Enhanced Modified Condition/Decision Coverage Using Exclusive-Nor Code Transformer

[1]Sangharatna Godboley, [2]G. S. Prashanth, [3]Durga Prasad Mohapatra, 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—**In regulated domains such as aerospace and safety critical domains, software quality assurance is subjected to strict regulations such as the DO-178B standard. MC/DC is a white box software testing criteria aiming to prove all the conditions involved in a predicate that can influence the predicate value in the desired way. Though MC/DC is a coverage criterion, existing automated test data generation approaches like CONCOLIC testing do not support MC/DC. In this paper, we propose an automated technique to generate a test suite that helps in achieving an increase in MC/DC coverage of a program under test. We use code transformation technique which consists of two steps: identification of predicates and generation of empty true-false if-else statements. The empty conditional statements are based on the concepts of exclusive nor (X-NOR) operations. This transformed program is inserted into the CREST TOOL. It drives CREST TOOL to generate test suite and increase the MC/DC coverage. Our technique helps to achieve a significant increase in MC/DC coverage as compared to traditional CONCOLIC testing.**

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

I. INTRODUCTION

In the early days of software development, software testing was considered only a debugging process for removing errors after the development of software. Software Testing is a process that detects important bugs with the objectives of having better quality software. [1] The testing process divided into a well-defined sequence of steps is termed as a software testing life cycle (STLC). The STLC consists the following phases: test planning, test design, test execution and test review/post execution. The different levels of testing Unit testing, Integration testing, System testing, and Acceptance testing. To systematically design test cases we have two important approaches:

A. Black-Box Testing: This technique examines the input and output values of the program, identifies the equivalence classes. Equivalence class test cases are designed by picking one representative value from each equivalence class. The boundary value test cases are designed as follows. Examine if any equivalence class is a range of values. Include the values at the boundaries of such equivalence classes in the test suite [2].

B. White-Box Testing: In this technique the test cases require a thorough knowledge of the internal structure of a program. It is also called as structural testing. This is based on an analysis of the code. It can either be covered –based or fault- based [2]. There are several white box coverage criteria:

- *Statement Coverage:* The idea of this coverage is that unless the statement is executed, there is no alternative method to detect whether the bug is there or not.
- *Branch Coverage:* It is stronger than statement coverage. It requires test cases to be designed so as to make each and every condition in the program to assume true and false values in turn [2]. It is also called as edge testing.
- *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. Example A OR B, test cases (TF), (FT) and (FF) provide MC/DC, for the above example.
- *Multiple condition coverage:* This is the strongest criteria. Here all possible outcomes of each condition in decision moving are under consideration. It requires sufficient test cases such that all points of entry are invoked at least once
- *Path coverage:* This requires designing test cases such that all linearly independent paths (or basis paths) in the program code are executed at least once. This can be defined in terms of the control flow graph (CFG) of a program.

PROBLEM DEFINITION: This section shows the overview of our work. Firstly, automated testing is discussed and then the objective of our proposed approach is discussed.

A. Automated Testing:

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

B. The objective of our approach:

The main aim is to develop an automated approach to generate test cases that can achieve MC/DC coverage. To reach our aim, we propose the concept of what to achieve an increase in MC/DC. The Concolic testing combination of concrete and symbolic testing was originally designed to achieve branch coverage [3]. In our approach, we have used CREST TOOL which is a concolic tester to test concolic testing. In our work we present the code transformer in which we insert program code under test and get the transformed program as output. Transformed program is nothing but additional nested if-else that's having true and false branches for each decision with the original program. This transformation is used to get an increase in MC/DC test data. This additional branch does not affect the original program. This transformed program is now inserted to CREST TOOL and we will get MC/DC test suite which consists test data. The tester also generates concrete input values for the transformed code to achieve an increase in MC/DC for the original program. Code transformer consists of two main steps first identifications of predicates and second generation of nested empty if-else statements. To generate nested if else statement we will follow Boolean derivative method. This Algorithm will be based on Exclusive -NOR operations to solve the predicates. This shows the independent effect of the each condition in the complex condition predicate. Third module is coverage analyses it will calculate the coverage percentage. We need to provide MC/DC test data for each and every clause and need to provide an original program in this and at last we will get the coverage percentage. In our observations when we are inserting our original program to concolic tester some MC/DC test data are generated, using these values and our program we will calculate a coverage percentage. Secondly by adding code transformer we will insert transformed program to councils tester and getting MC/DC test data using these values we will calculate a coverage percentage. Now we got two different values but we can see second values is some amount more it means we get an increase in MC/DC by using our approach code transformer.

