Automatic Test Case Generation Using Modified Condition/Decision Coverage Testing

Komal Anand

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

Automatic Test Case generation

using Modified Condition/Decision Coverage Testing

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

Master of Technology

in

Computer Science and Engineering

(Specialization: Software Engineering)

by

Komal Anand

(Roll No. 212cs3119)

under the supervision of

Prof. Banshidhar Majhi

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

June 2013

dedicated to my parents and friends...

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

Certificate

This is to certify that the work in the thesis entitled ***Automatic Test Case generation using Modified Condition/Decision Coverage*** by ***Komal Anand*** is a record of an original research work carried out by her under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of Master of Technology with the specialization of Software Engineering in the department of Computer Science and Engineering, National Institute of Technology Rourkela. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Place: NIT Rourkela
Date: May 30, 2014

Prof. Banshidhar Majhi
Professor, CSE Department
NIT Rourkela, Odisha

Acknowledgment

I am grateful to everyone who supported me throughout my thesis work. I would like to specially thank *Prof. Banshihar Majhi*, Professor and *Prof. D.P. Mohapatra*, my supervisor, for his consistent encouragement, incalculable guidance and co-operation to carry out this project, and for giving me an opportunity to work on this project and providing me with a great environment to carry my work with ease.

I would like to thank all my friends Mayank Mittal, Jyoti Shivhare, Prerna Kanojia, Sumana Maity, Priyesh Munjpara for their moral and ethical support and lab mates Basanti Minj, Santosh Behera, Sarojkant Mishra, Pankaj Gupta, Vijay Sarthi, Vipul Makvana for their encouragement and understanding. They made my life beautiful and helped me every time when I was in some problem.

Most importantly, none of this would have been possible without the love and patience of my Parents and my Sister. My family, to whom this thesis is dedicated to, has been a constant source of love, concern, support and strength all these years. I would like to express my heart-felt gratitude to them.

Komal Anand
Roll: 212cs3119

Abstract

An automated test generation technique is used to reduce the effort for software test [1]. Modified Condition/decision Coverage (MC/DC) is a type of white box testing technique which is used to show the coverage by proving all the conditions are involved in the predicate can affect the predicate value. MC/DC is a standard condition/decision coverage technique. For automated test input data generation, we are using an advanced code transformer which is an improvement on Boolean code transformer and Program code transformer by using Modified Quine Mccluskey Method [2] for Sum of product Minimization technique over the tabular method as well as the Quine Mccluskey method. By which the number of comparisons reduces and helps to achieve the increased MC/DC Coverage.

In this research work, we represent the coverage analysis for evaluating Modified Condition/Decision Coverage percentage. Basically, this research work is based on three modules. First module shows the coverage percentage analysis for Advanced Program Code Transformer (APCT). APCT is the modified version of Program Code Transformers. APCT uses modified Quine McCluskey method which is an optimization of Quine McCluskey method based on E-sum used for minimization of sum of product by which number of comparisons reduces. Second module shows the coverage analysis for the CONCOLIC Tester CREST Tool. Third module shows the analysis for Coverage Analyzer. In this paper, we have experimented 6 complex C programs and achieved variation of 3.81% average MC/DC coverage percentage after comparing with program code transformer(PCT) and Advanced program code transformer(APCT).

Keywords: *Software Testing; Coverage Analyser; Concolic Tester; Advanced Program Code Transformer; MC/DC*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Chapter 1

Introduction

1.1 Introduction

Software pervades everywhere in our society like Education field, Medical field, Business, Communication field and almost in every field. All the devices in every filed needs software, it is an essential part. A software faces various challenges, as its demand increases its complexity also increases. In order to raise the quality, the testing of software is required. That is why software testing is a very important stage in the software development cycle.

Software testing is a process of finding errors and a practical way to reduce defects and improve the reliability, dependability and quality of a software system [1]. If we discover more bugs in the early stage of the software development, then the software will be more successful. It is being evolved in the form of a process which is based on the strategies efforts. It is a process to evaluate software under various conditions to compare its results with the expected results. A software system is normally tested in these levels or strategies:

• **Unit testing:** It is a kind of root level testing. Programmers carry out this testing immediately after completing the coding of a single module. It consists of testing code modules.

• **Integration testing:** According the the integration plan, different modules are integrated, After different modules of a system have been coded and unit tested, to determine if they interface properly with each other.

• **System testing:** A system is fully developed and then tested on the basis of its requirements. The test cases are designed solely based on Software Requirement Specification document.

• **Acceptance Testing:** The customer itself test the system in order to accpet and reject the system.

1.2 Types of testing:

There are mainly two types of testing strategies:

(A) Black Box Testing: In order to design the test cases, Only functional specification is required. The internal structure of software is not considered. In this approach, we give input and see the output and do not see the internal process behind it.

(B) White Box testing: White box testing is also called as Structural testing or Glass box testing. To design white box test cases, knowledge about the internal structure of software is required. In this we take input and produce output and consider the internal coding of the software. It is done by the developers

There are various coverage criteria for white box testing: Let us discuss them using an example of simple c program code,

1. **Statement coverage:** In statement coverage every bug can be detected when all the statements of a module are executed at least once. To cover all the statement in the above program, two test cases are designed: (1) m=n=a, where a is any number, and (2) m=a, n=b. In test case 1 the loop will not get executed while in test case 2 the loop will be executed. Here two more test cases are also designed (3) m>n and (4) m<n. Therefore it is not a better coverage criteria because test case 3 and test case 4 are sufficient to execute all the statements in the code, but if we consider them

```
scanf(``%d%d'',&m, &n);
while(m!=n)
{
        If(m<n)
                n = n-m;
        else
                m = m-n;
}
printf(``m='', m);
printf(``n='', n);
```

Figure 1.1: A sample C program

only the condition and path of test case 1 will never be tested and all the errors cannot be found.

2. **Branch Coverage:** Each decision should take all possible outcomes at least once either true or false. In this case the test cases are (1) m>n, (2) m<n, (3) m=n, (4) m! =n,

3. **MC/DC Coverage:** It shows that all the conditions in a decision affect the output of the decision independently, and enhances the condition and decision coverage criterion.

4. **Modified Condition Coverage:** All possible outcomes of a decision are taken in order to invoke all entry points at least once.

1.3 Problem Definition

1.3.1 Automated Testing

We gave so much effort in the testing process in the whole software development process by repeating software tests, but manually repeating these steps is time consuming as well as costly. Once automated tests are created, we can run it over and over again without any extra cost and time as they are much faster than

the manual tests. There are two approaches to automate the testing process.

The **First** approach is to write scripts with all the test cases. This can be useful for the techniques where tests are performed repeatedly like regression testing by making a small change to ensure that this change will not affect the functionality of the system, but this can be costly to test all the tests again manually.

