# Automatic Generation of Testcases for High MCDC Coverage

*A Thesis Submitted*
*in Partial Fulfilment of the Requirements*
*for the degree of*

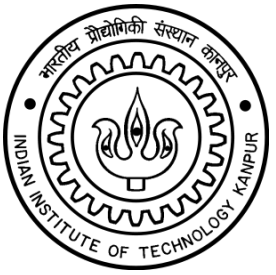
**Master of Technology (M.Tech)**

*by*

**Sonam Tiwari**
***Roll No. 12111041***

*Supervised By*

**Dr. Amey Karkare**
**Dr. Subhajit Roy**




DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

**INDIAN INSTITUTE OF TECHNOLOGY KANPUR**

**June 2014**

# CERTIFICATE

This is to certify that the work contained in the thesis entitled ***"Automatic Generation of Test Cases for High MCDC coverage"***, by *Sonam Tiwari (Roll No. 12111041)*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Amey Karkare 2/6/2014

**Dr. Amey Karkare**
Department of CSE
IIT Kanpur

2/6/2014

**Dr. Subhajit Roy**
Department of CSE
IIT Kanpur

**June 2014**

# ABSTRACT

Software Verification is a discipline of Software Engineering which aims at assuring that the software adhere with the requirements. Structural coverage is a testing mean complying that the requirements based testcases have exercised the code structure. In mission critical domains such as aviation industry, military software quality assurance is subject to strict regulations. Modified Condition/Decision Coverage (MCDC) is a whitebox testing metric with the objective of covering all the conditions and decisions in the program along with showing the independent effect of conditions on overall decision's outcome. This thesis presents a new approach to automatic generation of testcases for high MCDC coverage.

In the thesis, we have implemented concolic tester to automatically generate testcases. The short circuiting approach to evaluate conditions inside a decision in C-Language is used to decompose a decision into nested conditions using CIL. The approach involves two major modules - levelling module and CDG module. Levelling module generates testcases by flipping conditions at a specific nesting level, providing low coverage. CDG module utilizes program's control dependency information to find paths containing maximum number of uncovered conditions. We have also used MAXSAT to find maximum satisfiable set for path constraints, hence providing high coverage.

The advantages of these modules are utilized in a combined approach to first generate testcases using levelling module which then provides program state to CDG module. Experimental results show an average coverage of 78.275 % for combined approach, which is an improvement of 12.825 % over levelling module and 2.875 % over CDG module.

# *Acknowledgements*

I would like to express my sincere gratitude to my supervisors, Prof. Amey Karkare and Prof. Subhajit Roy for providing me with a platform to work on challenging areas of *Modified Condition Decision Coverage*. Their guidance ,support, patience, motivation and immense knowledge have been inspiration to my work.

I would like to express deepest appreciation to Swapnil Mahajan (MTech First year student) for all his inputs, discussions and contributions. I am also thankful to all my friends and lab mates for their encouragement, understanding and useful discussions.

I am deeply grateful to *BRNS, DAE* for sponsoring the project and encouraging research work. I would like to thank them for all suggestions, inputs and motivation for thesis.

*Sonam Tiwari*

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Software engineering is a systematic approach to software development. Software Quality assurance is one of the most important steps taking most of the efforts which aims at measuring the software quality according to the requirements using testing. As an example, Ariane 5 rocket failed just after 37 seconds of its launch due to malfunction of control software. Data conversion from 64 bit floating pointer number to 16 bit integer to be stored in a variable representing horizontal bias caused trap. According to DO - 178B standard, for certification of such software, test suites must provide MCDC coverage of source code. MCDC stands for *Modified Condition Decision Coverage* is a structure based testing strategy. MCDC is a stricter version to measure the test suite with the aim of checking independent effect of each condition in the decision's outcome. Modified Condition/ Decision Coverage[1] is used as an exit criteria for critical projects in avionics domain.

## 1.1 Problem Statement

Automated testing is a technique to reduce the time and effort taken for the testing process. For constructing an automatic testing tool, a coverage criteria is required which can be path coverage, branch coverage etc. According to the criteria, test suites are generated automatically by the tool given software as input.

Objective of the Thesis is to automatically generate testcases according to MCDC[1] criteria for C-Language with the aim of generating high coverage with minimum number of testcases. To achieve this, we have used the concept of Concolic testing[4] which is a combination of symbolic execution and concrete execution and was used introduced for path coverage.

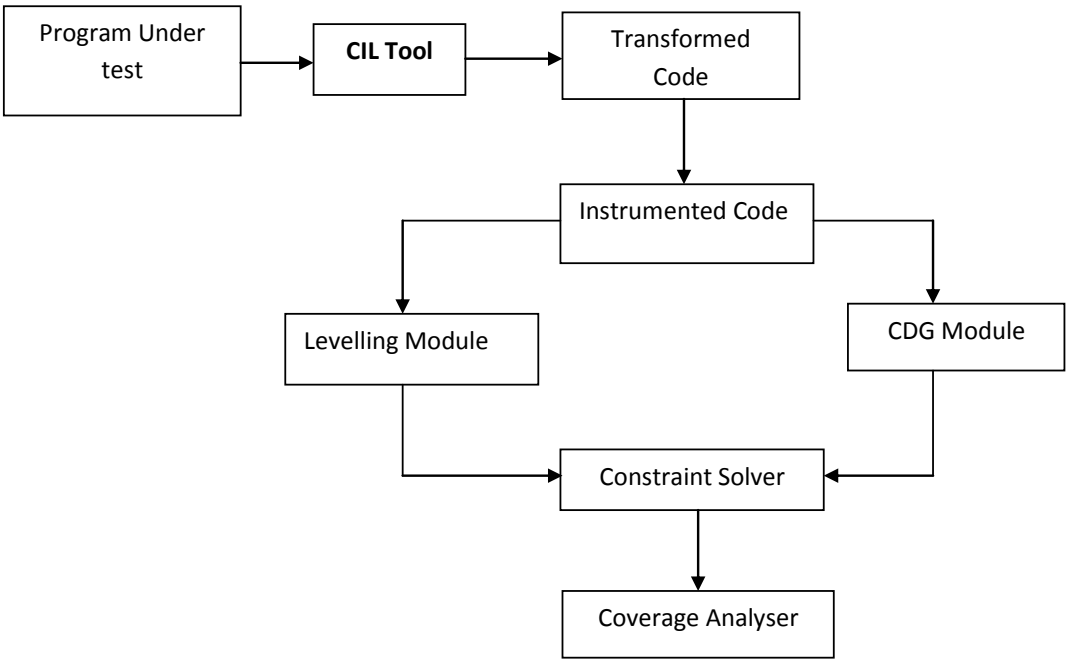



FIGURE 1.1: Major Modules of tool developed

*Figure 1.1* shows the major modules developed for our tool. In our approach we have used a tool called CIL [2] which helps us in statically analyzing the code over Abstract Syntax Tree. Using this tool, the decisions in the program are broken into nested if - else structure according to short circuiting way of evaluating conditions inside a decision in C. This short circuiting technique helps in relaxing the *Independent Effect* criteria of MCDC as all conditions are not required so as to see the independent effect of a condition [6].

We have developed our own concolic tester which has symbolic executor as its submodule to drive program through the test cases. Once the program's decisions are separated the problem of MCDC coverage criteria reduces to the decision coverage of transformed program. We generate test cases according to the MCDC criteria, to show independent effect of each condition's value on the overall outcome of decision, only one of the conditions is flipped in a sequence of nested If/Else structure.

To achieve above functionality we have introduced levelling module, in which nesting level is provided to each decision in the transformed program. For generating new testcases according to MCDC criteria, only nodes at one level are flipped keeping the lower level conditions as prefix of newly generated path constraint.

Levelling module can not cover most of the conditions if conditions are control dependent on other program statements that is, if there are some statements which determine whether a particular statement is executed or not. In order to overcome the drawback, we utilize Control Dependency information of program in the CDG module designed to produce high coverage. A metric called *score* is used which defines the number of uncovered nodes in *Control Dependence Graph* [10] according to this metric we find top **K** paths having maximum number of uncovered conditions. We use MAXSAT [19] on these top paths to find the maximum satisfiable set and generate testcases over this set.

## 1.2 Contributions

We contribute following to automatic testcase generation through this thesis:

- We have designed symbolic executor which can handle integers, floats, arrays, structures and pointers.
- We proposed levelling method in which testcases are generated according to MCDC criteria by flipping conditions at a specific nesting level.
- Control Dependency information of program is utilized to exploit control dependent conditions and increase the coverage.
- MAXSAT is utilized to generate maximum satisfiable set of path constraints increasing coverage in CDG module.

## 1.3 Organization of Thesis

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

**Chapter 2** contains the basic concepts used in the thesis. This chapter contains the definitions of coverages. We describe the steps used to get minimum number of testcases for complex decisions from basic gates - and, or, not and xor. We also provide a brief review of the related work relevant to our contribution and for the important concepts like Symbolic Execution, Concolic Testing, automated testcase generation and MCDC.

**Chapter 3** presents the detailed implementation of our tool. We introduce some formal definitions followed by concepts and algorithms. We describe the algorithms for Concolic Tester, Static Analyser which transforms and instruments the code, levelling technique which is one of our approach for testcase generation.

**Chapter 4** introduces Control Dependency Graph module a second approach to test-case generation. We describe the algorithm for creation, initialization of CDG, finding top **K** paths, feasible paths using MAXSAT.

**Chapter 5** deals with the experimental study and analysis of results obtained in terms of coverage and time taken to produce testcases by the tool over benchmark programs.

**Chapter 6** concludes the thesis with a summary of objective achieved. We also discuss future extentions that can be done to improve the results of testcase generator introduced in the thesis.

# Chapter 2

# Background Details

## 2.1 Basic Concepts

**Definition 2.1.** A condition is a boolean expression containing no boolean or logical operators - And(&&), Or(||) and XOR. These are the atomic expressions which can not be divided to further sub conditions.

**Definition 2.2.** A decision is a boolean expression composed of conditions with zero or more boolean operators. An expression with same condition appearing multiple times in a decision are considered as separate conditions.
Example: for decision expression (((u == 0) || (x >5)) && ((y <6) || (z == 0))), there are four conditions in the decision which are (u == 0), (x >5), (y <6) and (z == 0).

**MCDC Coverage:** MCDC stands for Modified Condition Decision Coverage, it enhances condition/decision coverage by introducing independent effect of each condition to the overall outcome of decision. Following are the MCDC coverage criteria:

- Each decision tries every possible outcome.
- Each condition in a decision takes on every possible outcome.
- Each condition in a decision is shown to independently affect outcome of decision.
- Each exit and entry point should be invoked atleast once.
- For a decision with n Boolean variables atleast n+1 test cases required.

**Logic Gates:** Logic gates are used for implementing logical functions on one or more inputs producing single output. And gate, Or gate, Xor gate and Not gate are considered

as the basic logic gates, knowing minimal number of test cases required for these logical operators provide a basis for examining more complex boolean expressions.

**And gate:** For n input *and* gate the minimum number of test cases required to fulfill MCDC criteria are following:

- All inputs are set to *true* such that the output for and gate comes to *true*.
- One by one each condition is set to *false* making the output *false*, this shows the independent effect of every condition as how the overall outcome changes with condition's value. This step produces a total of 'n' test cases.