II. Basic Concepts

In this section we will see some basic definitions and concepts regarding MC/DC coverage, concolic testing, logical gates and operators.

A. Definitions:

1. *Condition:* Boolean statement with no Boolean operator is called as condition or clause
2. *Decision:* Boolean expression consists of conditions and zero or many Boolean operators is called as decision or predicate. A decision without any Boolean operator is a condition. Example: Let's take an example:

   If ((a>100) && ((b<50) ||(c>40)) (1)

   Here in the if-statement whole expression is called as predicate or decision, && and || are the Boolean operators and (a>100), (b<50) and (c>40) are different conditions or clause).

3. *Logic Gates:* They are the fundamental building blocks of digital electronics and performs some logical functions. Most of the logic gates accept two binary inputs and result in single output in the form of 0 or 1.

Table 1 and Table 2 show the truth table for two and three variables respectively.

TABLE 1: TRUTH TABLE FOR TWO VARIABLES

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

B. Modified Condition/ Decision Coverage:

For the achieving MC/DC coverage, the following criteria a need to be satisfied :

- Every decision has to exercise for all outputs at least once.
- Every clause or condition in the decision has to exercise all outputs at least once.
- Every condition in the predicate has to independently affect the predicate's output.

TABLE 2: TRUTH TABLE FOR THREE VARIABLES

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

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

1. Develop a proper representation of the program.
2. Find the test inputs, which can be obtained from the requirement based tests of the software product.

3. Remove the masked test cases. The masked test case is one whose output for a particular gate are hidden from all others outputs.
4. Calculate MC/DC based on Table 3.
5. At last the results of the tests are used to confirm correct operation of the program. For the details of constructing the MC/DC table the readers may refer to [7].

TABLE 3: MC/DC RESULT FOR FOUR VARIABLES

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

C. *Concolic Testing:*

The concolic testing concept combines a concrete constraints execution and symbolic constraints execution to automatically generate test cases for full path coverage [5]. This testing generates test suites by executing the program with random values. At execution time both concrete and symbolic values are saved for executing path. The next iteration of the process forces is selected for different path. The tester selects a value from the path constraints and negates the values to create a new path value. Then the tester finds concrete constraints to satisfy the new path values. These constraints are inputs for all next execution. This process performed iteratively until exceeds the threshold value or sufficient code coverage obtained. Let us take an *example Fig. 1*, calculate speed category of bike when distance and time are given. Tester starts by executing the method with random strategy. Assume that tester has set the values of *distance=120* and *time=-5* in km and hours respectively. During execution time both *concrete* and *symbolic* values are saved for executing path. For input constraints to execute the similar path, it is must that each statement with decision branch calculates the similar value. The first statement (Line 6 in Fig.5) will execute as true, because initially the distance is equal to 120, which is more than zero.

$$(\text{Distance}>0) \qquad (2)$$

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

$$\neg(\text{time}>0) \qquad (3)$$

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

$$(\text{Distance}>0) \wedge \neg(\text{Time}>0) \qquad (4)$$

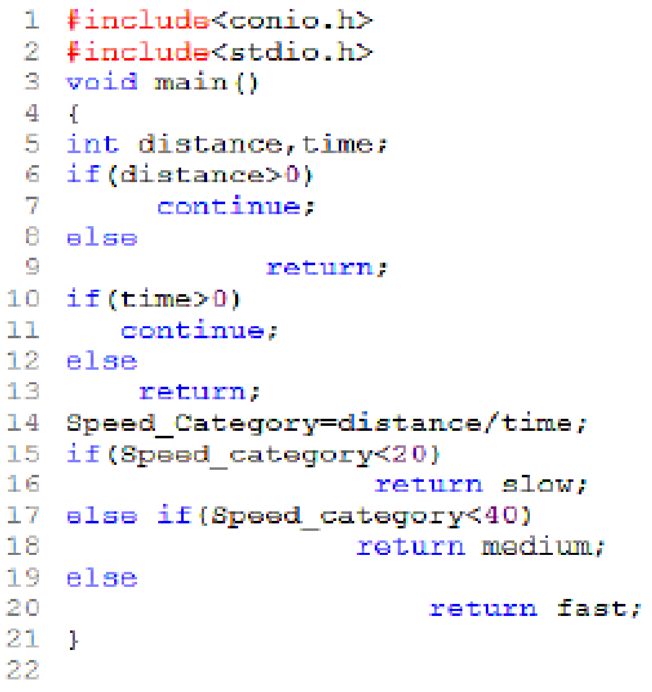
The method fails in the execution of the second condition, so it is altered by negating the branch constraints. When the last condition is negated, then the expression becomes:

