PCT 39
4.2 Coverage Calculated by Coverage Analyser for PCT 40
4.3 Time Constraint 41
4.4 Coverage percentage for BCT and PCT technique 42

5.1 Truth table for X-NOR operation 48
5.2 Truth table for first variable (a) after applying X-NOR operations 50
5.3 Truth table for second variable (b) after applying X-NOR operations 50
5.4 Truth table for third variable (c) after applying X-NOR operations 50
5.5 Summary of Characteristics of the program under test 53
5.6 Coverage Calculated by Coverage Analyser 54
5.7 Coverage percentage for XCT and X-NCT technique 55

Chapter 1

Introduction

Software engineering proposes systematic and cost-effective methods to software development process [1]. These methods have resulted from innovations as well as lessons learnt from past mistakes. Software engineering as the engineering approach to develop software. Software is usually subject to several types and cycles of verification and test. In the early days of software development, software testing was considered only a debugging process for removing errors after the development of software.

1.1 Software Testing

Software Testing is a process that detects important bugs with the objectives of having better quality software. This is the way to increase reliability of software projects [2]. The technique software testing is responsible for achieving good quality software and high software dependability. Software testing consists of the steps of execution of a system under some conditions and compares with expected results [3]. The conditions should have both normal and abnormal conditions to determine any failure under unexpected conditions.

1.1.1 Software Testing Goals

The main goals of software testing are divided into three categories and several subcategories as follows:

1. *Immediate Goal* :
    - Bug Discovery,
    - Bug Prevention

2. *Long-term Goals*:
    - Reliability,
    - quality,
    - customer,
    - satisfaction,
    - risk management
3. *Post Implementation Goals*:
    - Reduced maintenance cost,
    - Improved testing process

1.1.2 Software Testing Life Cycle

The testing process divided into a well-defined sequence of steps is termed as a software testing life cycle (STLC). The STLC consists the following phases:

- Test planning,
- Test design,
- Test execution
- Test review/post execution.

1.1.3 Software Testing Techniques

In software world it has been noticed that 100% efficient software testing is not possible. But an effective testing can solve this problem but to follow the effective testing is very difficult. The method to determine effective test case is known as Software Testing Technique. Two objectives are making the effective test cases that are detection of numbers of bugs and coverage of testing area. The different levels of testing Unit testing, Integration testing, Function Testing, System testing, and Acceptance testing. The detailed testing stages are followed:

1. **Unit Testing:** Each System Component of whole software is individually tested for all functionality and its interfaces.

2. **Integration Testing:** Process of mixing and testing multiple building blocks together. The individual tested component, when mixed with other components, is untested for interfaces. Therefore it may have bugs in integrated workspace. So, the purpose of this testing is to uncover this bug.

3. **Function Testing:** To measure systems functional component quality is the main purpose of functional testing. This is to expand the bugs related to problems between system behavior and specifications.

4. **System Testing:** Its objective is not to test particular function, but it tests the system on various platforms where bugs exist.

5. **Acceptance Testing:** This technique used by customer after software developed. Compares the process of the final status of project and agreement of acceptance criteria performed by the customer.

1.1.4 Software Testing Strategies

Testing strategies are mainly divided into two categories:

1. **Black Box Testing:** The structure of software is not considered only the functional requirements of the module are taken under consideration. In this the software system act as a black box taking input test data and and giving output results.

2. **White Box Testing:** As everything is transparent in glass like that in this software it visible in all aspects it is called as glass box testing. Structure, design and code of software should be studied for this type of testing. Also it is called as development or structural testing [3].

   There are several white box coverage criteria [4] [5]. Let us take a sample program as shown in Fig. 1.1.

   - **Statement Coverage**: In these coverage criteria each and every statement of a module is executed once, we can detect every bug. For example: If we want to cover the each line so we need to follow all test cases. Case 1- x=y=m, where m is any number. Case 2- x=m, y=m', where m and m' are different numbers. If Case 1 fails then all parts of the code never execute. Now consider the Case 2, here loop execute Case 3- $x < y$ and Case 4- $x > y$ will execute. This criterion

```
 1 #include <stdio.h>
 2 #include <iostream.h>
 3 int main(int argc, char *argv[])
 4 {
 5     int x,y,i;
 6     cin>>x;
 7     cin>>y;
 8     for(i=0;x!=y;i++)
 9      {
10        if (x<y)
11          y=y-x;
12        else
13          x=x-y;
14      }
15     cout<<x;
16     cout<<y;
17
18     return 0;
19 }
20
21
```

Figure 1.1: An example program for coverage criteria

is very poor criteria because Case 3 and Case 4 are sufficient for all statements in code. But, if both Case 3 and Case 4 will execute, so Case 1 never execute. Therefore it is a poor criteria.

- **Branch Coverage**: Each decision node traversed at least once. The possible outcomes are either TRUE or FALSE. For a last example Test cases are designed as: Test Case 1- x=y, Case- 2 $x \neq y$, Case3- $x > y$, Case4- $x < y$.
- **Modified Condition / Decision Coverage**: It enhances the condition coverage and decision coverage criteria by showing that each condition in a decision independently affects the result of the decision. For example, for the expression (A ||B), test cases (TF), (FT), and (FF) provide MC/DC.
- **Multiple Condition Coverage**: This is the strongest criteria. Here all possible outcomes of each condition in decision taking under consideration. It requires sufficient test cases such that all points of entry invoked at least once. Ex. If an AND results FALSE, no need to evaluate further steps, and if an OR result TRUE so again no need to evaluate further steps. Possible test cases: Case 1- A=TRUE, B=TRUE, Case 2- A=TRUE, B=FALSE, Case 3- A=FALSE, B=TRUE, Case 4- A=FALSE, B=FALSE.

1.2 Problem Statement

This section shows the overview of our work. First, automated testing is discussed and then the objective of our proposed approach is discussed.

1.2.1 Automated Testing

This achieved by using an automated test software or tool. Testing activity saved about 40% to 50% of the overall software development effort. Automated testing [6] appears as a promising technique to reduce test time and effort. This is used in regression testing, performance testing, load testing, network testing and security testing. This tool concept speeds up the test cycle as they can overcome the faster rate of the manual testing process. This can be done in two ways: first create scripts with all the required test cases embedded in them. Second design software that will automatically generate test cases and run them on the program or the system to be tested. This can be very complex and difficult to develop but once if it designed then we can save huge amount of time, cost, and effort. Therefore, it is possible to use these techniques to invoke the necessary information for test cases.

1.2.2 The objective of our approach

The main aim is to develop an automated approach to generate test cases that can achieve MC/DC coverage [2]. To reach our aim, we propose the concept. The CONCOLIC testing is the combination of concrete and symbolic testing and it was originally designed to achieve branch coverage [7].

In our approach, we have used CREST TOOL which is a CONCOLIC tester [8]. In our work we present the code transformer in which we insert program code under test and get the transformed program as output. Transformed program is nothing but additional nested if-else that's having true and false branches for each decision with the original program. This transformation is used to get an increase in MC/DC test data. This additional branch does not affect the original program. This transformed program is now inserted to CREST TOOL and we get MC/DC test suite which consists of test data. The tester also generates concrete input values for the transformed code to achieve an increase in MC/DC for the original program.

Coverage analyzer proposed to calculate the coverage percentage. We need to provide MC/DC test data for each and every clause and need to provide the original program. At last we get the coverage percentage. In our observations, when we are inserting our original program with CONCOLIC tester, some MC/DC test data are generated. Using these values and our program, we calculate a coverage percentage. Secondly by adding code transformer, we insert transformed program to CONCOLIC tester and get MC/DC test data using these values, again we calculate the coverage percentage. Now we get two different coverage percentages, but we can observe that the second percentage is improved by some value. Hence, we achieved an increase in MC/DC by using our approach code transformer.

1.3 Organization of the Thesis

The rest of the thesis is organized into chapters as follows:

Chapter 2 contains the basic concepts used in the rest of the thesis. The chapter contains the definitions of condition, decision, logic gates, group of conditions. We describe the determination of predicates. Then, we present some basic concepts of modified condition/ decision coverage with an example. Finally, we discuss the concepts of CONCOLIC testing with an example.

Chapter 3 provides a brief review of the related work relevant to our contribution. We discuss the work related to automated test data generation, modified condition/descision coverage, and CONCOLIC testing.

Chapter 4 presents the technique for Program Code Transformer(PCT). We introduce some formal definitions followed by concepts and algorithms for our program code transformer approach. We proposed the algorithms for Pogram Code Transformer, Quine-McMulsky Method, Generate Nested If-Else Conditions, and Coverage Analyzer. We discuss experimental study and comparision with related work.

Chapter 5 deals with the technique for Exclusive-NOR Code Transformer. We proposed the algorithm for Exclusive-NOR Code Transformer and Generate Nested If-Else Conditions. We discuss experimentas details, requirements, analysis of results, and comparision with related work.

Chapter 6 concludes the thesis with a summary of our contributions. We also briefly discuss the possible future extensions to our work.

Chapter 2

Basic Concepts

In this chapter, first we discuss some relevent definitions which will be used in our approach. Then, we discuss the concepts regarding MC/DC coverage, followed by a technique for Boolean derivative method and CONCOLIC testing approach.

2.1 Some Relevent Definitions

Below, we discuss some relevent definitions that will be used in our approach.

1. **Condition**: Boolean statement without any Boolean operator is called as condition or clause.

2. **Decision**: Boolean statement consisting of conditions and zero or many Boolean operators is called as decision or predicate. A decision with no Boolean operator is a condition [9].
   Example: Let's take an example: if((a>100) && ((b<50) ||(c>40))
   Here, in the if-statement whole expression is called as predicate or decision, && and ||are the Boolean operators and $(a > 100)$, $(b < 50)$ and $(c > 40)$ are different conditions or clause.

3. **Group of Conditions**: Boolean statement consisting of two or more conditions and one or more operators is called as a group of conditions.
   Example: statement1: if ((A && B) ||(C && D)).
   Here A, B, C, D are four different conditions and (A && B), (C && D) are two groups of conditions. Statement 1 is nothing but the decision statement.

Table 2.1: Truth table for two variables

x	y	2- NAND	2-NOR	2-XOR	2-XNOR
0	0	1	1	0	1
0	1	1	0	1	0
1	0	1	0	1	0
1	1	0	0	0	1

Table 2.2: Truth table for three variables

x	y	z	3-AND	3-OR	3-NAND	3-NOR	3-XOR	3-XNOR
0	0	0	0	0	1	1	0	0
0	0	1	0	1	1	0	1	1
0	1	0	0	1	1	0	1	1
0	1	1	0	1	1	0	0	0
1	0	0	0	1	1	0	1	1
1	0	1	0	1	1	0	0	0
1	1	0	0	1	1	0	0	0
1	1	1	1	1	0	0	1	1

4. **Logic Gates**: They are the fundamental building blocks of digital electronics and perform some logical functions. Most of the logic gates accept two binary inputs and result in single output in the form of 0 or 1. Table 2.1 and Table 2.2 show the truth table for two and three variables respectively [9].

2.2 Modified Condition/ Decision Coverage

MC/DC was designed to take the advantages of Multiple Condition testing when retaining the linear growth of the test cases. The main purpose of this testing is that in the application code each and every condition in a decision statement affects the outcome of the statement [10] [11]. MC/DC needs to satisfy the followings:

- Each exit and entry point in the code is invoked.
- Each and every condition in a decision statement is exercised for each possible output.
- Each and every possible output of every decision statement is exercised.
- Each and every condition in a statement is shown to independently affect the output of the decision stated.

Name	Schematic Representation	Code example	Truth Table
Input	A		
Output	C		
and Gate	A, B, C	C := A *and* B;	A B C T T T T F F F T F F F F
or Gate	A, B, C	C := A *or* B;	A B C T T T T F T F T T F F F
xor Gate	A, B, C	C := A *xor* B;	A B C T T F T F T F T T F F F
not Gate	A, B	B := *not* A;	A B T F F T

Figure 2.1: Schematic representation for different gates

To understand MC/DC approach completely we need to show the schematic representation of logical operator and the truth table of program code. Fig. 2.1 shows the schematic representation of the example predicate given below: Example: Z= (A ||B) && (C ||D) In this example A, B, C, D is four different conditions and Z is the output. For four conditions, we have sixteen combinations and outcomes respectively. MC/DC looks for the pair of test cases in which one condition changes the value and all others will remain as it is and it affects the output. Table 2.3 shows the representation for sixteen combinations.

To evaluate MC/DC using the gate level approach, each Boolean logical operator in a predicate in the code is examined to calculate whether the requirement-based test has observably exercised the operator using the minimum test. This concept is combination of condition coverage and decision coverage.

Following five steps are used to determine the MC/DC coverage:

1. Develop a proper representation of the program.
2. Find the test inputs, which can be obtained from the requirement based tests of the software product.
3. Remove the masked test cases. The masked test case is one whose output for a particular gate hidden from all others outputs.

Table 2.3: MC/DC result for four variables

	A	B	C	D		Z		A	B	C	D
1	F	F	F	F		F					
2	F	F	F	T		F		10	6		
3	F	F	T	F		F		11	7		
4	F	F	T	T		F		12	8		
5	F	T	F	F		F				7	6
6	F	T	F	T		T			2		5
7	F	T	T	F		T		3	5		
8	F	T	T	T		T		4			
9	T	F	F	F		F				11	10
10	T	F	F	T		T		2			9
11	T	F	T	F		T		3		9	
12	T	F	T	T		T		4			
13	T	T	F	F		F					
14	T	T	F	T		T				15	14
15	T	T	T	F		T					13
16	T	T	T	T		T				13	

4. Calculate MC/DC based on Table 2.3.

5. At last the results of the tests are used to confirm correct operation of the program. For the details of constructing the MC/DC table the readers may refer to [11].

2.3 Determination of Predicates

The method of determining predicate p_x is given here, which simply uses the Boolean derivative designed by Akers et al. [12] One benefit of this method is that the problem of redundancy of the same clause is handled properly, i.e. the fact that the clause appearing many times is represented explicitly. For a predicate p with variable x, let $p_x = true$, represents the predicate p and each occurrence of x is replaced by true and $p_x = false$, represents the predicate p and each occurrence of x is replaced by false. It may be noted that, here neither $p_x = true$ nor $p_x = false$ contains any occurrences of the clause x. Now, here we combine two expressions with the logical operator Exclusive OR:

$$p_x = p_{x=true} \oplus p_{x=false} \tag{2.1}$$

It turns out that p_x describes the exact conditions under which the value of x determines that of p. If the values for the clauses in p_x are taken so that p_x is true, then the truth value of x determines the truth value of p. If the clauses in p_x are taken so that p_x evaluates to false, then the truth value of p is independent of the truth value of x. Now, let's take an example: Consider the statement,

$$p = x \wedge (y \vee z) \tag{2.2}$$

If the major cause is x, then the Boolean derivative [13] finds truth assignments for y and z as follows:

$$p_x = p_{x=true} \oplus p_{x=false} \tag{2.3}$$

$$p_x = (true \wedge (y \vee z)) \oplus (false \wedge (y \vee z)) \tag{2.4}$$

$$p_x = (y \vee z) \oplus false \tag{2.5}$$

$$p_x = y \vee z; \tag{2.6}$$

This shows the deterministic answer, three choices of values make $y \vee z = true$, $(y = z = true)$, $(y = true, z = false)$, $(y = false, z = true)$.

