Setup:

Git Repository link: https://github.com/CSCI599/CSCI599Project
Java version: 7
External libraries: BCEL, Cobertura (along with any other dependency they might have).

Running:

The java file MainClass.java is the starting point of the analysis.

It expects 4 arguments:

1. Line number to run
2. Path to class file
3. Name of class file (without .class extension)
4. Name of the method

We have used Condition.java as the file we want to run the analysis on. It is a simple java file with several if statements in it.
The analysis might not work properly on very complex classes since we were not able to include all types of "IF" comparisons.

Understanding the output:

The output contains:

1) Total nodes in the CFG
2) Number of nodes which can reach the given line number
3) Conditions that the line number depends on
4) Some detailed information about the conditions like source line number, variable and its expected value etc.

Approach:

1) Prepare CFG and reachability information (similar to the ICAs)
2) Find the conditions (IF statements in byte code) on which a particular node depends on. This is done as follows:

* Find all branch statements that can reach the given node.
* The above set of branch statements may contain several conditions which do not affect the node we want to execute. There were two scenarios that we thought of:

I Always executed conditions

(i) Lets say statements B1, B2, ..., Bn are branch statements that can reach node N.

(ii) N is a child of node Bn.
(iii) The flow is such that node Bn will always be executed.
(iv) In this case even though B1...Bn-1 can reach N, the execution of N depends only on node Bn.
So we remove all branches before Bn from the set of branches that can affect N.

II Paths that always reach a given node.

(i) Lets say we want to execute a node N.
(ii) The location of N in the CFG is such that all paths will eventually reach it. In this case N is independent of any condition.
(iii) We can extend the above logic to determine the sub branches that N may be independent of:
Assume that branch B1 has two children: B2, B3.
B3 itself has two children: B4, B5.
Both B4 and B5 terminate at node N.
Here we can see that all paths from B3 will eventually reach N. So N is independent of B3.
However, N is not independent of B1.

Using this approach we determined the conditions on which a given node depends on.

3) The next step is to determine the variables on which a given condition depends on.
This is a complex step in terms of determining the value that a condition expects to be true.
This is because there can be different types of IF comparisons involving different types of data types.
We concentrated on baisc integer and string comparisons.
From the byte code, we determined the instructions that were loaded befor the IF statement.
These instructions load the variables and constants required for the comparison.
From the LocalVariableTable we determine the name and value of these variables.

4) calculate reaching definition: Verify that the variable is obtained from "outside" and is not modified within the code.
If it is modified, mark the variable.
For each node in the graph, we compute the reaching definitions. This iterative algorithm is used to compute reaching definition:

```
// Boundary condition
out[Entry] = ∅

// Initialization for iterative algorithm For each basic block B other than Entry
```

```
For each basic block B other than Entry {
        out[B] = ∅
        gen[B] = {d} // gen of B is definition(s) generated in block B (could
be ∅ if B does not contain defintion
        kill[B] = set of all other defs in the rest of program to the variable
(s) defined in B
}

// iterate
While (Changes to any out[] occur) {
        For each basic block B other than Entry {
                in[B] = ∪ (out[p]), for all predecessors p of B
                out[B]=gen[B] ∪ (in[B]-kill[B])
        }
}
```

5) Print output: Print the line numbers, conditions, variables (and their expected value if possible) that the given node depends on.
If the variable was "Marked" in the reaching definition as modified in the code, display a message saying that the variable was modified in the code (with the line number where it was modified).

