

Enhancement of Branch Coverage using Java Program Code Transformer

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

Bachelor of Technology

in

Computer Science and Engineering

by

Kumar Satyam
(Roll: 111CS0115)

under the supervision of

Prof. D.P. Mohapatra

NIT Rourkela



Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Orissa, India
May, 2015

Certificate



09 May, 2015

This is to certify that the work in the thesis entitled **Enhancement of Branch Coverage using JPCT** by **Kumar Satyam**, Roll No. 111CS0115, is a record of an original research work carried out by him under my supervision and guidance in partial fulfillment of the requisites for the award of the degree of Bachelor of Technology in Computer Science and Engineering. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Prof. Durga Prasad Mohapatra
Dept. of Computer Science and Engineering
National Institute of Technology
Rourkela - 769008

Acknowledgement

I would like to express my sincere gratitude to my supervisors, Prof. Dr. D.P. Mohapatra for providing me with a platform to work on challenging areas of Enhancement of Branch Coverage Techniques. His guidance, support, patience, motivation and immense knowledge have been inspiration to my work.

I would like to express deepest appreciation to Sangharatna Godbole (Phd Research Scholar) for all his inputs, discussions and contributions. I am thankful to all my lab mates for their encouragement, understanding and useful discussions.

Kumar Satyam

Contents

1	INTRODUCTION	2
1.1	Basic Definitions	2
1.2	Objective of our work	4
1.3	Organisation of the Thesis	4
2	LITERATURE REVIEW	5
2.1	MC/DC coverage using Code Transformer	5
2.2	Other Related Works	5
2.3	Summary	6
3	PROPOSED WORK	7
3.1	Java Program Code Transfer	7
3.2	Cobertura	7
3.2.1	Test Report using Cobertura	8
3.2.2	Sample of Cobertura Output	9
3.3	Schematic Representation	9
3.4	Flow of the Work	10
3.5	Proposed Algorithm	11
4	RESULTS	15
4.1	Input Program for Step 1 :	15
4.2	Output of Step 1 (Predicates Identified) :	16
4.3	Another input program for step 1 :	17
4.4	Output of the input 2(predicates identified)	18
4.5	Output of Step 2 (Generation of SOP) :	19
4.6	Output of Step 3 (Quine Mc-cluskey Method) :	20
4.7	Input program for Step 4	21
4.8	Output of Step 4 (Insertion of empty if-else statement)	22
5	CONCLUSION AND FUTURE WORK	23

List of Figures

3.1	Output sample of cobertura	9
3.2	Schematic representation of the work	10
3.3	Flow of the work	11
4.1	First Input prog for step 1	15
4.2	output of step 1	16
4.3	2nd input prog for step 1	17
4.4	output of 2nd input prog of step 1	18
4.5	output of step 2	19
4.6	output of step 3	20
4.7	input prog for step 4	21
4.8	output of step 4	22

Abstract

In traditional concolic testing branch coverage is low. Automated technique appears as a promising technique to reduce test time and effort. In this project we used a code transformation technique. We take input a simple java program and transform it using various algorithms. We have used four algorithms to transform the given code into a transformed code. We used Quine Mc-cluskey method and Petric methods to achieve the transformed code. After transforming the code, we pass it through a tool called Cobertura which gives the branch coverage of that transformed code. Here we observe that the percentage of branch coverage using the transformed code is greater than the coverage of original code. Hence, our technique helps to achieve a significant increase in branch coverage in comparison to traditional techniques.

Chapter 1

INTRODUCTION

In software testing field the importance of branch coverage cannot be neglected. A very slight increase in coverage can improve the test cases significantly. In this project we design a Code Transformer to increase the coverage of the program. We use four algorithm to design the PCT. We are doing it for JAVA programs only. We implemented the algorithm using JAVA programming language. The role of JPCT is very significant in increasing the coverage. We confirm the increase in the coverage by passing both the programs the original and the transformed through cobertura and note coverage percentage of both the cases.

1.1 Basic Definitions

Condition[4]: 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.

Decision[4]: 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 $((z == 0) || (t < 1)) \&\&((s > 10) || (g == 0))$, there are four conditions in the decision which are $(z == 0)$, $(t < 1)$, $(s > 10)$ and $(g == 0)$.

Black Box Testing : In Black Box testing the internal structure of the program being tested is not known to the tester. It is used for higher level of testing like acceptance and system testing. Black Box testing does not require programming and implementation knowledge.

White Box Testing : Unlike Black Box testing, in White Box testing the internal structure of the code is known to the tester. This strategy is used for lower levels of testing like unit testing and integration testing. Here, programming and implementation knowledge is required.

Statement Coverage : It is a type of coverage in which the execution of all the statements in a code is done at least once. It is used to evaluate the number of statement in the source code which has been executed. It is also known as the line coverage or segment coverage. The property of statement coverage is that it covers only the true conditions. Here, each and every line is not necessary to be checked. It measures the quality of the code written.

Branch Coverage[4] : Unlike the statement coverage it covers both true and false statements. IF statements, Case statements, loop control statements are decision statements. Branch Coverage is used to evaluate or validate that all branches in the code is reached.

Multiple Condition Coverage : In this coverage all the cases of condition for each decision is evaluated. This coverage is also known as Condition Combination coverage. The characteristic of this coverage is that if there will be p conditions then no. of test will be 2^p .

Modified Condition / Decision Coverage[4] : In this coverage every entry and exit point in the program has been invoked at least once. Every condition in the code has taken all possible outcomes once. Each condition has been shown to effect that decision outcome independently.