The **Second** approach is to design an automatic test case generation tool and run them on the system or program to be tested. Such a tool has long been considered critical, but once developed we can run it over and over again without any extra cost and time. Thousands of various complex test cases during every test run can be easily executed by automated software tests and providing better coverage that is impossible with manual.

1.3.2 Motivation

There are many approaches used for test case generation using branch coverage. But large and complex software like safety critical systems are required to satisfy the Modified Condition/ Decision coverage criteria so that a software can get a DO-178B Level A Standardize certification [3]. Hence it is necessary to develop test cases for MC/DC also by using various automated approaches to achieve MC/DC coverage.

1.3.3 Objective

There is an existing approach of CONCOLIC Testing which is used to achieve Branch Coverage [4] [5] [6]. In order to attain MC/DC Coverage we are trying to extend the CONCOLIC Testing technique. Our main objective in the approach is to attain MC/DC coverage and to generate test data to achieve the coverage. Hence we are using concolic testing to achieve better coverage, which was first used

to achieve the branch coverage. And we are using it to achieve MC/DC coverage with our code transformer.

In our approach, we are using advanced program code transformer which is a modified version of existing Boolean code transformer in which the Modified Quine McCluskey method [2] is applied in place of QuineMcCluskey method. It reduces the number of comparisons and helps in achieving better coverage percentage. And this code transformer gives various new branches which helps in concolic testing as it is made for branch coverage.

1.4 Organisation of thesis

The thesis is organized as follows: Chapter 2 discusses the basic concepts and definitions related to the thesis work. Chapter 3 describes the literature review done for this thesis. Chapter 4 discusses the main approach which is proposed, it is based on the advanced program code transformer. Chapter 5 Shows the implementation and experimental study of the Thesis work. Finally chapter 6 gives the summary of the whole thesis

Chapter 2

Chapter 2

Basic Concepts

In this chapter we are discussing about various basic terms related to modified condition/decision coverage testing [7], the definition of general terms like condition, decision, group of conditions etc, various criterion related to modified condition/ decision coverage and Boolean derivatives and the approaches for symbolic testing and concolic testing.

2.1 Basic Definitions:

Condition: A condition or a clause is a Boolean Expression without any Boolean Operator. It can not be broken further into more Boolean expressions [8].
Decision: A decision is a Boolean Expression composed of one or more conditions within a decision. Condition with Boolean operator is decision [8].
Group of Conditions: A predicate or decision is composed of two or more conditions with many Boolean Operators [8].

Let us take a statement S in a program
S= A OR (B AND C)
Here, A, B, C are conditions and S is a Decision statement
Condition Coverage: Each condition in a decision should take all possible outcomes at least once [3].
Decision Coverage: Each decision should take all possible outcomes at least one either true or false [3].

Table 2.1: Truth Table for A∨ B and minimum test fora decision with 2 conditions

	A	B	Output	A	B
1	T	T	T		
2	T	F	T	4	
3	F	T	T		4
4	F	F	F	2	3

2.1.1 Modified Condition/Decision Coverage:

It is a criterion for code coverage which was introduced by RTCA DO-178B standard [3]. It is a critical (level A) software, an improvement on Modified Condition Testing by overcoming its disadvantage like the linear growth of test cases is maintained [9].

It shows that the output of the statement must be affected by all the conditions in a decision statement. MC/DC must satisfy the following criteria:

1. All the points of entry and exit in the program must be invoked at least once.
2. All possible outcomes of a decision must be affected by each condition.
3. All possible outcomes of every decision must be exercised.
4. All the conditions in a decision must be exercised.

Consider an expression A OR B

In order to understand MC/DC technique, let us take a Boolean predicate and its schema.

Table 2.1, Consider the expression A∨B, for 2 variables we have 4 combinations and outputs. MC/DC Considers only those pairs of test cases in which the output is changing by changing only one condition.

For(A OR B)

independence of A: Take 2 + 4

independence of B: Take 3 + 4

Resulting test cases are : 2+ 3 + 4

(T , F) + (F , T) + (F , F)

2.1.2 Boolean Derivative

Kuhn and Ammann et al. [9] proposes an approach to solve the problem of solving the determination which all the predicates are evaluated independently for each clause. Boolean derivative method is good when the same clause are occurred many times explicitly. Let $P_{m=true}$ for every occurrence of m is true and $P_{m=false}$ with occurrence of m is false, where p is a predicate with m as clause. therefore, clause m occurs neither in $P_{m=true}$ nor in $P_{m=false}$. Now, these two expression for true and false values of predicate are combined with the Logical EXOR(Exclusive-OR) operation.

$$P_m = P_{m=true} \oplus P_{m=false}$$

so, we can say that P_m describes the exact conditions under which the value of m determines value of P. If the values for the clauses in P_m are taken so that P_m is true, then the truth value of P can be determined by the truth value of m. If the clauses in P_m are taken so that P_m evaluates to false, then the truth value of m does not affect the truth value of P.

Example: Consider the statement,

$$\mathrm{P} = \mathrm{m} \wedge (\mathrm{n} \vee \mathrm{r}) \qquad (2.2)$$

If m is the major cause, then the Boolean derivative finds truth assignments for n and r as follows:

$$P_m = P_m{=}\text{true} \oplus P_m{=}\text{false} \qquad (2.3)$$

$$P_m = (\text{true} \wedge (\mathrm{n} \vee \mathrm{r})) \oplus (\text{false} \wedge (\mathrm{n} \vee \mathrm{r})) \qquad (2.4)$$

$$P_m = (\mathrm{n} \vee \mathrm{r}) \oplus \text{false} \qquad (2.5)$$

$$P_m = \mathrm{n} \vee \mathrm{r}; \qquad (2.7)$$

A deterministic answer can be obtained, three choices of values make n $\vee$ r = true,

(n = r = true), (n = true; r = false), (n = false; r = true).

2.1.3 Symbolic testing

Symbolic Testing generates test data by using symbolic execution and this execution do not take concrete values for assigning as the program variables but it takes the symbolic expression. The main approach is to derive the constraints which describes the necessary condition for execution of certain path. The input variables comes as the solution of these constraints For system under test, we collect the path constraints in symbolic testing and these path constraints are solved using constraint solver. The solution represents the concrete test data that executes these paths.

2.1.4 Concolic Testing

In a sequential Program,Concrete inputs are generated randomly then these input tester executes the code as well as at each branch point along execution path [10], tester collects constraints the symbolic values, then at the end when execution stops, the sequence of symbolic constraints corresponding to each branch point is collected by the tester. The conjunction of such constraints are path constraints. Tester takes constraints value form path constraints and negate it in order to find the next oath constraint and then tester finds some concrete value for this new path constraints being constraint solver.