2.4 CONCOLIC Testing

In this section we discuss about CONCOLIC testing definition and process.

```
1  #include<conio.h>
2  #include<stdio.h>
3  void main()
4  {
5  int distance,time;
6  if(distance>0)
7       continue;
8  else
9               return;
10 if(time>0)
11    continue;
12 else
13     return;
14 Speed_Category=distance/time;
15 if(Speed_category<20)
16                       return slow;
17 else if(Speed_category<40)
18                      return medium;
19 else
20                           return fast;
21 }
22
```

Figure 2.2: An example program to explain CONCOLIC testing

2.4.1 Definition

The CONCOLIC testing [14]concept combines a concrete constraints execution and symbolic constraints execution [15] to automatically generate test cases for full path coverage. This testing generates test suites by executing the program with random values. At execution time both concrete and symbolic values are saved for executing path. The next iteration of the process forces is selected for different path. The tester selects a value from the path constraints and negates the values to create a new path value [16]. Then the tester finds concrete constraints to satisfy the new path values. These constraints are inputs for all next execution. This process performed iteratively until exceeds the threshold value or sufficient code coverage obtained.

Let us take an example Fig. 2.2, calculate speed category of bike when distance and time are given. Tester starts by executing the method with random strategy. Assume that tester has set the values of *distance=120* and *time=-5* in km and hours respectively. During execution time both concrete and symbolic values are saved for executing path. For input constraints to execute the similar path, it is must that each statement with decision branch calculates the similar value. The first statement (Line 6 in Fig. 2.2) will execute as true, because initially the distance is equal to 120, which is more than zero i.e.

$$(Distance > 0) \tag{2.7}$$

Now it's the time for a second branch statement which becomes false because time is automatic set as negative value i.e.

$$\neg(time > 0) \tag{2.8}$$

Therefore, the present branch statement is combined with the previous branch statement to form a new path statement:

$$(Distance > 0) \wedge \neg(Time > 0) \tag{2.9}$$

The method fails in the execution of the second condition, so it is altered by negating the branch constraints. When the last condition is negated, the expression becomes:

$$(Distance > 0) \wedge (Time > 0) \tag{2.10}$$

Now, this new path is passed to a solver to determine whether there exists an input that executes the new path. Definitely there will be many solutions but a tester picks one among all and executes for the next iteration. This time the input can be *distance=60* and *time=1* in km and hour respectively. Now it will execute without throwing any exception and return the category of speed. This path has the following constraints:

$$(Distance > 0) \wedge (time > 0) \wedge (speedcategory < 20) \wedge (speedcategory < 40) \tag{2.11}$$

where,

$$speedcategory = distance/time \tag{2.12}$$

This process continues until the stopping criteria is met. This could be possible only when the iteration exceeds the threshold value and sufficient coverage is obtained.

2.4.2 Process

The CONCOLIC testing process is carried out using the following six steps [16]:

1. **Symbolic Variables Declaration**: In starting, user has to decide which variable will be symbolic variables so that symbolic path formula is made.

2. **Instrumentation**: A target source code is statically instrumented with probes, which keep track of symbolic path conditions from a concrete execution path when the target code is executed. Ex: At each branch, a probe is inserted to track the branch condition.

3. **Concrete Execution**: The instrumented code is compiled and run with given input values. For the first time the target code assigned with random values. For the second time onwards, input values are getting from step 6.

4. **Evaluation of symbolic path formula X**: The symbolic execution module of the CONCOLIC testing executions collects symbolic path conditions over the symbolic input values at every branch point collides along the concrete execution path. Whenever s statement of the target code is executed, a corresponding probe inserted at s updates the symbolic structure of symbolic variables if statements are an assignment statement, or gathers a corresponding symbolic path condition c, if s is a branch statement. Therefore at last symbolic path formulas X is built at the last point of the ith execution by combining all path conditions c_1, c_2, c_3 where c_j is executed earlier than $c_{j+1} \forall 1 \leq j$.

5. **Evaluation of symbolic path formula X' for the next input values**: To find X' we have to negate one path condition c_j and removing after path conditions (i.e, c_{j+1}, c_n) of X'. If X' is not satisfiable, another path condition c'_j is negated and after path condition are removed, till satisfiable formula is getting. If there are no more paths to try, the algorithm stops executing.

6. **Choosing the next input values**: Constraints solver generates a model that satisfies X'. This model takes decision for next concrete input values and this procedure is repeated again from Step 3 with this input value.

2.5 Summary

We discussed relevant definitions which are useful to our approach. We explained definition, criterion, and process of modified condition/ decision coverage in detailed. We determined the boolean derivative predicates. At last of this chapter we discussed about the definition and process of the CONCOLIC testing.

Chapter 3

Review of Related Work

In this chapter we will discuss the existing work on Automated Testing for Branch Coverage and MC/DC [10] [17].

3.1 Automated testing for branch coverage

Automated test data generation for structural coverage is a very known topic of software testing. Search-based testing, symbolic testing, random testing and CONCOLIC testing are different type of automated branch coverage testing.

1. **Search-based testing**: The generation of test data is like a searched based optimization problem. McMinn [18] describes solutions in his survey. Solutions of this problem using Evolutionary Testing (ET) method are like Genetic Algorithm (GA) and like Hill Climbing (HC). These solutions are to achieve branch coverage.

2. **Symbolic testing**: Cadar et al. [19] Says that test data generated by symbolic execution used by the Symbolic testing technique. King et al. [15] describes that the execution assigns a symbolic statement instead of concrete values to code variables as a path is followed by the program structure. At last the result will show the concrete test data that execute these paths.

3. **Random testing**: An easy technique for automated test generation is described by Duran [20] [21] [22] [23]. If the technical meaning contrasts random with systematic, it is in the sense that fluctuations in physical measurements are random (unpredictable or chaotic) vs. systematic (causal or lawful). Godefroid et al. [24] say random testing provides low code coverage. The then branch of the conditional statement if (x ==

100) then has only one chance to be exercised out of 2^{32} if x is a 32-bit integer code input that is randomly initialized.

4. **CONCOLIC Testing**: Kim et al. [16] says the technique combines a concrete dynamic execution and a symbolic execution to automatically generate test cases for path coverage is known as CONCOLIC testing. In our approach we will use CONCOLIC tester CREST [25] an open source CONCOLIC testing tool for C code structures. CONCOLIC represent CONCrete + symbOLIC tests [7] [14] [16].

3.2 MC / DC Automatic Testing

Awedikian et al. [10] proposed an approach to automatically generate test data to satisfy MC/DC. The steps are as follows:

1. For each predicate, compute the sets for MC/DC coverage.
2. Following the proposed fitness function, compute:
    (a) Improved approach function
        i. Control dependencies
        ii. Data dependencies
    (b) Branching fitness function
3. Generate test data using Meta heuristic algorithms.

Liu et al. [26] proposed to replace the branch fitness with a flag cost function that considers the data dependence relationship between the use of the flag and its definitions and creates a set of conditions.

Bokil et al. have proposed a tool AutoGen that reduces the cost and effort for test data preparation by automatically generating test data for C code. Autogen takes the C code and a criterion such as statement coverage, decision coverage, or Modified Condition/Decision Coverage (MC/DC) as input and generates non-redundant test data that satisfies the specified criterion.

3.3 Other related works

Awedikian et al. [10] have given a concept for automatic MC/DC test generation. They used ET methods to generate test inputs to achieve MC/DC coverage. The concept is modified approach of the branch distance computation. They perform based on control and data dependencies of the code. Their objective was MC/DC coverage. However, a drawback of local maxima as the HC algorithm performs data search in limited scope. This shows the solution is not globally optimal.

Pandita et al. [27] have given an instrumental method for generating extra conditional statements for automating logical coverage and boundary value coverage. In this method they used symbolic execution. The coverage of the extra conditional statements increases the logical coverage and boundary value coverage of program code. However, the drawbacks are it does not effectively handle Boolean statements containing ||(OR) operators and it inserts many infeasible conditions into a program.

Hayhurst et al. [28] proposed a modified work of logic gate testing in MC/DC. From program they are creating a logic gate structure of the Boolean statements. Further they used Minimizing Boolean simplification method to decrease the number of logic gates. However, investigation of the process left.

Kuhn et al. [29] and Ammann et al. [4] proposed methods for generating test suite for making a clause which independently affects the result of the predicate. Their methods help to manually determine the independent effect. In particular they applied the Exclusive OR logic to calculate these conditions. However, investigation of automation of the method is left.

Xiao et al. [8] proposed suggestions on how we can use CONCOLIC testing tools [24] [19] and how we can improve CONCOLIC testing.

Ammann et al. [4] proposed predicate transformation issues. They have introduced a new way to express the requirement that tests " independently affect the outcome of a decision" by defining the defining the term determination, and separating minor and major clauses.

August et al. [30] presented paper which shows the ability of the mechanisms presented to overcome limits on ILP previously imposed by rigid program control structure. They

have proposed boolean minimization technique are applied to the network both to reduce dependence height and to simplify the component expression.

3.4 Summary

In this chapter we discussed about related work on Automated Testing for Branch Coverage and MC/DC. We also discussed other related works.

Chapter 4

Program Code Transformer Technique

This chapter presents a detailed explanation of the proposed automatic test generation [31] approach for MC/DC. Here we will see the formal definition and detailed description of our Program Code Transformer Technique.

4.1 Formal Definition

Our objective is to achieve structural coverage on a given program code under test (X), in the context of a given coverage criteria (Y). It uses the tester tool that aims to achieve coverage criterion (Y'). Therefore, our aim is to transform X to X' such that the problem of achieving coverage in X with respect to Y is transformed into the problem of achieving structural coverage in X' with respect to Y'. Few defined terms are the followings:

1. **COVERAGE (Y, X, M):** It shows the percentage of coverage achieved by a test suite (M) over a given program under test (X) with respect to given coverage criteria (Y).

2. **OUTPUT(X, I):** It shows the output result of a program code under test (X) subject to an input (I).

3. **(X $\vdash$ M):** It shows that a test suite (M) is generated by the tester tool β for the program (X) code under test.

We now defined our proposed approach. For a given X, the idea is to transform X to X', where X' = X+Z and Z is the code added to X such that the following requirements are met.

$$\textbf{R1:}\forall : [Output(X, I) = Output(X', I)], \tag{4.1}$$

where I is the collection of inputs to X.

The above statement's states that Z should not have any side effect on X. Z has a side effect if the execution of X' produces a different result from the one produced by the execution of X, when executed with same input I.

R2: If the test suite M1 is generated from X' by the tester tool β, then

$$\exists M1 : [((X' \rightarrow M1) \wedge Coverage(Y', X', M1) = 100\%) \Rightarrow (Coverage(Y, X, M1) = 100\%)] \tag{4.2}$$

The requirement states that if there exists a test suite M1 that achieves 100% coverage on X' with respect to Y', then coverage of M1 on X with respect to Y is 100%.

4.2 Our first proposed approach MC/DC Tester-I [MT-I]

Our approach developing MC/DC TESTER-I [MT-I], has central logic to extend the CONCOLIC testing to get increased MC/DC. Transformation of program code under test to include extra conditions is a feasible alternative to achieve increased MC/DC. After program transformation, we let it drive a CONCOLIC tester CREST to generate test suite. A proper representation of the test data generation by our MC/DC TESTER-I [MT-I]) is shown in Fig. 4.1. MT-I consists of three components:

1. Program code transformer,

2. Tester for CONCOLIC testing,

3. Coverage Analyser.

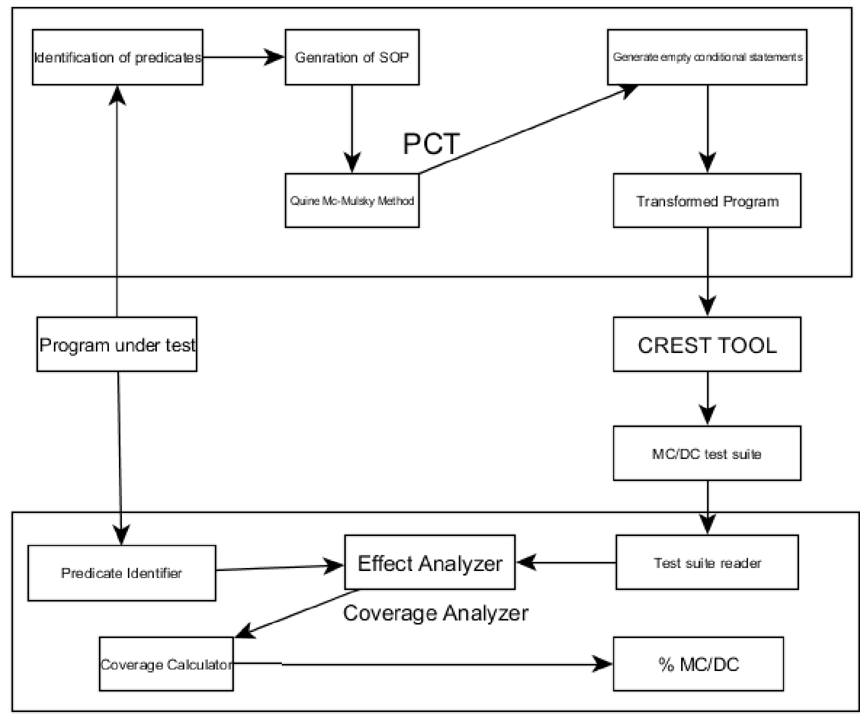

Figure 4.1: Schematic representation of our first approach [MT-I]

From the Fig. 4.1, program code under test is entered to the transformer, and it modifies the code by generating and adding conditional statements on the bases of the MC/DC coverage. We use the Boolean logic simplification technique to develop transformer. This approach converts a complex Boolean statement into a simpler form and generates additional statements from these simple expressions [13]. The transformed code is then passed to the CONCOLIC tester which executes all the branches of the transformed program and automatically generates the inputs for the feasible path. The original program code and the test data generated by the tester for the transformed program code are passed to the coverage analyzer. The analyzer calculates the percentage of MC/DC achieved in the program under test by the generated test data.

4.3 Program Code Transformer

We named our approach Program Code Transformer (PCT). The objective is based on the fact that MC/DC of a program is equivalent to testing of flip-flops and logic gates.

PCT converts each predicate in an entered program code to the standard sum-of product (SOP) form by Boolean algebra [13]. After this we use QUINE-Mc-MLUSKY Technique OR Tabulation Method to minimize the sum of product. The statement is then suppressed into simple conditions with empty true and false branches and inserted in the program before the predicate. The purpose of inserting empty true and false branches is to avoid duplicate statement executions as the original predicate and the statement in its branches are retained in the program during transformation. It is a simple process to retain the functional equivalence of the code and yet produces additional test cases for increased MC/DC coverage. Thus, PCT consists of mainly three steps and the second step consists of two sub steps as shown in Fig. 4.1. The pseudo-code representation of PCT is given in Algorithm1.