$$(\text{Distance}>0) \wedge (\text{Time}>0) \qquad (5)$$

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

$$(\text{Distance}>0) \wedge (\text{time}>0) \wedge \neg(\text{speed\_category}<20) \wedge \neg(\text{speed\_category}<40) \qquad (6)$$

where speed_category= distance/time. This process continues until the stopping criteria is met. This could be possible only when the iteration exceeds the threshold value and sufficient coverage is obtained.



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

FIGURE 1: AN EXAMPLE PROGRAM TO EXPLAIN CONCOLIC TESTING

III. RELATED WORKS

Awedikian et al. [3] have given a concept for automatic MC/DC test generation. They used ET methods to generate test inputs to achieve MC/DC coverage. The concept is modified approach of the branch distance computation [10]. They perform based on control and data dependencies of the code. Their objective was MC/C coverage. However, a drawback of local maxima as the HC algorithm performs data search in limited scope. This shows the solution is not globally optimal.

Pandita et al. [11] have given an instrumental method for generating extra conditional statements for automating logical coverage and boundary value coverage. In this method they used symbolic execution. The coverage of the extra conditional statements increases the logical coverage and boundary value coverage of program code. However, the drawbacks are it does not effectively handle Boolean statements containing || (OR) operators and it inserts many infeasible conditions into a program.

Hayhurst et al. [6] Proposed a modified work of logic gate testing in MC/DC. From program they are creating a logic gate structure of the Boolean statements. Further they used Minimizing Boolean simplification method to decrease the number of logic gates. However, investigation of the process left.

Kuhn [8] and Ammann et al. [2] Proposed methods for generating test suite for making a clause which independently affects the result of the predicate. Their methods help to manually determine the independent effect. In particular they applied the Exclusive OR logic to calculate these conditions. However, Investigation of automation of the method is left.

Bokil et al. [4] have proposed a tool *AutoGen* that reduces the cost and effort for test data preparation by automatically generating test data for C code. *AutoGen* takes the C code and a criterion such as statement coverage, decision coverage, or Modified Condition/Decision Coverage (MC/DC) as input and generates non-redundant test data that satisfies the specified criterion.

IV. GENERATION OF MC/DC TEST DATA

This section presents an explanation of the proposed approach i.e Automated test suite generation approach for MC/DC coverage. Before describing our approach, first we present some definitions that will be used in our approach.

A. Definitions

Our aim is to achieve structural coverage on a given program under test (M), with given coverage criteria (N). It uses the concolic tester tool its objective is to achieve coverage criterion (N′). Therefore, our main objective is to transform M to M′ such that the problem of achieving coverage into M with respect to N is transformed into the problem of achieving structural coverage in M′ with respect to N′. Now we will define some important terms, that will be used in our approach.

- *COVERAGE (N, M, Z)* It indicates the percentage of coverage achieved by test cases (Z) over a given program code under test (M) with respect to given coverage strategy (N).
- *OUTPUT (M,I)* It indicates the result of a program under test (M) subject to an input(I).
- *(M → Z)* It shows that test cases (M) are generated by the concolic tester tool for the program code (M) under test.

For a given M, the concept is to transform M to M′, where M′=M+L and L is the code added to M such that the following requirements are met.

$$\text{R1: : } \forall[\text{Output } (M, I) = \text{Output } (M', I)] \qquad (7)$$

Where I is the collection of inputs to M. Above states that L should not consist of any effect on M. L Consisting effect if the execution of the M′ produces a different result from the one produced by the execution of M when executed from the same input I.

R2: If the test suite T1 is generated for M′ by the tester tool,

$$\exists T1:[((M'\rightarrow T1)\wedge Coverage(N',M',T1)=100\%)\Rightarrow(Coverage(N,M,T1)=100\%)\ ] \qquad (8)$$

The requirement states that if there exists a test suite T_1 that achieves 100% coverage on M′ with respect to N′, then coverage of T_1 on M with respect to N is 100%.