For Example, Let us take a function WEIGHT [11] and parameters for mass and weight are set randomly as 22 and 5.0. Now the program is executed with generated random inputs and performs concolic testing . Both the values concrete and symbolic are collected for executed path while execution. The first if statement is executed when first branch instruction is encountered.
The mass is taken as 22.0 the branch predicates evaluates to true.

```
1 static int weightcategory (doublemass, double length) {
2       if(mass>0.0)
3         continue;
4       else
5         return;
6       if(length>0.0)
7         continue;
8       else
9           return;
10
11      double bmi =mass/(length*length);
12
13      if (bmi<18.5)
14          return underweight;
15
16      else if(bmi<25.0)
17        return normal;
18      else
19          return overweight;
20      }
```

Figure 2.1: A program to show Concolic Testing.

Example, Then 2nd if statement is executed and next branch will be find. Then This length is set to negative value then the branch predicate evaluates to false. Then the branch constraints should be ¬(length >0.0) Then we find new path by combining it with previous constraint (mass>0.0)∧(lenght>0.0).
The function will return after execution of else statement corresponding to second if statement. For path, the one branch constraints will negate by which the path constraints will changed. When the last branch constraints negated, the changed path are (mass>0.0)∧(lengh>0.0).

In order to determine the input which makes constraint true, a test data length= 1.0 and mass =50.0 is the solution which satisfies the constraints. Again with this input, the function will executes but the path constraints are collected again. The function will return overweight category with this input. Then the execution path have the following constraints

(mass>0.0)∧(lenght>0.0)∧ ¬(bmi<18.5)∧ ¬(bmi<25.0)

Then this whole process will continue till we meet the stopping criteria . And stopping criteria can be when we obtain the proper code coverage or the number of iteration exceeds the threshold. Suppose, when there are no inputs are existing which satisfies the constraints and constraints are not feasible, the constraint solver will not be able to compute test inputs for a path.

Chapter 3

Chapter 3

Related Work

In this chapter, we are presenting a literature survey related to the research work in the area of Modified Condition/Decision Coverage testing and automated Testing.

3.1 Test Generation for Branch Coverage

Concolic testing technique is used to generate the test cases that is used for branch coverage [4]. Similarly there are various testing techniques which are used for branch coverage like search based , symbolic testing and random testing.

3.1.1 Random Testing

Random Testing is a simple and effective technique for Automated Testing. We can generate a large number of independent inputs randomly and run the program using these random inputs. The results are then compared with a system specification. The test is a failure if any input leads to incorrect results. The Random Test inputs can be generated in the negligible time using Random Testing. But, it can not test all possible behavious of the program and it do not provide better code coverage.

3.1.2 Symbolic Testing

Its a technique proposed by King [12]. In this technique concrete values are not assigned to the program variable, in place of that a symbolic expression is assigned. The technique is used to get some constraints that show some conditions used for execution of certain path and gives some solutions. This solution is then given as an input variable and in object oriented software. The path constraints are collected for some application, and solved using a path constraint solver . The concrete test data which executes these paths will be shown by the solutions solved using constraint solver.

3.1.3 Concolic Testing

CONCOLIC [12] is a combination of CONCrete and symbOLIC techniques [13], which performs symbolic and concrete execution simultaneously. CONCOLIC (CONCrete + symbOLIC) testing [13] (also known as dynamic symbolic execution [Tillman et al. [14] And the white - box fuzzing [6]) combines concrete dynamic analysis and static symbolic analysis to automatically generate test suite to explore execution paths of a target program. Therefore, it is necessary to check if CONCOLIC testing [13] can detect bugs in open source applications in a practical manner through case studies [15] [16]. In this technique, path constraints are collected at the time of concrete execution of the system under test. As soon as the execution finishes the path constraints are modified. The solution of this modified constrained will be used again in order to find another path and the process continue till we reach to a stopping criteria. The stopping criteria can be number of iterations exceeding a threshold or when a sufficient code coverage is obtained.

3.2 Literature Review

Bokil et al. [4] Have given a AutoGen tool which automatically generates test data for C code which helps in reducing the cost and effort for test data preparation. There are various coverage criterion like statement coverage, decision coverage, or Modified Condition/Decision Coverage (MC/DC) with these coverage criterions Autogen takes the C code as input and generates test data that are non-redundant and satisfies the specified criterion.

Das, A. et al. [17] has given an approach for augmentation of MC/DC test case generation. The approach deals with automatic generation of MC/DC test suite. The author proposed the concept by presenting Boolean Code Transformer (BCT). BCT is based on Karnaugh map minimization technique.

Godboley, S. et al. [7] proposed another approach to enhance MC/DC using exclusive- nor code transformer. The approach reduces the effort of minimizing the sum of product by simple X-NOR operator. This approach overcomes the disadvantages of old concepts.

Godboley, S. et al. [17] proposed an approach to increase MC/DC using a program code transformer. Program Code Transformer was based on the Quine-McCluskey minimization method. The objective of the paper was to automatically generate MC/DC [18] test suite.

Vitthal et. al. [2] proposed an approach called Modified quine Mccluskey (MQM)method. By using MQM method performance of digital circuits can be increased by reducing the number of min-literals or minterms in Boolean Expression. Algebraic approach is used to reduce the number of comparisons between

minterms while E-sum is used to eliminate repitition. MQM is much simple and faster than Quine McCluskey method due to less number of required comparisons. Thus the method can be used to achieve speed in minimizing the Boolean function manually and improve performance of conventional method [17].

Chapter 4

Chapter 4

Proposed work

In this chapter, we are discussing our proposed work automated test generation using advanced program code transformer for Modified Condition/Decision Coverage (MC/DC). We are giving the formal definition and detailed description of various modules used in the proposed approach like Program Code Transformer, Concolic Tester and Coverage Analyser.

4.1 Formal Definitions

To achieve structural coverage is our primary purpose on a given program under test (PUT) with respect to a given coverage criterion (C). An automatic tool ω for test generation is used in order to achieve coverage in the context of other coverage criterion C'.

Therefore, we transform PUT to PUT' in a way that the problem to obtain structural coverage in PUT with respect to C is converted into the problem to attain structural coverage in PUT' with respect to C'.

There are few terms defined below used in our approach:

COVERAGE (C, P U T, TS) [2] It shows the percentage that Test Suite(TS) achieves the coverage over a given program under test (PUT) with respect to given coverage criteria (C).

OUTPUT (P U T, I) It shows the output result of a program code under

test (P U T) subject to an input (I).

(P U T $\vdash$ TS) It shows that the tester tool ω generates a test suite (TS) for the program code under test (P U T). We have to transform P U T to PUT' where PUT' = P U T+R for a given PUT and R is the code added to PUT such that the following requirements are met.