Prime Implicant : It is a term in Sum of product expression that cannot be concatenated with another term to delete a variable.

Concolic Testing [8]: It is a interbred software authentication method that relates concrete execution with symbolic execution. It uses executable paths in the way as symbolic execution. The main motto of Concolic Testing is to find bugs in real-world software instead of demonstrating code correctness.

1.2 Objective of our work

The main objective of our work is to increase the branch coverage of a java program. To achieve this task our aim is to design a Java Program Code Transformer. The JPCT will take a java program as input and give a transformes program as output. The branch coverage of the transformed program should be greater than the original program. We will compare this increase in the coverage by a tool called Cobertura. First we will take the coverage of the original program through this tool and then the coverage of the transformed program would be compared with it. Our main motto is to increase the value of the coverage.

1.3 Organisation of the Thesis

Chapter 2 gives a brief review of the existing work done related to our work. We describe the work related to program code transformer.

Chapter 3 gives an introduction to our proposed work, flow of the work and tools used in our project. In this section we also described the steps used in the formation of Java Program Code Transformer. This chapter also gives a brief introduction of the Algorithms used in this project.

Chapter 4 consists of the results obtained after implementation of the algorithmms. It also gives screenshots of the input taken in all the steps. All the figures and screenshots are explained briefly.

Chapter 5 concludes the project and gives a summary of our work. It also describes the future scope of the project.

Chapter 2

LITERATURE REVIEW

2.1 MC/DC coverage using Code Transformer

Godbole [4] proposed an approach to improve MC/DC coverage using code transformation technique. MC/DC is a standard coverage criterion but existing automated test data generation methods like Concolic Testing do not support it. To tackle this problem, an automated approach to generate test data has been found that helps in achieving an increment in MC/DC coverage of a code being tested. To achieve this, code transformation method is used. This is done by inserting transformed program into a CREST TOOL [4]. It gives test suits and increase the coverage.

2.2 Other Related Works

Das [3] proposed an approach to generate MC/DC test data automatically. This approach helps to increase MC/DC coverage by approximately 21 percent as compared to the existing Concolic testing.

Godbole et al.[6] proposed an approach to analyse time of Evaluation of coverage percentage for C programs using Advanced Program code transformer. Godbole et al.[8] proposed a framework to compute MC/DC percentage for distributed test case generation. This approach uses many client nodes to generate the non-redundant test cases in a distributed and scalable manner.

Tiwary [5] proposed an approach to generate test cases automatically for high MC/DC coverage. To achieve this, the concept of

Concolic testing which is a combination of symbolic execution and concrete execution is used.

Godbole et al.[9] proposed an approach to measure percentage of coverage of C code using code Slicer and Crest Tool. An augmented method is used to generate a test suite that helps in measuring coverage percentage of a program.

2.3 Summary

In this chapter we discussed about related work on Enhancement of Branch Coverage and Program Code Transformer. We also discussed other related works.

Chapter 3

PROPOSED WORK

Mainly two component levels are there in our proposed framework.

3.1 Java Program Code Transfer

It modifies the program under test by generating and inserting additional condition statements based on the requirements of Branch coverage criterion.

Steps

- First scan the program and list out the predicates . For this we use the 1st algorithm.
- Add all the predicates in a list.
- For each predicate, generate we generate corresponding SOP expression. Method is called Petric Method.
- Then Minimize the SOP expression using Quine-Mcluskey method.
- Then re-construct the predicates using minimized SOP expression. Method is called Reverse Petric method.
- Then we generate the nested if-else statement for each predicate.

3.2 Cobertura

It is a coverage tool. It shows how many lines of code are touched. In most cases we'll use cobertura to see how good our regression test

are. We can also use it to see how many lines of codes are reached or to improve code that is accessed a lot.

It wil generate HTML or XML reports. Cobertura is meant to be used with ant but it also works with the command line and plugins are under development for Maven2 and Eclipse.

3.2.1 Test Report using Cobertura

The first column shows package name, second column shows how many classes are in the package. The third column is named lined coverage. Line coverage is the amount of lines of code that are executed. A useful line is a line of code where something happens (so no curly braces or new lines).

The next column gives us information about the branch coverage. Branch coverage is the coverage of the decision points in your class (this can be if/else/then loop). When your code only reaches the first part the if loop and skips the else part you will have 50 percent branch coverage. We were probably thinking we'll never get a higher coverage than 50 percent with if/else/then loop. With cobertura it is possible to execute our code as many times as we want to test all the branches we have.

The last column is the so-called McCabe cyclomatic code complexity. This is a way to determine how complex our code is. One of the methods McCabe uses is measuring the amount of decision point in our code. A higher rating means more complex code.

A higher coverage does not mean that our code is bug-free. It is very easy to write code that touches many lines but doesn't do really anything. We should always keep in mind to write useful tests and then cobertura will be a usefu tool.

3.2.2 Sample of Cobertura Output










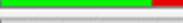

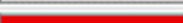






Package ^	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	38	50% 	62% 	1.078
net.jvw.fdb5	3	77% 	91% 	1
net.jvw.fdb5.logic	3	57% 	73% 	0
net.jvw.fdb5.lucene	2	45% 	58% 	0
net.jvw.fdb5.model	5	66% 	80% 	1.286
net.jvw.fdb5.model.ibatis	9	70% 	N/A 	1
net.jvw.fdb5.model.rss	1	69% 	0% 	1.222
net.jvw.fdb5.struts	13	21% 	29% 	1.571
net.jvw.taqlibs.tooltip	2	0% 	N/A 	1.2

Figure 3.1: Output sample of cobertura

3.3 Schematic Representation

The following figure gives an overview of how we would proceed with the input program and after getting the transformed program we have to check its increase through a coverage tool called Cobertura. The output of the Java Program Code Transformer will become the input of the Cobertura.