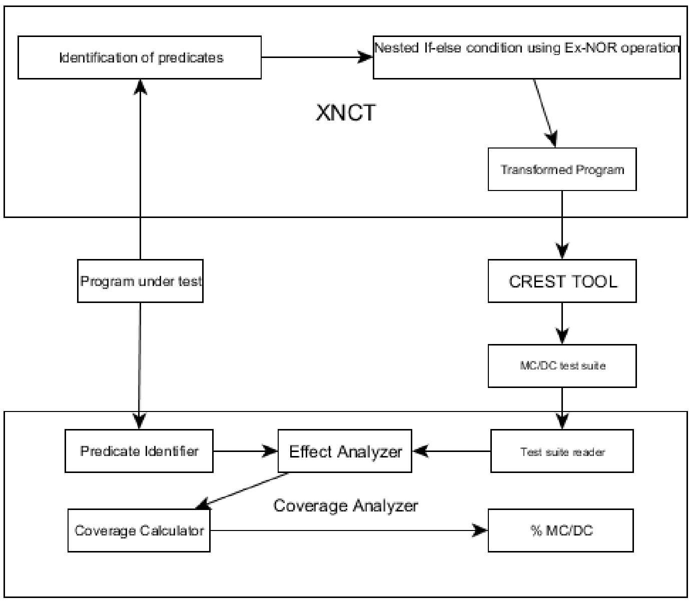


FIGURE 2: SCHEMATIC REPRESENTATION OF OUR APPROACH

B. Our Proposed Approach

The main purpose of our proposed MC/DC tester is to extend the Concolic testing to get increased MC/DC coverage. Transformations of programs under test to include extra conditions are a feasible alternative to attain that aim. After program transformation, let us drive a Concolic tester

CREST Tool to generate MC/DC test suite. A representation of the test data Suite generation is described in Fig. 2. The approach consists of three components:

A. X-NOR Code Transformer

B. Concolic Testing Tool

C. Coverage Analyser

Fig. 2 describes the schematic representation of our approach. A program under test is inserted as input to the X-NOR Code Transformer. It changes the code by generating and adding extra conditional statements for the MC/DC coverage. This approach performs for every clause of the Boolean statement to generate additional statements after identifying the predicates. The transformed program is then passed to the Concolic testing tool which executes all the branches of the transformed program and generates the input for the feasible path. The original program and the test data suite generated by the concolic tester for the transformed program code is supplied to a coverage analyzer. The coverage analyzer calculates the percentage of MC/DC coverage achieved in the program under test by the generated test suite.

C. X-Nor Code Transformer (X-NCT)

We have named the code transformer as X-NCT i.e Exclusive Nor Code Transformer. It uses the Exclusive-Nor gate to calculate the conditions under which each condition in a predicate statement can independently determine the output of a predicate. It assigns each occurrence of the condition in an expression first as true and then as false and then performs X-Nor operation. The output of the X-Nor operation gives the condition under which the clause independently affects the expression results. These additional additional conditions with empty true false inserted. The purpose of inserting empty true and false branches is to avoid duplicate statement executions. Thus, X-Nor comprises mainly of two major steps:

i. Identification of Predicates

ii. Generation of Nested If-Else statements

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

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

Algorithm1: Exclusive-Nor Code Transformer.

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

Algorithm2: generate Nested IfElse XNCT.

```
Input: p // predicate p
Output:Statement list //list of statement in c
Begin
for each && condition c∈p do
            T_a←p
            T_b←p
        for each occurence of condition c a of c∈T a do
                    c_a←TRUE
        end for
        for each occurence of condition c b of c∈T b do
                    c_b←FALSE
        end for
T_c← Exclusive-Nor (T_a, T_b)
Create an If staement S_1 with T_c as the predicate
Create an If statement S_2 with c as the condition
Create an empty Truebranch T_B1
c'← Generate negation (c)
Create a Nested-ELSE-IF statement S_3 with c' as the condition
Create an empty True Branch T_B2
Statement_ list←addList (strcat (S_1, S_2, T_B1, S_3, T_B3))
end for
return Statement_list
```