Req1: $\forall$: [Output(P U T, I)=Output(P U T',I)], where I is input set for P U T. When the results of PUT and PUT' violates for same input I then, PUT' will have side effects.

Req2: If the tester tool ω generates the test suite TS' from PUT', then $\exists$ TS'[((PUT' TS') Coverage(C', PUT', TS') = 100%) (Coverage(C, PUT, TS') = 100%)] The requirement states that if there exists a test suite TS' that achieves 100% coverage on PUT' with respect to C', then coverage of TS' on PUT with respect to is 100%.

4.2 Proposed Approach

We gave a name as Advanced MC/DC Tester to our approach and our work is based on three modules. Advanced Program Code Transformer, Concolic tester and Coverage Analyzer.

In the whole process we take program code as input and then insert it to the code transformer. We are using an advanced program code transformer which produces the transformed program; the transformer modifies the program by adding some condition statements.

We are using advanced program code transformer in which a program is identified with various conditions which shows the predicates and then it is simplified into simpler predicates as it can be complex in several programs and generate nested-if else condition.

Concolic tester takes the transformed program as input and determines the feasible paths and test inputs by determining its feasible branches that can be reached in a program.

Then the third module is coverage analyser which takes the original program that is test input and test output of input files as input and calculates MC/DC coverage by using coverage calculator.

These three modules in our approach can be discussed further in this chapter:

4.3 Advanced Program Code Transformer:

The code transformer module in our approach is named as advanced program code transformer. In this module, the process consists of various steps. APCT takes input program i.e. C program and identifies the predicate, these predicates are need to be simplified, so it generates sum of product and then simplify it, using modified Quine mcCluskey method. Then these simplified conditions are used to find the various branches of program and decompose it into simpler conditions with empty true and false branches and these conditions are inserted into the original predicate.

The reason why empty true and false branches are inserted is to avoid the execution of duplicate statement which can be the original predicate and predicates after the transformation. And it is helpful in order to retain functional equivalence of a program after that, in generation of additional test cases for increased MC/DC coverage.

The algorithms for various steps in APCT are given as:

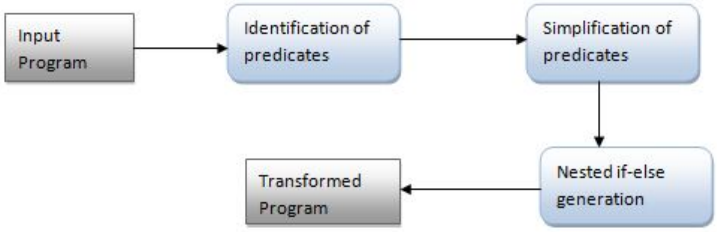

Figure 4.1: Code Transforming Steps.

Algorithm1: To obtain advanced Code Transformation

Input: PUT // PUT is the program under test in C syntax

Output: PUT' // PUT' is the transformed program

Begin

/* *Identification of predicates* */

for each statement s ∈ PUT

do

{

if (&& or ||) occurs in s

{

then

1 Predicate_List ← add-to-List(s) // List of predicates

end if

}

end for

}

/* *Simplification of predicates* */

{

for each predicate p Predicate_List do

2 P_ SOP ← gen_sum_of_product(p)

// Generates in the form of SOP expression

3 P_ Minterm ← Convert_to_Minterm (P_SOP)

// Converting in the minterm form

4 P_Simplifeid ← Mini_ SOP_ MQM (P_Minterm)

// Minimizes the SOP

end for

}

/**Nested if–else Generation* */

5 List_Statement ← generate_Nested_If–else APCT(P_ Simplified)

// Generating conditional statements

6 $PUT' \leftarrow$ insert_ code (List_Statement,PUT)

// Forming in the form of C syntax

end for

7 return PUT'

4.3.1 Identification of predicates:

Line 1 Represents the identification of predicates which scans the input program and identifies all the predicate and these predicates are added to a list. Predicates are the conditional statements with Boolean operators, so the process scan for Boolean operators like && (AND)and || (OR) operators.

4.3.2 Simplification of predicates:

The predicates are identified can be complex, so they need to be simplified and simplification of predicates involves two steps

(a) SOP Generation:

In algorithm 1 line 2, represents generation of SOP in which predicates are passed which are identified in the first step. And these predicates are connected into sum of Product (SOP). We are using SOP instead of POS because the whole structure should be in AND operator condition which is not flexible to the standard format and for the OR operator conditions, the structure of POS will be failed.

(b) SOP Minimization:

The predicates can be complex forms and can be redundant, so the are need to simplified or minimized and for which various minimization technique is use. Like k-map Quine Mccluckey method and Modified Quine McCluskey method. Lines 3-4 in Algorithm1 calls another Algorithm 2 which minimizes the generated SOP expression. For that we use modified Quine McCluskey method or Tbulation

method. There are some other techniques to minimize SOP like Quine McCluskey or K-Map method, but we use MQM which has advantages to overcome the problems of other techniques. The proposed algorithm uses E-SUM and algebraic approach to reduce number of comparisons between mintermlist. E-SUM is used to keep track of all eliminated variable in mintermlist or Boolean term. E-SUM based MQM algorithm is presented using following step-by-step approach:

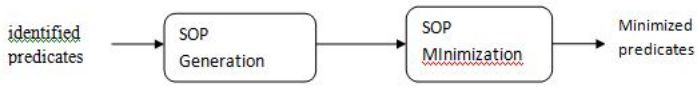

Figure 4.2: Sum of Product Minimization.

Algorithm2: Minimization of SOP Modified Quine-McCluskey Method

Input: p_Minterm

Output: p_Simplified

1. Transform the given Boolean function into canonical SOP form and obtain binary notation for each minterm.

2. All the minterms are arranged into groups according to number of 1's in their binary notation. All minterms in one group should contain same number of 1's. Then initialize E-SUM of all minterms to 0.

3. Compare mintermlist in adjacent groups according to MQM matching principal. Use algebraic approach to reduce number of comparison between mintermslist in adjacent group. In algebraic approach mintermlist in nth group having least minterm as x is compared with all minterms in (n+1)th group having least minterm as x+2p where p=0,1,2,3..so on. Once there are any two minterms of nth and (n+1)th group satisfying MQM matching principal and having least minterm as x and y respectively. Then combine the two minterm list by taking E-SUM of resulting minterm list as"E-sum of combining minterm list + Current MPW (i.e. y-x)". Checkmark (✓)

minterm lists which can combined to form new minterm list. Now repeat the same procedure for all other minterm list.