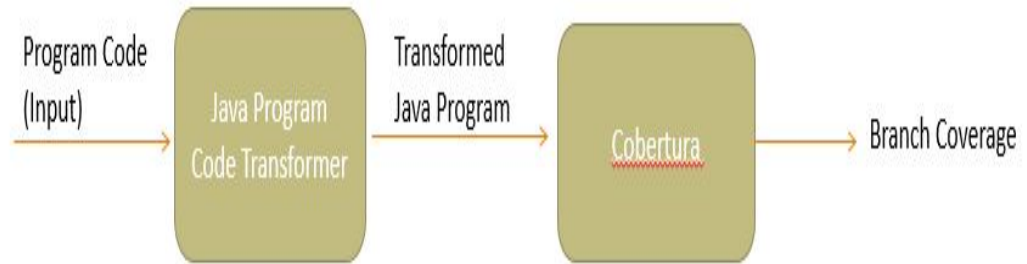


Figure 3.2: Schematic representation of the work

3.4 Flow of the Work

The following figure shows the flow of our work. To design a Java Program Code Transformer we have basically four modules. Identification of predicates, Generation of SOP, Quine McClusky method and the insertion of nested empty if-else statements. We take a java program as input and gives a transformed code as output using these four modules.

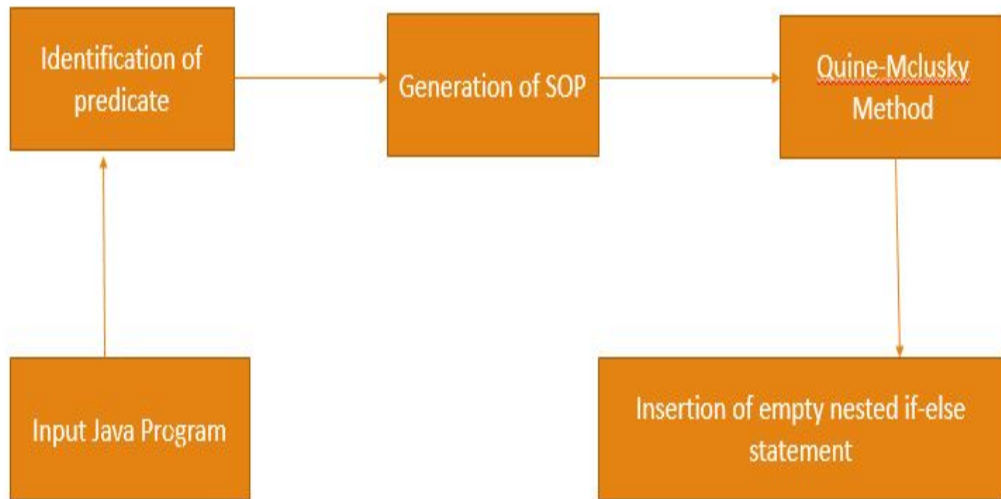


Figure 3.3: Flow of the work

3.5 Proposed Algorithm

We have used three Algorithms in this project. first Algorithm is used for identification of predicates. This algorithm scans the input program and gives the list of predicate found in that as output. second algorithm is used to generate sum of product. It takes the list of predicates as input and convert them into sum of product form. It also minimizes the SOP using the Quine McClusky method. Finally, third algorithm inserts the nested empty if-else statement in the program and gives transformed program as the output.

Explanation of 1st Algorithm : 1st algorithm give the procedure to design Program Code Transformer[7]. First it search for the Predicates[4] and if it finds one , immediately add that predicate in a list called add_in_list. Then for each predicate in the list it calls a function called gen_sumofproduct to give the corresponding SOP statement. After generating the SOP statement it minimizes it using a function Minimiz_QM and in the last step it inserts nested empty if-else statements in the given input program to give the

1st Algorithm: JPCT [7].

Input: Y // Program Y is in Java

Output: Y ' // program Y ' is transformed program

Begin

```
1: for each statement  $s \in Y$  do
2:   if && or ||or unary ! found in  $s$  then
3:     List_Predicate  $\leftarrow$  add_in_List( $s$ )
4:   end if
5: end for
6: for each predicate  $p \in$  List_Pred do
7:   P_SOP  $\leftarrow$  gen_sumofproduct( $p$ ) //Generation of SOP
8:   P_Minterm  $\leftarrow$  convert_to_Minterm(P_SOP)
9:   P_Min  $\leftarrow$  Minimize_QM(P_Minterm)
10:  List_Statement  $\leftarrow$  Generation_nested_if-else_JPCT(P_Min)
11:  Y '  $\leftarrow$  entry_of_code(Statement_Record,Y)
12: end for
13: Exit
```

transformed program.

Explanation of 2nd algorithm: This algorithm is used for converting the predicates into SOP form. Here we convert each minterm into binary form and process them to find Sop. We process this task by checking each bits individually.

2nd Algorithm: QM Method [4].

