Increase in Modified Condition/Decision Coverage Using Program Code Transformer

[1]Sangharatna Godboley, [2]G. S. Prashanth, [3]Durga Prasad Mohapatro and [4]Bansidhar Majhi
Department of Computer Science and Engineering
National Institute of Technology
Rourkela, India
[1]sanghu1790@gmail.com, [2]g.saiprashanth@gmail.com, [3]durga@nitrkl.ac.in and [4]bmajhi@nitrkl.ac.in

***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. 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 which consist of the following major steps: Identification of predicates, Simplification of sum of product by QUINE-McMLUSKY method, and generating empty true-false if-else statements. This transformed program is inserted into the CONCOLIC tester (CREST TOOL) to generate test data for increased MC/DC coverage. Our approach helps to achieve an increase in MC/DC coverage as compared to the traditional CONCOLIC testing.**

Keywords—MC/DC; concolic testing; CREST tool; program code transformer; coverage analyzer

I. INTRODUCTION

Software testing is the way to increase reliability of software projects. 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. The conditions should have both normal and abnormal conditions to determine any failure under unexpected conditions. 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 [5]

Testing strategies are mainly divided into two categories [16]:

A. 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.

B. 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.

There are several white box coverage criteria. Let us take a sample program shown in Figure 1.

i. 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 each line, 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 won't execute. Now consider the case 2, here loop will execut.e

Case 3- x <and case 4- x >y will execute. This criterion is very poor criteria because case 3 and case 4 are sufficient for all statements in code. But if these both will execute case 1 will never execute so it is poor criteria.

ii. 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□=y , case3-x >, case4- x<y.

iii. 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 OR B), test cases (TF), (FT) and (FF) provide MC/DC.

iv. 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.

```
#include <stdio.h>
#include <iostream.h>
int main(int argc, char *argv[])
{
    int x,y,i;
    cin>>x;
    cin>>y;
    for(i=0;x!=y;i++)
     {
       if (x<y)
         y=y-x;
       else
         x=x-y;
     }
    cout<<x;
    cout<<y;

    return 0;
}
```

Figure 1. An example program

A. Problem Definition

In this section we will describe automated testing followed by discussion of our work.

1) To perform Automated Testing: Testing process can save 50% of the whole software development. Due to tight budget limitations thorough testing is unfeasible. Automate testing saves effort and time. There are two approaches to automate the testing process. The first approach is to write scripts for all the embedded test cases. This can be beneficial for regression testing, which contains the repetition of all tests of the system, after this some changes are made to ensure that it does not affect the other process in the system. In this condition it is very costly to repeat manually the same tests every time. The second approach is to design a tool which generates test cases automatically and run them on the system or program to be tested. Such a tool is very difficult to develop, but once if it is developed then it can save a big amount of time. Using tools to perform tasks that are repetitive, complex and time consuming that can help eliminate the possibility of errors resulting from mental fatigue and release humans for tasks that that can't be automated, such as technical reviewing [7].

2) To achieve increased MC/DC: Branch coverage is used by automated test-generation approaches. We will extend an existing approach to get an increase in MC/DC. Here we are talking about CONCOLIC TESTING to achieve MC/DC. It is used to achieve branch coverage. We introduce a code transformer technique to enable concolic testing to achieve increased MC/DC. This transformer introduces several new branches in the program. We entered only nested conditional statements with empty branches in the program. Constraints of branches are necessarily satisfied to achieve an increase in MC/DC. We then switch from transformer to concolic testing. The generated concrete input values for the transformed code achieve an increase in MC/DC for the program under the code.

II. BASIC CONCEPTS

In this section we will discuss the basic concepts which will be used our approach. This section will describe some useful definitions, MC/DC coverage description, Boolean derivative method and concolic testing approach.

A. Definition

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 are called as decision or predicate. A decision with no Boolean operator is a condition.

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 and B) or (C and D)). Here A, B, C, D are four different conditions and (A and B), (C and D) are two groups of conditions. Statement 1 is nothing but the decision statement.