4. Eliminate repeated or identical mintermlist from combined mintermlist in all group. Two mintermlist in same group become identical if their corresponding E-SUM, least minterm and largest minterm are equal.

5. Repeat step described in 3 and 4 to minimize given Boolean function until it is impossible to combine minterm list.

6. Collect all non-checked (i.e. not ✓marked) mintermlist as prime implicant.

7. Now the redundant prime implicants are removed with the help of prime implicant chart in Quine Mccluskey method.

4.3.3 Nested if else generation

This is the last step in algorithm 1, in which some statements are generated which are additional conditional statement and thus statements are combined with original program statement. Nested if else algorithm scans all the if-else statements in C-syntax and identified the conditions in a group of conditions which are connected with && and || conditions. If first condition is satisfied then make an statement with that conditional and then next as else statement, similarly for each condition in a group, a if statement and corresponding else statement is created. This ensures that each condition is evaluated for both true and false values and this process is repeated for the simple conditions if they are the part of statement in the program.

Algorithm3: generateNestedIfElse.

Input: p // minimized SOP predicate p

Output: List_Statement //list of statements in C syntax

Begin

{

```
   for every && and || operator connected group of condition ∈ p
do
{
    for all the condition s1 ∈ group of condition
do
{
     if s1 is the first condition then {
1        make an if statement k with s1 as the condition
2        List_Statement ← add-to-list(k)
     else
{
3        make a nested if statement k with s1 as the condition
4        make an empty Truebranch Tb and an empty False branch Fb in order,
5        List_Statement ← add list(strcat(k,Tb,Fb))
     end if
}
    end for
}
6        make an empty False branch Fb for the first condition
7        List_Statement ← add list(Fb)
  end for
}
  for each condition ∈ p any group of condition
do
{
8        repeat lines 1, 4 and 5
  end for
}
                    if p is an else if predicate then
9        make an if(false) statement s
```

```
10      make an empty Truebranch Tb
11      List_Statement ← add list(strcat(k,Tb))
  end if }
12      return List_Statement
}
```

The above algorithm takes minimal predicates as input and produce the list of statements in the if- else in C syntax as the output. This algorithm finds the group of conditions which are connected with boolean operators like && or || operators, and identifies the condition. As soon as the first condition is identified, the algorithm makes an if statement with its condition and add this to the statement list. Similarly, whole program gets scanned for all the conditions connected with && or || operators. For each true and false values for each evaluated condition creates if and its corresponding else conditions.

4.3.4 CONCOLIC Testing

The program code under test are transformed from advanced program code transformer and these transformed program are then passed to the concolic tester tool that is CREST tool. Through the random test generation , the tester achieves the branch coverage. This tester is called as Concolic Tester. A concolic tester is that which performs concrete as well as symbolic testing. The extra generated expressions lead to generation of extra test cases for the transformed program. The identical test suites may not be generated because of random strategy in which different runs of concolic testing is done. The generated test cases depend on the path on each run. The test cases are stored in input text files which form a test suite.

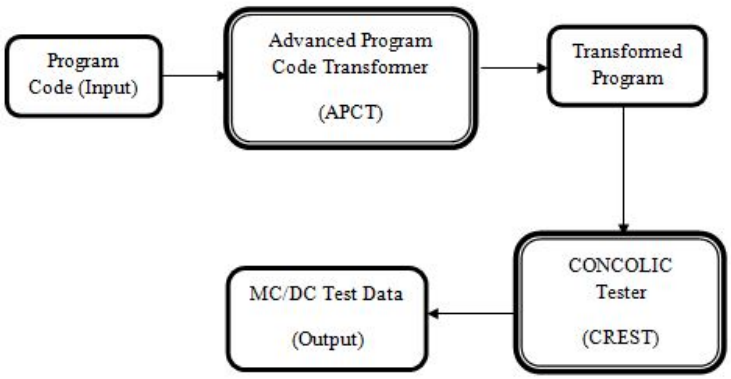

Figure 4.3: Process to generate test cases using tester.

4.3.5 Coverage Analyzer

Coverage analyzer determines the coverage percentage attained by generated test cases. It determines the coverage percentage achieved by test cases. We need to calculate the extent to which a program feature has been performed by the test suite. It also finds inadequacy of test cases and provides an insight on those aspects of an implementation that have not been tested. In our approach, it is essentially used to calculate if there are any changes in coverage performed by the test suite generated by the CREST TOOL using our approach. The entered program to test and the test data generated are passed to the coverage Analyzer. Coverage Analyser (CA) evaluates 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 following formula:

$$MC/DCCoverage = \frac{\sum_{i=1}^{n} I\_i}{\sum_{i=1}^{n} C\_i} * 100\% \tag{4.1}$$

Algorithm4:MC/DC COVERAGE ANALYSER

Input: P,Test Suite // Program P and Test Suite obtained

Output: MC/DCcoverage // % MC/DC achieved for P

Begin

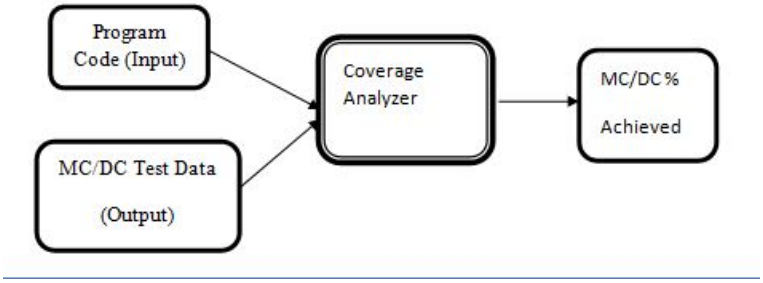

Figure 4.4: Coverage Analyzer.

*/*Identification of predicates*/*

for each statement s∈P do

if && or ||occurs in s then

1. Predicate_List ← add-to-List(s)

end if

end for

/ Determine the outcomes */*

for each predicate p∈ Predicate_List do

for each condition c∈p do

for each test case tc ∈ Test Suite do

if c evaluates to TRUE and calculate the outcome of p with tc

then

2. True Flag←TRUE

end if

if c evaluates to FALSE and calculate the outcome of p without td then

3. False Flag←TRUE

end if

end for

if both True Flag and False Flag are TRUE then

4. $List_I$←add-to-List(c)

End if

5. $List_c$ ←add-to-List(c)

end for

end for

/* *Calculate the MC/DC coverage percentage* */

6. MC DC COVERAGE $\leftarrow$ (SIZEOF($List_I$)_SIZEOF ($List_c$))X 100%.

In algorithm 4, There is a predicate identifier, which identifies the predicates which are read with the test data td and check whether the test data makes all the conditions in a predicate both true and false, as well as check whether conditions independently determine the predicate outcome. finally, when the number of conditions are identified which are independently affecting the outcome together with the total number of conditions in each predicate then, it is passed to the coverage calculator to calculate the MC/DC coverage percentage.