Evaluation:

We have experimented our implementation using the following application:

```
==============================================
==========
package com.csci599.project;



public class Condition {

public static void main(String[] args) {
int n = 10000;
n = n + 4;
int a = Integer.parseInt(args[0]), b = Integer.parseInt(args[1]), c =
Integer
.parseInt(args[2]), d = Integer.parseInt(args[3]), e = Integer
.parseInt(args[4]), f = Integer.parseInt(args[5]), g = Integer
.parseInt(args[6]), h = Integer.parseInt(args[7]);
String x1 = "xy";
if (b == 3 && c == a && x1.equalsIgnoreCase("xy")) {
if (d == 5) {
if (a == 6) {
System.out.println("A = 6");
}
if (a == 7) {
```

```
System.out.println("A != 6");
System.out.println("This is the line to execute");
}
} else {
if (a == 7) {
System.out.println("A = 7");
} else {
System.out.println("A != 7");
}
}
} else {
if (e == 8) {
System.out.println("E = 8");
} else {
System.out.println("E != 8");
}
}

if (f >= 3 && g <= 4) {
if (h == 5) {
e = n + 1 * (a / 4);
if (e == 6) {
System.out.println("E = 6");
} else {
System.out.println("E != 6");
}
} else {
if (e == 7) {
System.out.println("E = 7");
} else {
System.out.println("E != 7");
}
}
} else {
if (e == 8) {
System.out.println("E = 8");
} else {
System.out.println("E != 8");
}
}
System.out.println("Start of WHILE Loop");
int x = 1;
while (x < 10) {
System.out.println("This is X WHILE: " + x);
x++;
}

x = 1;
System.out.println("Start of DO WHILE Loop");
```

```
do {
System.out.println("This is X DO WHILE: " + x);
x++;
} while (x < 10);

System.out.println("Start of FOR Loop");
for (x = 1; x < 10; x++) {
System.out.println("This is X FOR: " + x);
if (x == 8) {
break;
}
}
System.out.println("This is the end");
}
}
```

===
==========

Our goal is to execute line 22, which contains instruction "System.out.println("This is the line to execute");"

To analyze the above class we run the analysis using the following command:

```
java -cp ./bin:./bcel-5.2.jar com.csci599.project.MainClass 22 bin/com/
csci599/project/ Condition main
```

This is the output of the analysis:

```
=================================================
==========
Total Nodes: 208
1 independent conditions out of 6
135(line number 22) can be reached by 73 other nodes
Node at position 135 depends on the following conditions
Total dependency conditions: 5

119: if_icmpne[160](3) -> iload 7
  99: if_icmpne[160](3) -> iload_2
  93: ifeq[153](3) -> iload 6
  83: if_icmpne[160](3) -> iload 6
  77: if_icmpne[160](3) -> iload 6

Conditions:

Instruction:  119: if_icmpne[160](3) -> iload 7
Must evaluate to : true
Depends on variable: a
For instruction to be TRUE, a must be : 7
```

```
Source code line number: 20


Instruction:   99: if_icmpne[160](3) -> iload_2
Must evaluate to : true
Depends on variable: d
For instruction to be TRUE, d must be : 5
Source code line number: 16


Instruction:   93: ifeq[153](3) -> iload 6
Must evaluate to : true
Depends on variable: x1
For instruction to be TRUE, x1 must be : xy
Source code line number: 15
The variable x1 does not come from outside. It is redefined at position: 73


Instruction:   83: if_icmpne[160](3) -> iload 6
Must evaluate to : true
Depends on variable: c
For instruction to be TRUE, c must be : a
Source code line number: 15


Instruction:   77: if_icmpne[160](3) -> iload 6
Must evaluate to : true
Depends on variable: b
For instruction to be TRUE, b must be : 3
Source code line number: 15
======================================================
```

As can be implied from the output, the following conditions need to be satisfied to successfully execute line 22 :

1. a == 7
2. d == 5
3. x1 == "xy"
4. c == a
5. b == 3

(note that x1 is defined inside the application and not passed as an argument)

So running class Condition with the following arguments will NOT cover line 22:

```
java -cp target:./lib/cobertura.jar com.csci599.project.Condition 7 3 5 5 0 0 0 0
```

However, after reading the analysis report and changing the arguments, this command will cover line 22:

```
java -cp target:./lib/cobertura.jar com.csci599.project.Condition 7 3 7 5 0 0 0 0
```

See case-a and case-b coverage reports located in cobertura/reports directory.