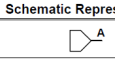
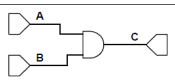
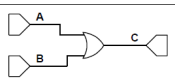
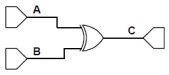
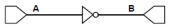
Name	Schematic Representation	Code example	Truth Table
Input			
Output			
and Gate		C := A *and* B;	A B C T T T T F F F T F F F F
or Gate		C := A *or* B;	A B C T T T T F T F T T F F F
xor Gate		C := A *xor* B;	A B C T T F T F T F T T F F F
not Gate		B := *not* A;	A B T F F T

Figure 2. Schematic representation showing different gates

B. 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. 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.

To understand MC/DC approach completely we need to show the schematic representation of logical operator and the truth table of program code. Figure 2 shows the schematic representation of the example predicate given below [7]:
Example: Z= (A OR B) AND (C OR 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. Shaded part shows the effect of MC/DC. Figure 3 shows the representation for sixteen combinations [7].

C. Determination of Predicates

The method of determining predicate p_x has given here, simply uses the Boolean derivative designed by Akers [2]. One benefit of this method is that the problem of redundant of the same clause is handled finely, 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. Note 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:

$$p_x = p_{x = true} \oplus p_{x = false} \quad (1)$$

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 [3]:

Consider the statement, p=x ∧ (y ∨z). If the major cause is x, then the Boolean derivative finds truth assignments for y and z as follows:

$$p_x = p_{x=true} \oplus p_{x=false}$$
$$p_x = (true \wedge (y \vee z)) \oplus (false \wedge (y \vee z))$$
$$p_x = (y \vee z) \oplus false$$
$$p_x = y \vee z$$

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

	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			15	14
14	T	T	F	T	T				13
15	T	T	T	F	T			13	
16	T	T	T	T	T				

Figure 3. The figure shows the sixteen combinations

D. Concolic Testing Process

Concolic testing is the combinational approach of concrete and symbolic testing. Concrete values and symbolic constraints are most important for path execution when the program under process. The next execution code compulsorily takes a different path. Many concolic test generators achieve this point by first negating one of the symbolic constraints that evaluate the execution path and then by solving the outcome set of constraints. This will go on till the stopping criteria are met. There are two stopping criteria's first having fixed threshold value, if it exceeds it will stop further iterations and second based on sufficient code coverage, once if it covered sufficient coverage it stop further iterations. The concolic testing process is carried out using the following six steps:

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) 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 i^{th} execution by combining all path conditions c_1, c_2, c_3 where c_j is executed earlier than c_{j+1} for all $1 \leq j$.

5) 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 repeats from Step 3 again with this input value.

III. Related Work

In this section we will discuss about Automated Testing for Branch Coverage and MC/DC.

A. 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 [14] describes solutions in his survey. Solutions of this problem using Evolutionary Testing (ET) method are like Genetic Algorithm (GA) and like Hill Climbing (HC) [4]. These solutions are to achieve branch coverage.

2) Symbolic testing

Cristian Cadar et al. [6] Says that test data generated by symbolic execution used by the Symbolic testing technique. J. King [10] [11] 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 J. [8]. 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). Patrice Godefroid et al. [15] 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 232 if x is a 32-bit integer code input that is randomly initialized.

4) Concolic Testing

Moonzoo Kim et al. [9] 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 [1] an open source concolic testing tool for C code structures. Concolic represent CONCrete + symbOLIC tests [16].

B. MC / DC Automatic Testing

Awedikian et al. [4] proposed an approach to automatically generate test data to satisfy MC/DC. The steps are as follows:

1. For each predicate, computes 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 X. et al. [12] 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.

IV. MC /DC TESTER

This section presents a detailed explanation of the proposed automatic test generation approach for MC/DC. Here we will see the formal definition and detailed description of our approach..

A. 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→M): It shows that a test suite (M) is generated by the tester tool □ for the program (X) code under test. 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.

R1: ∀: [Output(X,I)=Output(X’,I)], 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 □, then

∃M1: [((X’ → M1) ∧ Coverage(Y’, X’, M1) = 100%) ⇒ (Coverage(Y, X, M1) = 100%)]

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%.

B. A brief Description of our approach [MT]

Our approach developing MC/DC TESTER (MT), 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 (MT) is shown in Figure 5. MT consisting of three components:

1. Program code transformer,
2. Tester for concolic testing,
3. Coverage Analyser.

From the Figure 4, 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. 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.

C. 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's 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 Figure 5. The pseudo-code representation of PCT is given in Algorithm1.