***Algorithm1: Program Code Transformer*.**

Input:X

Output:X'

Begin

for each statement s∈X **do**

  if && or ||occurs in s **then**

    List_Predicate←adding_in_List(s)

  end if

end for

for each predicate p ∈ List_Predicate **do**

  P_SOP ← gen_sum_of_product(p)

  P_Meanterm ← Convert_to_Minterm(P_SOP)

  P_Simplifeid ← Mini_Sumofproduct_Tabulation(P_Minterm)

  List_Statement ← generate_Nested_Ifelse_PCT(P_Simplified)

  X'← insert_code(List_Statement,X)

end for

return X'

4.3.1 Description of Algorithm 1

1. **Identification of predicates** The objective of this step (First for loop in Algorithm 1) is to identify the predicates in program code under test. This step is executed once in the whole process. All conditional expressions with Boolean operators are predicates. Further process will proceed after this.

2. **Simplification** From Algorithm 1 (line 2-4). First it generates the sum of product standard form and then it uses Tabulation method to minimize expression identified above.

   (a) **Sum of Product:** Line 2 in Algorithm 1 takes a predicate as input and generates the standard sum of product form. Here, we may give a justification of generating sum of product (SOP), not product of sum (POS) because the structure of the POS will fail for OR operator condition. All should be in AND operator condition which doesn't show flexibility of the standard format.

   (b) **Minimization:** Lines 3-4 in Algorithm1 are responsible for calling another algorithm Algorithm 2 for minimizing expressions. Here we use Quine-Mc-Mlusky Technique or Tabulation method to minimize expression. Another technique could be Karnaugh Map, but we will use Tabulation method which having advantage which overcoming the problem of Karnaugh maps.

3. **Nested If-else Generation:** Using line 5-6 in Algorithm 1 the additional conditional expressions are generated and inserted into the program code under test. From previous step we get minimized expression in SOP form. Using Algorithm 3, we generate empty If-else conditions. Line 7 returns the transformed program. The pseudo-code representation of Minimisation of SOP Tabulation Method is given in Algorithm 2.

***Algorithm2: Minimization of SOP Tabulation Method*.**

Input:P_Minterm

Output:P_Simp

Begin

for each min_term m∈P_Minterm **do**

  1 List_M ← Convert_to_binary(minterm)

end for

2 List_L ←sort(List_M)

for each List l∈L **do**

  for each group_first to group_last∈groups **do**

    for each bit∈toatal_bits **do**

      3 one_bit_diff_term←Compare(grp_current,grp_next)

    end for

    if 1_bit_diff_term=1&&existed_legal_dash_position **then**

      4 bit will replaced with char – and put chekchar t

else

5 put check char ∗ for uncompared group

end if

end for

end for

6 Prime_Implicant ← Uncompared any more and indicated with ∗

7 essential_Prime_Implicant← Coveragetable(minterms,Prime_Implicant)

8 simplified_function P_Simp←assigning_variables and compliment_variables to test Prime_Implicant

4.3.2 Description of Algorithm 2

Algorithm 2 performs mainly five steps. Lines 1-2 show the conversion of minterm to binary form. Lines 3-5 shows the comparison between groups and marking un-compared group. Line 6 determines prime implicant. Line 7 determines essential prime implicant. Line 8 shows the use of Patrik's method to get simplified function. The pseudocode representation for generating empty if-else conditional statements is given in Algorithm 3.

Algorithm3: PCT_generateNestedIfElse.

Input:p

Output:Statement_list //list of statement in c

Begin

for each && connected cond_grp∈p **do**

for each condition a∈cond_grp **do**

if a is the firstcondition **then**

make an if statement m with a as the condition

Statement_list←add_list(m)

else

make a nested if statement m with a as the condition make an empty Truebranch Tb and an empty Falsebranch Fb in order

Statement_list←add_list(strcat(m,Tb,Fb))

end if

end for

make an empty Falsebranch Fb for the first condition

Statement_list←add_list(Fb)

end for

```
1  #include <stdio.h>
2  godboley_Weight (int p, int q, int r, int s)
3  int main(int argc, char *argv[])
4  {
5   godboley_Weight (75,65,89,95);
6      return 0;
7  }
8  godboley_Weight (int p, int q, int r, int s){
9    if ((p>70)&&((q<80)||(r<90)||(s<100))){
10    printf("weight must be more than 70 kg "); }
11   else{
12    printf("weight may be more or less than 70 kg");}
```

Figure 4.2: An example showing concept of PCT

for each condition∈p and ∉ any cond_grp **do**
  repeat line 4,8 and 9
end for
if P is an else if predicate **then**
  make an if(false) statement m
  make an empty Truebranch Tb
  Statement_list←add_list(strcat(m,Tb))
end if
return Statement_list

4.3.3 Description of Algorithm 3

The PCT generateNestedIfElse method decomposes the minimized SOP expression into a set of nested if else constructs. The minimized SOP expression contains simple conditions or groups of conditions connected with OR operator. The conditions within a group of conditions are connected with AND operators. For every condition in each group. This method creates nested if conditions and corresponding else conditions (Lines 1-7 in Algorithm 3). This ensures that each condition is evaluated to both true and false values. The iteration over all the groups ensures that the process is applied to every condition in the predicate. Line 8 repeats these steps for simple condition if they are part of the minimized SOP expression. Lines 9-11 insert a dummy if statements in the program if the identified predicate was an else-if predicate. Line 12 returns list of statements. The generated nested if-else statements are then inserted into the program under test before the particular predicate. The insert_Code method in Algorithm 1 performs this task.

```
1  #include <stdio.h>
2  godboley_Weight (int p, int q, int r, int s)
3  int main(int argc, char *argv[])
4  {
5   godboley_Weight (75,65,89,95);
6      return 0;
7  }
8  godboley_Weight(int p,int q,int r,int s){
9   if(p>50)
10      if(q<80)
11       { }
12       else{ }
13   else{ }
14   if(p>50)
15       if(r<90)
16       { }
17       else{ }
18   else{ }
19   if(p>50)
20       if(s<100)
21       { }
22       else{ }
23   else{ }
24  if ((p>70)&&((q<80)||(r<90)||(s<100))){
25      printf("weight must be more than 70 kg "); }
26  else{
27      printf("weight may be more or less than 70 kg");}
28  }
29
```

Figure 4.3: Transformed form of Fig. 4.2

4.3.4 Example of PCT

We explain the working of the PCT with an example. Consider the example of *godboley-Weight* function shown in Fig. 4.2. After identifying the predicates, we generate SOP form and minimize it using Tabulation method; We get the following form: $(((p > 70)\&\&(q < 80))\ ||((p > 70)\&\&(r < 90))||((p > 70)\&\&(s < 100)))$; The transformed code for *godboley-Weight* function is shown in following Fig. 4.3. The above program contains empty true and false branches. This confirms that the Transformation of code does not have any effect on the program because there are no executable statements in these empty branches.

4.3.5 Complexity of PCT

The overall time complexity of PCT is O(n+mn)=O(mn), where m is the number of predicates in a program and n is the number of statements in a program code.

4.4 CONCOLIC Testing

The transformed program of a program code under test from the PCT is passed to the CREST TOOL. This tester achieves branch coverage through random test generation. CON-

COLIC tester is a combination of concrete and symbolic testing. The extra generated expressions lead to generation of extra test cases for the transformed program. Because of random strategy different runs of the CONCOLIC tester may not generate identical test cases. The generated test cases depend on the path on each run. All test cases are stored in text files which form a test suite.

4.5 MC/DC Coverage Analyser

It determines the MC/DC coverage achieved by a test suite. It is required to calculate the extent to which a program feature has been performed by test cases. In our approach, it is essentially used to calculate if there are any changes in MC/DC coverage performed by the test cases generated by the CREST TOOL using our approach. Coverage Analyser (CA) examines the extent to which the independent effect of the component conditions on the calculation of each predicate of the test data takes place. The MC/DC coverage achieved by the test cases T for program input p denoted by MC/DC coverage is calculated by the formula:

$$MC/DCcoverage = (\Sigma_{i=1ton} I_i \div \Sigma_{i=1ton} c_i) \times 100 \tag{4.3}$$

Algorithm4 :MCDC COVERAGE ANALYSER.

Input:X,Test_Suite // Program X and Test_Suite is collection of Test cases
Output:MC/DCcoverage // % MC/DC achieved for X
Begin
for each statement s∈X **do**
  if && or ||occurs in s **then**
    List_Predicate←adding_in_List(s)
  end if
end for
for each predicate p∈List_Predicate **do**
  for each condition c∈p **do**
    for each test_case t_d ∈ Test_Suite **do**
      if c evaluates to TRUE and calculate the outcome of p with t_d **then**
        True_Flag←TRUE
      end if
      if c evaluates to FALSE and calculate the outcome of p without t_d **then**
        False_Flag←TRUE
      end if

```
      end for
      if both True_Flag and False_Flag are TRUE then
        I_List←adding_in_List(c)
      end if
      C_List←adding_in_List(c)
    end for
  end for
  MC_DC_COVERAGE←(SIZEOF(I_List)/SIZEOF(C_List))× 100%
```

4.5.1 Description of Algorithm 4

Algorithm 4 describes the coverage analyzer. It takes a program and test suite as input and produces coverage percentage. Line 1 shows identification of predicates. Lines 2-5 show the determination of outcomes. Line 6 calculates the coverage percentage.

4.6 Experimental Study

In this section we observe experimental study with some requirements details, result, and comaparision.

4.6.1 PCT details

Program Code transformer is built up of five modules viz. Predicate Identifier, Sum of Product Generator, Qune-McMulsky Technique, Empty Nested If-Else conditional statement generator, and Code Inserter. The size of prototype of PCT is 2577 lines of code.

The *Predicate Identifier* module reads the program under test which is written in C language. The scanning is line by line execution. Wherever this module detects boolean operators like &&, ||, and !, the module separte the whole line as it is and saves all predicate to another file. The *Sum-of-Product Generator* module converts each predicate to *Sum-of-Product(SOP)* form by using Boolean algebra laws. The formed SOP may be complex so we required to test by using minimizing technique. For our approach, we have considered *Quine-McMulsky* method. The reason behind choosing this technique is the disadvantage of K-map. The K-map technique is easy to use upto four variables and in extreme case five

```
310 int
311 get_process(prio, ratio, job)
312      int prio;
313      float ratio;
314      struct process ** job;
315 {
316     int length, index;
317     CREST_int(length);
318     CREST_int(index);
319     CREST_int(prio);
320     struct process **next;
321     if(prio > MAXPRIO)
322     { }
323     else { }
324     if(prio < 0)
325     { }
326     else { }
327     if(prio > MAXPRIO || prio < 0) return(BADPRIO); /* Somebody goofed */
328     if(ratio < 0.0)
329     { }
330     else { }
331     if(ratio > 1.0)
332     { }
333     else { }
334     if(ratio < 0.0 || ratio > 1.0) return(BADRATIO); /* Somebody else goofed */
335     length = prio_queue[prio].length;
336     index = ratio * length;
337     index = index >= length ? length -1 : index; /* If ratio == 1.0 */
338     for(next = &prio_queue[prio].head; index && *next; index--)
339         next = &(*next)->next; /* Count up to it */
340     *job = *next;
341     if(*job)
342     {
343     *next = (*next)->next; /* Mend the chain */
344     (*job)->next = (struct process *) 0; /* break this link */
345     prio_queue[prio].length--;
346     return(TRUE);
347     }
```

Figure 4.4: Transformed program from PCT

variables, beyond which it is very difficult to use. In a program we generally cannot expect number of conditions to be fixed, it may be in any number. But the Qune-McMulsky method is usefull for n number of conditions. After minimisation, we get the simplified form of SOP. The *Empty Nested If-Else conditional statement* generator module breaks the minimized form of SOP in small simple condition and passes it to *Code Inserter* module. The *Code Inserter* module inserts the transformed conditions into the program just above the predicate detects. The process is repeated for all the predicates detected by *Predicate Identifier*. Fig. 4.4 shows a transformed output from our PCT technique.

4.6.2 CREST details

In our experiments we have used CREST as the CONCOLIC tester. CREST is written in C language and works for programs written in C language only. CREST performs symbolic execution and concrete execution simultaneously. There are many search strategies like DFS (Bounded Depth First Search), CFG (Control Flow Directed Search), Random, and Uniform strategies that are used in CREST.

CREST accepts C program and selects the concrete values for the symbolic variables. The values for all the variables in a program are is saved in an input file. The number of input files depends on number of iterations provided and on number of covered branches. The values in input files are nothing but the test data or test cases. The collection of test cases is test suite.

CREST is used on any modern LINUX or Mac operating systems. The main limitations of CREST is that it can solve path constraints with integer variables. It cannot solve path constraints with float, string, pointer variables, functions call, and native calls.

The process of CREST compilation and execution are shown from Fig. 4.5 to Fig. 4.14. Fig. 4.5 the example program that we have taken from chapter 4. Fig. 4.6 shows the compilation of original program with number of branches and number of nodes. Fig. 4.7 shows the run execution of the compiled program and results in number of reachable functions, reachable branches, and covered branches by using any search strategies and number of iterations provided. The input file is automatically generated and it consists of concrete values as shown in Fig. 4.8. The coverage file consists of node number. This file is automatically generated as shown in Fig 4.9. Fig. 4.10 shows the transformed program of original program. Rest of the figures from Fig. 4.11 to Fig. 4.14 shows the same process for transformed program and from the figures it is clearly evident that all the values have been increased.