D. Exclusive-NOR operation

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

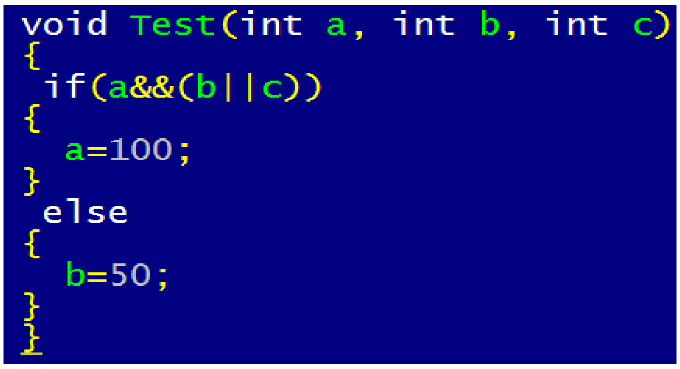
E. *Example for X-NCT*

We describe the concept of X-NCT through Fig. 3 to Fig. 7. Fig. 3 shows the original program. After applying Exclusive NOR operation, the results are shown in Table 4, Table 5, and Table 6 in the form of truth tables. We obtain the final transformed program as in Fig. 4.
The followings are the steps for three variables:

1. *(true AND (b OR C)) X-NOR (false AND (b OR c)) =!(b OR c)* (9)

2. *(a AND (true OR c)) X-NOR (a AND (false OR c))=(!a OR c)* (10)

3. *(a AND (b OR true)) X-NOR (a AND (b OR false))=(!a OR b)* (11)



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

FIGURE 3: AN EXAMPLE FOR X-NOR OPERATION

F. *Complexity for X-NCT*

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

TABLE 4: TRUTH TABLE FOR FIRST VARIABLE (a) AFTER APPLYING X-NOR OPERATIONS

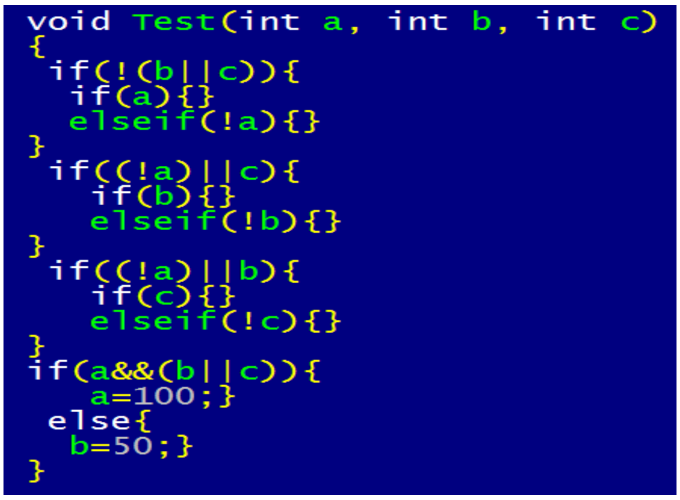
b	c	n=(true AND (b OR c))	n=(false AND (b OR c))	m X-NOR n
T	T	T	F	F
T	F	T	F	F
F	T	T	F	F
F	F	F	F	T

TABLE 5: TRUTH TABLE FOR SECOND VARIABLE (b) AFTER APPLYING X-NOR OPERATIONS

a	c	m=(a AND (true OR c))	n=(a AND (false OR c))	m X-NOR n
T	T	T	T	T
T	F	T	T	T
F	T	F	F	T
F	F	F	F	T

TABLE 6: TRUTH TABLE FOR THIRD VARIABLE (c) AFTER APPLYING X-NOR OPERATIONS

a	b	m=(a AND (b OR true))	n=(a AND (b OR false))	m X-NOR n
T	T	T	T	T
T	F	T	F	F
F	T	F	F	T
F	F	F	F	T



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

FIGURE 4: TRANSFORMED PROGRAM

G. *Concolic Tester*

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

H. *MC/DC Coverage Analyser*

It calculates the MC/DC coverage percentage achieved by a test suite consisting of test data. It is to determine the extent to which a program feature has been performed by test data. Also analyzer finds inadequacy of test cases and provides an insight on those aspects of an implementation that have not been tested. In our approach Coverage Analyzer is essentially used to calculate if there is any change in MC/DC coverage performed by the test suite generated by the concolic tetser *CREST TOOL*. The inserted program for testing and the test suite generated are passed to the coverage analyzer. Coverage analyzer examines the extent to which independent effect of the component clauses on the determining each predicate by the test suite. The MC/DC percentage coverage achieved by the test cases 'T' for program input 'p' denoted by MC/DC coverage is calculated by the formula :