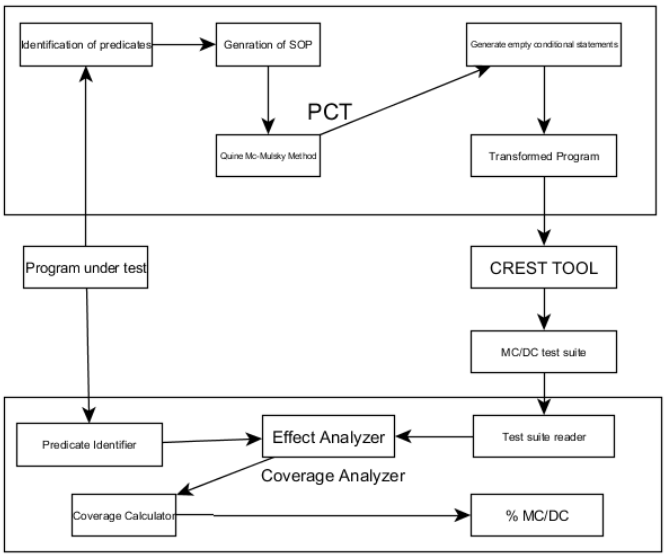


Figure4. The Figure shows the concept of our approach

Algorithm1: Program Code Transformer.
Input: X // X is the program in C syntax
Output: X′ // X′ is the transformed program
Begin
/* Identification of predicates */
for each statements ∈ X **do**
if && or ||occurs in s **then**
1 List Predicate ← adding_in_List (s)
// List of predicates
end if
end for
/* Simplification of predicates */
for each predicate p ∈ List Predicate **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_Sumofproduct_Tabulation (P_Minterm)
// Minimizes the SOP
/* Nested if-else Generation */
5 List_Statement ← generate_Nested_If-else PCT(P_Simplified)
// Generating conditional statements
6 X′ ← insert_code (List_Statement,X)
// Forming in the form of C syntax
end for
7 return X′

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
/* Minterm to Binary form */
for each min term m∈P Minterm **do**
1 List_M ← Convert_to_binary (minterm)
// Listing the binary form after convertion from minterm
end for
2 List L ← sort(List_M)
//According to the number of one in each binary No.
/* comparison of groups and marking of un-compared group*/
for each List l ∈ L **do**
for each group first to group last ∈ groups **do**
for each bit∈total bits **do**
3 one_bit_diff term ← Compare (grp_current, grp_next) // Comparing with groups
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
/* Selection of prime implicant */
6 Prime_Implicant ← Uncompared anymore and indicated with ∗ /* Selection of essential
Prime implicant */
7 essential_Prime_Implicant ←Coveragetable(minterms,Prime Implicant)
// Marking essential elements
/* According to PATRIC METHOD */
8 simplified_function_P_Simp←assigning variables and compliment variables to test Prime_Implicant

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 pseudo code representation for generating empty if-else conditional statements is given in Algorithm 3.

```
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. An example showing concept of PCT

Algorithm3: PCT generateNestedIfElse.

Input: P // minimized SOP predicate P
Output: Statement list //list of statements in C syntax
Begin
for each && connected cond_grp ∈ p **do**
for each condition a ∈ cond_grp **do**
if a is the first condition **then**
1 make an if statement m with a as the condition
2 Statement_list←add list(m)
else
3 makes a nested if statement m with a as the condition
4 making an empty Truebranch Tb and an empty Falsebranch Fb in order,
5 Statement_list←add list(strcat(m,Tb,Fb))
end if
end for
6make an empty False branch Fb for the first condition
7 Statement_list←add list(Fb)
end for
for each condition ∈ p and =∈ any cond_grp **do**
8 repeat lines 1, 4 and 5
end for
if P is an else if predicate **then**
9 make an if(false) statement m
10make an empty Truebranch Tb
11Statement list←add list(strcat(m,Tb))
end if
12 return Statement_list

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.

Example of PCT:
We explain the working of the PCT with an example. Consider the example of *godboleyWeight* function shown in Figure 8. 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 *godboleyWeight* function is shown in following Figure 9. 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.

2. 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.