TABLE 2.1: Minimum Test Cases for 3 input And gate

| Test Case Number | Input A | Input B | Input C | Output |
|---|---|---|---|---|
| 1 | T | T | T | T |
| 2 | F | T | T | F |
| 3 | T | F | T | F |
| 4 | T | T | F | F |

**Or gate:** For n input *or* gate the minimum number of test cases required to fulfill MCDC criteria are following:

- All inputs are set to *false* such that the output for or gate comes to *false*.
- One by one each condition is set to *true* making the output *true*, this shows the independent effect of every condition as how the overall outcome changes with condition's value. This step results in a total of 'n' test cases.

TABLE 2.2: Minimum Test Cases for 3 input Or gate

| Test Case Number | Input A | Input B | Input C | Output |
|---|---|---|---|---|
| 1 | F | F | F | F |
| 2 | T | F | F | T |
| 3 | F | T | F | T |
| 4 | F | F | T | T |

**Xor gate:** With respect to minimum number of test cases according to MCDC *xor* gate differs from And & Or gate as there are different sets of test cases satisfying MCDC criteria. Truth table for 2 input xor gate is shown in *Table 2.3*, any combination of three

TABLE 2.3: Truth Table for Xor gate

| Test Case Number | Input A | Input B | Output |
|---|---|---|---|
| 1 | T | T | F |
| 2 | T | F | T |
| 3 | F | T | T |
| 4 | F | F | F |

test cases will satisfy MCDC for xor gate.

Hence, mimimum number of test cases satisfying MCDC criteria for 2 input xor gate requires one of the following set of test cases:

- testcases 1,2 and 3
- testcases 1,2 and 4
- testcases 1,3 and 4
- testcases 2,3 and 4

**Not gate:** The *not* gate works on single input which may be a single condition or logical expression. Minimum number of test cases required are as follows:

- The only input set to *false* such that output comes to *true*.
- The only input set to *true* such that outcome comes to *false*.

To evaluate MCDC using the basic constructs or gates each decision is examined for all operators to determine whether tests have exercised the operator according to minimum number of test cases as discussed above. Following are the steps for identifying MCDC test cases :

- Construct representation of source code.
- Determine test inputs from requirement based tests and create truth table for decisions.
- Masked test cases are the one in which output of a particular gate is hidden from observed output.

- MCDC testcases are determined satisfying all the criteria listed above according to minimum number of test cases required for basic constructs.
- Last step involves verifying the output obtained using these test cases with the desired outcomes to confirm correct operation of software.

TABLE 2.4: Minimum test cases for example decision

| Test Case Number | A: (u == 0) | B: (x > 5) | C: (y < 6) | D: (z == 0) | ( (A ‖ B) && (C‖ D) ) |
|---|---|---|---|---|---|
| 1 | F | F | F | F | F |
| 2 | **F** | **F** | **F** | **T** | **F** |
| 3 | F | F | T | _ | F |
| 4 | F | T | F | F | F |
| **5** | **F** | **T** | **F** | **T** | **T** |
| 6 | F | T | T | _ | T |
| **7** | **T** | _ | **F** | **F** | **F** |
| **8** | **T** | _ | **F** | **T** | **T** |
| **9** | **T** | _ | **T** | _ | **T** |

Table 2.4 shows truth table for decision expression ( ((u == 0) || (x > 5)) && ((y < 6) || (z == 0)) ). Each condition is represented as A, B, C & D as there are 4 conditions we need a minimum of 5 set of test cases in order to fulfill MCDC criteria. One of the possible set of test case is highlighted in table as testcase {2, 5, 7, 8, 9}. In the truth table, _(underscore) symbol signifies that the value of condition is dont care that is, it can take any value true/false without affecting the overall outcome of decision.

## 2.2 Concolic Testing

Concolic testing [4] is a hybrid approach to software verification that combines *Symbolic Execution*, which involves representing program variables in terms of symbolic variables and *Concrete Execution*, running program on particular inputs. This testing concept was first introduced for path coverage. In this technique, first the program is run by intializing symbolic variables with random inputs and the path condition is obtained along with symbolic execution done on the path obtained. New path is directed from the previous path by flipping or negating last condition seen.

*Figure 2.1* shows an example program which checks if a given year is a leap year or not. In order to elaborate concolic testing, we start with initializing year with random input, where the value of *year = 2000*. The execution of program starts with this value of year saving both concrete and symbolic values of variables in the executing path. As 2000 is

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int year;
6
7    printf("Enter a year to check if it is a leap year\n");
8    scanf("%d", &year);
9
10   if ( year%400 == 0)
11     printf("%d is a leap year.\n", year);
12
13   else if ( year%100 == 0)
14     printf("%d is not a leap year.\n", year);
15
16   else if ( year%4 == 0 )
17     printf("%d is a leap year.\n", year);
18
19   else
20     printf("%d is not a leap year.\n", year);
21
22   return 0;
23 }
```

FIGURE 2.1: Example Program to explain Concolic Testing

a leap year therefore condition at line 10 gets executed and the path condition obtained for current path is:

$$(year \ \% \ 400) == 0 \tag{2.1}$$

In order to explore new paths, the last condition obtained from the previous path condition is negated. In the example above, new constraint generated is:

$$\neg(year \ \% \ 400) == 0 \tag{2.2}$$

This constraint is examined by a solver in order to check whether this is a feasible path for program. If it is then new set of values are generated for symbolic variables and program is executed with these values. There may be many solutions which satisfy the path constraint but the solver picks one among them. Let the input generated for constraint given by *Equation 2.2* be year = 1900. The new path condition obtained by executing program is as follows:

$$\neg(year\%400 == 0) \wedge (year\%100 == 0) \tag{2.3}$$

The above process is repeated till there is no new feasible path left to be explored. The series of iteration for example program produces following constraints for complete path coverage:

$$\neg(year\%400 == 0) \wedge \neg(year\%100 == 0) \wedge (year\%4 == 0) \tag{2.4}$$

$$\neg(year\%400 == 0) \wedge \neg(year\%100 == 0) \wedge \neg(year\%4 == 0) \tag{2.5}$$

### 2.2.1 Concolic Testing Process

The concolic testing process is carried out using following steps:

1. **Determination of Symbolic Variables:** Symbolic variables are the one for which tester generates test inputs. These variables can be function arguments, user inputs etc. These are the variables which drive the behaviour of program under test.
2. **Code Instrumentation:** In this phase, the input code is instrumented with additional code statically at compile time which keeps track of path conditions symbolically when the program is executed with concrete inputs.
3. **Concrete Execution:** This step involves executing program with the test inputs. These inputs are random for the first run, for successive iterations the test values are generated using a solver based on the directed path conditions obtained form previous runs.
4. **Symbolic Execution:** Symbolic execution module involves representing program variables in terms of symbolic variables identified in step 1. During program execution, whenever there is an assignment statement the code is instrumented statically with probe, such that each program variable is a function of symbolic variables or collects symbolic path condition whenever there is an *if* condition.
5. **Generating new constraints:** The symbolic path condition obtained from previous run is directed to negate the last condition and generate a new constraint for which there is a feasible path to be explored in the program.
6. **Generating test inputs:** The new path constraint generated in step 5 is sent to constraint solver to check its feasibility. The test inputs are generated for feasible

constraints, are set as new inputs and we move to step 3. Steps 3, 4, 5 and 6 are repeated till there are no new feasible path left in program for exploration.

## 2.3 Related work

*"Symbolic Execution and Program Testing"* - an approach to testing was proposed by James c. King. In the approach, rather than passing concrete values symbolic values - variables with some values are passed to the program under test. The advantage of approach is one symbolic execution represents a large set of concrete execution.

**CUTE** : CUTE [7] stands for *"Concolic Unit Testing Engine"* which combines Symbolic execution and Concrete execution to generate test inputs to explore all feasible execution paths.

**DART** : DART [5] stands for *"Directed Automated Random Testing"* , this was a tool developed by Patrice Godefroid, Nils Klarlund and Koushik Sen as a unit testing method. The tool uses concept of concolic testing for path coverage. It uses following techniques:

- Automatic interface extraction.
- Automatic test driver generation for random testing.
- Dynamic analysis of program behaviour to automatically generate test inputs in order to drive execution along alternate paths,

Zeina Awedikian [9] proposed an approach to automatically generate test inputs according to MCDC criteria. The steps followed in the approach are as follows:

1. For each decision, calculate the sets of MCDC coverage using truth table.
2. Following are the proposed fitness function:
   - **Branch Fitness:** It is the branch distance function describing how far is the decision statement under test from making its outcome true.
   - **Dependency Fitness:** This function describes the series of decision statements from entry which will help in reaching the decision under test obtained with the help of *Control Flow Graph*.
3. Generate test inputs using Meta heuristic algorithms.

Liu et al. [12] proposed a unified algorithm to calculate fitness function which replaces the branch fitness with a flag cost function, that considers the data dependence relationship between the definition and use of flag function creating a set of conditions.

Bokil et al. [13] have proposed a tool *AutoGen* that reduces the cost and effort for test input preparation by automatically generating test inputs for C programs. Tool takes the program and a criterion such as path coverage, statement coverage, decision coverage, or Modified Condition/Decision Coverage (MCDC) as input and generates test inputs satisfying the specified criterion.

Program Code Transformation thesis [20] at NIT Rourkela proposed approach to transform each boolean expression in to Sum of Products form and minimize using QUINE-Mc-MLUSKY Technique. This introduces additional conditions in the tranformed program. For concolic testing they used a tool called CREST used for test case generation according to branch coverage technique.

# Chapter 3

# Methodology

This chapter gives a detailed description of the algorithm implemented for automatic test case generation.

## 3.1 Definition

Given a C-Program **P** under test, our objective is to find a set of test cases for high MCDC coverage of **P** with less number of testcases. To do so, the input program **P** is instrumented with additional code **X** which helps to achieve the desired coverage and **X** has no effects on output of program **P**. We use the following notations:

- Output(P, I) - The output of the program **P** when run with input **I**.
- I – Input generated by the tool for program **P**.
- Coverage(P, I) – This metric gives the coverage percentage when a program P is run on input **I**.

## 3.2 Modules

To implement MCDC [1] test case generator, we use an extention of *Concolic Testing* [4] approach. The Program P under test is transformed to break the decisions into nested conditional structure which is done automatically by CIL [2]. The objective now reduces to Decision Coverage of transformed program P, for MCDC the criteria of *independent effect of each condition in the decision's outcome* is exploited by levelling algorithm.

The approach also uses program's *Control Dependency* information for increased MCDC coverage with optimum number of test cases.

The Major Components are:

- Static Analyser.
- Symbolic Executor.
- TestCase Generator.
- Control Dependence Graph (CDG Module).
- Coverage Module.

*Figure 3.1* shows the schematic representation of the tool developed.

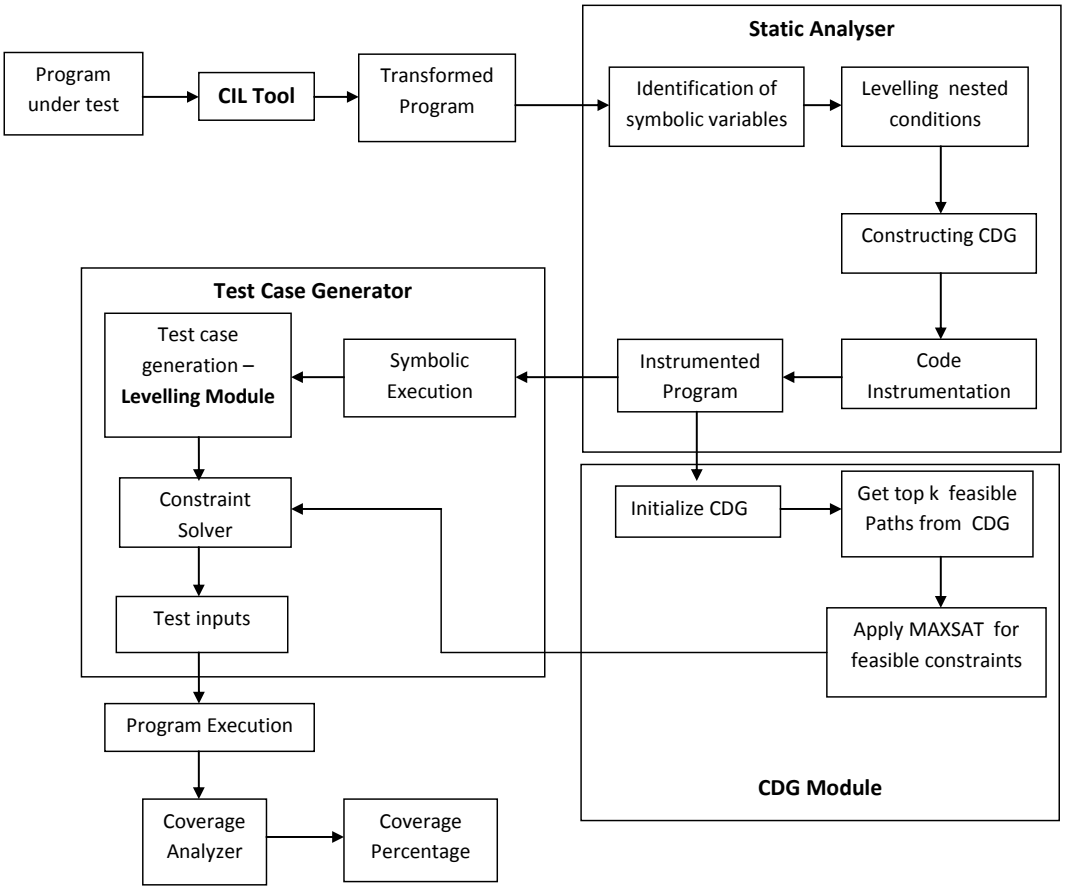



FIGURE 3.1: Schematic Representation of approach

As seen in *Figure 3.1*, the program **P** is sent to static analyser which in our case is CIL (C Intermediate Language) that transforms the decisions in to nested structure.

Next we identify the symbolic variables of transformed program and level the decisions. In the approach, we use two major modules - Levelling Module and CDG Module. In Levelling Module, we direct the path conditions obtained in such a way that conditions at a particular level are flipped together keeping the parent conditions as it is with the aim of showing independent effect of each condition inside decision [1].

For CDG module, program's Control Dependence Graph is constructed using the CFG and Dominator relationship [17]. Top k paths are found on the basis of score representing total number of uncovered conditions in the subtree of a node. MAXSAT [19] is used to find the feasible set of conditions. Instrumentation of additional code **X** is done as a part of analyzer. This transformed code is then used to generate new path conditions using Levelling module and CDG module. The path condition obtained in the form of prepositional formula is then given to constraint solver in order to obtain test inputs. After generation of test inputs, program is driven with these inputs and coverage analyzer gives the program's coverage percentage.

## 3.3 Static Analyser

Static analysis of program involves traversing the program's Abstract Syntax Tree generated by a tool called CIL – C Intermediate Language [2]. The functionalities of module are explained below.

### 3.3.1 Extraction of Program Interface:

This module involves finding all the variables and their types that drive the function under test. These variables represent *Symbolic Values* for Program P which includes function arguments, user inputs and global variables. These Symbolic variables are the test inputs for which values are generated.

*Figure 3.2* shows an example C-Program. In the program, there are no function arguments or global variables. User inputs are - *max, a, b* and *c* these values determine the behaviour of function under test. Hence, after visiting the Abstract Syntax Tree of program we extract the *Symbolic Variables* as - max, a, b and c.

### 3.3.2 Test Driver Instrumentation:

After extraction of interface, program P is instrumented with test driver D. The function of D is to initialize all the *Symbolic Variables* with random values. All the symbolic