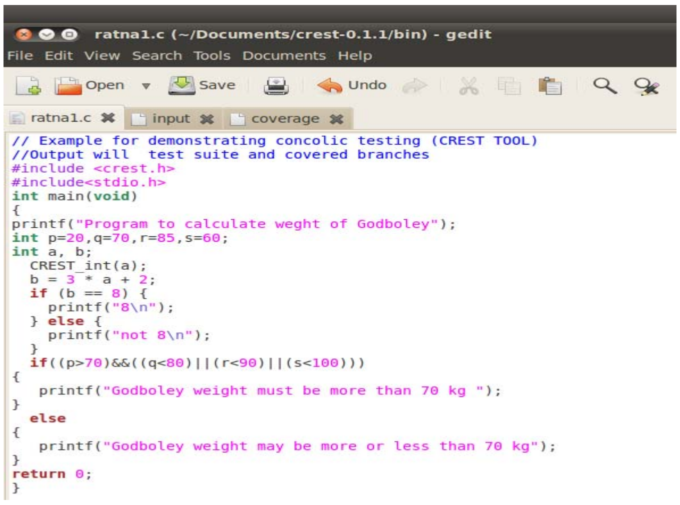
$$\text{MC/DC coverage} = \left(\Sigma_{n \text{ to } i=1} I\_i \div \Sigma_{n \text{ to } i=1} c\_i\right) \times 100 \qquad (11)$$

Algorithm3:MCDC COVERAGE ANALYSER.

```
Input: X, Test_Suite // Program X and Test Suite is collection
of Test cases
Output: MC/DC coverage        // % MC/DC achieved for X
Begin
for each statement s∈X do
    if && or ||occurs in s then
        List_Predicate←adding_in_List (s)
    end if
end for
for each predicate p∈List Predicate do
        for each condition c∈p do
            for each test_case $t_d$ ∈Test_Suite do
if c evaluates to TRUE and calculate the outcome of p with $t_d$
then
            True_Flag←TRUE
end if
if c evaluates to FALSE and calculate the outcome of p
without $t_d$ then
            False Flag←TRUE
end if
            end for
if both True_Flag and False_Flag are TRUE then
                I_List←adding in List (c)
end if
                C_List←adding in List(c)
        end for
end for
MC_DC_COVERAGE←(SIZEOF(I_List)_SIZEOF(C
List))× 100%
```

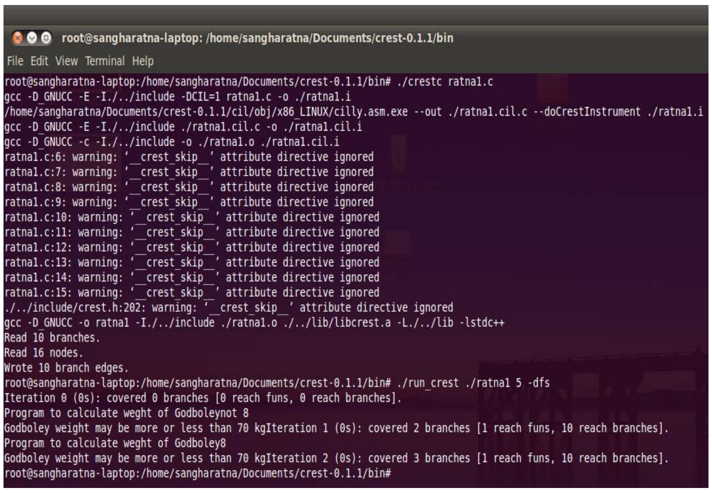
V. IMPLEMENTATION

We have implemented our approach using CREST TOOL[1]. Crest tool is a tester to execute concolic testing. We insert one program, then it gives the test suite and coverage as output. It is based on three different strategies: a) DFS b) CFG and c) RANDOM. In Figures 5 to 8, we have considered the DFS strategy. The stopping criterion is either reaching the threshold value or covering all the branches.


ratna1.c (~/Documents/crest-0.1.1/bin) - gedit
File Edit View Search Tools Documents Help
Open
Save
Undo
ratna1.c
input
coverage