```
#include <stdio.h>
godboley_Weight (int p, int q, int r, int s)
int main(int argc, char *argv[])
{
 godboley_Weight (75,65,89,95);
    return 0;
}
godboley_Weight(int p,int q,int r,int s){
 if(p>50)
     if(q<80)
      { }
      else{ }
  else{ }
  if(p>50)
      if(r<90)
      { }
      else{ }
  else{ }
  if(p>50)
      if(s<100)
      { }
      else{ }
  else{ }
 if ((p>70)&&((q<80)||(r<90)||(s<100))){
     printf("weight must be more than 70 kg "); }
 else{
     printf("weight may be more or less than 70 kg");}
 }
```

Figure 5. Transformed form of figure8

D. 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. Concolic 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.

E. 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:

$$\text{MC/DC coverage} = (\Sigma_{i=1 \text{ to } n} I\_i \div \Sigma_{i=1 \text{ to } n} c\_i) \times 100 \quad (2)$$

Algorithm4:MC/DC COVERAGE ANALYSER
Input:X,Test Suite // Program X and Test Suite obtained
Output:MC/DCcoverage // % MC/DC achieved for X
Begin
/*Identification of predicates*/
for each statement s∈X **do**
 if && or ||occurs in s **then**
  1 List_Predicate ← adding_in_List(s)
 end if
end for
/* Determine the outcomes */
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
    2 True Flag←TRUE
   end if
   if c evaluates to FALSE and calculate the outcome of p without t d **then**
    3 False Flag←TRUE
   end if
 end for
 if both True Flag and False Flag are TRUE **then**
  4 I_List←adding_in_List(c)
 end if
  5 C_List←adding_in_List(c)
 end for
end for
/* Calculating the MC/DC coverage percentage */
  6 MC DC COVERAGE←(SIZEOF(I_List)_SIZEOF(C List))× 100%

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.

V. IMPLEMENTATION

In this section we will see a performance of concolic tester i.e CREST TOOL. Crest tool is the tester to execute concolic testing. We need to insert one program then it gives test suite and coverage as an output. It's based on three different strategy a) DFS b) CFG and c) RANDOM. In following figures we have taken 'DFS' strategy. Stopping criteria are either threshold value or till all branches covered. The step by step process is shown in figure 10 to 13.

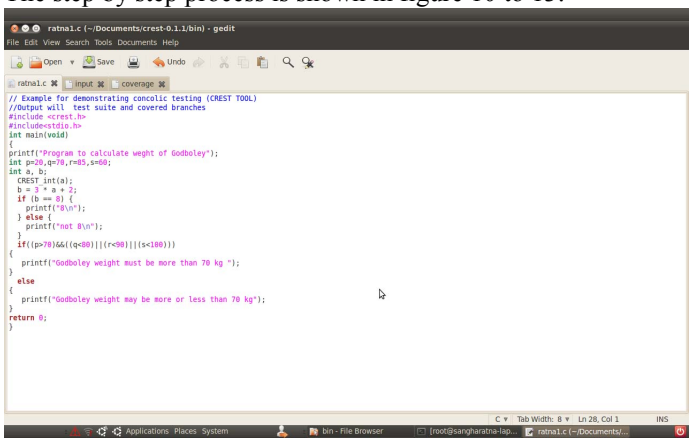


Figure 6. An example of program to demonstrate crest tool

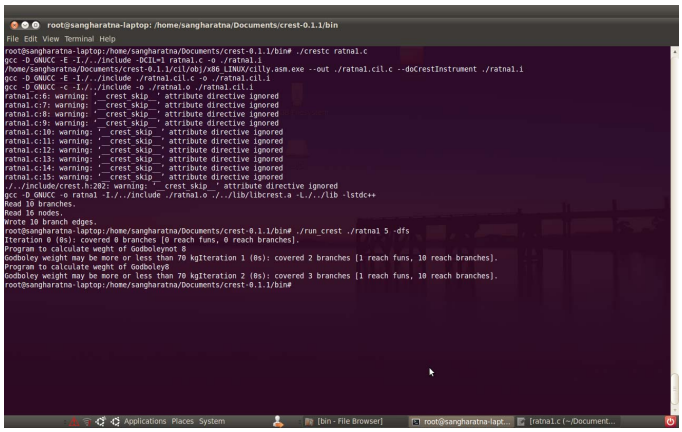


Figure 7. This shows the compilation and execution of program which gives the number of branches and nodes

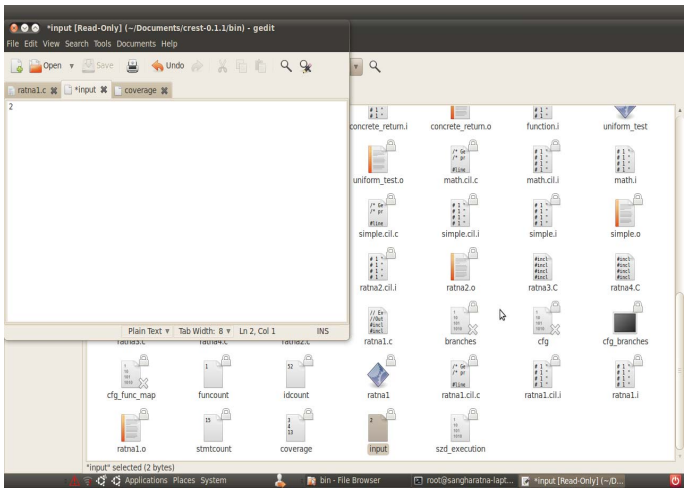


Figure 8. Input file consists of a test data set

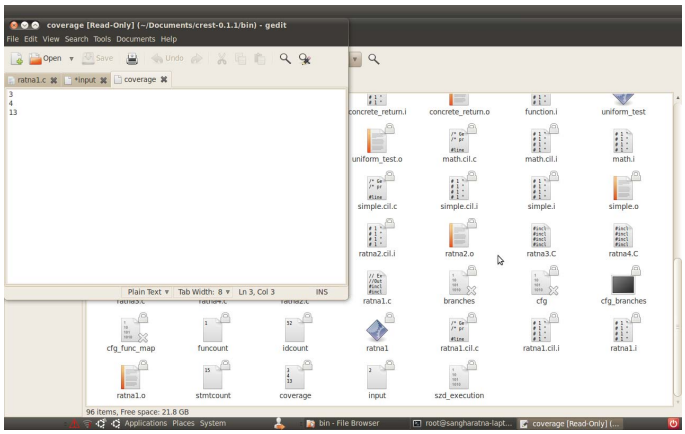


Figure 9. The coverage file shows the covered nodes

VI. Conclusion

In this work 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. Transformer follows four steps including minimization of sum of product by Tabulation Method. Code transformer gives an automated implementation of the boolean Derivative method. Also we have proposed the algorithm for coverage analyzer which calculates the coverage percentage. After observing the whole concept we conclude that we got increased in MC/DC coverage.