```
1  int findGrade()
2  {
3    int m,total, a, b, c, sum, per;
4
5    printf("Enter total marks & marks obtained by the student in 3 subjects: ");
6    scanf("%d%d%d%d",&m,&a,&b,&c);
7    printf("m=%d, a=%d, b=%d, c=%d\n",m,a,b,c);
8
9    if((a>m)||(b>m)||(c>m))
10     printf("\n\t!!Wrong data !!\n");
11
12   else
13   {  sum = a+b+c;
14
15     if(sum>=240)
16      printf("Grade : A\n");
17
18    else if((sum>=180)&&(sum<240))
19      printf("Grade : B\n");
20
21    else if((sum>=120)&&(sum<180))
22      printf("Grade : C\n");
23
24    else
25      printf("Result : Fail\n");
26   }
27
28 return 0;
29 }
```

FIGURE 3.2: Example Program 1

variables are added to argument list of the function under test so that the test inputs can be passed during function call itself. *Figure 3.3* shows Test Driver for the example program in *Figure 3.2*.

```
int testDriver()
{

 int max, a, b, c;

 max = rand();
 a = rand();
 b = rand();
 c = rand();

 findGrade(max, a, b, c);

 //other function calls..

 return 0;
}
```

FIGURE 3.3: Instrumented Test Driver

### 3.3.3 Code Transformation:

Code Transformation involves breaking the *Decisions* in program to nested If/Else structure which is already taken care by CIL [2]. This nested structure is based on how the conditions are evaluated in C example, for (A && B) type of boolean expressions condition B is evaluated only if condition A is true. Similarly for (A || B) type of boolean expressions if condition A is true, condition B is not evaluated and if condition A is false, then only condition B gets evaluated. *Figure 3.4* shows the transformed code for our example program.

### 3.3.4 Levelling the nested structure of conditions:

In this phase, nesting level is assigned to each decision in the program which is a part of *Levelling Module*. This phase is important as further conditions will be directed for testcase generation according to the levels, required by *Independent Effect* one of the MCDC criteria [1]. The pseudocode representation of the phase is given in Algorithm 1.

***Algorithm 1: Levelling Nested Structure***

```
1  Input : Program P
2  Output : P with instrumented code
3  begin
4  for each statement S ∈ P do
5    if S is conditional then
6        AddLevel(S, 1)
7        EnQueue(Q, S)
8    end if
9    while Q is not Empty do
10     Parent = DeQueue(Q)
11     for each statement S' in true block and false block of Parent do
12       if S' is conditional then
13         AddLevel(S', findLevel(Parent) + 1)
14         EnQueue(Q, S')
15       end if
16     end for
17   end while
18 end for
19 end
```

```
1   int findGrade(void)
2   {
3     int m ;
4     int a ;
5     int b ;
6     int c ;
7     int sum ;
8
9     {
10
11    printf((char const   * __restrict  )"Enter total marks & marks obtained by the student in 3
    subjects: ");
12
13    scanf((char const   * __restrict  )"%d%d%d%d", & m, & a, & b, & c);
14
15    printf((char const   * __restrict  )"m=%d, a=%d, b=%d, c=%d\n", m, a, b, c);
16
17    if (a > m) {
18
19      printf((char const   * __restrict  )"\n\t!!Wrong data !!\n");
20    } else
21
22    if (b > m) {
23
24      printf((char const   * __restrict  )"\n\t!!Wrong data !!\n");
25    } else
26
27    if (c > m) {
28
29      printf((char const   * __restrict  )"\n\t!!Wrong data !!\n");
30    } else {
31
32      sum = (a + b) + c;
33
34      if (sum >= 240) {
35
36        printf((char const   * __restrict  )"Grade : A\n");
37      } else
38
39      if (sum >= 180) {
40
41        if (sum < 240) {
42
43          printf((char const   * __restrict  )"Grade : B\n");
44        } else {
45
46          goto _L;
47        }
48      } else
49      _L: /* CIL Label */
50
51      if (sum >= 120) {
52
53        if (sum < 180) {
54
55          printf((char const   * __restrict  )"Grade : C\n");
56        } else {
57
58          printf((char const   * __restrict  )"Result : Fail\n");
59        }
60      } else {
61
62        printf((char const   * __restrict  )"Result : Fail\n");
63      }
64    }
65
66    return (0);
67  }
68  }
```

FIGURE 3.4: Example Program showing code transformation

***Algorithm1 Description:*** The algorithm takes program P as input and generates P with some instrumented code. In line number 5 to 8 each statement of program is visited and if statement matches conditional type then nesting level 1 is assigned. A queue is used to assign levels in Breadth First Manner. For each element of queue, statement becomes parent(line number 10) and all the statements in its true and false block is visited one by one in Line numbers 9 to 11. If statement is conditional, level is assigned as parent's level + 1 (line number 13) and the statement is queued for exploring its block later. Table 3.1 shows levels assigned to conditional statements after code transformation for example program.

TABLE 3.1: Nesting Level Table

| Condition | Statement id | level | Parent id |
|---|---|---|---|
| a >m | 2 | 1 | 0 |
| b >m | 4 | 2 | 2 |
| c >m | 6 | 3 | 4 |
| sum $\geq$ 240 | 9 | 4 | 6 |
| sum $\geq$ 180 | 11 | 5 | 9 |
| sum $<$ 240 | 12 | 6 | 11 |
| sum $\geq$ 120 | 15 | 6 | 11 |
| sum $<$ 180 | 16 | 7 | 15 |

### 3.3.5 Code Instrumentation:

In order to achieve functionalities such as symbolic execution, getting path condition in the form of a level tree at run time, CDG construction program P is instrumented with various function calls at appropriate location in the program.

## 3.4 Symbolic Executor

Symbolic Execution [3] is a program analysis method which means executing a program with symbolic values rather than concrete values, where symbolic values are the representation of program variables driving its behaviour. Each assignment statement in the program is represented as a function or in terms of *Symbolic Variables*. Every conditional statement expression is represented as a prepositional formula in terms of Symbolic Variables. In our implementation, we maintain a symbol table for program variables containing symbolic values and concrete values. For an assignment statement, symbol table is looked up for each variable in the right hand side and replaced with its symbolic value. Table 3.2 shows symbol table for example program.

TABLE 3.2: Symbol Table for Example Program

| Variable | Symbolic value | Concrete Value |
|---|---|---|
| m | s0 | 100 |
| a | s1 | 80 |
| b | s2 | 80 |
| c | s3 | 80 |
| sum | (s1 + s2 + s3) | 240 |

### 3.4.1 Customized Heap Allocation:

In the implementation, symbolic execution for data types like int and float are obtained by maintaining a symbol table as explained above. Symbolic Execution for complex data types such as *Structures, Arrays or Pointers* is achieved by maintaining a heap structure as follows:

- *Arrays :* For an assignment statement such that array variables are accessed to set value of any program variable, memory is allocated in the table with the array name, index, assigned symbolic name and concrete value.While constructing prepositional formula from the expressions in conditions the allocated table is looked up for a symbolic name and if found, condition is obtained in terms of symbolic variable. If not found, a new entry for the variable is inserted int the table and the symbolic value assigned is used. An example symbol table is given in Table 3.3.

TABLE 3.3: Array Datatype Sample Symbol Table

| Array Name | Index | Symbolic Value | Concrete Value |
|---|---|---|---|
| a | 0 | a0 | 12 |
| a | 5 | a5 | 7 |
| b | 3 | b3 | 15 |
| c | 2 | c2 | 3 |
| a | 1 | a1 | 4 |

- *Pointers :* As pointers store address of variables they point to, whenever there is an assignment statement of a variable's address to a single indirection pointer(p) or double indirection pointer (ptr) as shown in the equations 3.1 and 3.2 below. Two entries are added to the symbol table, one for mapping the address stored in pointer variable and other for mapping address stored to its variable.

$$p = \&var \tag{3.1}$$

$$ptr = \&p \tag{3.2}$$

The Symbol table values for above assignment are shown in Table 3.4 considering var's symbolic value was already inserted. For pointer p, we add two entries signifying p stores an address, mapping p to addr and this address belongs to a variable var and hence the entry addr to var. Depending on the number of pointer indirection symbol table is looked up for respective symbolic values iteratively.

TABLE 3.4: Pointer Symbol Table Entry

| Name | Symbolic Value | Concrete Value |
|---|---|---|
| p | addr#var | <address of var > |
| addr#var | var | - |
| var | s0 | 55 |
| ptr | addr#p | <address of p > |
| addr#p | p | - |

For example, program checks if a pointer is NULL or not for this the value stored by the pointer needs to be an address an thus we need one table lookup to get the address. Suppose we have a condition *p > 10 we need to lookup for the value of variable p points to thus we need symbolic value of var which is 's0' and the condition becomes s0 > 10. In we have a pointer with 'n' indirections then we need to lookup the table 2*n + 1 times to fetch the symbolic value.

- *Structures :* Structures are user defined data types and there can be nested structures in the program. For each structure object a table is created mapping its address to the table, the fields of the structure are the contents of the table having symbolic value and concrete values. In case of nested structures, the struct field points to a new table where fields of this nested structure can be accessed and this process continues recursively till there are no more nesting of structures.

  For example, in order to obtain symbolic value for $p.a.b.c$ where p,a and b are of struct type *Figure 3.5* shows the dynamic table allocation. Since, p is of type structure we create a table for p's fields now we found that field a is again of type structure, again is table is allocated which is pointed by p's field a and similalry for b. Assuming c is not a struct type, the table of b contains one entry for c. The number of table lookups required depends on the nesting level of structure whose field is to be accessed.

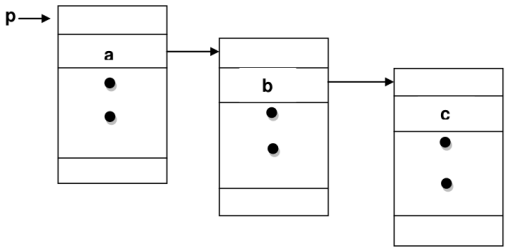



FIGURE 3.5: Symbol Table for Structure Type

## 3.5 TestCase generator for Levelling Module

Once the program P is run with random values a Path Condition is obtained, in order to direct program through different branch we negate one or more conditions obtained from previous run. The new directed path condition obtained is then sent to the constraint solver Z3 [15] to check if new path is feasible if yes we get values for Symbolic Variables and if infeasible we search for a new path by negating next level conditions.