Input: A_Minterm

Output: A_Simp

Begin

```
1: for each min_term a ∈ A_Minterm do
2:   List_V ← convert_to_binary(minterm)
3: end for
4: List_Z ← sort(List_V)
5: for each List z ∈ Z do
6:   for each group_first to group_last ∈ groups do
7:     for each bit ∈ total_bits do
8:       1_bit_diff_term ← Compare(current_group, next_group)
9:     end for
10:    if 1_bit_diff_term = 1 && existed_legal_dash_position then
11:      replace the bit with char – and put check char t
12:    else
13:      give check char * for uncompered group
14:    end if
15:  end for
16: end for
17: A_Implicant ← Uncompered any more and indicated including *
18: essential_A_Implicant ← Covtable(minterms,Prime_Implicants)
19: simplified_function A_Simp ← assigning_variables to test Prime_Implicant
20: Exit
```

3rd Algorithm: Insertion of nested empty if-else statement[4].

Input: h

Output: list_of_statements

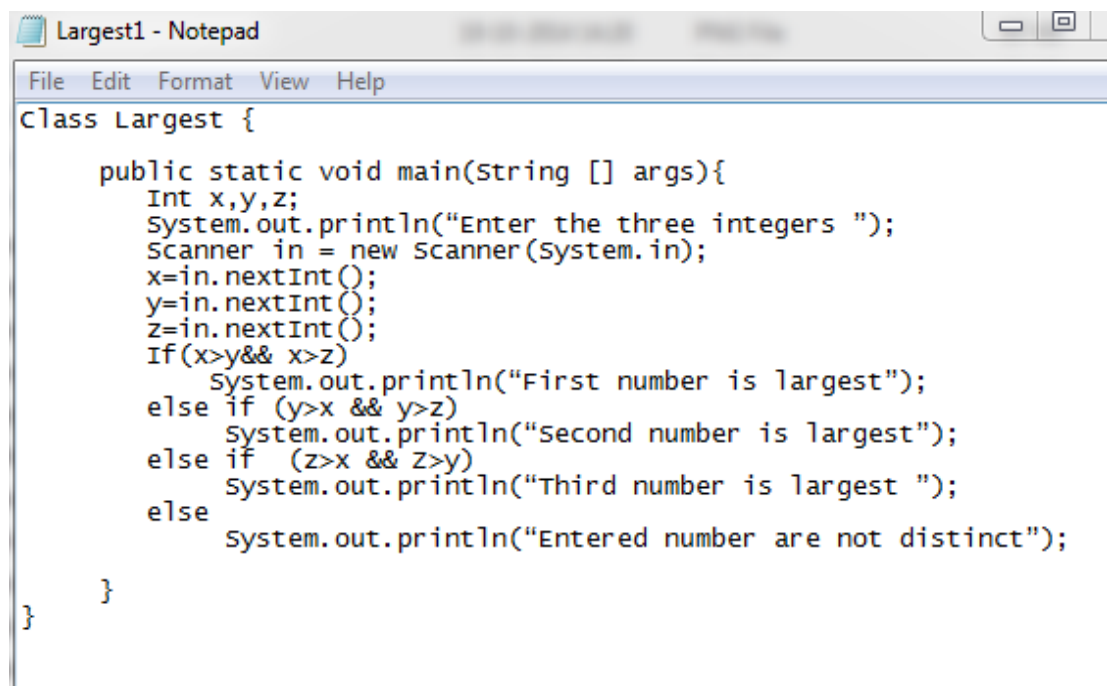
Begin

```
1: for each && connected con_group  $\in$  h do
2:   for each condition c  $\in$  con_group do
3:     if c is first_condition then
4:       insert an IF clause n with c as the condition list_of_statement  $\leftarrow$ 
         add_list(n)
5:     else
6:       insert a nested IF statement n with c as the condition insert an
         empty true branch TB and an empty false branch FB in order
         list_of_statement  $\leftarrow$  add_list(strcat(n,TB,FB))
7:     end if
8:   end for
9:   insert an empty false branch FB for the 1st condition list_of_statement  $\leftarrow$ 
         add_list(FB)
10: end for
11: for each condition  $\in$  h and not  $\in$  any con_group do
12:   repeat line 4,8 and 9
13: end for
14: if P is an else if predicate then
15:   insert an false if statement n
16:   make an empty truebranch TB list_of_statement  $\leftarrow$  add_list(strcat(n,TB))
17: end if
18: return list_of_statement
```