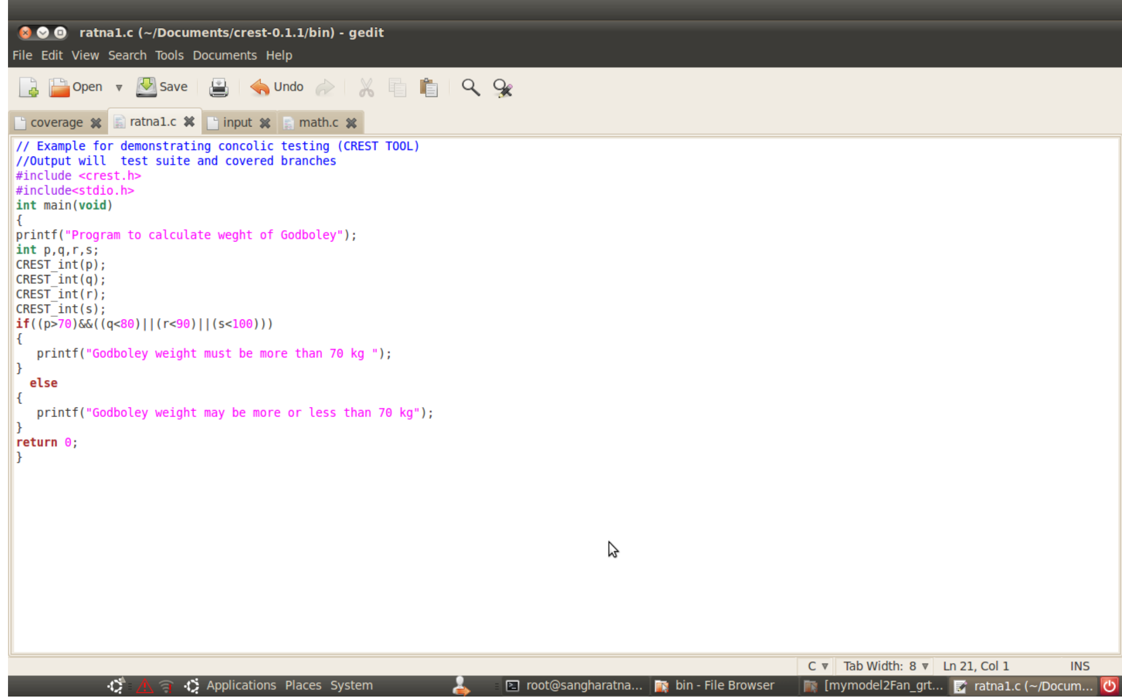

Figure 4.5: An Example program from chapter 4

```
root@sangharatna-laptop: /home/sangharatna/Documents/crest-0.1.1/bin
File Edit View Terminal Help
root@sangharatna-laptop:/home/sangharatna/Documents/crest-0.1.1/bin# ./crestc ratna1.c
gcc -D_GNUCC -E -I./../include -DCIL=1 ratna1.c -o ./ratna1.i
/home/sangharatna/Documents/crest-0.1.1/cil/obj/x86_LINUX/cilly.asm.exe --out ./ratna1.cil.c --doCrestInstrument ./ratna1.i
gcc -D_GNUCC -E -I./../include ./ratna1.cil.c -o ./ratna1.cil.i
gcc -D_GNUCC -c -I./../include -o ./ratna1.o ./ratna1.cil.i
ratna1.c:6: warning: '__crest_skip__' attribute directive ignored
ratna1.c:7: warning: '__crest_skip__' attribute directive ignored
ratna1.c:8: warning: '__crest_skip__' attribute directive ignored
ratna1.c:9: warning: '__crest_skip__' attribute directive ignored
ratna1.c:10: warning: '__crest_skip__' attribute directive ignored
ratna1.c:11: warning: '__crest_skip__' attribute directive ignored
ratna1.c:12: warning: '__crest_skip__' attribute directive ignored
ratna1.c:13: warning: '__crest_skip__' attribute directive ignored
ratna1.c:14: warning: '__crest_skip__' attribute directive ignored
ratna1.c:15: warning: '__crest_skip__' attribute directive ignored
./../include/crest.h:202: warning: '__crest_skip__' attribute directive ignored
gcc -D_GNUCC -o ratna1 -I./../include ./ratna1.o ./../lib/libcrest.a -L./../lib -lstdc++
Read 8 branches.
Read 13 nodes.
Wrote 6 branch edges.
root@sangharatna-laptop:/home/sangharatna/Documents/crest-0.1.1/bin#
Applications Places System   root@sangharatna...   bin - File Browser   [mymodel2Fan_grt...   ratna1.c (~/Docum...
```

Figure 4.6: Compilation of original program

```
root@sangharatna-laptop: /home/sangharatna/Documents/crest-0.1.1/bin
File Edit View Terminal Help
root@sangharatna-laptop:/home/sangharatna/Documents/crest-0.1.1/bin# ./crestc ratna1.c
gcc -D_GNUCC -E -I./../include -DCIL=1 ratna1.c -o ./ratna1.i
/home/sangharatna/Documents/crest-0.1.1/cil/obj/x86_LINUX/cilly.asm.exe --out ./ratna1.cil.c --doCrestInstrument ./ratna1.i
gcc -D_GNUCC -E -I./../include ./ratna1.cil.c -o ./ratna1.cil.i
gcc -D_GNUCC -c -I./../include -o ./ratna1.o ./ratna1.cil.i
ratna1.c:6: warning: '__crest_skip__' attribute directive ignored
ratna1.c:7: warning: '__crest_skip__' attribute directive ignored
ratna1.c:8: warning: '__crest_skip__' attribute directive ignored
ratna1.c:9: warning: '__crest_skip__' attribute directive ignored
ratna1.c:10: warning: '__crest_skip__' attribute directive ignored
ratna1.c:11: warning: '__crest_skip__' attribute directive ignored
ratna1.c:12: warning: '__crest_skip__' attribute directive ignored
ratna1.c:13: warning: '__crest_skip__' attribute directive ignored
ratna1.c:14: warning: '__crest_skip__' attribute directive ignored
ratna1.c:15: warning: '__crest_skip__' attribute directive ignored
./../include/crest.h:202: warning: '__crest_skip__' attribute directive ignored
gcc -D_GNUCC -o ratna1 -I./../include ./ratna1.o ./../lib/libcrest.a -L./../lib -lstdc++
Read 8 branches.
Read 13 nodes.
Wrote 6 branch edges.
root@sangharatna-laptop:/home/sangharatna/Documents/crest-0.1.1/bin# ./run_crest ./ratna1 5 -dfs
Iteration 0 (0s): covered 0 branches [0 reach funs, 0 reach branches].
Program to calculate weght of GodboleyGodboley weight may be more or less than 70 kgIteration 1 (0s): covered 1 branches [1 reach funs, 8 reach branches].
Program to calculate weght of GodboleyGodboley weight must be more than 70 kg Iteration 2 (0s): covered 3 branches [1 reach funs, 8 reach branches].
Program to calculate weght of GodboleyGodboley weight must be more than 70 kg Iteration 3 (0s): covered 5 branches [1 reach funs, 8 reach branches].
Program to calculate weght of GodboleyGodboley weight must be more than 70 kg Iteration 4 (0s): covered 7 branches [1 reach funs, 8 reach branches].
Program to calculate weght of GodboleyGodboley weight may be more or less than 70 kgIteration 5 (0s): covered 8 branches [1 reach funs, 8 reach branches].
root@sangharatna-laptop:/home/sangharatna/Documents/crest-0.1.1/bin#
```

Figure 4.7: Run execution of original program

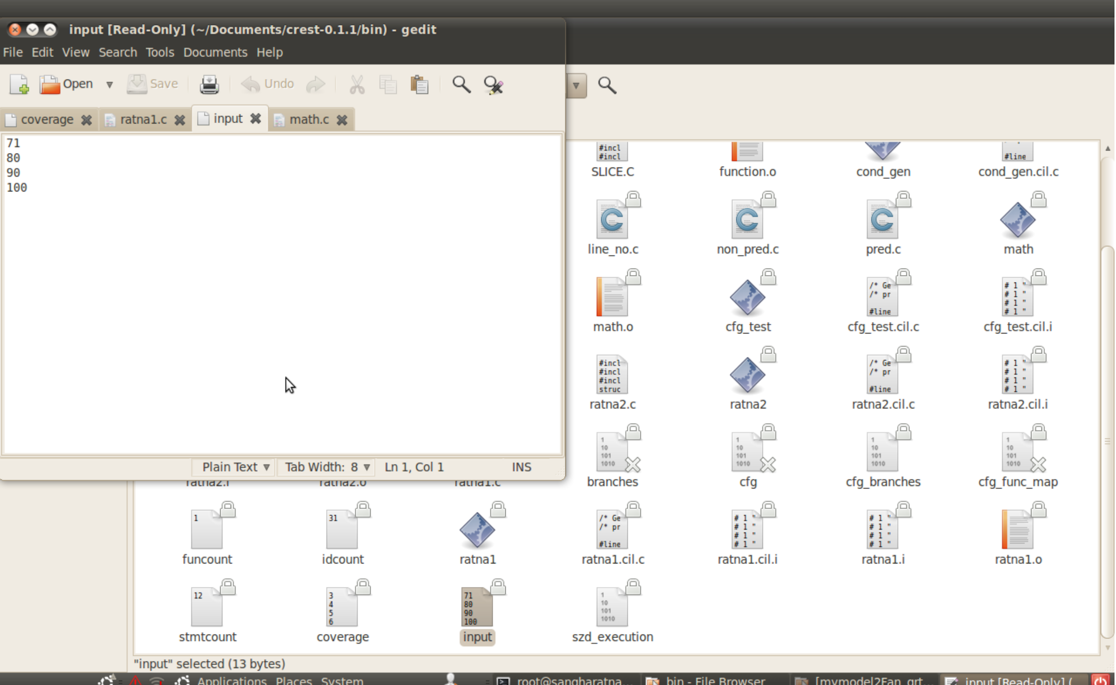

Figure 4.8: Generated Test data in input file for original program

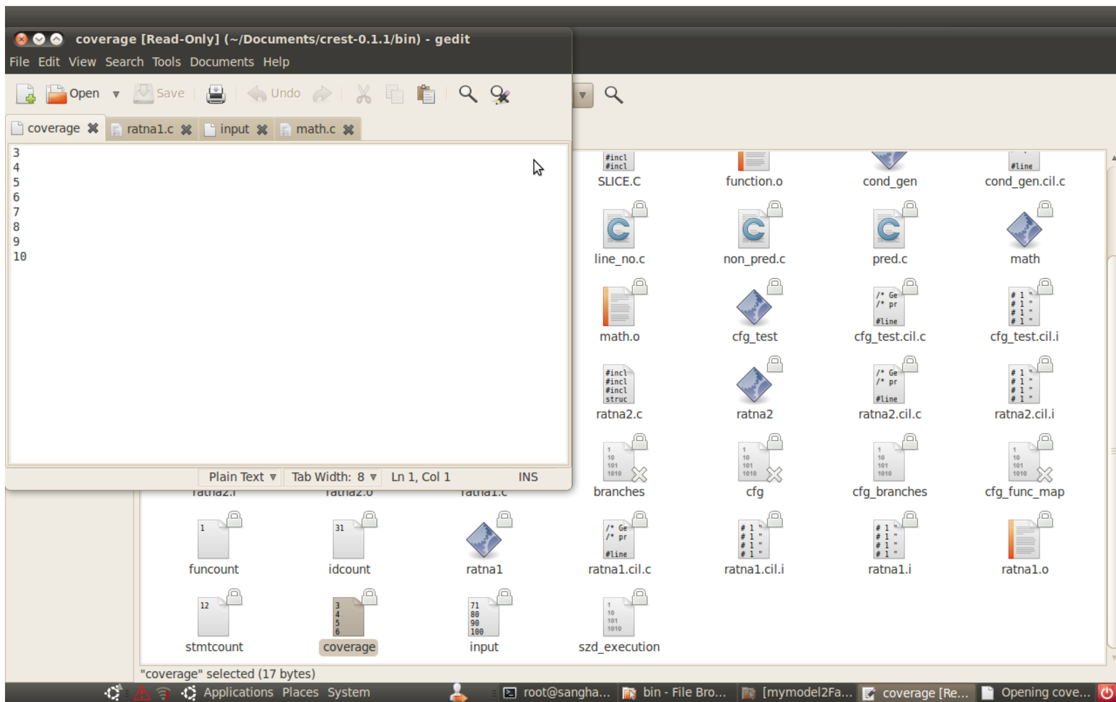

Figure 4.9: Automatically generated coverage file for original program

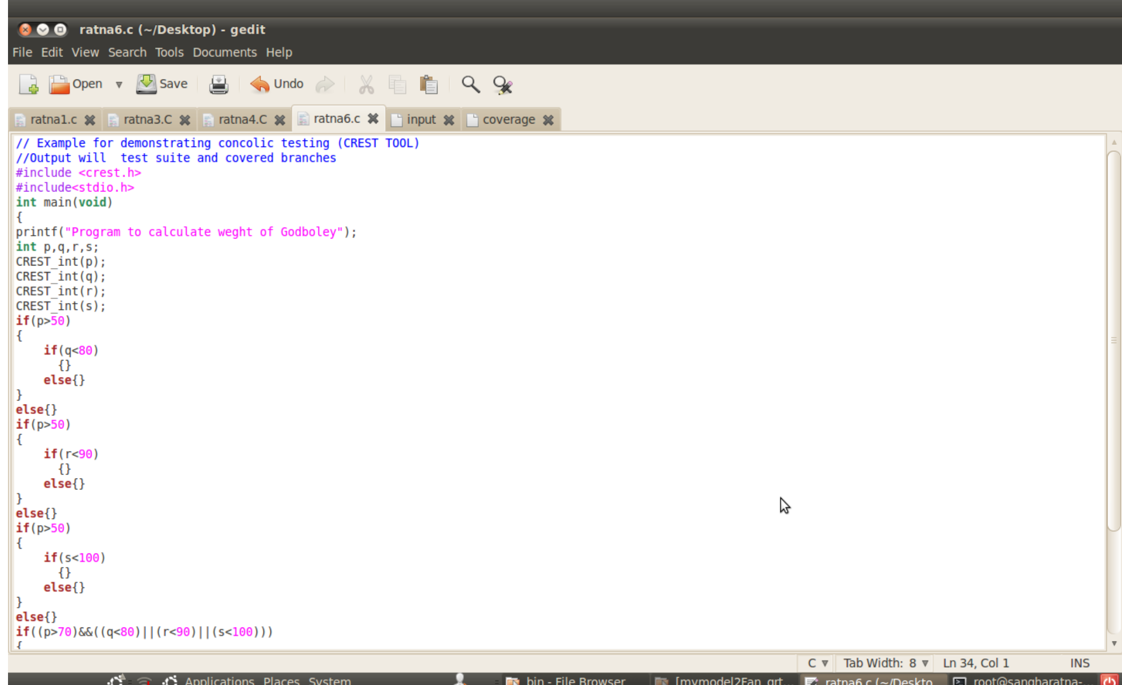

Figure 4.10: Transformed program of original program

```
root@sangharatna-laptop: /home/sangharatna/Documents/crest-0.1.1/bin
File Edit View Terminal Help
ratna1.c:8: warning: '__crest_skip__' attribute directive ignored
ratna1.c:9: warning: '__crest_skip__' attribute directive ignored
ratna1.c:10: warning: '__crest_skip__' attribute directive ignored
ratna1.c:11: warning: '__crest_skip__' attribute directive ignored
ratna1.c:12: warning: '__crest_skip__' attribute directive ignored
ratna1.c:13: warning: '__crest_skip__' attribute directive ignored
ratna1.c:14: warning: '__crest_skip__' attribute directive ignored
ratna1.c:15: warning: '__crest_skip__' attribute directive ignored
./../include/crest.h:202: warning: '__crest_skip__' attribute directive ignored
gcc -D_GNUCC -o ratna1 -I./../include ./ratna1.o ./../lib/libcrest.a -L./../lib -lstdc++
Read 8 branches.
Read 13 nodes.
Wrote 6 branch edges.
root@sangharatna-laptop:/home/sangharatna/Documents/crest-0.1.1/bin# ./run_crest ./ratna1 5 -dfs
Iteration 0 (0s): covered 0 branches [0 reach funs, 0 reach branches].
Program to calculate weght of GodboleyGodboley weight may be more or less than 70 kgIteration 1 (0s): covered 1 branches [1 reach funs, 8 reach branches].
Program to calculate weght of GodboleyGodboley weight must be more than 70 kg Iteration 2 (0s): covered 3 branches [1 reach funs, 8 reach branches].
Program to calculate weght of GodboleyGodboley weight must be more than 70 kg Iteration 3 (0s): covered 5 branches [1 reach funs, 8 reach branches].
Program to calculate weght of GodboleyGodboley weight must be more than 70 kg Iteration 4 (0s): covered 7 branches [1 reach funs, 8 reach branches].
Program to calculate weght of GodboleyGodboley weight may be more or less than 70 kgIteration 5 (0s): covered 8 branches [1 reach funs, 8 reach branches].
root@sangharatna-laptop:/home/sangharatna/Documents/crest-0.1.1/bin# ./crestc ratna6.c
gcc -D_GNUCC -E -I./../include -DCIL=1 ratna6.c -o ./ratna6.i
/home/sangharatna/Documents/crest-0.1.1/cil/obj/x86_LINUX/cilly.asm.exe --out ./ratna6.cil.c --doCrestInstrument ./ratna6.i
gcc -D_GNUCC -E -I./../include ./ratna6.cil.c -o ./ratna6.cil.i
gcc -D_GNUCC -c -I./../include -o ./ratna6.o ./ratna6.cil.i
ratna6.c:6: warning: '__crest_skip__' attribute directive ignored
ratna6.c:7: warning: '__crest_skip__' attribute directive ignored
ratna6.c:8: warning: '__crest_skip__' attribute directive ignored
ratna6.c:9: warning: '__crest_skip__' attribute directive ignored
ratna6.c:10: warning: '__crest_skip__' attribute directive ignored
ratna6.c:11: warning: '__crest_skip__' attribute directive ignored
ratna6.c:12: warning: '__crest_skip__' attribute directive ignored
ratna6.c:13: warning: '__crest_skip__' attribute directive ignored
ratna6.c:14: warning: '__crest_skip__' attribute directive ignored
ratna6.c:15: warning: '__crest_skip__' attribute directive ignored
./../include/crest.h:202: warning: '__crest_skip__' attribute directive ignored
gcc -D_GNUCC -o ratna6 -I./../include ./ratna6.o ./../lib/libcrest.a -L./../lib -lstdc++
Read 20 branches.
Read 28 nodes.
Wrote 30 branch edges.
Applications Places System    bin - File Browser  [mymodel2Fan_grt...  ratna6.c (~/Deskto...  root@sangharatna-...
```