```
1   int binarySearch(int A[], int l, int r, int key)
2   {
3       int m;
4
5       while( l <= r )
6       {
7           m = l + (r-l)/2;
8
9           if( A[m] == key )
10              return m;
11
12          if( A[m] < key )
13              l = m + 1;
14          else
15              r = m - 1;
16      }
17
18      return -1;
19  }
```

FIGURE 3.6: Example Program 2

*Figure 3.6* shows an example program which uses binary search algorithm to find an element in the array. The transformed program with loops unrolled is shown in *Figure 3.7* we will use this program to understand the algorithm stated below for directing path conditions. Let the initial path condition obtained by random run be as stated in *Equation 3.3*

$$PC = (l <= r) \wedge \neg(*(A+m) == key) \wedge (*(A+m) < key) \wedge (l <= r) \wedge (*(A+m) == key) \tag{3.3}$$

```
1   int binarySearch(int *A , int l , int r , int key )
2   {
3     int m ;
4
5     {
6     {
7     if (l <= r) {
8       m = l + (r - l) / 2;
9       if (*(A + m) == key) {
10        return (m);
11      }
12      if (*(A + m) < key) {
13        l = m + 1;
14      } else {
15        r = m - 1;
16      }
17    }
18    if (l <= r) {
19      m = l + (r - l) / 2;
20      if (*(A + m) == key) {
21        return (m);
22      }
23      if (*(A + m) < key) {
24        l = m + 1;
25      } else {
26        r = m - 1;
27      }
28    }
29    }
30  #line 18
31    return (-1);
32  }
33  }
```

FIGURE 3.7: Transformed Example Program 2

The algorithm to construct new path conditions from the level tree obtained is given in Algorithm 2.

---

***Algorithm 2: Directing Path Conditions for TestCase Generation***

---

***Input :*** Program P

***Output :*** Set of Test cases

***Initialize :*** Program is run on random inputs and path condition PC is obtained in form of level tree

**begin**

level $\leftarrow$ 1

```
7  position ← 0
8  EnQueue(Q, PC)
9  While isEmpty(Q) is False do
10   PC ← DeQueue(Q)
11   if position=0 then
12     PC' ← Negate all conditions at level 1
13     level ← level + 1
14     position ← position + 1
15   else if isEmpty(level) then
16    PC ← DeQueue(Q)
17    level ← 1
18    position ← 0
19   else
20     i ← 0
21     while i <position && node != NULL do
22       moveToNextNode() at level 1
23       i++
24     end while
25     if i <position then
26       level ← level + 1
27       position ← 1
28     end if
29     currLevel ← 1
30     while currLevel <level do
31       PC' ← PC' and conditions at currLevel
32       currLevel++
33     end while
34     PC' ← PC' and flip all conditions at level
35     position ← position + 1
36   end if
37   if atleast one condition in PC' is not seen then
38     Get results from Constraint Solver for PC'
39     Run P with new values
40     EnQueue(PC')
41   else
42      go to line 9
43   end if
44 end While
45 end
```

***Algorithm2 Description:*** Let us assume that the path condition obtained when program is run on random inputs is given by *Equation 3.3*. The level tree obtained for path condition is shown in *Figure 3.8* where (l ≤ r) is the condition at level 1 and each node has parent, child and next pointers. The conditions at level 1 seem to be same but they are different due to unrolling of loop. The next pointer points to next node in the same level for a path condition. Parent and child pointers are according to the nesting predecessor and successor relationship. As we can see, (*(A+m)!=key) is at level 2 with parent as (l ≤ r) and points to the next node in the same level which is (*(A+m)==key) having a different parent.

In the algorithm, position signifies number of the node if all nodes at a level are enumerated and level signifies nesting level assigned previously. Position and level together signify the position of parent node whose children are to be flipped in the next run. All nodes at same level are connected to each other in the level tree. During the first iteration (line numbers 11 to 14) all the conditions obtained at level 1 are negated and the path condition obtained is shown in *Equation 3.4*. We negate all conditions of same parent at same level because these are independent to each other and to cover two independent conditions minimum number of testcases required is two.

$$PC'(level1) = \neg(l <= r) \wedge \neg(l <= r) \tag{3.4}$$

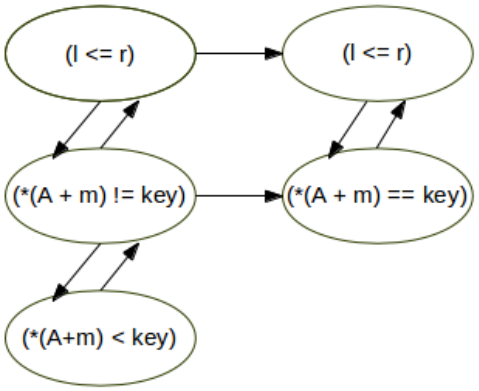



FIGURE 3.8: Path condition as level tree

Line number 15 checks if the next level is completely empty, that is, if no parent is left whose children are yet to be flipped or all the nodes in the current level are leaf nodes, then the testcases from this path condition is completely explored and

we move to the next path condition. From line number 21 to 24 we move to the correct position in the level tree at level. If all the children of the nodes at current level are explored then we move to next level with position as 1 in lines 25 to 27. Once position is located, a new path condition is constructed by adding all the parent conditions as prefix until the desired level (line numbers 30 - 33). All nodes at level are flipped or negated and conjuncted with the path condition. Running algorithm in the level tree of *Figure 3.8* following path conditions are generated:

$$PC'(level2, position1) = (l <= r) \wedge (*(A + m) == key) \tag{3.5}$$

$$PC'(level2, position2) = (l <= r) \wedge \neg(*(A + m) == key) \tag{3.6}$$

$$PC'(level3, position1) = (l <= r) \wedge \neg(*(A + m) == key) \wedge \neg(*(A + m) < key) \tag{3.7}$$

After constructing new path conditions these prepositional formula are given to constraint solver to get the values for symbolic variables that makes the path condition feasible. The program is run with the new inputs generated and the path condition obtained is queued, process gets repeated till there are no new path conditions obtained or there are successive number of more than ten tries to find a new condition which increases coverage and we either got infeasible constraints or the constraints did not improve the coverage. This is where levelling module stops, in our approach we have followed the combined levelling and CDG module so, once levelling module is complete CDG module starts which is discussed in Chapter 4.

## 3.6 Coverage Module

This module measures the coverage according to the test case obtained. All the conditions that are partially or completely seen are extracted, each side true/false is given a weight of 1. The coverage percentage [20] is evaluated according to the following formula:

$$\text{Coverage} = \frac{total\_weight\_of\_conditions \times 100}{2 \times total\_conditions}$$

***Algorithm 3: MCDC Coverage***

1 ***Input :*** Program P, Test Cases
2 ***Output :*** MCDC coverage percentage
3 **begin**
4 **for** each statement s in Program P **do**
5 **if** s is conditional statement **then**
6 C_List ← AddToList(condition(s))
7 **end if**
8 **end for**
9 weight ← 0
10 **for** each testcase t ∈ testcases **do**
11 **for** each condition c ∈ path_condition **do**
12 **if** c evaluates to TRUE **then**
13 TRUE_FLAG ← True
14 **end if**
15 **if** c evaluates to FALSE **then**
16 FALSE_FLAG ← False
17 **end if**
18 **if** both TRUE_FLAG and FALSE_FLAG are True **then**
19 weight ← weight + 2
20 **else if** TRUE_FLAG or FALSE_FLAG are True **then**
21 weight ← weight + 1
22 **end if**
23 **end for**
24 **end for**
25 Coverage ← $\frac{weight \times 100}{2 \times sizeOf\,(C\_List)}$
26 **end**

***Algorithm3 Description:*** From line number 4 to 8, C_List which contains total number of conditions in the program is populated. For each testcase and each condition, populate its TRUE_FLAG or FALSE_FLAG. If both true and false side of the condition is seen then we add a weight of 2, if only one of the sides is seen weight of 1, which is partial weight, is added in line numbers(18 - 22). The coverage is calculated using the weights and total decision statements in the program.

## 3.7 Summary

In this chapter, we discussed about the static analyser, symbolic execution engine, working of *Levelling Module* and coverage analyser. The drawback of levelling module is we obtain less coverage when there are control dependent statements, levelling module will not be able to generate the best coverage. To overcome this, we present second approach followed using CDG Module in the next chapter.

# Chapter 4

# CDG Module

In Chapter 3, we introduced *Levelling* technique for testcase generation. The technique lacks in producing a good coverage if program has control dependent statements. These are the statements whose execution depends on some other statement in the program. We use *Control Dependency Information*[9] of program P with the objective of exploring maximum number of conditions giving high coverage. In the module, we choose a path such that it contains maximum number of uncovered conditions, that is conditions whose atleast one side is not seen. The phases of CDG module are as follows :

- CDG Construction
- CDG Initialization
- Finding Top K paths
- Finding feasible paths using MAXSAT
- Updation of CDG score

## 4.1 Control Dependence Graph Construction:

Control Dependence Graph [10] summarizes the sequence of conditions required to execute a particular statement in the program. A statement Y is control dependent on statement X if X determines whether Y executes or not. In order to determine dependency following 2 conditions must hold between X and Y:

- There is a path from X to Y.
- X is not postdominated by Y. Y post dominates X if all paths from X reaches Y.

*CDG* is constructed from CFG and Post Dominance [16] Relationship which is obtained by finding dominance relationship in the reverse CFG of program. The above two points are standard procedure to construct CDG [17] and we have implemented the algorithm describing all steps as given below:

---

**Algorithm 1: Constructing CDG**

---