References

[1] Crest tool. Website. http://www.code.google.com/p/crest.
[2] Akers, S. B. On a theory of boolean functions. Journal Society Industrial Applied Mathematics, 7(4), pp. 487 – 498.
[3] Ammann, P., Offutt, J., and Huang, H. Coverage criteria for logical expressions. 14th International Symposium on Software Reliability Engineering (ISSRE03), IEEE Computer
Society Press, pp. 99–107.
[4] Awedikian, Z., Ayari, K., and Antoniol., G. MC/DC automatic test input data generation. GECCO'09, Ed., ACM, pp. Pages 1657–1664. In proceedings of the 11th Annual conference on Genetic and evolutionary computation.
[5] Chauhan, N. Software Testing Principles and Practices, 1st ed. 9780198061847. Oxford University Press ,YMCA Library Building, Jai Singh Road, New Delhi 110001, Naresh
Chauhan, Assistant Professor, Dept. of Computer Engineering YMCA University of Science and Technology Faridabad, 31st Jan 2010.
[6] Cristian Cadar, D. D., and Engler, D. Klee:unassisted and automatic generation of high-coverage tests for complex system programs. In Software Maintenance (San Diego, CA, December 2008), In USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008).
[7] Hayhurst, K. J., Veerhusen, D. S., Chilenski, J. J., and Rierson, L. K. A practical tutorial on modified condition/ decision coverage. National Aeronautics and Space Administration , Langley Research Center Hampton, Virginia 23681-2199.
[8] J., D., and Ntafos, S. An evaluation of random testing. IEEE Trans. Software Eng.SE-10 (July 1984), 438–444.
[9] Kim, M., Kim, Y., and Choi, Y. 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.
[10] King, J. A new approach to program testing. In proceedings of the International Conference on Reliable Software, ACM press, pp. 228–233.
[11] King, J. Symbolic execution and program testing. Communications of the ACM,19(7),
ACM press, pp. 385–394.
[12] Liu X., Liu H., W. B. C. P., and X., C. A unified fitness function calculation rule for flag conditions to improve evolutionary testing. In proceeding of the 20th IEEE/ACM international conference on Automated Software Engineering, ACM, pp. 337–341.
[13] Mano, M. Digital Design, 3rd ed. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.
[14] McMinn, P. Search-based software test data generation. a survey:Research articles. Softw. Test. Verif. Reliab. (JUNE 2004), 105–156.
[15] Patrice Godefroid, N. K., and Sen, K. Dart: directed automated random testing. In proceeding of the 2005 ACM SIGPLAM conference on Programming Language Design and Implementation, PLDI'05, ACM, pp. 213–223.
[16] A. Das, "Automatic Genereation of MC / DC Test Data", M. Tech. thesis, IIT, Kharagpur, April 2012.