Figure 4.11: Compilation of transformed program

```
root@sangharatna-laptop: /home/sangharatna/Documents/crest-0.1.1/bin
File Edit View Terminal Help
root@sangharatna-laptop:/home/sangharatna/Documents/crest-0.1.1/bin# ./crestc ratna6.c
gcc -D_GNUCC -E -I./../include -DCIL=1 ratna6.c -o ./ratna6.i
/home/sangharatna/Documents/crest-0.1.1/cil/obj/x86_LINUX/cilly.asm.exe --out ./ratna6.cil.c --doCrestInstrument ./ratna6.i
gcc -D_GNUCC -E -I./../include ./ratna6.cil.c -o ./ratna6.cil.i
gcc -D_GNUCC -c -I./../include -o ./ratna6.o ./ratna6.cil.i
ratna6.c:6: warning: '__crest_skip__' attribute directive ignored
ratna6.c:7: warning: '__crest_skip__' attribute directive ignored
ratna6.c:8: warning: '__crest_skip__' attribute directive ignored
ratna6.c:9: warning: '__crest_skip__' attribute directive ignored
ratna6.c:10: warning: '__crest_skip__' attribute directive ignored
ratna6.c:11: warning: '__crest_skip__' attribute directive ignored
ratna6.c:12: warning: '__crest_skip__' attribute directive ignored
ratna6.c:13: warning: '__crest_skip__' attribute directive ignored
ratna6.c:14: warning: '__crest_skip__' attribute directive ignored
ratna6.c:15: warning: '__crest_skip__' attribute directive ignored
./../include/crest.h:202: warning: '__crest_skip__' attribute directive ignored
gcc -D_GNUCC -o ratna6 -I./../include ./ratna6.o ./../lib/libcrest.a -L./../lib -lstdc++
Read 20 branches.
Read 28 nodes.
Wrote 30 branch edges.
root@sangharatna-laptop:/home/sangharatna/Documents/crest-0.1.1/bin# ./run_crest ./ratna6 50 -dfs
Iteration 0 (0s): covered 0 branches [0 reach funs, 0 reach branches].
Program to calculate weght of GodboleyGodboley weight may be more or less than 70 kgIteration 1 (0s): covered 4 branches [1 reach funs, 20 reach branches].
Program to calculate weght of GodboleyGodboley weight may be more or less than 70 kgIteration 2 (0s): covered 10 branches [1 reach funs, 20 reach branches].
Program to calculate weght of GodboleyGodboley weight may be more or less than 70 kgIteration 3 (0s): covered 11 branches [1 reach funs, 20 reach branches].
Program to calculate weght of GodboleyGodboley weight may be more or less than 70 kgIteration 4 (0s): covered 12 branches [1 reach funs, 20 reach branches].
Program to calculate weght of GodboleyGodboley weight may be more or less than 70 kgIteration 5 (0s): covered 13 branches [1 reach funs, 20 reach branches].
Program to calculate weght of GodboleyGodboley weight may be more or less than 70 kgIteration 6 (0s): covered 17 branches [1 reach funs, 20 reach branches].
Program to calculate weght of GodboleyGodboley weight must be more than 70 kg Iteration 7 (0s): covered 18 branches [1 reach funs, 20 reach branches].
Program to calculate weght of GodboleyGodboley weight may be more or less than 70 kgIteration 8 (0s): covered 18 branches [1 reach funs, 20 reach branches].
Program to calculate weght of GodboleyGodboley weight must be more than 70 kg Iteration 9 (0s): covered 19 branches [1 reach funs, 20 reach branches].
Program to calculate weght of GodboleyGodboley weight must be more than 70 kg Iteration 10 (0s): covered 19 branches [1 reach funs, 20 reach branches].
Program to calculate weght of GodboleyGodboley weight may be more or less than 70 kgIteration 11 (0s): covered 19 branches [1 reach funs, 20 reach branches].
Program to calculate weght of GodboleyGodboley weight may be more or less than 70 kgIteration 12 (0s): covered 19 branches [1 reach funs, 20 reach branches].
Program to calculate weght of GodboleyGodboley weight must be more than 70 kg Iteration 13 (0s): covered 20 branches [1 reach funs, 20 reach branches].
Program to calculate weght of GodboleyGodboley weight must be more than 70 kg Iteration 14 (0s): covered 20 branches [1 reach funs, 20 reach branches].
Program to calculate weght of GodboleyGodboley weight may be more or less than 70 kgIteration 15 (0s): covered 20 branches [1 reach funs, 20 reach branches].
Program to calculate weght of GodboleyGodboley weight must be more than 70 kg Iteration 16 (0s): covered 20 branches [1 reach funs, 20 reach branches].
Program to calculate weght of GodboleyGodboley weight must be more than 70 kg Iteration 17 (0s): covered 20 branches [1 reach funs, 20 reach branches].
root@sangharatna-laptop:/home/sangharatna/Documents/crest-0.1.1/bin#
Applications Places System    bin - File Browser  [mymodel2Fan_grt...  ratna6.c (~/Deskto...  root@sangharatna-...
```

Figure 4.12: Run execution of transformed program

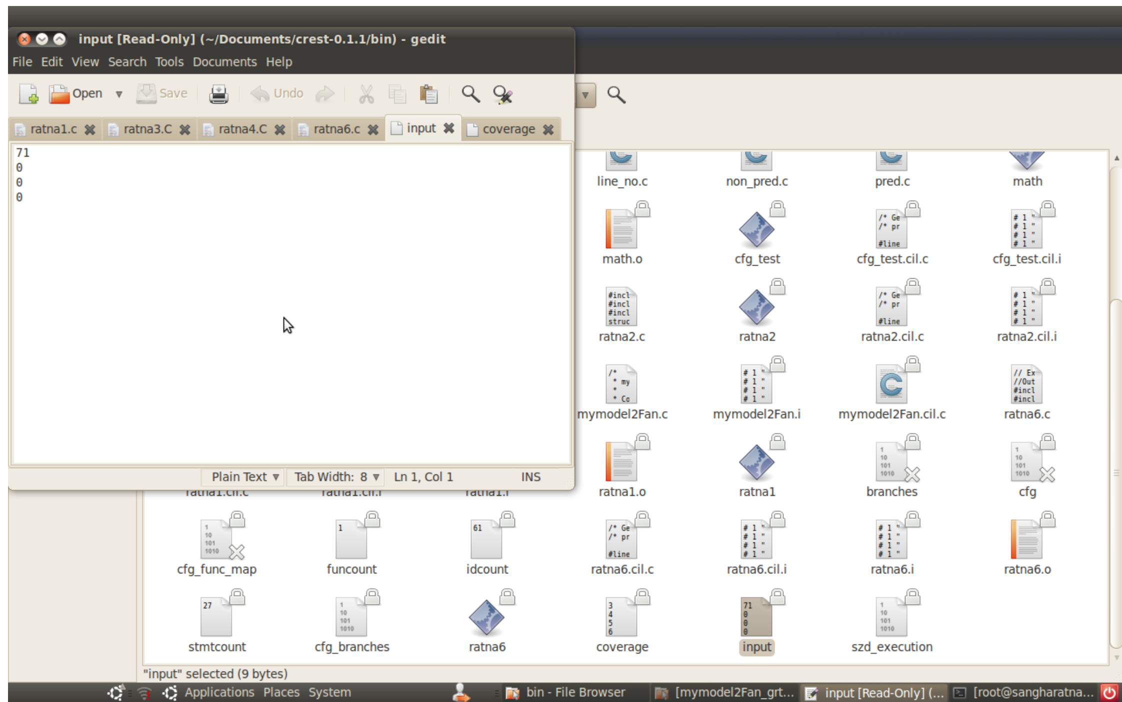

Figure 4.13: Generated Test data in input file for transformed program

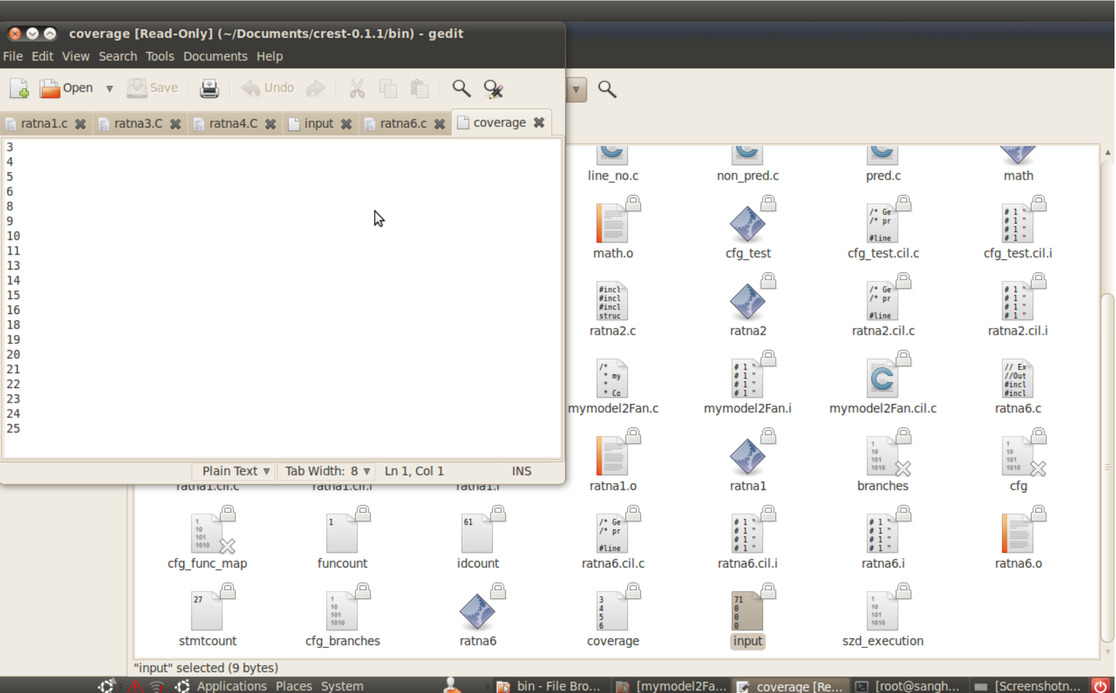

Figure 4.14: Automatically generated coverage file for transformed program

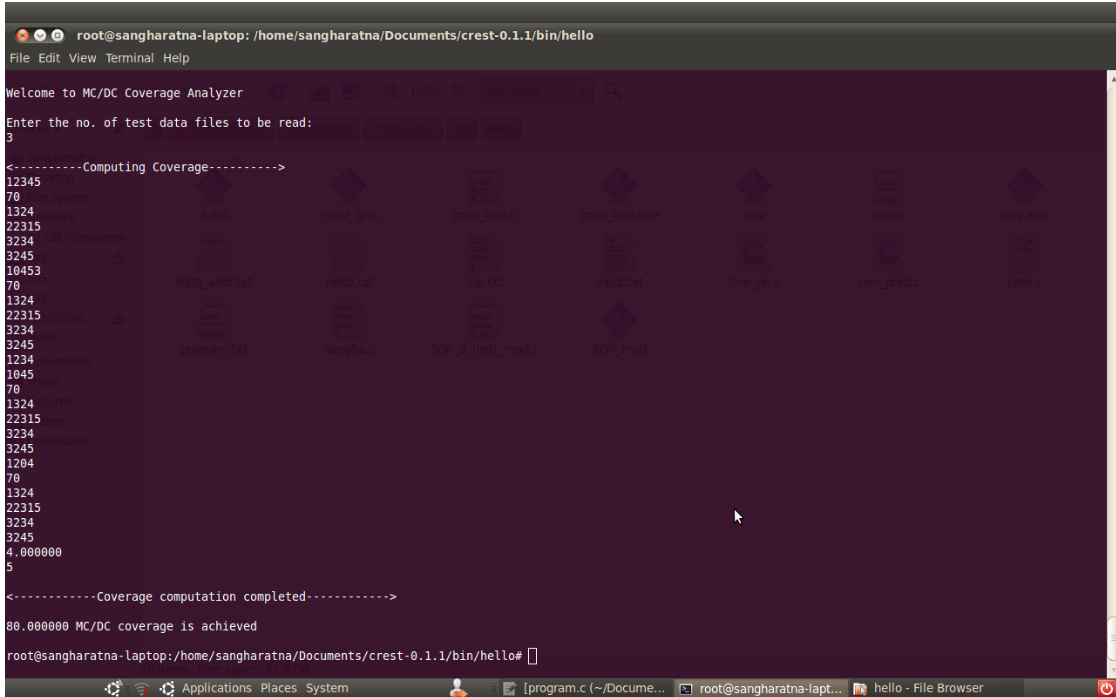

Figure 4.15: Screenshot of Coverage analyser

4.6.3 CA details

Coverage Analyser is built up of four modules viz. Predicate Identifier, Test Suite Reader, Effect Analyzer, and Coverage Calculator. The size of prototype is 971 lines of codes. The Predicate Identifier module in CA is same with that of PCT. The second module Test Suite Reader reads each test cases generated from CREST tool and passes it to the third module Effect Analyzer. The Effect Analyzer module reads each predicate and test cases and checks whether the test data makes each condition in a predicate both true and false. It also check whether the conditon independently affect the result of whole predicate or not. Finally, it identifies the number of independently affected conditons and the total number of conditions and passes them to the fourth module. The fourth module Coverage Calculator module calculates the percentage of MC/DC achieved by the test suite. Fig. 4.15 shows the output of coverage analyser.