```
1  Input : Program P
2  Output : Control Dependence Graph
3  begin
4  Cfg ← Control Flow Graph of P
5  PostDominance Tree ← Dominance relation on reverse Cfg
6    create an 'entry' node with id 0
7      for each statement S ∈ P do
8        for each successor A of S do
9          if A does not postdominate S then
10           makeEdge(node(S) → node(A))
11         else
12           EnQueue(Q, A.predecessors)
13           pred ← DeQueue(Q)
14           while there is a predecessor of A and A does not postdominate pred do
15             EnQueue(Q, pred.preds)
16             pred ← DeQueue(Q)
17           end while
18           if isEmpty(Q) then
19             makeEdge( node(entry) → node(A))
20           else
21             makeEdge( node(pred) → node(A))
22           end if
23       end for
24     end for
25 end
```

***Algorithm1 Description:*** In lines 4-5 cfg and post dominance tree of program P is constructed. For each statement's successor A, if A doesnot occur at all the paths from S then A is control dependent on statement S (lines 9-10). Otherwise, all the predecessors of A are searched till a node is obtained which is not post dominated by A and if such node is not found that means A will be executed whenever program runs and

is not dependent on any node (lines 14-21). CDG generated for our example program of *Figure 3.7* is shown in *Figure 4.1*.

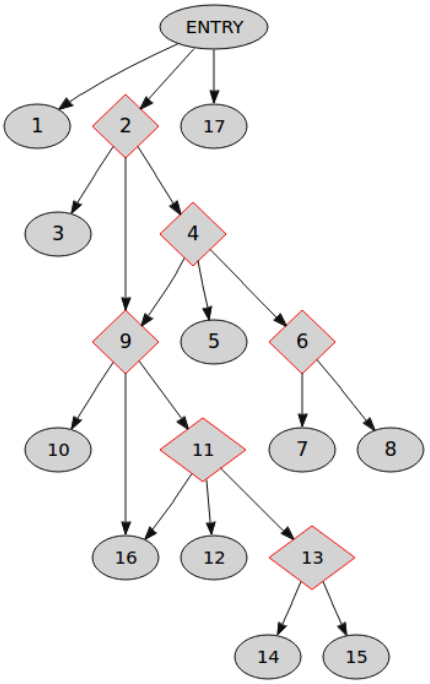



FIGURE 4.1: CDG for Example Program 2

The CDG shown above consists of nodes and edges where diamond shaped nodes are the decision statements. Each node has a number representing the statement id, following table shows the statement id of conditional nodes -

TABLE 4.1: Mapping decisions to statement id

| Condition | Statement id | Line number |
|---|---|---|
| (l ≤ r) | 2 | 7 |
| (*(A + m) == key) | 4 | 9 |
| (*(A + m) <key) | 6 | 12 |
| (l ≤ r) | 9 | 18 |
| (*(A + m) == key) | 11 | 20 |
| (*(A + m) <key) | 13 | 23 |

In the program of *Figure 3.7* we observe that if condition of node 4 is true then we return from the function. Thus the execution of statement 6 and 9 depends on the outcome of node 4. Also, execution of 9 depends on the outcome of 2 as the path going to the true side of 2 might go to exit. Thus we say that statement 9 is control dependent on statement 2 and 4.

## 4.2 CDG Initialization:

In this phase, CDG nodes are initialized with a metric called *Score*. This metric defines the total number of conditions in the subtree whose true or false side or both has not been seen. The initialization is done in the reverse breadth first search manner. Each internal node has score equal to maximum of sum of all scores in true block side and false block side. The leaf conditional nodes have a score of one if any of the true or false side is not seen otherwise has a score of zero if both true and false outcomes is seen. Procedure for initializing or updating scores of all CDG nodes is given in *Algorithm 2*.

**Algorithm 2: Initialize CDG**

```
1  Input : CDG of P
2  Output : CDG with score
3  begin
4  for each node 'n' traversed in reverse BFS order do
5    if n is conditional leaf node then
6      if both side of n has been seen then
7        setScore(n, 0)
8        setOutcome(n,1)
9      else if true side seen then
10       setScore(n, 1)
11       setOutcome(n, 0)
12     else
13       setScore(n,1)
14       setOutcome(n,1)
15     end if
16   else
17     sumTrue ← ∑ (scores of all nodes in true side)
18     sumFalse ← ∑ (scores of all nodes in false side)
19     if sumTrue >sumFalse then
20       setScore(n, sumTrue)
21       setOutcome(n,1)
22     else
23       setScore(n, sumFalse)
24       setOutcome(n,0)
25     end if
26     if both sides of n not seen then
```

```
27          setScore(n, getScore(n)+1)
28        end if
29      end if
30  end for
31  end
```

***Algorithm2 Description:*** In the algorithm above, score gives total number of uncovered conditions and outcome tells whether maximum score is found at true/false block of the condition. We use a function named setOutcome which updates node n with 0/1 representing true/false side where we get maximum uncovered conditions. In line numbers 5-15 score is assigned to the conditional leaf nodes if condition's both side has been seen initialize a score of 0. If one side is seen then score is initialized as 1.

The nodes are visited in reverse breadth first Order. For, internal conditional nodes the sum of all the scores of nodes in true block and false block is calculated and the maximum of these score is assigned to the node done in lines 17-25. If conditional node itself is not been true or false then a score of 1 is added as the uncovered side of this condition also needs to be explored (lines 26-28). CDG initialized with scores after some runs is shown in *Figure 4.2* where each node represents (statement id, score, outcome).

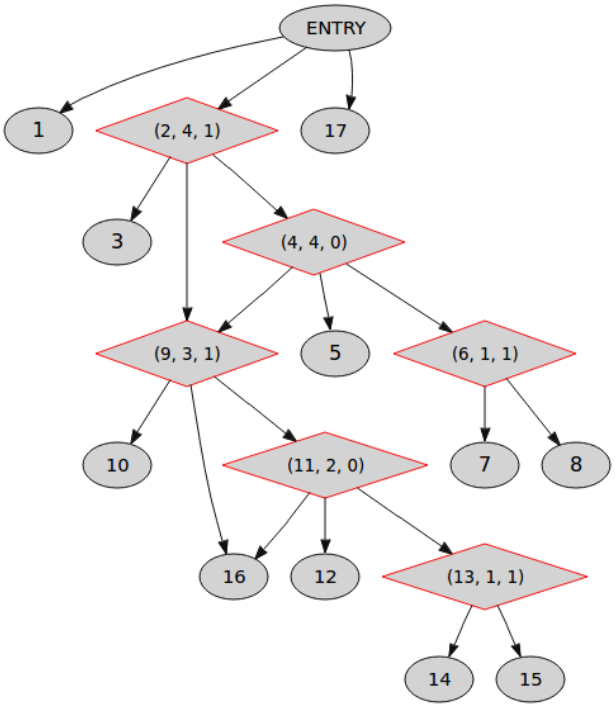



FIGURE 4.2: CDG Initialized for Example Program 2

Table 4.2: Outcome of Conditions seen

| Condition | Statement id | Line number | True side | False side |
|---|---|---|---|---|
| (l ≤ r) | 2 | 7 | 1 | 1 |
| (*(A + m) == key) | 4 | 9 | 1 | 1 |
| (*(A + m) <key) | 6 | 12 | 0 | 1 |
| (l ≤ r) | 9 | 18 | 1 | 0 |
| (*(A + m) == key) | 11 | 20 | 1 | 0 |
| (*(A + m) <key) | 13 | 23 | 0 | 0 |

Table 4.2 shown above, represents the conditon's outcome seen at a particular point. We will consider this table to update score in CDG shown in *Figure 4.2*. Starting with node 13, from table we know that both side of 13 has not been seen yet therefore we give it a score of 1 and outcome 1. Similarly for node 6, its true side is not seen so give it a score of 1 and outcome 1. For internal node, lets consider node 4, sum of true side of node 4 is a score of 0 and false side is score of 4 (summing score of 9 and 6) and the outcome desired is 0. Similarly, other nodes are updated with score and outcome.

## 4.3 Finding top K paths according to weight:

In this phase, top **K** paths are found based on score calculated from previous phase such that topmost path has maximum number of uncovered conditions. At every node, next node in the path is obtained by checking whether true/false block have maximum number of uncovered conditions using score at every node. The parameter **K** is user dependent it can be varied to cover more number of conditions, for our experimental purposes we have used **K** as five. The procedure for determining top k paths is given in *Algorithm 3*.

**Algorithm 3: Top K paths**

```
1 Input : CDG of P
2 Output : K paths
3 begin
4 while k >0 do
5   push(s, root)
6   while isEmpty(s) is false do
7     outcome = getOutcome(s, top)
8     pop(s)
9     if outcome then
```