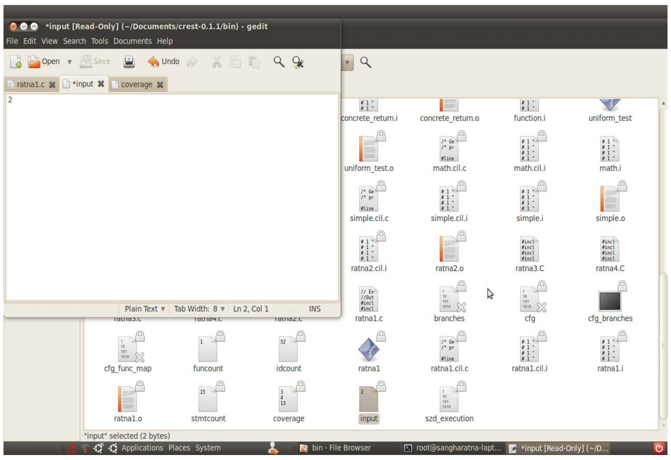
```
// Example for demonstrating concolic testing (CREST TOOL)
//Output will  test suite and covered branches
#include <crest.h>
#include<stdio.h>
int main(void)
{
printf("Program to calculate weght of Godboley");
int p=20,q=70,r=85,s=60;
int a, b;
  CREST_int(a);
  b = 3 * a + 2;
  if (b == 8) {
    printf("8\n");
  } else {
    printf("not 8\n");
  }
  if((p>70)&&((q<80)||(r<90)||(s<100)))
{
   printf("Godboley weight must be more than 70 kg ");
}
  else
{
   printf("Godboley weight may be more or less than 70 kg");
}
return 0;
}
```

FIGURE 5: AN EXAMPLE OF PROGRAM TO DEMONSTRATE CREST TOOL


root@sangharatna-laptop: /home/sangharatna/Documents/crest-0.1.1/bin
File Edit View Terminal Help

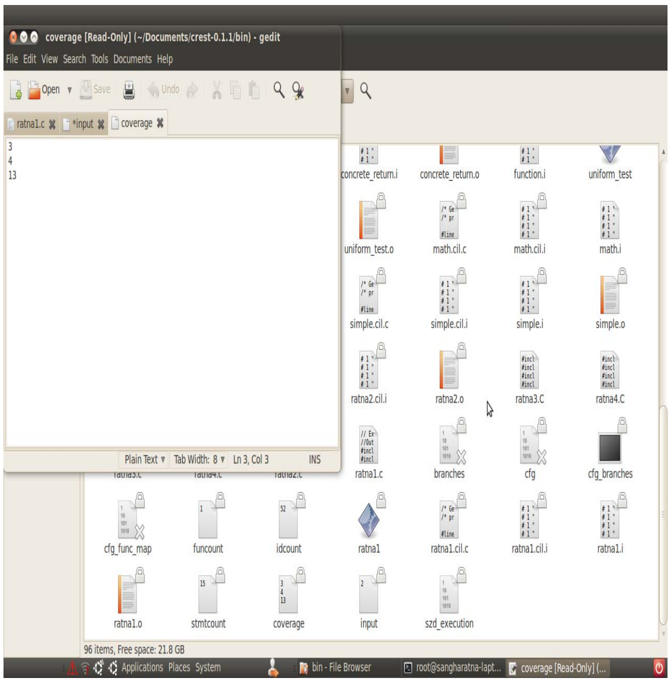
```
root@sangharatna-laptop:/home/sangharatna/Documents/crest-0.1.1/bin# ./crestc ratna1.c
gcc -D_GNUCC -E -I./../include -DCIL=1 ratna1.c -o ./ratna1.i
/home/sangharatna/Documents/crest-0.1.1/cil/obj/x86_LINUX/cilly.asm.exe --out ./ratna1.cil.c --doCrestInstrument ./ratna1.i
gcc -D_GNUCC -E -I./../include ./ratna1.cil.c -o ./ratna1.cil.i
gcc -D_GNUCC -c -I./../include -o ./ratna1.o ./ratna1.cil.i
ratna1.c:6: warning: '__crest_skip__' attribute directive ignored
ratna1.c:7: warning: '__crest_skip__' attribute directive ignored
ratna1.c:8: warning: '__crest_skip__' attribute directive ignored
ratna1.c:9: warning: '__crest_skip__' attribute directive ignored
ratna1.c:10: warning: '__crest_skip__' attribute directive ignored
ratna1.c:11: warning: '__crest_skip__' attribute directive ignored
ratna1.c:12: warning: '__crest_skip__' attribute directive ignored
ratna1.c:13: warning: '__crest_skip__' attribute directive ignored
ratna1.c:14: warning: '__crest_skip__' attribute directive ignored
ratna1.c:15: warning: '__crest_skip__' attribute directive ignored
./../include/crest.h:202: warning: '__crest_skip__' attribute directive ignored
gcc -D_GNUCC -o ratna1 -I./../include ./ratna1.o ./../lib/libcrest.a -L./../lib -lstdc++
Read 10 branches.
Read 16 nodes.
Wrote 10 branch edges.
root@sangharatna-laptop:/home/sangharatna/Documents/crest-0.1.1/bin# ./run_crest ./ratna1 5 -dfs
Iteration 0 (0s): covered 0 branches [0 reach funs, 0 reach branches].
Program to calculate weght of Godboleynot 8
Godboley weight may be more or less than 70 kgIteration 1 (0s): covered 2 branches [1 reach funs, 10 reach branches].
Program to calculate weght of Godboley8
Godboley weight may be more or less than 70 kgIteration 2 (0s): covered 3 branches [1 reach funs, 10 reach branches].
root@sangharatna-laptop:/home/sangharatna/Documents/crest-0.1.1/bin#
```