4.6.4 Experimental Reqiurements

In our experiments, we have considered seventeen example programs written in C language. Some programs are open soure and some other programs are taken from student

Table 4.1: Summary of Characteristics of the program under test for PCT

S.No	Program	LOC	LOC'	Function	Predicate	Branch	Branch'	Edges	Edges'
1	Triangle	63	75	1	2	16	24	21	40
2	Next Date	106	135	6	3	32	46	42	70
3	ATM	150	241	1	10	54	98	78	163
4	Library	221	242	6	4	66	82	96	134
5	TCAS	272	338	10	10	88	146	151	299
6	Schedule	327	349	16	4	100	116	181	213
7	Tic-tac-toe	279	375	6	11	126	198	233	379
8	Elevator	445	530	6	8	158	224	273	571
9	Tokenizer	509	578	19	6	162	194	332	492
10	Ptok2	569	672	24	9	168	254	347	512
11	Replace	608	733	20	15	200	280	376	656
12	Ptok1	725	893	19	18	284	379	433	679
13	Phonex	1030	1198	33	19	348	578	647	928
14	ProgramSTE	1051	1254	28	25	384	538	687	1163
15	ProgramTR	1117	1311	42	23	362	474	708	1148
16	Sed	8678	10143	70	48	2690	4623	3727	5428
17	Grep	12562	13743	126	53	3768	6279	5249	7346

projects. All the experiments perfomed in a system having 1.85 GHz processing with 1 GB RAM and having Ubuntu Linux operating system installed in it..

To measure the improvement in MC/DC we required to observe two observations, first observation is calculation of MC/DC coverage percentage without code transformation. Second observation is calculation of MC/DC coverage percentage with code transformer. We have proposed code transformation technique i.e Program Code Transformer(PCT) and Coverage Analyser(CA) to calculate coverage percentage.

Table 4.1 shows the characteristics of the programs under test. The column *program*, shows the name of all example programs. Column *LOC* shows the number of lines of codes in a program. LOC' shows the number of lines of codes in transformed program by using PCT respectively. *Function* and *Predicate* columns shows the number of functions and predicates in a program. Branch column shows the number of branches in a program. Branch' shows the number of branches in a transformed program by using PCT. Edges column shows the number of branch edges in a program. Edges' shows the number of branch edges in transformed program using PCT.

Table 4.2: Coverage Calculated by Coverage Analyser for PCT

S.No	Program	M_Cov	M_Cov_PCT	INC_Using_PCT
1	Triangle	75%	100%	25%
2	Next Date	71%	88.2%	17.2%
3	ATM	70%	94.7%	24.7%
4	Library	75%	100%	25%
5	TCAS	52.3%	73.7%	20.8%
6	Schedule	62.5%	82.9%	20.4%
7	Tic-tac-toe	65%	85%	20%
8	Elevator	63.9%	81.9%	18%
9	Tokenizer	64.7%	83%	18.3%
10	Ptok2	63%	78%	15%
11	Replace	57.8%	79%	21.2%
12	Ptok1	64.6%	85.2%	20.6%
13	Phonex	63.7%	81.8%	18.1%
14	ProgramSTE	62.4%	82.7%	20.3%
15	ProgramTR	64%	87%	23%
16	Sed	58.5%	73.4%	14.9%
17	Grep	53.8%	74.6%	20.8%

4.6.5 Results

We present the experimental results in Table 4.2. M_Cov column shows the MC/DC coverage percentage calculated by covergae analyser. None of the seventeen programs under consideration has 100 % coverage. The minimum coverage percentage of 52.9 % is for TCAS and maximum coverage percentage of 75 % is for the two programs (Triangle and Library). Our main objective is to improve our MC/DC coverage percentage and try to achieve 100 % coverage. We propose PCT transformation technique to do what so all the original programs get transformed. M_Cov_PCT column shows the MC/DC coverage percentage calculated by coverage analyser for PCT. We observed that we got an increase in MC/DC percentage. Two programs achieved 100 % viz. Triangle and Library. The minimum coverage percentage for M_Cov_PCT is 73.4 %. PCT transformation technique having some disadvantage, so we have proposed an efficient technique which overcomes the problem of PCT is Exclusive-NOR Code Transformer(X-NCT). The average increase in MC/DC coverage percentage achieved for seventeen program by using Program Code Transformer technique is 20.194 %.

Table 4.3: Time Constraint

S.No	Programs	PCT (sec)	CREST (sec)	CA (sec)	TOTAL TIME (sec)
1	Triangle	0.002673	0	3	3.002673
2	Next date	0.004548	0	3	3.004548
3	ATM	1.166958	1	3	5.166958
4	Library	0.007316	7	5	12.007316
5	TCAS	1.792756	2	5	8.792756
6	Schedule	0.010618	0	3	3.010618
7	Tic-tac-toe	1.682375	2	4	7.682375
8	Elevator	1.824791	0	4	5.824791
9	Tokenizer	1.473162	0	3	4.473162
10	Ptok2	2.001572	1	3	6.001572
11	Replace	1.523179	0	6	7.523179
12	Ptok1	1.918721	1	2	4.918721
13	Phonex	0.763241	2	4	6.763241
14	ProgSTE	1.629315	6	5	12.629315
15	ProgTR	1.427561	5	6	12.427561
16	Sed	2.347128	1	4	7.347128
17	Grep	2.71568	0	4	6.71568

4.6.6 Time constraints

In this section we disuss about time taken by our approach to complile and execute. Time constraints is one of the important parameter in software development phases. We calculate the time effort so that we can observe that time taken to calculate MC/DC coverage percentage is efficient. Time constraint of seventeen programs is shown in Table 4.3.

Fig. 4.16 shows the graph for time constraints. Seventeen programs executed through code transformer, CREST tool, and coverage analyzer and recorded their compilation and execution time. The unit of time taken is in *seconds*. The average time taken for seventeen programs is 6.89950 sec.

4.6.7 Comparison Between BCT and PCT

Das et al. [32] proposed a Boolean Code Transfomer technique in which he has taken ten example program, according to his work we may observe our reults in Table 4.4. M-Cov-BCT column represent the MC/DC coverage percentage for transformed program by using transformation techniwue BCT. INC-Cov-BCT column shows the increase in cover-

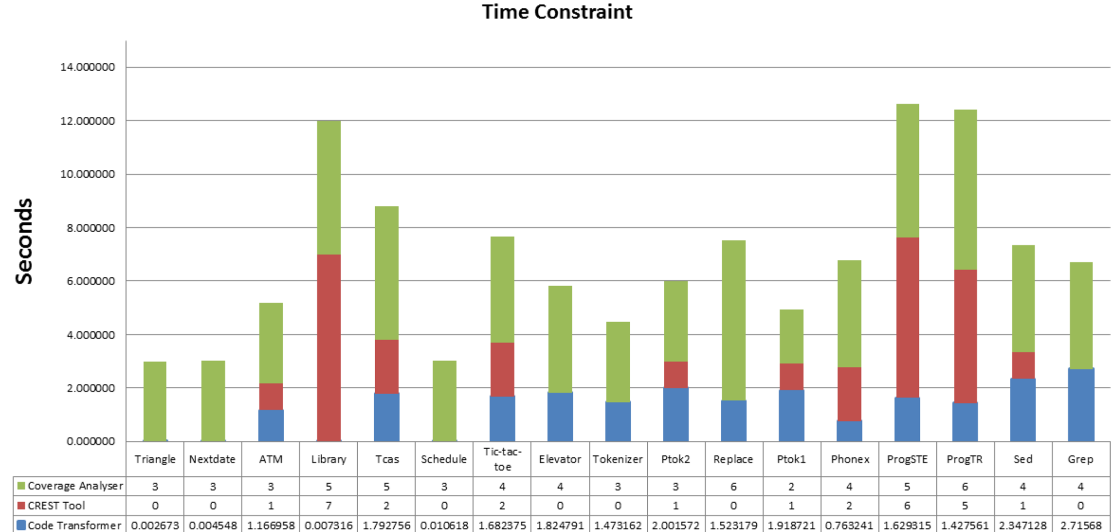

Figure 4.16: Time Constarint Graph

Table 4.4: Coverage percentage for BCT and PCT technique

S.No	Program	M-Cov-BCT	INC-Cov-BCT	M-Cov-PCT	INC-Cov-PCT
1	Triangle	100%	25%	100%	25%
2	Next Date	89.5%	18.5%	88.2%	17.2%
3	ATM	93.3%	23.3%	94.7%	24.7%
4	Library	100%	25%	100%	25%
5	TCAS	70.6%	17.7%	73.7%	20.8%
6	Schedule	84.5%	22%	82.9%	20.4%
7	Tic-tac-toe	86%	21%	85%	20%
8	Elevator	82.5%	18.6%	81.9%	18%
9	Tokenizer	81%	17.7%	83%	18.3%
10	Replace	75%	17.2%	79%	21.2%

age percentage. The average coverage percentage for BCT is 20.6 %. M-Cov-PCT column represents the MC/DC coverage percentage for transformed program by using transformation technique PCT. INC-Cov-PCT column shows the increase in coverage percentage. The average coverage percentage for PCT is 21.06 %. The increased coverage percentage from BCT to PCT is 0.46 % as shown in Fig. 4.17.

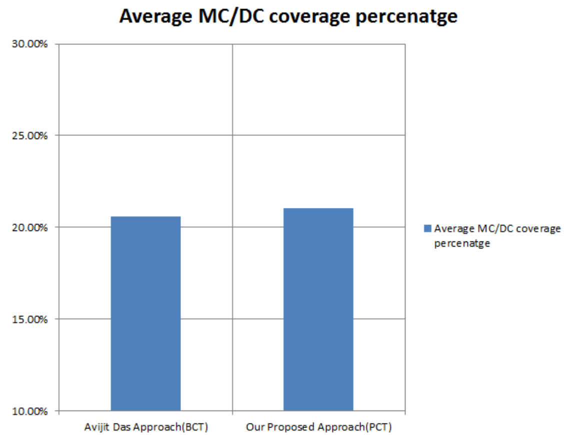

Figure 4.17: Comparison between transformation techniques BCT and PCT

4.7 Limitation of PCT

It is possible that for a predicate PCT can skip to traverse for some conditions in same predicate. For example if first condition of a predicate fails and the very next operator is (OR) operator than PCT won't execute further conditions.

Let's take a predicate S1: if(a && b), where a and b are arbitrary boolean conditions. The MC/DC coverage requires the test set of (t,t),(t,f), and (f,t) to satisfy the predicate $a \wedge b$. Now, after transformation using PCT technique, the test set generated will be (t,t),(t,f), and one of either (f,t) or (f,f). If (f,t) choosen then it will allow a to independently determine the predicate outcome otherwise (f,f) does not allow.

The demerit of PCT forced us to propose new code transformer technique which execute each condition for each predicate. In next chapter we discuss about Exclusive-NOR Code transformer in details.

4.8 Conclusion

We proposed program code transformer technique to improve our MC/DC percentage. We achieved 20.194 % coverage percentage for seventeen programs. We discussed experimental study for our approach. As compared to other approach our transformer achieve 0.46 % more. We discussed the time constraint parameter and we conclude that the average time taken for seventeen programs is 6.89950 seconds.

Chapter 5

Exclusive-Nor Code Transformer Technique

This chapter presents an explanation of the proposed approach i.e Automated test suite generation approach for MC/DC coverage. In this technique we have used X-NOR operator because it is more efficient than X-OR operator. X-NOR operator required less number of gates in place of X-OR. X-NOR is less complex than X-OR. Compliment of X-OR results X-NOR. Before describing our approach, first we present some definitions that will be used in our approach.

5.1 Our Proposed Approach MC/DC Tester-II [MT-II]

The main purpose of our proposed MC/DC tester is to extend the CONCOLIC testing to get increased MC/DC coverage. Transformations of programs under test to include extra conditions are a feasible alternative to attain that aim. After program transformation, let us drive a CONCOLIC tester CREST Tool to generate MC/DC test suite. A representation of the test data Suite generation is described in Fig. 5.1. The approach consists of three components:

1. X-NOR Code Transformer
2. CONCOLIC Testing Tool
3. Coverage Analyser

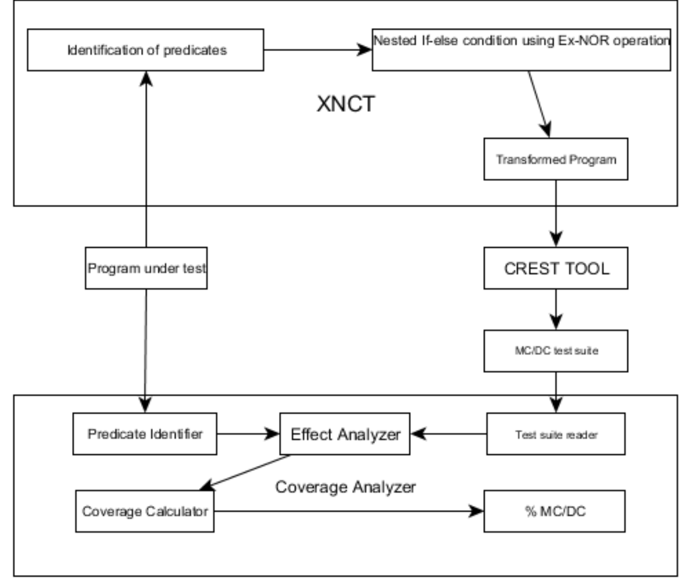

Figure 5.1: Schematic representation of our second approach [MT-II]

Fig. 5.1 describes the schematic representation of our approach. A program under test is inserted as input to the X-NOR Code Transformer. It changes the code by generating and adding extra conditional statements for the MC/DC coverage. This approach performs for every clause of the Boolean statement to generate additional statements after identifying the predicates. The transformed program is then passed to the CONCOLIC testing tool which executes all the branches of the transformed program and generates the input for the feasible path. The original program and the test data suite generated by the CONCOLIC tester for the transformed program code is supplied to a coverage analyzer. The coverage analyzer calculates the percentage of MC/DC coverage achieved in the program under test by the generated test suite.

5.2 X-Nor Code Transformer (X-NCT)

We have named the code transformer as X-NCT i.e Exclusive Nor Code Transformer. It uses the Exclusive-Nor gate to calculate the conditions under which each condition in a predicate statement can independently determine the output of a predicate. It assigns each occurrence of the condition in an expression first as true and then as false and then performs X-Nor operation. The output of the X-Nor operation gives the condition under

which the clause independently affects the expression results. These additional additional conditions with empty true false inserted. The purpose of inserting empty true and false branches is to avoid duplicate statement executions.

Thus, X-Nor comprises mainly of two major steps:

1. Identification of Predicates
2. Generation of Nested If-Else statements

Algorithm1: Exclusive-Nor Code Transformer.

Input:X //program X is in C syntax
Output:X' //program X' transformed
Begin
// start first step
for each statement s∈X **do**
  if && or ||or unary !occurs in s **then**
    List_Predicate←adding_in_List(s)
  end if
end for// stop first step // start second step
for each predicate p ∈ List_Predicate **do**
  List_Statement ← generate_Nested_IfElse_XNCT(p)//call algorithm2
  X'← insert_code(List_Statement,p)
end for
return X' // stop second step and return Transformed Program

5.2.1 Description of Algorithm 1

Step1: *Identification of Predicate:* From line number 1 to 5 in algorithm 1. In this step our aim is to determine predicate on the basis of all conditional statements and Boolean operators &&, ||and Unary!. This step is executed once in the whole process. The second step is executed on the basis of each predicate.

Algorithm2: generate_Nested_IfElse_XNCT.