```
10          for all nodes n in True block of stack top do
11            push(n)
12            Add n to the current path
13          end for
14        else
15          for all nodes n in False block of stack top do
16            pop(s)
17            push(n)
18            Add n to the current path
19          end for
20        end if
21      end while
22      Update CDG scores such that nodes in the current path has been seen with respective
23  outcome
24  end while
25  end
```

***Algorithm3 Description:*** The path starts from root and outcome of each node is found which determines whether maximum number of uncovered conditions are there in true block or false block done in line numbers 5-8. From line numbers 9 to 20 all the conditions in either true block or false block is discovered thus directing the new path conditions. For determining second, third and so on best paths we update the scores of all the nodes in previous path such that respective outcome of node is seen hence decreasing the overall weight of path.

*Figure 4.3* shows the topmost path for CDG initialized. The path from entry is highlighted and marked with the outcome of previous node needed in order to reach next node. At every node we check visiting true/false side will cover more number of conditions. Thus the path obtained is 2 - 4 - (9,6) - 11 - 13 and the path constraint is given as follows:

$$PC'(TopPath) = (l <= r) \ \wedge \ \neg(*(A+m) == key) \ \wedge \ (*(A+m) < key) \wedge \ (l <= r) \\ \wedge \neg(*(A+m) == key) \ \wedge \ (*(A+m) < key) \tag{4.1}$$

Since, the path obtained in *Figure 4.3* is feasible we update scores of CDG such that these nodes are seen and then find the top path in updated CDG as shown in *Figure 4.4*. The second top path is highlighted in the figure with the outcomes labelled which is 2 - 4 - 9 - 11 - 13. For first path there were 4 uncovered conditions and testcase for

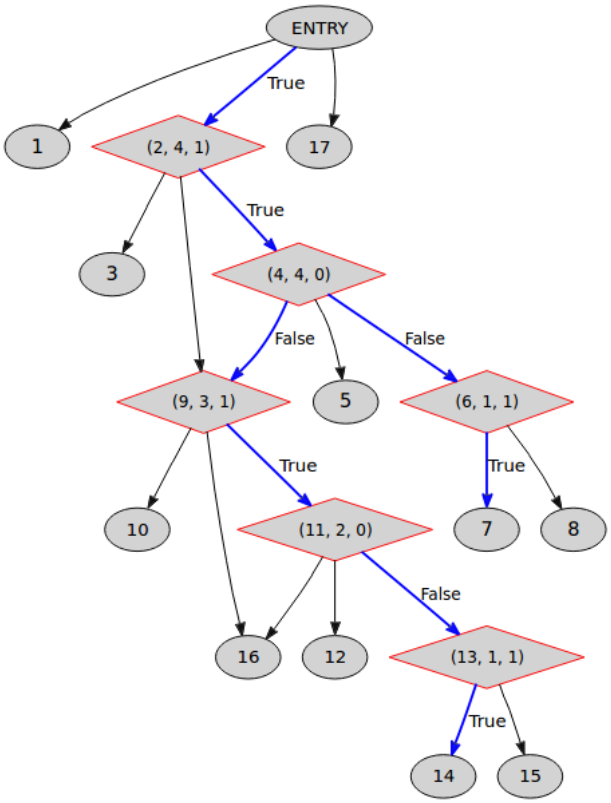



FIGURE 4.3: Top Path for CDG

the path constraint will cover nodes 6 and 11. Thus in this path we are still left with two uncovered conditions 9 and 13, these conditions will be covered by successive paths. Once top paths are found, we revert back the scores changed in CDG.

## 4.4 Finding maximum weight feasible paths using MAXSAT:

The objective of this phase is to find prefix path that is a path starting from entry without any infeasible node in between, having maximum number of uncovered conditions. The topmost paths found in previous phase is checked for its feasiblity by determining maximum number of satisfiable conditions using MAXSAT [19]. Once satisfiable condition set is known, prefix set of conditions is found by checking the nodes in the path from entry. The weight of these paths determines the best satisfiable path and testcases are obtained against its path conditions .

Since the CDG for example program 2 of *Figure 3.7* has no infeasible paths, let us consider CDG of example program 1 *Figure 3.4* and Table 3.1 for mapping of statement ids to conditions. After covering nodes 2, 4 and 6 both sides say we have seen the following path condition before initializing CDG.

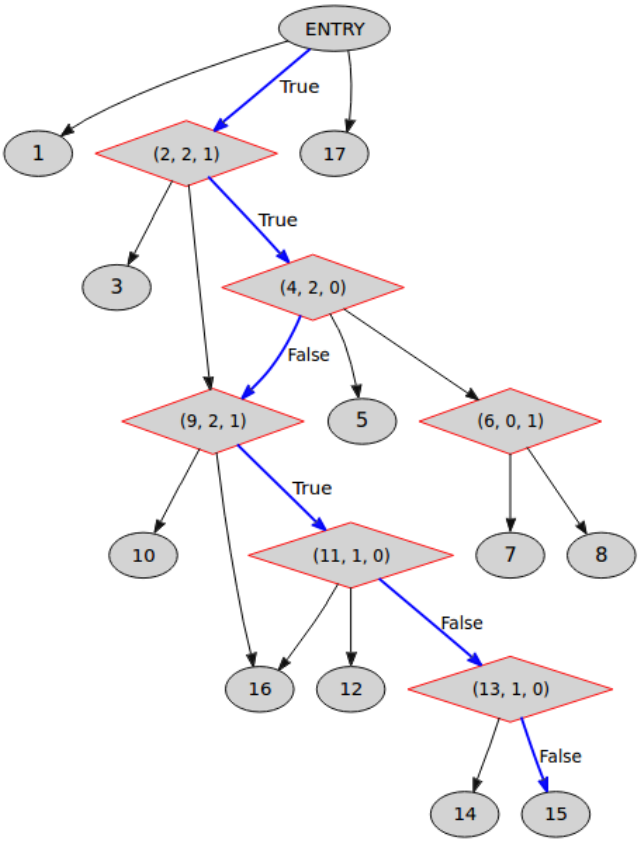



FIGURE 4.4: New path with updated scores

$$PC = \neg(a > m) \wedge \neg(b > m) \wedge \neg(c > m) \wedge \neg(sum >= 240) \wedge (sum >= 180) \wedge (sum < 240) \tag{4.2}$$

Running *Algorithm 3* to find the top path we get the path highlighted in *Figure 4.5*. Path constraint for this path is checked for its feasibility using MAXSAT [19], but the path is infeasible as false of node 9 and false of node 12 makes it infeasible. The conditions for these nodes are:

$$\begin{aligned} PC = \neg(a > max) \wedge \neg(b > max) \wedge \neg(c > max) \wedge \neg(sum >= 240) \wedge \\ (sum >= 180) \wedge \neg(sum < 240) \wedge (sum >= 120) \wedge (sum < 180) \end{aligned} \tag{4.3}$$

MAXSAT [19] will solve the above constraint and give the satisfiable set till node 11, therefore the prefix feasible path obtained will be : 2 - 4 - 6 - 9 - 11. This process is repeated for every path and then we check for the best path amongst them that is a

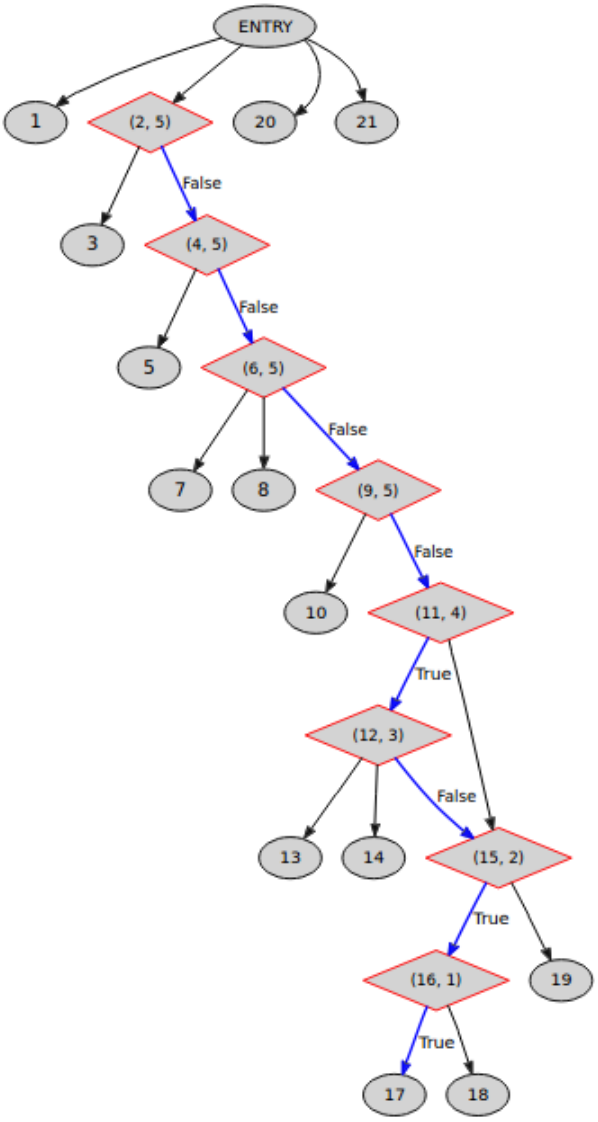



FIGURE 4.5: Infeasible Path for CDG

path which can cover maximum number of conditions and determine testcases for paths accordingly.

## 4.5 Updation of CDG score:

After each test run using the testcases for paths obtained from prevous phase, scores of nodes are updated by first updating the score of leaf nodes in the outcome side (true / false block) to 0 and hence propagating this score upwards towards root with similar procedure as given in *Algorithm 2*. This process of updating scores, finding paths, using MAXSAT, getting testcases, running programs over generated testcases are repeated

either till no paths are left to be explored or there are successive more that ten failed tries to increase the coverage.

## 4.6 Summary

In this chapter, we discussed about utilizing control dependency information for higher coverage. Concept of score defined as a heuristic to drive through various paths and MAXSAT with the objective of getting high coverage were introduced. In next chapter we present and analyse the results obtained by Levelling module ,CDG Module and combined levelling and CDG module.

# Chapter 5

# Experimental Results

In this chapter, we observe the results of experiments over a set of benchmark programs along with analyzing the results obtained from various approach discussed in previous chapters.

## 5.1 Results

Table 5.1 shows summary of results obtained, the major columns define levelling module, CDG module and the combined approach followed. For each module, coverage obtained, number of testcases produced and time taken by the module, which includes code instrumentation and static analysis is shown.

The levelling module, in which we provide nesting level to conditions and direct path conditions so that conditions at a specific level is only flipped, doesnot handle the infeasible constraints, if found we search for a new condition that can be directed. The module searches for new feasible path conditions till there are more than or equal to ten successive tries which results in either infeasible path constraints, path constraints such that all the conditions were already seen or there are no more path conditions to be explored. If any of these are true, the module stops and gives the results in form of Coverage, testcases obtained and time taken for the complete run.

CDG module consists of CDG construction, defining score for each node, finding top K paths, using MAXSAT to find the feasible set of constraints, running the program through best paths where score defines the total number of uncovered conditions in the

subtree. As we are running the module when no conditions were covered, a single test-case covers many conditions at once. The results are produced over forty benchmark programs which are explored till either there are no paths left in CDG or there are more than or equal to ten unsuccessful attempts to increase coverage.

Combined module works with both levelling module and CDG module. At first levelling module explores all possible conditions according to MCDC. The state of program that is, the number of conditions covered, from levelling module is used to initialize CDG and explore maximum number of uncovered conditions.

TABLE 5.1: Results obtained by three approaches

| S No | Program | Function | Levelling Module | | | CDG Module | | | Combined Module | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Coverage | TestCases | Time | Coverage | TestCases | Time | Coverage | TestCases | Time |
| 1-10 | Linpack | idamax | 45 | 4 | 2.028 s | 63 | 4 | 2.080 s | 72 | 6 | 2.180 s |
| | | dscal_ur | 25 | 2 | 2.016 s | 85 | 6 | 2.088 s | 85 | 6 | 2.056 s |
| | | ddot_ur | 23 | 1 | 2.012 s | 65 | 4 | 1.988 s | 65 | 4 | 2.212 s |
| | | daxpy_ur | 28 | 2 | 2.004 s | 67 | 5 | 2.156 s | 67 | 5 | 2.244 s |
| | | dscal_r | 41 | 2 | 2.016 s | 83 | 4 | 2.020 s | 83 | 4 | 1.932 s |