FIGURE 6: SCREEN SHOT SHOWING THE COMPILATION AND EXECUTION OF THE PROGRAM IN FIG. 5.



```
2
```

FIGURE 7: INPUT FILE CONSISTING OF TEST SUITE



```
3
4
13
```

FIGURE 8: COVERAGE FILE SHOWING THE COVERED NODES

VI. CONCLUSION AND FUTURE WORK

We have presented an approach to automate the test data generation procedure to achieve increased MC/DC coverage. We have used the existing concolic tester i.e CREST tool with a code transformer based on exclusive nor (X-NOR) operation to generate test data for MC/DC. Code transformer gives an automated implementation of the boolean derivative method. Also, we have proposed an algorithm for coverage analysis which calculates the coverage percentage. The advantage of our approach is that it achieves a significant increase in MC/DC coverage.

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

REFERENCES

[1] Crest tool. Website. http://www.code.google.com/p/crest.

[2] Paul Ammann, Jeff Offutt, and Hong Huang. "Coverage criteria for logical expressions". 14th International Symposium on Software Reliability Engineering(ISSRE03), IEEE Computer Society Press, pages 99–107, 2003.

[3] Zeina Awedikian, Kamel Ayari, and Giuliano Antoniol. "Mc/dc automatic test input data generation". ACM In proceedings of the 11th Annual conference on Genetic and evolutionary computation, New York, NY,USA, pages 1657–1664, 2009.

[4] Prasad Bokil, Priyanka Darke, Ulka Shrotri, and R. Venkatesh. "Automatic test data generation for C programs". Third IEEE International Conference on Secure Software Integration and Reliability Improvement. The 1st Workshop on Testing Technologies and Tools for Critical Industry Applications, TCS, pages 359–368, 2009

[5] Naresh Chauhan. "*Software Testing Principles and Practices*". 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, 1st edition, 31st Jan 2010.

[6] Kelly Hayhurst, Dan S. Veerhusen, John J. Chilenski, and Leanna K. Rierson. "A practical tutorial on modified condition/decision coverage". 2001.

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

[8] D. Richard Khun. "Fault classes and error detection capability of specification-based testing". ACM Trans. Soft. Eng. Methodol. 8:pages 411-424., October 1999.

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

[10] Phil McMinn. "Search-based software test data generation". *a survey:Research articles.*, Softw. Test. Verif. Reliab: pages 105–156, JUNE 2004.

[11] Rahul Pandita, Tao Xie, Nikolai Tillmann, and Jonathan de Halleus. "Guided test generation for coverage criteria". In *Software Maitenance*. IEEE International Conference, 2010.

[12] Mall R. "*Fundamentals of Software Engineering*". Prentice Hall, 3rd edition, 2009.

[13] Koushik Sen, Darko Marinov, and Gul Agha. "Cute: a concolic unit testing engine for c". International Conference, ACM, In ESEC/FSE-13: Proceedings of the 10th European, pages 263 – 272, 2005.