Chapter 5

Chapter 5

Experimental Study

5.1 Experimental Study

We are using an open source Concolic tester tool i.e. CREST for our experiments. The main objective of our experiments is to determine better MC/DC coverage. We have used several programs to test the coverage. Here we are showing the implementation using an example for triangle C program as in figure 5.1. The different modules are:

5.1.1 Advanced Program Code Transformer

Nested if-else generator and code inserter, he predicate identifier scans the program to identify the predicates, SOP generator module takes a predicate and convert it to sum-of-product(SOP) form using Boolean algebraic laws and SOP minimizer module then simplify the predicate using modified Quine Mccluskey method and then nested if else generator breaks the simplify predicates into simple conditions and passes these conditions to code inserter modules which inserts these conditions into the program before the location of predicate. This process is repeated for all the predicates.

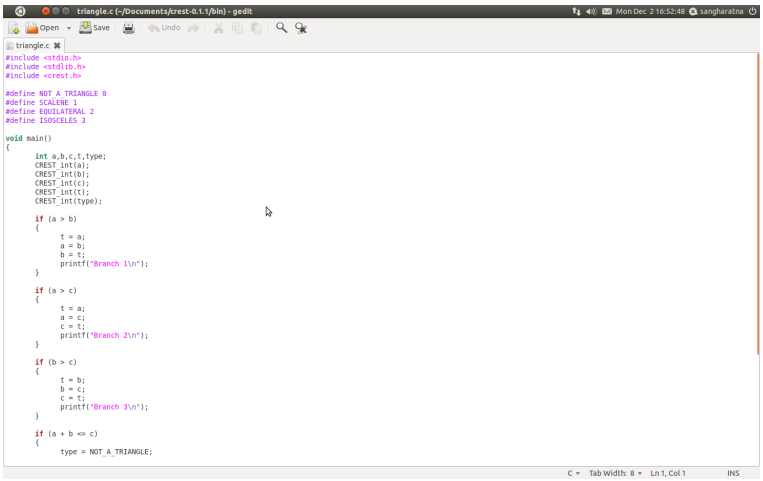

Figure 5.1: Triangle Program in C

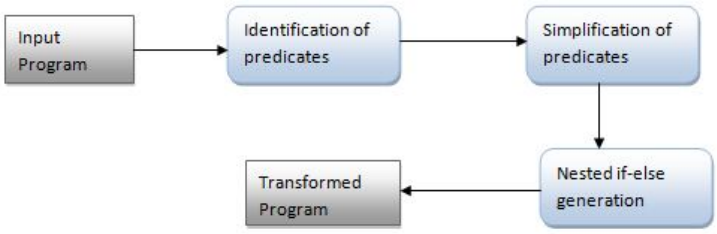

Figure 5.2: Program code transformation using Modified quine McCluskey

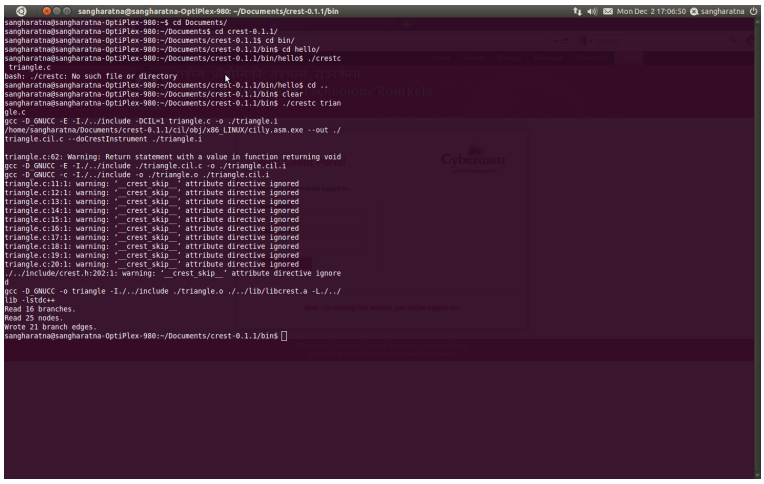

Figure 5.3: Output after crest Compilation

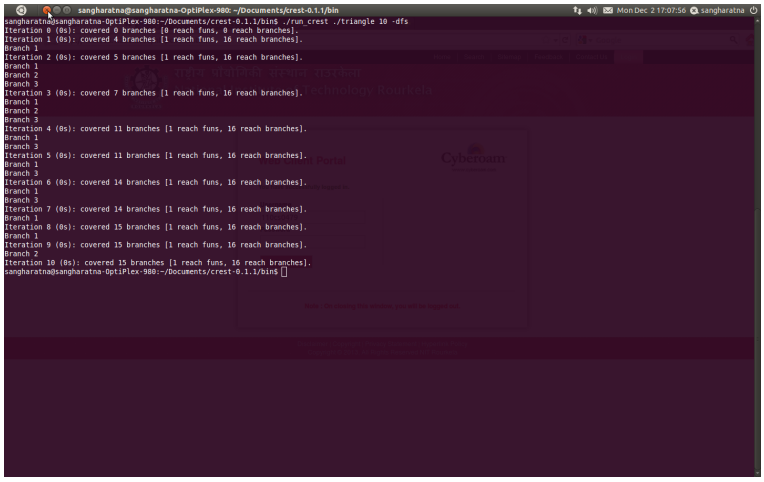

Figure 5.4: Output for number of iterations generated by CREST in DFS

5.1.2 Concolic Tester:CREST:

Crest is concolic tester; it performs symbolic execution and concrete execution together. The symbolic constraints which are generated are solved to generate input. There are two main strategies are used in CREST are Depth flow search and control flow directed search. The Figure 5.4 represents the DFS search for triangle program. Figure 5: shows the compilation for the test program which number of nodes(vertices) in the program, number of branches in the program, and the number of branch edges remaining in the graph (CFG)as shown in figure 5.4.