| | | ddot_r | 38 | 2 | 2.008 s | 61 | 3 | 1.994 s | 61 | 3 | 2.064 s |
| | | daxpy_r | 40 | 2 | 2.008 s | 65 | 4 | 2.080 s | 65 | 4 | 2.160 s |
| | | dgesl | 80 | 11 | 2.164 s | 87 | 13 | 2.176 s | 95 | 17 | 2.628 s |
| | | dgefa | 80 | 8 | 2.148 s | 68 | 6 | 2.248 s | 80 | 8 | 1.940 s |
| | | matgen | 75 | 2 | 2.152 s | 50 | 1 | 1.884 s | 75 | 2 | 2.160 s |
| 11-15 | WCET | select | 63 | 3 | 2.020 s | 63 | 4 | 2.176 s | 66 | 4 | 3.104 s |
| | | expint | 71 | 4 | 1.992 s | 71 | 4 | 2.028 s | 71 | 4 | 2.040 s |
| | | qurt_fabs | 100 | 2 | 1.800 s | 100 | 2 | 1.760 s | 100 | 2 | 1.800 s |
| | | qurt_sqrt | 57 | 2 | 1.840 s | 50 | 1 | 1.984 s | 57 | 2 | 2.072 s |
| | | qurt | 66 | 2 | 1.628 s | 66 | 2 | 2.116 s | 66 | 2 | 1.896 s |
| 16-19 | BRNS | selectMedian | 31 | 3 | 2.020 s | 31 | 4 | 2.284 s | 31 | 3 | 2.192 s |
| | | UpCounter | 95 | 10 | 1.852 s | 95 | 10 | 2.200 s | 95 | 10 | 1.876 s |
| | | AAS | 56 | 7 | 2.036 s | 97 | 14 | 2.560 s | 97 | 15 | 2.360 s |
| | | GroupAAS | 35 | 5 | 2.610 s | 61 | 8 | 2.696 s | 64 | 10 | 3.000 s |
| 20-22 | Heap Sort | adjust | 93 | 5 | 2.028 s | 81 | 4 | 2.036 s | 93 | 5 | 1.868 s |
| | | heapify | 80 | 3 | 1.928 s | 70 | 3 | 2.036 s | 80 | 3 | 2.056 s |
| | | heapSort | 100 | 2 | 2.008 s | 100 | 2 | 2.380 s | 100 | 2 | 1.720 s |
| 23 | Insertion Sort | insertionSort | 79 | 4 | 2.008 s | 83 | 4 | 2.056 s | 83 | 5 | 2.052 s |
| 24 | Bubble Sort | bubbleSort | 70 | 3 | 1.956 s | 83 | 5 | 2.104 s | 83 | 4 | 1.924 s |
| 25-26 | Quick Sort | partition | 62 | 3 | 2.036 s | 87 | 3 | 1.992 s | 100 | 4 | 1.829 s |
| | | quickSort | 60 | 2 | 2.028 s | 60 | 3 | 1.984 s | 60 | 2 | 2.004 s |
| 27 | Merge two arrays | merge | 89 | 4 | 1.912 s | 96 | 5 | 2.172 s | 96 | 5 | 1.956 s |
| 28-29 | Tic Tac Toe | checkwin | 84 | 13 | 2.156 s | 90 | 14 | 2.988 s | 98 | 20 | 2.568 s |
| | | tictactoeMain | 38 | 2 | 2.020 s | 52 | 5 | 2.576 s | 56 | 6 | 2.532 s |
| 30 | Triangle | triangleCheck | 47 | 5 | 2.028 s | 79 | 9 | 2.320 s | 79 | 9 | 3.496 s |
| 31 | Binary Search in Array | binarySearch | 41 | 2 | 2.004 s | 91 | 5 | 2.036 s | 91 | 5 | 1.928 s |
| 32 | ATM | atm | 95 | 8 | 2.020 s | 95 | 8 | 3.140 s | 95 | 8 | 1.960 s |
| 33 | Median of 2 Sorted Arrays | getMedian | 87 | 5 | 2.020 s | 75 | 3 | 2.112 s | 87 | 5 | 2.048 s |
| 34-40 | TCAS | Own_Below_Threat | 100 | 2 | 1.744 s | 100 | 2 | 1.780 s | 100 | 2 | 1.744 s |
| | | Own_Above_Threat | 100 | 2 | 1.724 s | 100 | 2 | 1.740 s | 100 | 2 | 1.724 s |
| | | Positive_RA_Alt_Thresh | 100 | 5 | 1.724 s | 100 | 5 | 1.868 s | 100 | 5 | 1.724 s |
| | | Inhibit_Biased_Climb | 100 | 2 | 1.728 s | 100 | 2 | 1.760 s | 100 | 2 | 1.728 s |
| | | Non_Crossing_Biased_Climb | 37 | 2 | 2.008 s | 37 | 3 | 2.124 s | 37 | 2 | 2.048 s |
| | | Non_Crossing_Biased_Descend | 37 | 2 | 2.008 s | 37 | 3 | 2.952 s | 37 | 2 | 2.004 s |
| | | alt_sep_test | 61 | 5 | 2.008 s | 61 | 5 | 2.144 s | 61 | 5 | 2.068 s |

From *Table 5.1* we observe that for levelling module, most of the programs produces less coverage when compared to maximum coverage that can be achieved. Although, for 6 progams from heapSort, TCAS and WCET 100 % coverage is achieved. The minimum

coverage percentage for this module is 23 %. For the above set of programs, maximum number of testcases obtained is 13 and minimum is 1. The less coverage is due to infeasible paths which shows scope of improvement in the module. Time taken by module is around 2 seconds 12 milliseconds which includes the time taken to instrument the program and generate all possible testcases. The average coverage obtained with levelling approach is **65.225 %** for the set of forty programs.

For CDG Module, we observe an increase in coverage of programs as compared to levelling module. The maximum coverage obtained is 100 percent and minimum is 31. The maximum number of testcases obtained is 14. The average coverage obtained by using CDG module is **75.125 %**. The time taken for generating test cases using CDG module is approximate 2 seconds and 300 milliseconds including program instrumentation and static analysis.

The results obtained by using combined approach shows an increase in the coverage for most of the programs over levelling module. Some programs like matgen, GroupAAS shows an increase in coverage from CDG module. Around 52 percent of the programs are observed to get a coverage of more than 80 %. The average coverage obtained by using combined approach is **78.275 %**. Time taken by the module is approximate 2 seconds and 500 milliseconds which involves all the instrumentation and static analysis of code along with the execution time of levelling and CDG module. The maximum number of testcases obtained by using this approach is 20 and hence increase in coverage.

## 5.2 Comparison on TestCases

Table 5.2 shows the number of testcases generated when a module is forced to exploit conditions and reach various thresholds used for comparison. In order to compare number of testcases produced by levelling module, CDG module and combined module, we have set thresholds to be 40 percent, 60 percent and 80 percent. The star mark used in table shows that for a program the module cannot reach the threshold even after exhaustive search.

For program AAS, the number of testcases generated by levelling and combined module is more than that of CDG module for same threshold percentage. Programs such as dgesl, expint produces same number of testcases for all three approaches. For programs such as dgefa, AAS, checkwin we observe that CDG module can reach a coverage

threshold of 80 percent in less number of testcases than levelling module and combined approach. ATM program shows that levelling module provides threshold coverage with less number of testcases as compared to CDG module and combined module. Whereas combined module always generate testcases greater than or equal to levelling and CDG module.

TABLE 5.2: Comparison on Number of TestCases

| | **Program** | **Function** | **40 percent** | | | **60 percent** | | | **80 Percent** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | **Levelling** | **CDG** | **Combined** | **Levelling** | **CDG** | **Combined** | **Levelling** | **CDG** | **Combined** |
| **1-10** | **Linpack** | idamax | 3 | 3 | 3 | * | 4 | 6 | * | * | * |
| | | dscal_ur | * | 3 | 3 | * | 4 | 4 | * | 5 | 5 |
| | | ddot_ur | * | 3 | 3 | * | 4 | 4 | * | * | * |
| | | daxpy_ur | * | 3 | 3 | * | 4 | 4 | * | * | * |
| | | dscal_r | 2 | 2 | 2 | * | 3 | 3 | * | 4 | 4 |
| | | ddot_r | * | 3 | 3 | * | 3 | 3 | * | * | * |
| | | daxpy_r | 2 | 2 | 2 | * | 4 | 4 | * | * | * |
| | | dgesl | 3 | 3 | 3 | 6 | 7 | 6 | 11 | 11 | 11 |
| | | dgefa | 2 | 4 | 2 | 5 | 5 | 5 | 13 | * | 13 |
| | | matgen | 1 | 1 | 1 | 2 | * | 2 | * | * | * |
| **11-15** | **WCET** | select | 1 | 1 | 1 | 4 | 3 | 2 | * | * | * |
| | | expint | 2 | 2 | 2 | 4 | * | 4 | * | 3 | 4 |
| | | qurt_fabs | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 |
| | | qurt_sqrt | 1 | 1 | 1 | * | * | * | * | * | * |
| | | qurt | 1 | 1 | 1 | 2 | 2 | 2 | * | * | * |
| **16-19** | **BRNS** | selectMedian | * | * | * | * | * | * | * | * | * |
| | | UpCounter | 3 | 3 | 3 | 5 | 4 | 5 | 8 | 7 | 8 |
| | | AAS | 5 | 3 | 5 | 8 | 4 | 8 | 12 | 8 | 12 |
| | | GroupAAS | * | 4 | 6 | * | 8 | 9 | * | * | * |
| **20-22** | **Heap Sort** | adjust | 1 | 1 | 1 | 2 | 2 | 2 | 4 | 4 | 4 |
| | | heapify | 1 | 1 | 1 | 3 | 3 | 3 | 3 | * | 3 |
| | | heapSort | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 |
| **23** | **Insertion Sort** | insertionSort | 1 | 1 | 1 | 2 | 2 | 2 | * | 4 | 5 |
| **24** | **Bubble Sort** | bubbleSort | 1 | 1 | 1 | 2 | 3 | 3 | * | 5 | 4 |
| **25-26** | **Quick Sort** | partition | 2 | 2 | 2 | 3 | 2 | 3 | * | 3 | * |
| | | quickSort | 2 | 1 | 2 | 2 | 2 | 2 | * | * | * |
| **27** | **Merge two arrays** | merge | 2 | 2 | 2 | 3 | 2 | 2 | 4 | 4 | 4 |
| **28-29** | **Tic Tac Toe** | checkwin | 3 | 2 | 3 | 5 | 2 | 5 | 10 | 8 | 10 |
| | | tictactoeMain | * | 3 | 3 | * | 5 | * | * | * | * |
| **30** | **Triangle** | triangleCheck | 5 | 2 | 5 | * | 4 | 6 | * | * | * |