Input:p // predicate p
Output:Statement_list //list of statement in c
Begin
for each && condition c∈p **do**

```
    T_a←p
    T_b←p
    for each occurence of condition c_a of c∈T_a do
      c_a←TRUE
    end for
    for each occurence of condition c_b of c∈T_b do
      c_b←FALSE
    end for
    T_c← Exclusive-Nor(T_a,T_b)
    Create an If staement S_1 with T_c as the predicate
    Create an If staement S_2 with c as the condition
    Create an empty Truebranch T_B1
    c'← Generate_negation(c)
    Create a Nested-ELSE-IF statement S_3 with c' as the condition
    Create an empty True Branch T_B2
    Statement_list←addList(strcat(S_1,S_2,T_B1,S_3,T_B3))
  end for
  return Statement_list
```

5.2.2 Description of Algorithm 2

Step2: *Generation of Nested If-Else Statements:* From line number 6 to 9, generate nested if-else statements. This process is performed using the Boolean derivative method. Line-7 calls Algorithm 2 to add extra conditional statements. Algorithm 2 is based on the Boolean derivative method which is executed for every condition in the predicate. The Exclusive-Nor method in line-9 accepts two predicates and performs X-Nor operation on them and returns a new predicate. From line 1-8, two temporary predicates forming input to the Exclusive-Nor method, before performing operation clause under test, are replaced; once by true and then by false. The true output of the new predicate that is returned by the Exclusive- Nor method depicts the situation under which the condition under test in the identified predicate can independently affect its results. The generated negative method in Line 13 of Algorithm 2 accepts a clause as input and returns a new predicate that is the negation of the input condition. The generated nested if-else statements are then inserted into the original program just above the predicate and performed by insert code method in algorithm 1 for each predicate.

Table 5.1: Truth table for X-NOR operation

a	b	$z = a \odot b$
0	0	1
0	1	0
1	0	0
1	1	1

5.3 Exclusive-NOR operation

In X-Nor MC/DC coverage, test data are successfully performed by taking initially a=0, b=0 and output Z=1. Now the independent value of a=1 where b=0 remains unchanged, the output of whole predicate is different (Z=0) means the individual value of is affected. In case 1 the pair of MC/DC coverage is case 3 with respect to a. Suppose a=1, b=1 and output Z=0, now the independent value of b=0, a=1 unchanged and Z=1, the output of predicate is changing it means the value of b affects the whole predicate. In case 3 the pair of MC/DC coverage is case 4 with respect to b. Therefore, the X - Nor technique is used for MC/DC test data suite. Another alternative of this concept is to use exclusive-OR operation which can perform MC/DC coverage. There is no advantage to use in place of each other but they are two different methods or concepts to generate nested if-else. The X - Nor concept follows the laws of Boolean algebra for an Exclusive-NOR operation shown in Table 5.1 to achieve an increase in MC/DC coverage.

5.4 Example for X-NCT

We describe the concept of X-NCT from Fig. 5.2, Fig. 5.3, and Fig. 5.2 shows the original program. After applying Exclusive NOR operation, the results are shown in Table 5.2, Table 5.3, and Table 5.4 in the form of truth tables. We obtain the final transformed program as in Fig. 5.3. The followings are the steps for three variables:

$$(true\&\&(b \mid\mid c)) \odot (false\&\&(b \mid\mid c)) = !(b \mid\mid c) \tag{5.1}$$

$$(a\&\&(true \mid\mid c)) \odot (a\&\&(false \mid\mid c)) = (!a \mid\mid c) \tag{5.2}$$

$$(a\&\&(b \mid\mid true)) \odot (a\&\&(b \mid\mid false)) = (!a \mid\mid b) \tag{5.3}$$

```
void Test(int a, int b, int c)
{
 if(a&&(b||c))
{
  a=100;
}
 else
{
  b=50;
}
}
```

Figure 5.2: An example for X-NOR operation

```
void Test(int a, int b, int c)
{
 if(!(b||c)){
  if(a){}
  elseif(!a){}
}
 if((!a)||c){
   if(b){}
   elseif(!b){}
}
 if((!a)||b){
   if(c){}
   elseif(!c){}
}
if(a&&(b||c)){
   a=100;}
 else{
  b=50;}
}
```

Figure 5.3: Transformed program for Fig. 5.2

Table 5.2: Truth table for first variable (a) after applying X-NOR operations

<table>
<tr><th>b</th><th>c</th><th>m=(true && (b ||c))</th><th>n=(false && (b ||c))</th><th>m $\odot$ n</th></tr>
<tr><td>T</td><td>T</td><td>T</td><td>F</td><td>F</td></tr>
<tr><td>T</td><td>F</td><td>T</td><td>F</td><td>F</td></tr>
<tr><td>F</td><td>T</td><td>T</td><td>F</td><td>F</td></tr>
<tr><td>F</td><td>F</td><td>F</td><td>F</td><td>T</td></tr>
</table>

Table 5.3: Truth table for second variable (b) after applying X-NOR operations

<table>
<tr><th>a</th><th>c</th><th>m=(a && (true ||c))</th><th>n=(a && (false ||c))</th><th>m $\odot$ n</th></tr>
<tr><td>T</td><td>T</td><td>T</td><td>T</td><td>T</td></tr>
<tr><td>T</td><td>F</td><td>T</td><td>T</td><td>T</td></tr>
<tr><td>F</td><td>T</td><td>F</td><td>F</td><td>T</td></tr>
<tr><td>F</td><td>F</td><td>F</td><td>F</td><td>T</td></tr>
</table>

Table 5.4: Truth table for third variable (c) after applying X-NOR operations

<table>
<tr><th>a</th><th>b</th><th>m=(a && (b ||true))</th><th>n=(a && (b ||false))</th><th>m $\odot$ n</th></tr>
<tr><td>T</td><td>T</td><td>T</td><td>T</td><td>T</td></tr>
<tr><td>T</td><td>F</td><td>T</td><td>F</td><td>F</td></tr>
<tr><td>F</td><td>T</td><td>F</td><td>F</td><td>T</td></tr>
<tr><td>F</td><td>F</td><td>F</td><td>F</td><td>T</td></tr>
</table>

5.5 Complexity for X-NCT

The overall time complexity of X-NCT is [O (X+MX) =O (MX)] where M is the number of predicates and X is the number of statements in a program respectively.

5.6 CONCOLIC Tester

The transformed program of a program under test from X-NCT is passed to CONCOLIC tester CREST Tool [1]. This tester achieves branch coverage through random test generation. CONCOLIC Tester is a combination of concrete and symbolic testing. The additional generated expressions lead to generation of extra test cases for the transformed program. Because of random strategy, different execution of the CONCOLIC tester may not generate identical test cases. Test cases generation depends on the path of each execution. All test cases stored in text files forms the test suite. We have already discussed the CREST tool in Section 4.4.

5.7 MC/DC Coverage Analyser

It calculates the MC/DC coverage percentage achieved by a test suite. The MC/DC percentage coverage achieved by the test suite 'T' for program input 'p', is calculated by using equation number 4.3. The algorithm for MC/DC Coverage Analyzer is already discussed in Section 4.5.

5.8 Experimental Study

In this section we observe experimental study with some requirements details, result, comaparision with existance work, and analysis of results.

5.8.1 X-NCT details

Exclusive-NOR code Transformer is built up of four modules viz. Predicate Identifier, X-NOR operator, Empty Nested If-Else conditional statement generator, and Code Inserter. The size of prototype of X-NCT is 1381 lines of code.

```
125     new_process->pid = pid;
126     new_process->priority = prio;
127     new_process->next = (struct process *) 0;
128     status = enqueue(prio, new_process);
129     if(status)
130     {
131         free(new_process); /* Return process block */
132     }
133     }
134     if(status) next_pid--; /* Unsuccess. Restore pid */
135     return(status);
136 }
137
138 int upgrade_prio(prio, ratio) /* increment priority at ratio in queue */
139      int prio;
140      float ratio;
141 {
142     int status;
143     CREST_int(status);
144     struct process * job;
145     if(!(prio < 1))
146     {
147          if(prio > MAXLOPRIO)
148          {printf("1\n"); }
149          else if(!(prio > MAXLOPRIO))
150          {printf("2\n"); }
151     }
152     if(!(prio > MAXLOPRIO))
153     {
154           if(prio < 1)
155           { printf("3\n");}
156           else if(!(prio < 1))
157           { printf("4\n");}
158     }
159     if(prio < 1 || prio > MAXLOPRIO) return(BADPRIO);
160     if((status = get_process(prio, ratio, &job)) <= 0) return(status);
161     /* We found a job in that queue. Upgrade it */
162     job->priority = prio + 1;
```

Figure 5.4: Transformed program from X-NCT

The Predicate Identifier module in X-NCT is same as PCT. For each predicates the rest of three modules are executed. The advantage of choosing X-NOR operator is the less number of gates and less complex than X-OR gate. In this technique X-NOR operator module takes a predicate and creates two new predicates by assigning *true* and *false* to the particular condition and performs X-NOR operation to generate a new predicate. It repeats this step for every condition in the predicate. The Empty Nested If-Else conditional statement generator generates addirional condiitons and passes all these condition to the Code Inserter Module. The Code Inserter module again is similar to that of PCT. Fig. 5.4 shows a transformed output from our X-NCT technique.

5.8.2 Experimental Reqiurements

Table 5.5 shows the characteristics of the programs under test. Column program shows the name of all example programs. Column LOC shows the number of lines of codes in a program. LOC" shows the number of lines of codes in transformed program by using X-NCT. Function and Predicate columns shows the number of functions and predicates in

Table 5.5: Summary of Characteristics of the program under test

S.No	Program	LOC	LOC"	Function	Predicate	Branch	Branch"	Edges	Edges"
1	Triangle	63	92	1	2	16	40	21	66
2	Next Date	106	164	6	3	32	80	42	138
3	ATM	150	310	1	10	54	178	78	308
4	Library	221	273	6	4	66	114	96	186
5	TCAS	272	440	10	10	88	262	151	590
6	Schedule	327	381	16	4	100	148	181	277
7	Tic-tac-toe	279	509	6	11	126	450	233	528
8	Elevator	445	666	6	8	158	466	273	870
9	Tokenizer	509	648	19	6	162	320	332	662
10	Ptok2	569	749	24	9	168	338	347	738
11	Replace	608	830	20	15	200	480	376	826
12	Ptok1	725	957	19	18	284	477	433	812
13	Phonex	1030	1325	33	19	348	947	647	1138
14	ProgramSTE	1051	1556	28	25	384	1064	687	1672
15	ProgramTR	1117	1478	42	23	362	800	708	1488
16	Sed	8678	11565	70	48	2690	6725	3727	6719
17	Grep	12562	14826	126	53	3768	9389	5249	8513

a program. Branch column shows the number of branches in a program. Branch" shows the number of branches in a transformed program by using X-NCT. Edges column shows the number of branch edges in a program. Edges" shows the number of branch edges in transformed program using X-NCT respectively.

5.8.3 Results

We present the experimental results in Table 5.6. M_Cov_X-NCT column shows the MC/DC coverage percentage calculated by coverage analyzer for X-NCT. In this column we can observe that most of the programs achieved increase in MC/DC coverage percentage. Four programs viz. Triangle, Next-Date, ATM, and Library programs achieved 100 % MC/DC coverage percentage. The minimum coverage percentage is 76.2 % for this technique. INC_Using_X-NCT column shows the differnce between the M_Cov and M_Cov_X-NCT and it means increase in coverage percentage by using X-NCT technique. The average increase in MC/DC coverage percentage for seventeen program by using by using Exclusive-NOR Code Transformer technique is 25.4470 %.

Table 5.6: Coverage Calculated by Coverage Analyser

S.No	Program	M_Cov	M_Cov_X-NCT	INC_Using_X-NCT
1	Triangle	75%	100%	25%
2	Next Date	71%	100%	29%
3	ATM	70%	100%	30%
4	Library	75%	100%	25%
5	TCAS	52.3%	86.4%	33.5%
6	Schedule	62.5%	89.6%	27.1%
7	Tic-tac-toe	65%	92%	27%
8	Elevator	63.9%	88.3%	24.4%
9	Tokenizer	64.7%	87.3%	22.6%
10	Ptok2	63%	84%	21%
11	Replace	57.8%	83.7%	25.9%
12	Ptok1	64.6%	88.3%	23.7%
13	Phonex	63.7%	87%	23.3%
14	ProgramSTE	62.4%	88.2%	25.8%
15	ProgramTR	64%	89.7%	25.7%
16	Sed	58.5%	79.7%	21.2%
17	Grep	53.8%	76.2%	22.4%

5.8.4 Comparison Between XCT and X-NCT

Das et al. [32] proposed another technique called as Exclusive OR Code Transfomer technique in which he has taken seventeen example program, according to his work we may observe our reults in Table 5.7. M-Cov-XCT column represent the MC/DC coverage percentage for transformed program by using transformation technique XCT. INC-Cov-XCT column shows the increase in coverage percentage. The average coverage percentage for BCT is 24.84 %. M-Cov-X-NCT column represents the MC/DC coverage percentage for transformed program by using transformation technique X-NCT. INC-Cov-X-NCT column shows the increase in coverage percentage. The average coverage percentage for X-NCT is 25.447 %. The increased coverage percentage from XCT to X-NCT is 0.607 % as Fig. 5.5

5.8.5 Analysis of Result

As can be observed from Table 4.2 and 5.6 we have compared our results in a bar graph shown in Fig. 5.6. Blue colured bar represent the level of MC/DC coverage percentage for original example program. None of the programs achieved 100 % MC/DC coverage percentage. Percentage varies from 52.9% to 75 % at the maximum. Red colured bar

Table 5.7: Coverage percentage for XCT and X-NCT technique

S.No	Program	M-Cov-XCT	INC-Using-X-CT	M-Cov-X-NCT	INC-Using-X-NCT
1	Triangle	100%	25%	100%	25%
2	Next Date	100%	29%	100%	29%
3	ATM	100%	30%	100%	30%
4	Library	100%	25%	100%	25%
5	TCAS	83.3%	30.4%	86.4%	33.5%
6	Schedule	87.5%	25%	89.6%	27.1%
7	Tic-tac-toe	91.6%	26.6%	92%	27%
8	Elevator	87.5%	23.6%	88.3%	24.4%
9	Tokenizer	88.2%	23.5%	87.3%	22.6%
10	Ptok2	83.5%	20.5%	84%	21%
11	Replace	81.5%	23.7%	83.7%	25.9%
12	Ptok1	88.3%	23.7%	88.3%	23.7%
13	Phonex	87%	23.3%	84%	23.5%
14	PrograSTE	86.6%	26.9%	88.2%	25.8%
15	ProgramTR	89.4%	22.4%	89.7%	25.7%
16	Sed	79.7%	21.2%	79.7%	21.2%
17	Grep	76.2%	22.4%	76.2%	22.4%