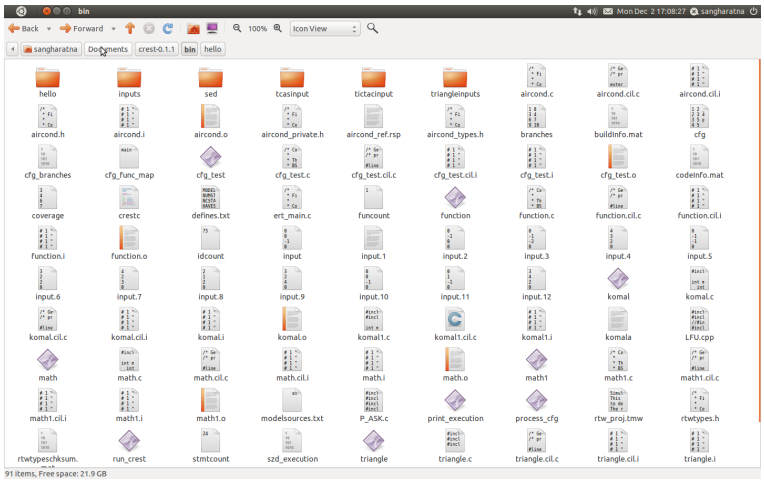

Figure 5.5: Input Test files

There are various number of input test cases are generated and the test cases may vary depending on the path along which the CONCOLIC execution starts in each run. The generated cases are stored in text files which form a test suite as in figure 5.5

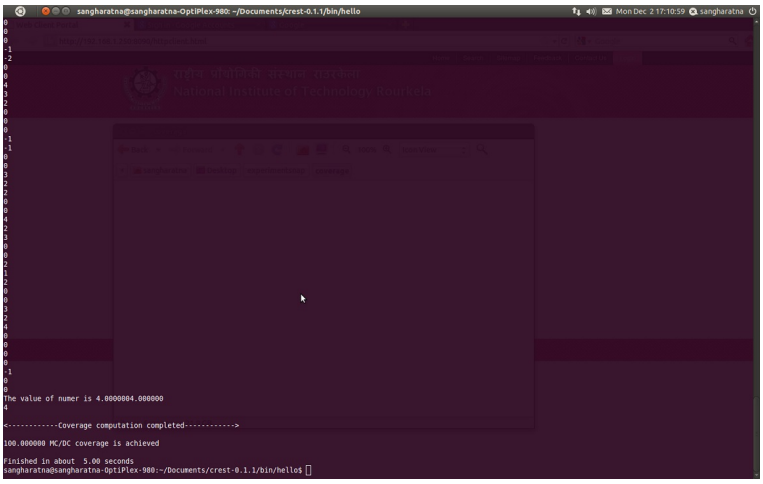

Figure 5.6: Coverage Percentage Achieved

5.1.3 Coverage Analyzer:

Further these test data with the program under test are passed to the coverage analyzer then these test datas examine the extent to which the independent effect of component conditions on the evaluation of each predicate by the test data takes place. And tries to achieve the more coverage percentage.
There are four modules in coverage Analyzer: Predicate identifier which is same as the Advanced Program Code Transformer, Test Suite reader module that reads each test data and passes it to the effect analyzer module, The effect analyzer which then reads each identified predicate and test data and then checks whether the test data makes each condition in a predicate both true and false and identifies conditions which have independent effect on predicate outcome and passes to the Coverage calculator module, and the last is Coverage calculator which computes the percentage of MC/DC achieved by the test suite, as shown in figure 5.6.

5.2 Analysis of MC/DC Coverage percentage

In previous research papers [17] [7], work for an increase in MC/DC coverage percentage has been done. In this paper, we are analyzing coverage percentage. To calculate coverage percentage, the C program is fed to the Advanced Program Code Transformer based on sum of product. The transformer has four steps,

including the minimization of SOP in which modified Quine McCluskey(MQM) method is applied by using which we can reduce the number of comparisons between mintermlist while E-sum is used to eliminate repetition. Modified Quine McCluskey is simple and faster than Quine-McCluskey method due to less number of required comparisons.

After transforming the C program, we pass it to CREST tool to automatically generate MC/DC test cases. With the use of the original C program and test cases we are evaluating coverage percentage. In our experimental study, we have taken 6 complex programs. Table 5.1 shows the name of 6 programs with their lines of code (LOC), MC/DC Coverage for original programs(Mcov_Original), MC/DC coverage analyser using a program code transformer(Mcov_PCT), MC/DC coverage analyser for Advanced program code transformer(MCov_APCT). Table 5.2 shows the Variation in MC/DC percentage using Program code transformer i.e $Mcov\_PCT - Mcov\_Original$, Variation in MC/DCcov using Advanced program code transformer i.e $Mcov\_APCT - Mcov\_Original$ and Variation in Mcov using Program Code Transformer and Advanced Program Code Trnsformer i.e $Mcov\_PCT - Mcov\_APCT$. Some programs are student assignment and some are open source programs. Dairy Management program is very complex in nature. Since it is more than 1000 lines of code. Triangle, Timer, Contact manager, Student record, Snake Game have Good MC/DC coverage percentage.

From the observation of Table 5.2 we can observe that variation in average MC/DC coverage percentage is 3.81% for 6 programs. We achieve an increase in APCT MC/DC.

The Modified Quine McCluskey method is used in a way that when the number of minterms is high, the number of comparisons increases between two adjacent groups, and the condition becomes worst when there are all possible combinations in the minterm list are taken. (For n number of variables, number

of comparisons is 2^n). Therefore, in the first pass, the number of comparisons between adjacent minterms in Quine McCluskey (QM) method is $\sum_{i=0}^{n-1} {}^{n}C_i * {}^{n}C_{i+1}$, where i is a group number. But in Modified Quine McCluskey (MQM) method the number of comparisons reduces to $\sum_{i=0}^{n-1} {}^{n}C_i$ * (n-i) or $n * 2^{n-1}$, which is much lesser than the Quine McCluskey Method. The number of comparisons reduces because of E-sum is used to eliminate the repetitions.

Table 5.1: Coverage Percentage using PCT and APCT.

S.No.	Program	LOC	Mcov Original	Mcov PCT	Mcov APCT
1	Triangle	75	75%	100%	100%
2	Contact Manager	224	50%	84%	87%
3	Student Record	390	58.7%	74.9%	81.2%
4	Calender	430	63.4%	81.5%	86.4%
5	Snake Game	533	62.7%	82.4%	85.9%
6	Dairy manager	1250	59.1%	79.6%	84.8%

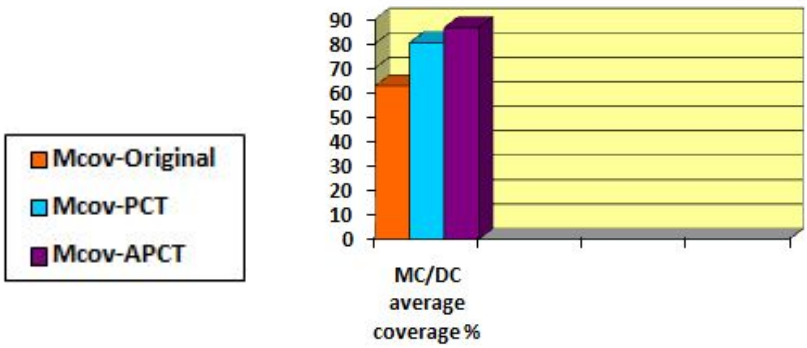

Figure 5.7: Graph showing the average Mcov_Original, Mcov_PCT and Mcov_APCT Percentage.