Chapter 4

RESULTS

4.1 Input Program for Step 1 :

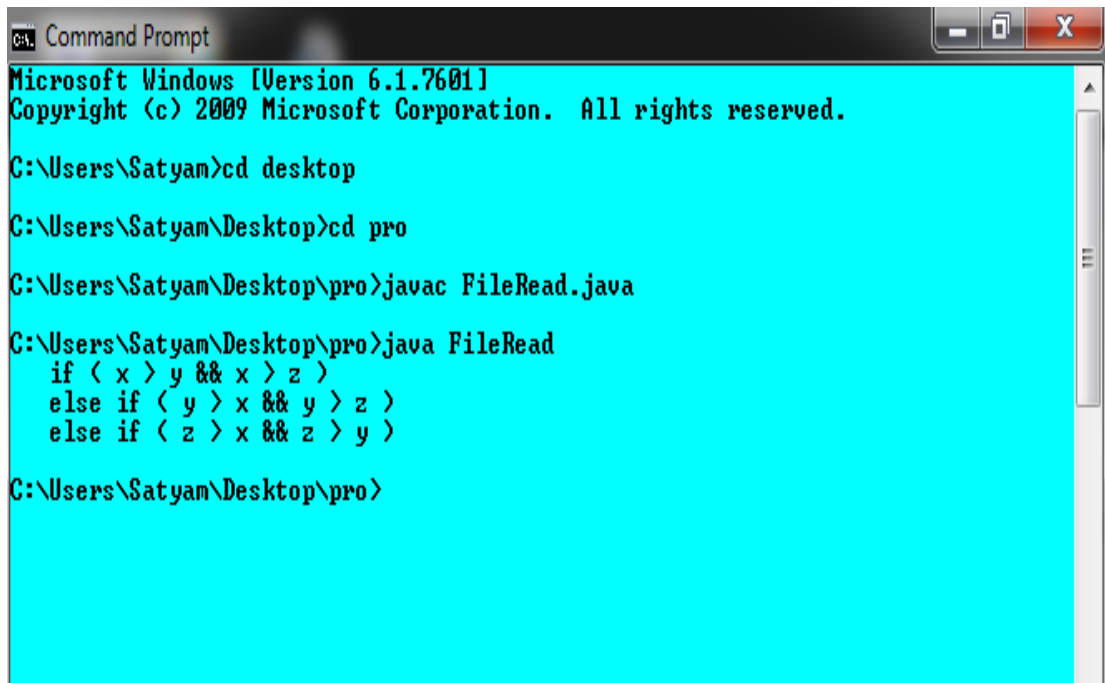


```
File Edit Format View Help
Class Largest {
    public static void main(String [] args){
        Int x,y,z;
        System.out.println("Enter the three integers ");
        Scanner in = new Scanner(System.in);
        x=in.nextInt();
        y=in.nextInt();
        z=in.nextInt();
        If(x>y&& x>z)
            System.out.println("First number is largest");
        else if (y>x && y>z)
            System.out.println("Second number is largest");
        else if (z>x && z>y)
            System.out.println("Third number is largest ");
        else
            System.out.println("Entered number are not distinct");
    }
}
```

Figure 4.1: First Input prog for step 1

The above figure is a simple JAVA code to find the largest number among three numbers. From this program we have to find the predicates. predicates are those if-else statement which contains AND, OR or NOT operator. We use 1st algorithm to find the predicates from this program.

4.2 Output of Step 1 (Predicates Identified) :



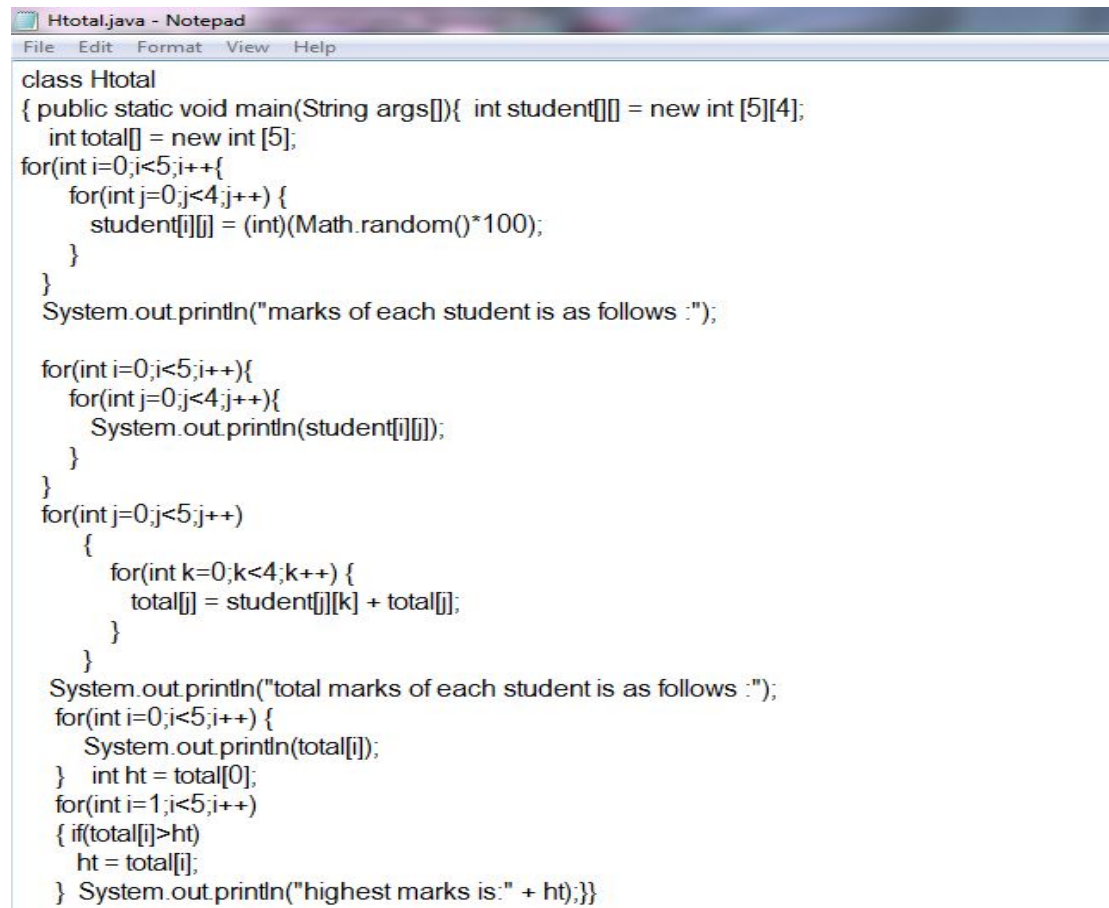
```
Command Prompt
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Satyam>cd desktop
C:\Users\Satyam\Desktop>cd pro
C:\Users\Satyam\Desktop\pro>javac FileRead.java
C:\Users\Satyam\Desktop\pro>java FileRead
  if ( x > y && x > z )
  else if ( y > x && y > z )
  else if ( z > x && z > y )
C:\Users\Satyam\Desktop\pro>
```

Figure 4.2: output of step 1

In the above figure, 3 predicates got identified from the input program. Based on these predicates, sum of predicates will get generated.