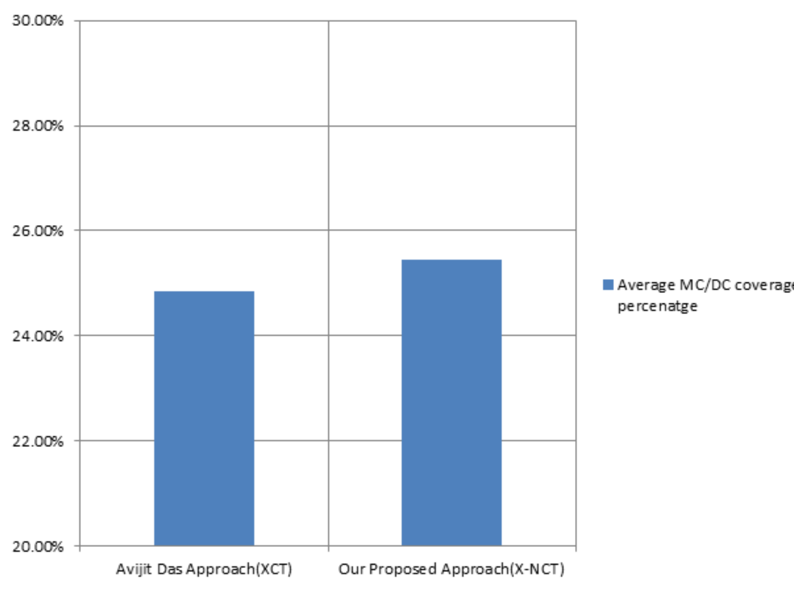

Figure 5.5: Comparison between transformation techniques XCT and X-NCT

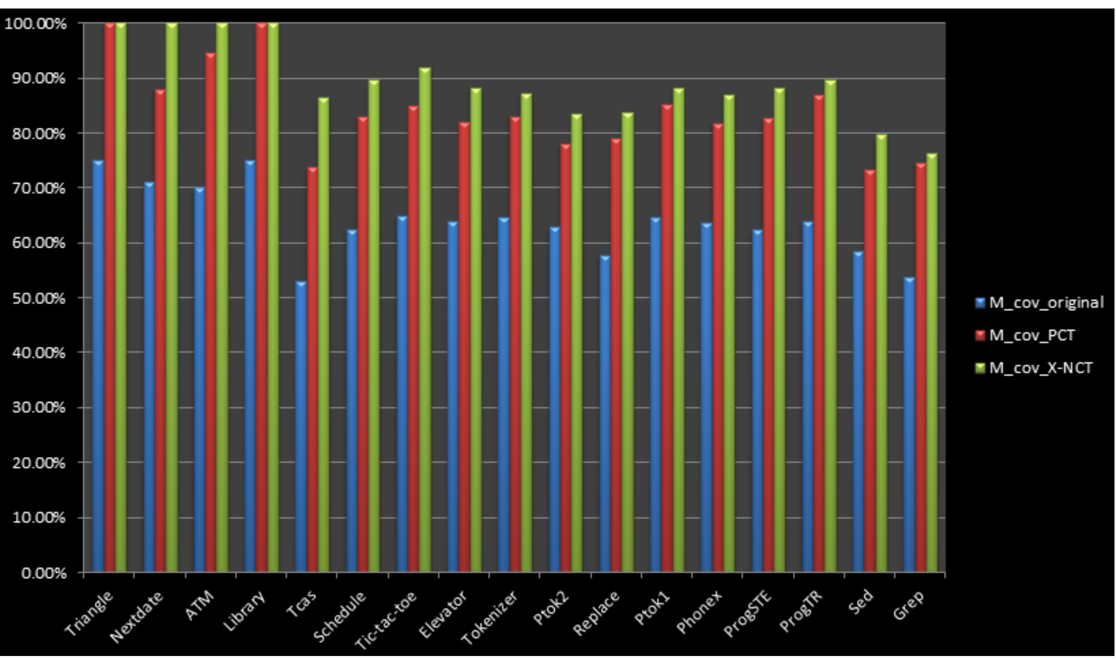

Figure 5.6: Analysis of all example program for Evaluated MC/DC coverage percentage

represent the level of MC/DC coverage percentage for transformed program using Program Code Transformer technique to achieve increase in percentage. Two programs (Triangle and Library) achieved 100 % as shown in Fig. 5.6. Percentage varies from 73.4 % to 100 %. Green colured bar represent the level of MC/DC coverage percentage for transformed program using Exclusive-NOR Code Transformer. Four programs achieved 100 % coverage as shown in Fig. 5.6. Percentage varies from 76.2 % to 100 %.

As we can observed we have compared our improved results for both the transforamation technique in Fig. 5.7. Blue colured bar represent the increase in MC/DC coverage percentage by using Program Code Transformer technique. Percentage varies from 15 % to 25 %. Red colured bar represent the increase in MC/DC coverage percentage by using Exclusive-NOR Code Transformer. Percentage varies from 21 % to 33.5 % as shown in Fig.5.7.

From Fig. 5.8 we analyse the average coverage percentage for seventeen example programs. First bar represent the average MC/DC coverage percentage by using Program

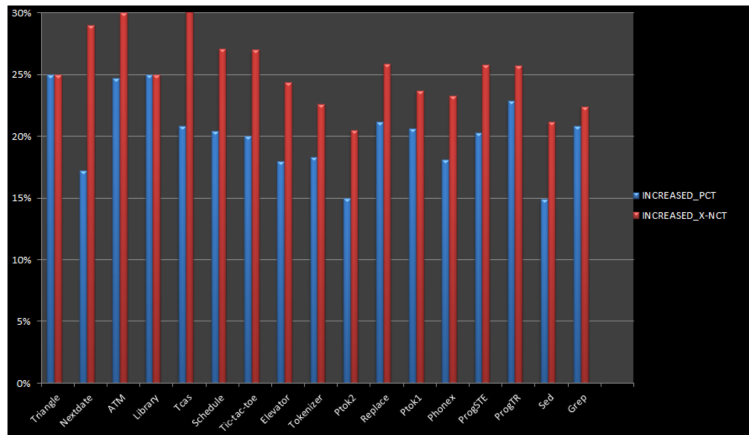

Figure 5.7: Increase in MC/DC percentage comaparision analysis for all example programs

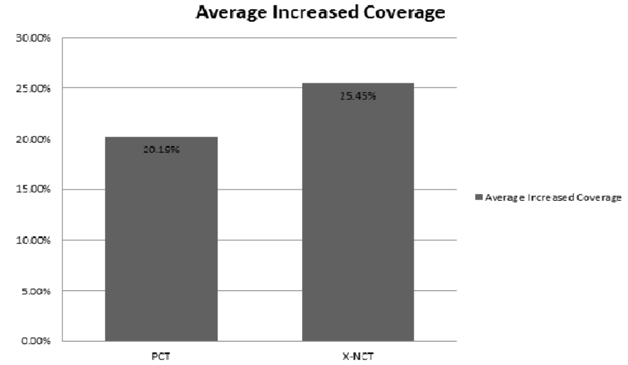

Figure 5.8: Comparison analysis for PCT and X-NCT

Code Transformer and the value is 20.19 %. Second bar represent the average MC/DC coverage percentage by using Exclusive-NOR Code Transformer and the value is 25.447%.

5.9 Conclusion

We proposed Exclusive-NOR Code Transformer technique to improve our MC/DC percentage. We achieved 25.447%. coverage percentage for seventeen programs. We discussed experimental study for our approach. As compared to other approach we achieved 0.607 % more.

Chapter 6

Conclusions and Future Work

In this thesis we have proposed a novel approach to automatically increase the MC/DC coverage of a program under test. Here we have presented an approach to automate the test data generation procedure to achieve increased MC/DC coverage. We have used existing CONCOLIC tester i.e crest tool with a code transformer based on sum of product (SOP) boolean logical concept to generate test data for MC/DC. In the following, we summarize the important contributions of our work. Finally, some suggestions for future work are given.

6.1 Contributions

In this section, we summarize the important contributions of our work. There are three important contributions, *Program Code Transformer*, *Exclusive-NOR Code Transformer*, and *Coverage Analyser*.

6.1.1 Program Code Transformer

Program Code Transformer follows four steps including minimization of sum of product by Tabulation Method. Code transformer gives an automated implementation of the boolean derivative method. Our experimentation on example programs show 21.06% average increase in MC/DC using our PCT approach. The average time taken for seventeen programs is 6.89950 seconds.

6.1.2 Exclusive-Nor Code Transformer

Exclusive-NOR Code Transformer based on exclusive nor (X-NOR) operation to generate test data for MC/DC. The advantage of our approach is that it achieves a significant increase in MC/DC coverage. Our experimentation on example programs show 25.447% average increase in MC/DC using our X-NCT approach.

6.1.3 Coverage Analyser

Also we have presented the coverage analyzer which calculates the coverage percentage after accepting original programs and test cases.

6.2 Future Work

We briefly outline the following possible extensions to our work.

- We are planning to extend the CONCOLIC Tester (CREST) to solve path constraints with float or pointer variables. It will then be possible for our approach to achieve 100% MC/DC coverage for most programs. In practice software developers and testers want to generate the minimum number of test cases so that the time and effort required for testing does not become an overhead. Therefore, a future version of our approach will have the option of selection of test cases so that the total number of test cases required to satisfy MC/DC can be reduced.

- Our work can be extended to compute MC/DC coverage percentage for a sliced version of program to improve more coverage percentage. We may use *CodeSurfer* tool to slice the program written in C language.

- Our work can also be extended in parallel distributed testing to increase scalability. We may use *SCORE-0.1.1* tool to resolve problem faced during use of CREST i.e bit vector calculation. In this concept we design client server architecture for CONCOLIC tester.

Dissemination of Work

1. Sangharatna Godboley, G.S.Prashanth, Durga Prasad Mohapatra and Bansidhar Majhi.Increase in Modified Condition/Decision Coverage Using Program Code Transformer, In proceedings of *2013 3rd IEEE International Advance Computing Conference (IACC) , Ajay kumar Garg College of Engineering Gaziyabad(U.P)*, Pages: 1401-1408, 22nd-23rd Feb 2013. IEEE Catalog Number: CFP1339F-CDR, ISBN: 978-1-4673-4528-6

2. Sangharatna Godboley, Sai Prashanth, Durga Prasad Mohapatra and Bansidhar Majhi.Enhanced Modified Condition/Decision Coverage Using Exclusive-NOR Code Transformer, In proceedings of *2013 IEEE International Multi Conference on Automation, Computing, Control, Communication and Compressed Sensing (IMAC4S), School of Electronics, St. Joseph's College of Engineering and Technology, Palai, Kottayam,India*, 22nd - 23rd March 2013. ISBN:978-1-4673-5088-4

3. Sangharatna Godboley, Durga Prasad Mohapatra, and Bansidhar Majhi.Evaluation of Coverage Percentage Using Exclusive-Nor Code Transformer and CREST Tool,Inernational. Journal of Computer Applications in Technology, Special issue on*Software Architecture, Evaluation and Testing for Emerging Paradigms*, 2013.(Communicated)

Bibliography

[1] R. Mall, *Fundamentals of Software Engineering.* New Delhi, India: PHI Learning Private Limited, 3rd ed., 2009.

[2] Rtca, "Software considerations in airborne systems and equipment certification," *October*, no. December 1992, 1992.

[3] N. Chauhan, *Software Testing Principles and Practices.* 2010.

[4] P. Ammann, J. Offutt, and H. Huang, "Coverage criteria for logical expressions," in *In 14th International Symposium on Software Reliability Engineering (ISSRE03*, pp. 99–107, IEEE Computer Society Press, 2003.

[5] R. Pandita, T. Xie, N. Tillmann, and J. de Halleux, "Guided test generation for coverage criteria," *Software Maintenance, IEEE International Conference on*, vol. 0, pp. 1–10, 2010.

[6] M. D. Hollander, "Automatic unit test generation," Master's thesis, Delft University of Technology, July 2010.

[7] R. Majumdar and K. Sen, "Hybrid concolic testing," in *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, (Washington, DC, USA), pp. 416–426, IEEE Computer Society, 2007.

[8] X. Qu and B. Robinson, "A case study of concolic testing tools and their limitations," in *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement*, ESEM '11, (Washington, DC, USA), pp. 117–126, IEEE Computer Society, 2011.

[9] M. M. Mano, *Digital Design.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 3rd ed., 2001.

[10] Z. Awedikian, K. Ayari, and G. Antoniol, "Mc/dc automatic test input data generation," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, GECCO '09, (New York, NY, USA), pp. 1657–1664, ACM, 2009.

[11] K. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, "A practical tutorial on modified condition/decision coverage," 2001.

[12] S. B. Akers, "On a theory of boolean functions," pp. 487 – 498, Journal Society Industrial Applied Mathematics, 7(4), December 1959.

[13] A. L. White, *Programming Boolean expressions for testability*, pp. 3110–3122. IEEE, 2004.

[14] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *In ESEC/FSE-13: Proceedings of the 10th European*, pp. 263–272, ACM, 2005.

[15] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, pp. 385–394, July 1976.

[16] M. Kim, Y. Kim, and Y. Choi, "Concolic testing of the multi-sector read operation for flash storage platform software," *Under Consideration for publication in Formal Aspects of Computing*, 2011. CS Dept. KAIST, Daejeon,South Korea and School of EECS, Kyungpook National University, Daegu, South Korea.

[17] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol. 9, no. 5, pp. 193–200, 1994.

[18] P. McMinn, "Search-based software test data generation: a survey: Research articles," *Softw. Test. Verif. Reliab.*, vol. 14, pp. 105–156, June 2004.

[19] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: automatically generating inputs of death," in *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, (New York, NY, USA), pp. 322–335, ACM, 2006.

[20] D. L. Bird and C. U. Munoz, "Automatic generation of random self-checking test cases," *IBM Syst. J.*, vol. 22, pp. 229–245, Sept. 1983.

[21] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, (Minneapolis, MN, USA), IEEE Computer Society, 2007.

[22] D. J. and S. Ntafos, "An evaluation of random testing," *IEEE Trans. Software Eng. SE-10*, pp. 438–444, july 1984.

[23] D. D. Cristian Cadar and D. Engler, "Klee:unassisted and automatic generation of high-coverage tests for complex system programs," in *Software Maitenance*, (San Diego, CA), In USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008), December 2008.

[24] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, (New York, NY, USA), pp. 213–223, ACM, 2005.

[25] "Crest. http://code.google.com/p/crest.."

[26] B. W. P. C. X. Liu ., H. Liu and X. Cai, "A unified fitness function calculation rule for flag conditions to improve evolutionary testing," (Long Beach, CA,USA), pp. 337–341, In proceeding of the 20th IEEE/ACM international conference on Automated Software Engineering, ACM, 2005.

[27] N. T. Rahul Pandita, Tao Xie and J. de Halleus, "Guided test generation for coverage criteria," in *In Software Maintainance*, IEEE International Conference, 2010.

[28] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, "A practical tutorial on modified condition/ decision coverage," vol. NASA/TM-2001-210876, May 2001. National Aeronautics and Space Administration , Langley Research Center Hampton, Virginia 23681-2199.

[29] D. R. Kuhn, "Fault classes and error detection capability of specification-based testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 8, pp. 411–424, October 1999.

[30] D. August, J. W. Sias, J.-M. Puiatti, S. A. Mahlke, D. A. Connors, K. M. Crozier, and W.-M. Hwu, "The program decision logic approach to predicated execution," in *Computer Architecture, 1999. Proceedings of the 26th International Symposium on*, pp. 208–219, IEEE, 1999.

[31] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, (Washington, DC, USA), pp. 443–446, IEEE Computer Society, 2008.

[32] A. Das, "Automatic generation of mc/dc test data," Master's thesis, IIT Kharagpur, April 2012.