| **31** | **Binary Search in Array** | binarySearch | 2 | 2 | 2 | * | 3 | 3 | * | 5 | 5 |
| **32** | **ATM** | atm | 1 | 1 | 2 | 3 | 4 | 4 | 5 | 6 | 7 |
| **33** | **Median of 2 Sorted Arrays** | getMedian | 1 | 1 | 1 | 2 | 2 | 2 | 5 | * | 5 |
| **34-40** | **TCAS** | Own_Below_Threat | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 |
| | | Own_Above_Threat | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 |
| | | Positive_RA_Alt_Thresh | 2 | 2 | 2 | 3 | 3 | 3 | 4 | 4 | 4 |
| | | Inhibit_Biased_Climb | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 |
| | | Non_Crossing_Biased_Climb | * | * | * | * | * | * | * | * | * |
| | | Non_Crossing_Biased_Descend | * | * | * | * | * | * | * | * | * |
| | | alt_sep_test | 5 | 4 | 5 | 5 | 4 | 5 | * | * | * |

Thus, from the table above we conclude that there are some cases for which levelling module performs better than CDG module. Combined approach results in more number of testcases with higher coverage as there are programs like dgefa and heapify for which CDG can not reach higher threshold.

## 5.3 Overall Analysis

In this section, we will compare the results obtained by using Levelling module, CDG module and the combined approach.

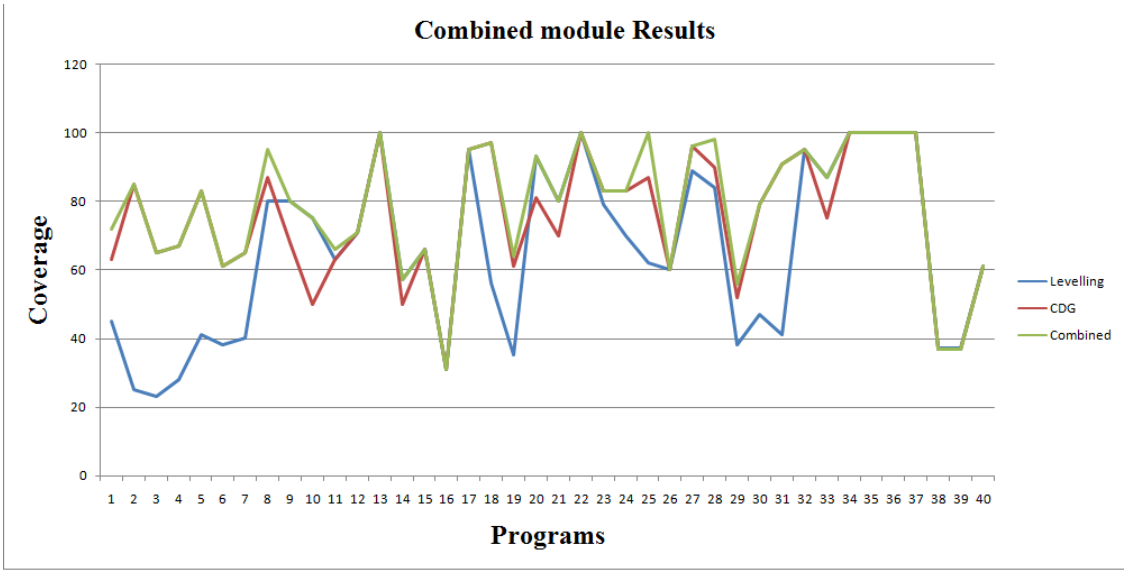



FIGURE 5.1: Analysis of Coverage

From the graph shown in *Figure 5.1* we observe the coverage obtained by three approaches for a set of forty programs. As seen in section 5.1, the combined approach gives a coverage either equal to levelling module or CDG module for most of the cases. Whereas we also observerd that combined approach results in more coverage than levelling and CDG in many cases.

In the graph shown above, blue line signifies levelling module, red represents CDG module and green shows combined module. We observe that coverage obtained by combined approach that is green line is mostly higher than other modules resulting in best coverage. From levelling to combined approach we get an increase of 12.825 % in the coverage and CDG to combined approach results in an increase in 2.875 % of increase in coverage.

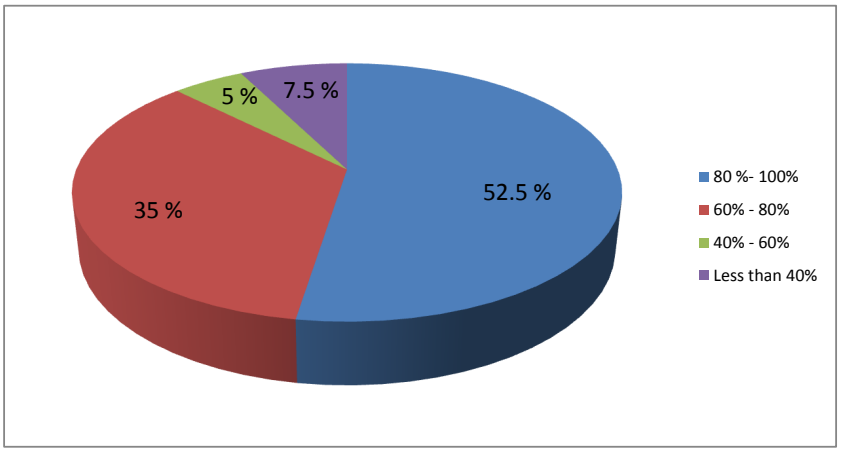



FIGURE 5.2: Distribution of Coverage

In *Figure 5.2*, we show the distribution of program according to different range of coverage for combined approach. Around 52.5 % of programs from set of forty programs achieved a coverage greater than 80 percent and 35 % were able to cover more than 60 percent while 12.5 % of programs generate testcases for coverage less than 60 percent. Thus, we conclude here that Combined approach is better than levelling module and CDG module, providing higher coverage.

# Chapter 6

# Conclusions and Future Work

In the thesis, we have proposed a new approach to automatic testcase generation for high MCDC coverage of program under test. Here we have developed concolic tester which is used along with levelling module and CDG module to generate MCDC testcases.

Levelling module consists of providing nesting levels and directing path conditions such that all nodes at same level are flipped together. This module along with concolic executor provides an average coverage of 65.225 percent for set of forty programs and average time taken for the module is 2.012 seconds.

CDG module consists of CDG creation, initialization, finding top **K** paths, using MAXSAT to generate path constraints having maximum number of uncovered conditions to generate testcases along with concolic tester. Average coverage of 75.125 percent is obtained with average time taken of 2.3 seconds. The combined approach of levelling module followed by CDG module provides an average coverage of 78.275 percent and the time taken using this approach is 2.5 seconds. In terms of coverage, the combined approach outperforms both levelling module and CDG module.

Table 6.1: Features supported by Tool

| **Category** | **Features** | **Supported by Tool** |
|---|---|---|
| Data Types | **Integers, Floats, Arrays, Pointers, Structures** | ✓ |
| | Char, Strings, Enum, Typedef, Union, Multidimensional Arrays | × |
| Operators | **Arithmetic, Logical, Relational, Ternary operators** | ✓ |
| | Bitwise operators | × |
| Control Structures | **If/else, Loop** | ✓ |
| | File Handling | × |
| | Heap manipulation | × |

Table 6.1 shows a brief summary of the constraints on the tool developed. The benchmark programs and results are according to the features supported by the tool. Hence, the results are subject to change once the unsupported features are handled.

We briefly outline the following possible extentions to work -

- In the thesis, interfunctional prodecure calls are not handled. Including this will let us perform symbolic execution for these functions returning values. These values can be generated and hence will increase the coverage.
- For symbolic execution on struct in C, we have used a customized heap allocation approach which creates a nesting of symbol tables for each structure object. This process utilizes lot of memory and is inefficient as it involves searching symbolic name for a variable in nested tables. Optimizing this can reduce memory required and time for table lookup.
- Machine learning techniques can be used to predict infeasible paths of the program in advance.
- Slicing is a program decomposition method in which statements are extracted relevant to a particular computation. The work can be extended by calculating MCDC percentage on a sliced version of program to improve coverage.

# Bibliography


[1] 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.

[2] *CIL-Infrastructure for C Program Analysis and Transformation.* Website http://www.cs.berkeley.edu/ necula/cil/.

[3] Koushik Sen, Cristian Cadar. *Symbolic Execution for Software Testing : Three Decades Later*, Imperial College London, University of California Berkley.

[4] Koushik Sen.*Concolic Testing*, EECS Department, UC Berkeley, CA, USA.

[5] Patrice Godefroid, Nils Klarlund, Koushik Sen. *DART:Directed Automated Random Testing.* Bell Laboratories, Lucent Technologies, University of Illinois at Urbana-Champaign.

[6] John Joseph Chilenski and Steven P. Miller. *Applicability of Modified Condition/Decision Coverage to Software Testing* Software Engineering Journal, September 1994, Vol. 9, No. 5, pp.193-200.

[7] Koushik Sen, Darko Marinov, Gul Agha - University of illinois. *A Concolic Unit Testing Engine for C* 13th ACM SIGSOFT international symposium jointly with 10th European Software engineering conference.

[8] King, J.*Symbolic Execution and Program Testing.* Communications of the ACM,19(7), ACM press, pp. 385-394.

[9] Zeina Awedikian. *Automatic Generation of Test Input data for MC/DC test coverage.* Soccer Lab, Ecole Polytechnique de Montreal.

[10] Kenneth M. Anderson CSCI, Spring 2000, Judith Stafford. *A Compositional Approach to Interprocedural Control Dependence Analysis.* Software Engineering Research Laboratory, University of Colorado, Boulder.

[11] Michael Whalen, Gregory Gay, Dongjiang You *Observable Modified Condition/Decision Coverage* in proceedings of 2013 International Conference on Software Engineering.

[12] B.W.P.C.X.Liu, H.Liu and X. Cai, *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, 2005.

[13] Prasad Bokil, Priyanka Darke, Ulka Shrotri, R. Venkatesh *Automatic Test Data Generation for C Programs* In proceeding SSIRI'09 proceedings of 2009 Third IEEE International Conference on Software Integration and Reliability Improvement.

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

[15] Z3-Constraint Solver. Website- http://z3.codeplex.com/.

[16] *PostDominance* - http://en.wikipedia.org/wiki/Dominator_%28graph_theory%29

[17] *Control Dependence Graph* - www.cs.colorado.edu/~kena/classes/5828/s00/lectures/lecture15.pdf.

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

[19] *Maximum Satisfiability Problem* en.wikipedia.org/wiki/Maximum_satisfiability_problem.

[20] Sangharatna Godboley. *Improved Modified Condition/ Decision Coverage using Code Transformation Techniques.* Department of Computer Science and Engineering NIT, Rourkela May 2013.