4.3 Another input program for step 1 :



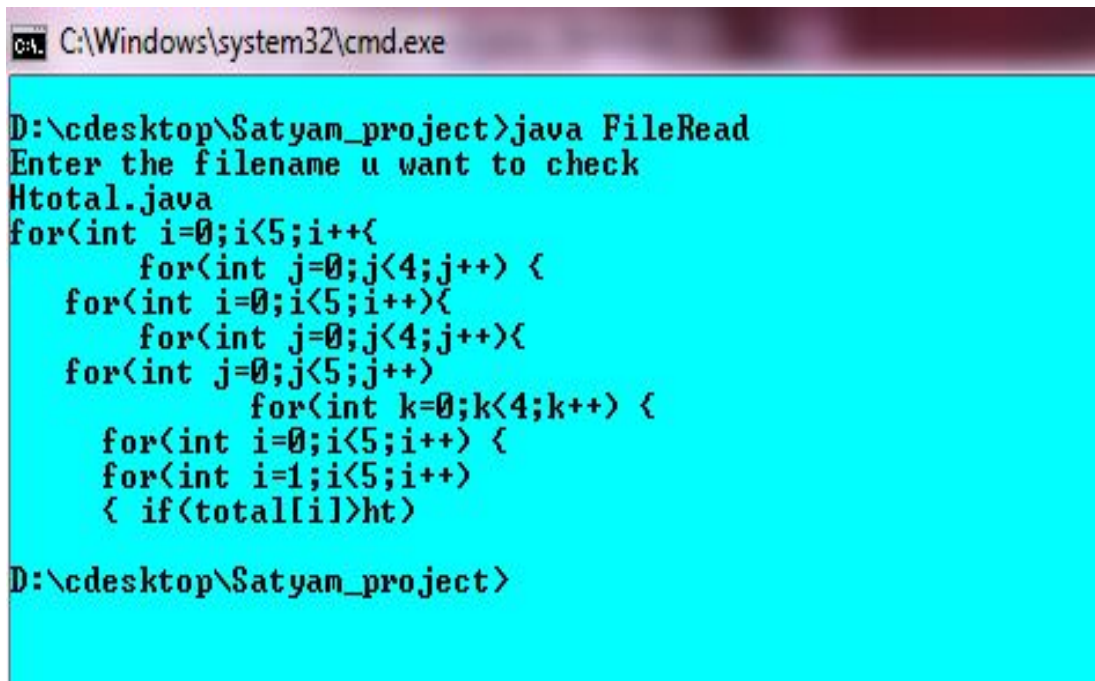
```
class Htotal
{ public static void main(String args[]){ int student[][] = new int [5][4];
  int total[] = new int [5];
  for(int i=0;i<5;i++){
    for(int j=0;j<4;j++){
      student[i][j] = (int)(Math.random()*100);
    }
  }
  System.out.println("marks of each student is as follows :");

  for(int i=0;i<5;i++){
    for(int j=0;j<4;j++){
      System.out.println(student[i][j]);
    }
  }
  for(int j=0;j<5;j++){
    {
      for(int k=0;k<4;k++) {
        total[j] = student[j][k] + total[j];
      }
    }
  }
  System.out.println("total marks of each student is as follows :");
  for(int i=0;i<5;i++) {
    System.out.println(total[i]);
  } int ht = total[0];
  for(int i=1;i<5;i++)
  { if(total[i]>ht)
    ht = total[i];
  } System.out.println("highest marks is:" + ht);}}
```

Figure 4.3: 2nd input prog for step 1

In the above figure another input program for step is taken.

