Increase in Modified Condition/Decision Coverage Using Program Code Transformer

1Sangharatna Godbole, 2G. S. Prashanth, 3Durga Prasad Mohapatro and 4Bansidhar Majhi
Department of Computer Science and Engineering
National Institute of Technology
Rourkela, India
1sanghu1790@gmail.com, 2g.saiprashanth@gmail.com, 3durga@nitrkl.ac.in and 4bmajhi@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 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 execute
Case 3- x>y 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=FALSE, case 2- A=TRUE, B=FALSE, case 3- A=FALSE, B=TRUE, case 4- A=FALSE, B=FALSE.
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.

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, 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 \( p_x = \text{true} \), represents the predicate p and each occurrence of x is replaced by true and \( p_x = \text{false} \), represents the predicate p and each occurrence of x is replaced by false. Note here neither \( p_x = \text{true} \) nor \( p_x = \text{false} \) contains any occurrences of the clause x. Now, here we combine two expressions with the logical operator Exclusive OR:

\[
p_x = p_x = \text{true} \oplus p_x = \text{false}
\]

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, \( p = x \land (y \lor z) \). If the major cause is x, then the Boolean derivative finds truth assignments for y and z as follows:

\[
p_x = (p_x = \text{true} \oplus p_x = \text{false})
\]

\( p_x = (p_x = \text{true} \land (y \lor z)) \oplus (p_x = \text{false} \land (y \lor z)) \)

\[p_x = (y \lor z) \oplus \text{false}
\]

This shows the deterministic answer, three choices of values make \( y \lor z \) true, \((y = z = \text{true})\), \((y = \text{true}, z = \text{false})\), \((y = \text{false}, z = \text{true})\).

\[
\begin{array}{cccccccc}
A&B&C&D&Z&A&B&C&D \\
1&F&F&F&F&10&6&10&6 \\
2&F&F&T&T&11&7&11&7 \\
3&F&T&F&T&12&8&12&8 \\
4&F&T&T&T&13&9&13&9 \\
5&T&T&F&T&14&10&14&10 \\
6&T&T&T&T&15&11&15&11 \\
7&T&T&T&T&16&12&16&12 \\
\end{array}
\]

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 where 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. \( c_{j+1}, c_{j} \) ) 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
An easy technique for automated test generation is path is followed by the program structure. At last the result statement instead of concrete values to code variables as a path is followed by the program structure. At last the result statement instead of concrete values to code variables as a structure. Concolic represent CONCrete + symbOLIC tests as concolic testing. In our approach we will use concolic tester automatically generate test cases for path coverage is known as concolic testing. Moonzoo Kim et al. [9] says the technique combines a concrete dynamic execution and a symbolic execution to achieve structural coverage in X with respect to Y is transformed into the problem of achieving coverage of X with respect to Y. Z has a side effect if the execution of X, when executed with same input I. R1: ∀: [Output(X,I)=Output(X’,I)], then (Y,X,M) shows the percentage of MC/DC coverage achieved in the program under test by the generated test data. 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). 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 code under test (X) subject to an input (I).
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 3M1: ([(X’ → M1) ∧ Coverage(Y’, 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 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 Algorithm 1.

**Algorithm 1: 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 Simplified ← 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 Algorithm 1 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.

**Algorithm 2: Minimization of SOP Tabulation Method.**

**Input:** P Minterm  
**Output:** P Simplified  
**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 conversion from minterm  
end for  
/* comparison of groups and marking of un-compared group*/  
for each list l ∈ List M do  
  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  
  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)) || (q<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 
end for 

6 make 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 
10 make an empty Truebranch Tb 
11 Statement_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.
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:

\[
MC/DC \text{ coverage} = \left( \frac{\sum_{i=1}^{n} I_i - \sum_{i=1}^{n} C_i}{1} \right) \times 100\%
\]  

Algorithm 4: MC/DC COVERAGE ANALYSER

Input: X, Test Suite // Program X and Test Suite obtained
Output: MC/DC coverage // % MC/DC achieved for X

Begin
/*Identification of predicates*/
for each statement s∈ X do
  if && or || occurs in s then
    I 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 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)) \times 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.
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