Table 5.2: Variation in the Coverage Percentage.

S.No.	Program	Variation in Mcov% using PCT	Variation in Mcov% using APCT	Variation in PCT and APCT Mcov%
1	Triangle	25%	25%	0%
2	Contact Manager	34%	37%	3%
3	Student Record	16.2%	22.5%	6.3%
4	Calender	18.1%	23%	4.9%
5	Snake Game	19.7%	23.2%	3.5%
6	Dairy manager	20.5%	25.7%	5.2%

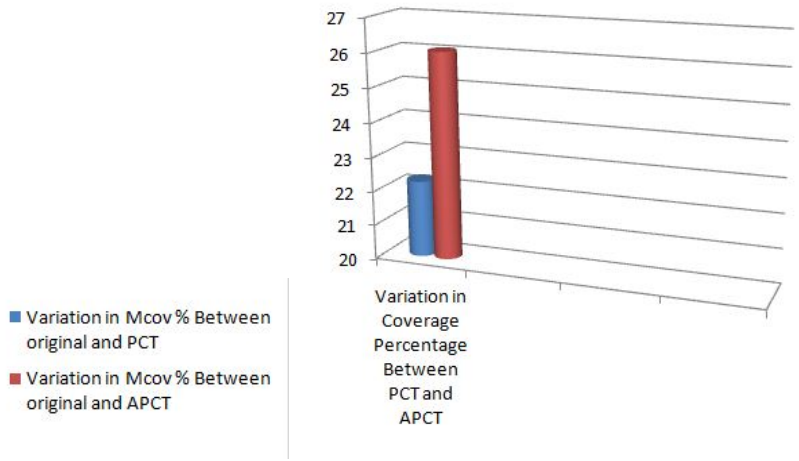

Figure 5.8: Graph showing variation in Mcov Original, Mcov-PCT and Mcov-APCT.

As we can observe in our APCT architecture we achieved 3.81 % improved coverage percentage as compared to the PCT architecture. We know that MC/DC Coverage depends on entered empty nested if-else statements for each condition in each predicate of program under execution. It shows that number of coverage is dependent on number of conditions covered. Now, In K-map minimization technique, the simplification of sum of product is very difficult beyond 6 variables. Similarly in QM, according to more number of min terms for n variables, it is very

difficult to cover more number of variables. Since, min term comparisons for QM technique is:

$$\sum_{i=0}^{n-1} {}^{n}\mathrm{C}_i * {}^{n}\mathrm{C}_{i+1}.$$

In MQM minimization technique, we can cover more number of conditions as compared to the K map and QM technique because MQM reduces the number of comparisons of min term to

$$\sum_{i=0}^{n-1} {}^{n}\mathrm{C}_i * \text{(n-i) or } n * 2^{n-1}$$

and covering the more number of conditions by reducing the time complexity.

Chapter 6

Chapter 6

Conclusion

In this research work we have discussed an approach to automatically increase the MC/DC Coverage. We have used a concolic tester tool i.e CREST tool and a code transformer which is based on sum of product(SOP) Boolean logic concept to generate test data for MC/DC. The Transformer has four steps including minimization of SOP in which the Modified Quine McCluskey method is applied by using which we can reduce number of comparison between minterm list while E-sum is used to eliminate repetition. MQM is simple and faster than Quine-McCluskey method due to less number of required comparisons. Thus method can be used to achieve speed in minimizing the Boolean function manually and to improve performance of conventional method and by which we can get the better coverage.

In this work, we have proposed the Advanced Program Code Transformer, CREST Tool, and Coverage Analyzer with their working and descriptions. We have done experiments for 6 complex programs. In that we have calculated MC/DC coverage percentage of the original C program, for program code transformer and advanced program code transformer by our proposed architecture. Hence, We conclude that the variation or enhancement of MC/DC coverage percentage is 3.81%. The effort for calculations reduces the overall effort by using the Modified Quine Mccluskey methods. The Figure 3 shows a Graph of the average Coverage Percentage for 6 Programs taking Original programs and Transformed Programs(using PCT and APCT) and Figure 4 shows a graph for overall variation

in MC/DC coverage percentage using Program Code Transformer and Advanced program Code Transformer in average for all the 6 programs and from the graph, we can clearly see that the Coverage % using APCT is more than PCT by 3.81%.

Bibliography

[1] N. Chauhan, *Software Testing: Principles and Practices*. Oxford University Press, 2010.

[2] V. Jadhav and A. Buchade, "Modified quine-mccluskey method," *arXiv preprint arXiv:1203.2289*, 2012.

[3] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, "A practical tutorial on modified condition," *Decision Coverage*, 2001.

[4] P. Bokil, P. Darke, U. Shrotri, and R. Venkatesh, "Automatic test data generation for c programs," in *Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009. Third IEEE International Conference on*, pp. 359–368, IEEE, 2009.

[5] W. Miller and D. L. Spooner, "Automatic generation of floating-point test data," *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 223–226, 1976.

[6] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *ACM Sigplan Notices*, vol. 40, pp. 213–223, ACM, 2005.

[7] S. Godboley, G. Prashanth, D. Mohapatra, and B. Majhi, "Enhanced modified condition/decision coverage using exclusive-nor code transformer," 2013.

[8] M. M. Mano, *Digital Design: For Anna University, 4/e*. Pearson Education India.

[9] P. CREST, J. Offutt, and H. Huang, "Coverage criteria for logical expressions," in *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, pp. 99–107, IEEE, 2003.

[10] 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*, pp. 443–446, IEEE Computer Society, 2008.

[11] M. D. Hollander, *Automatic unit test generation*. PhD thesis, Masters thesis, Delft University of Technology, 2010.

[12] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[13] K. Sen, D. Marinov, and G. Agha, *CUTE: a concolic unit testing engine for C*, vol. 30. ACM, 2005.

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

[15] M. Kim, Y. Kim, and Y. Choi, "Concolic testing of the multi-sector read operation for flash storage platform software," *Formal Aspects of Computing*, vol. 24, no. 3, pp. 355–374, 2012.

[16] D. R. Kuhn, "Fault classes and error detection capability of specification-based testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 8, no. 4, pp. 411–424, 1999.

[17] S. Godboley, G. Prashanth, D. P. Mohapatro, and B. Majhi, "Increase in modified condition/decision coverage using program code transformer," in *Advance Computing Conference (IACC), 2013 IEEE 3rd International*, pp. 1400–1407, IEEE, 2013.

[18] 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*, pp. 1657–1664, ACM, 2009.