4.4 Output of the input 2(predicates identified)

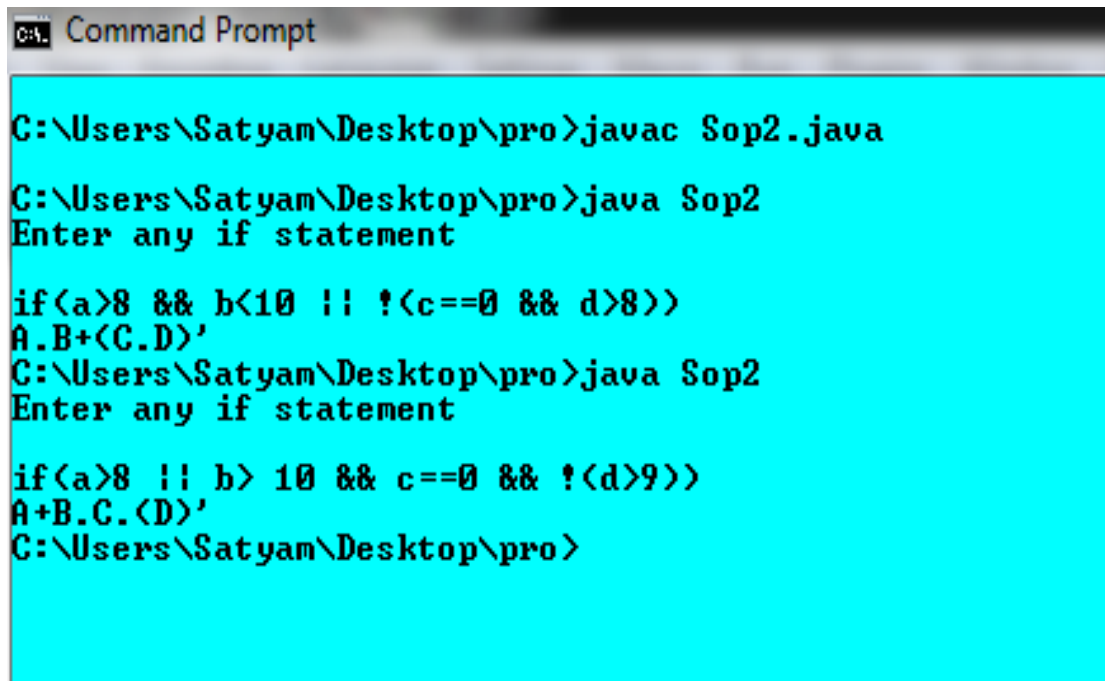


```
C:\Windows\system32\cmd.exe
D:\cdesktop\Satyam_project>java FileRead
Enter the filename u want to check
Htotal.java
for(int i=0;i<5;i++){
    for(int j=0;j<4;j++) {
    for(int i=0;i<5;i++){
        for(int j=0;j<4;j++){
        for(int j=0;j<5;j++)
            for(int k=0;k<4;k++) {
                for(int i=0;i<5;i++) {
                for(int i=1;i<5;i++)
                { if(total[i]>ht)
D:\cdesktop\Satyam_project>
```

Figure 4.4: output of 2nd input prog of step 1

In the above figure the output of the second program for step one is depicted. It contains all the predicates available in the previous figure.

4.5 Output of Step 2 (Generation of SOP) :



```
C:\Users\Satyam\Desktop\pro>javac Sop2.java
C:\Users\Satyam\Desktop\pro>java Sop2
Enter any if statement
if(a>8 && b<10 !! !(c==0 && d>8))
A.B+(C.D)'
C:\Users\Satyam\Desktop\pro>java Sop2
Enter any if statement
if(a>8 !! b> 10 && c==0 && !(d>9))
A+B.C.(D)'
C:\Users\Satyam\Desktop\pro>
```

Figure 4.5: output of step 2

In the above figure, sum of product of two if statements has been generated. We used 2nd Algorithm to generate the sum of product. Here, we substituted A for $a > 8$, B for $b < 10$, C for $c == 0$ and D for $d > 9$ in the first statement and likewise in the second.

4.6 Output of Step 3 (Quine Mc-cluskey Method)

:

```
C:\Users\Satyam\Downloads>java -cp Minimize.jar MinimizedTable 1 2 4 5 6 7
SIMPLIFY PRODUCT TERMS:
<ab'c + a'b'c> can be reduced to b'c in pass 1: Done
<abc' + a'bc'> can be reduced to bc' in pass 1: Done
<ab'c + ab'c'> can be reduced to ab' in pass 1: Done
<abc' + ab'c'> can be reduced to ac' in pass 1: Done
<a'b'c + ab'c'> can be reduced to b'c in pass 1: Already included
<ab'c' + ab'c'> can be reduced to ab' in pass 1: Already included
<abc + ab'c'> can be reduced to ac in pass 1: Done
<a'bc' + abc'> can be reduced to bc' in pass 1: Already included
<ab'c' + abc'> can be reduced to ac' in pass 1: Already included
<abc + abc'> can be reduced to ab in pass 1: Done
<ab'c + abc> can be reduced to ac in pass 1: Already included
<abc' + abc> can be reduced to ab in pass 1: Already included
Unable to reduce b'c in pass 2
Unable to reduce bc' in pass 2
<ab + ab'> can be reduced to a in pass 2: Done
<ac + ac'> can be reduced to a in pass 2: Already included
<ac' + ac'> can be reduced to a in pass 2: Already included
<ab' + ab'> can be reduced to a in pass 2: Already included
Unable to reduce b'c in pass 3
Unable to reduce bc' in pass 3
Unable to reduce a in pass 3

DETERMINE ESSENTIAL PRIME IMPLICANTS:
Minterm 1 is covered by 1 prime implicant.
Minterm 2 is covered by 1 prime implicant.
Minterm 4 is covered by 1 prime implicant.
Minterm 5 is covered by 2 prime implicants.
Minterm 6 is covered by 2 prime implicants.
Minterm 7 is covered by 1 prime implicant.
6 minterms remain
[ b'c => a'b'c, ab'c ] is the only implicant that covers a'b'c
5 minterms to go.
4 minterms to go.
[ bc' => a'bc', abc' ] is the only implicant that covers a'bc'
3 minterms to go.
2 minterms to go.
[ a => ab'c', ab'c, abc', abc ] is the only implicant that covers ab'c'
1 minterm to go.
0 minterms to go.
Minterm Numbers: [1,2,4,5,6,7]
Expression: Not Given
Sum of products: a'b'c + a'bc' + ab'c' + ab'c + abc' + abc
Prime implicants: [ b'c => a'b'c, ab'c ], [ bc' => a'bc', abc' ], [ a => ab'c',
ab'c, abc', abc ]
Minimized: b'c + bc' + a

C:\Users\Satyam\Downloads>
```

Figure 4.6: output of step 3

In the above figure the output of the QM method is depicted. It contains the minimized sop expression.

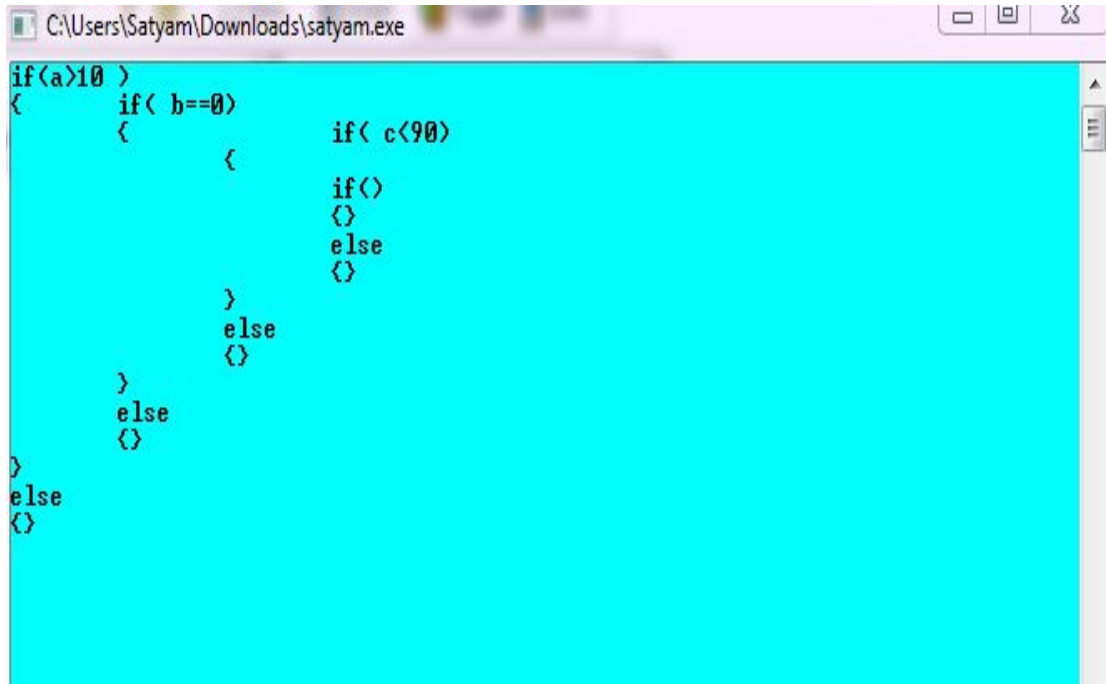
4.7 Input program for Step 4

```
class My
{
public static void main()
{
    if(a>10 && b==0&& c<90)
        System.out.println("insert");
    else
        System.out.println("Do not insert");
}
}
```

Figure 4.7: input prog for step 4

Above Figure is input program for the last step of JPCT.

4.8 Output of Step 4 (Insertion of empty if-else statement)



```
C:\Users\Satyam\Downloads\satyam.exe
if(a>10)
{
    if(b==0)
    {
        if(c<90)
        {
            if()
            {}
            else
            {}
        }
        else
        {}
    }
    else
    {}
}
else
{}

```

Figure 4.8: output of step 4

Above figure depicts the output of the last step. It has nested empty if-else statements in it.

Chapter 5

CONCLUSION AND FUTURE WORK

All the algorithms to design a code transformer have been successfully implemented and desired results were obtained. The transformed program were passed through cobertura tool and an increase of branch coverage have been recorded. Since the algorithms were implemented only for JAVA programming language, it can be generalised for every programming language.

Bibliography

- [1] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, “emphA practical tutorial on modified condition/ decision coverage”, vol. NASA/TM-2001-210876, May 2001. National Aeronautics and Space Administration , Langley Research Center Hampton, Virginia 23681-2199.
- [2] Mall R, “*Fundamental of Software Engineering*” Prentice Hall , 3rd edition , 2009.
- [3] A. Das, “*Automatic generation of MC/DC test data*”, Master’s thesis, IIT Kharagpur, April 2012.
- [4] S. Godbole, “*Improved Modified Condition/ Decision Coverage using Code Transformation Techniques*”, M.Tech Thesis, NIT Rourkela, May 2013.
- [5] S. Tiwary, “*Automatic generation of test cases for high MC/DC coverage*”, M.Tech Thesis, IIT Kanpur, June 2014.
- [6] Sangharatna Godbole, Durga Prasad Mohapatra, “*Time Analysis of Evaluating Coverage Percentage for C Program using Advanced Program Code Transformer*”, 7 th CSI International Conference on Software Engineering, 7 th CSI CONSEG, 2013.
- [7] Sangharatna Godbole, GS Prashanth, Durga Prasad Mohapatra and Bansidhar Majhi, “*Increase in Modified Condition/Decision Coverage using program code transformer*”, Advance Computing Conference (IACC), 2013 IEEE 3rd International, 22/2/2013.
- [8] Sangharatna Godbole, Subhrakanta Panda, Durga Prasad Mohapatra, “*SMCDCT: A Framework for Automated MC/DC Test Case Generation Using Distributed Concolic Testing*”, Springer International Publishing, 5/2/2015.

- [9] Sangharatna Godbole, Avijit Das, Kuleshwar Sahu, Durga Prasad Mohapatra and Banshidhar Majhi, “*Measuring Coverage Percentage for C Programs using Code Slicer and CREST Tool*”.