EE360T/EE382V: Software Testing
Problem Set 3

Out: Mar 1, 2020; **Due: Mar 15, 2020 11:59pm**
Submission: *.zip via Canvas
Maximum points: 40

Control-flow Graphs and Java Classfiles

You are to construct a partial[1] control-flow graph from the bytecode[2] of a given Java class using the Bytecode Engineering Library (BCEL)[3].

To illustrate, consider the following class `C`:

```
package pset3;

public class C {
    int max(int x, int y) {
        if (x < y) {
            return y;
       } else return x;
    }
}
```

which can be represented in bytecode as (for example, the output of `javap -c`):

```
Compiled from "C.java"
public class pset3.C extends java.lang.Object{
public pset3.C();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

int max(int, int);
  Code:
   0:   iload_1
   1:   iload_2
   2:   if_icmpge       7
   5:   iload_2
   6:   ireturn
   7:   iload_1
   8:   ireturn

}
```

Figure 1 illustrates the corresponding (partial) control-flow graph with *single* exit point: we introduced a "dummy" node for each method to represent a single exit point – all method nodes that represent a return instruction have an edge to the unique exit point.

Consider the following class `CFG` that models control-flow graph in a Java bytecode program:

[1] This assignment ignores the labels that are traditionally annotated on nodes and edges, as well as the edges that correspond to `jsr[_w]` or `*switch` bytecodes.
[2] http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings
[3] http://commons.apache.org/bcel/

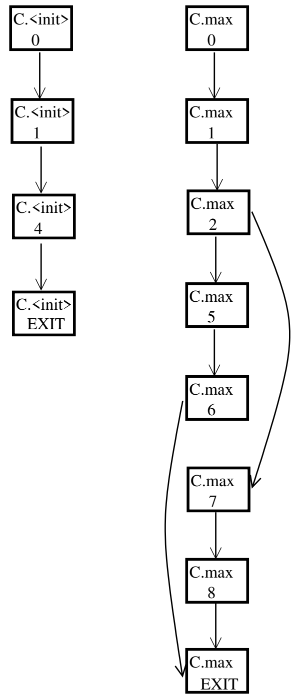

Figure 1: Partial control-flow graph with unique exit points.

```
package pset3;

import java.util.*;

import org.apache.bcel.classfile.JavaClass;
import org.apache.bcel.classfile.Method;

public class CFG {
    Set<Node> nodes = new HashSet<Node>();
    Map<Node, Set<Node>> edges = new HashMap<Node, Set<Node>>();

    public static class Node {
        int position;
        Method method;
        JavaClass clazz;

        Node(int p, Method m, JavaClass c) {
            position = p;
            method = m;
            clazz = c;
        }

        public Method getMethod() {
            return method;
        }

        public JavaClass getClazz() {
            return clazz;
        }

        public boolean equals(Object o) {
            if (!(o instanceof Node)) return false;
```

```
            Node n = (Node)o;
            return (position == n.position) && method.equals(n.method) && clazz.equals(n.clazz);
        }

        public int hashCode() {
            return position + method.hashCode() + clazz.hashCode();
        }

        public String toString() {
            return clazz.getClassName() + '.' + method.getName() + method.getSignature() + ": " + position;
        }
    }

    public void addNode(int p, Method m, JavaClass c) {
        addNode(new Node(p, m, c));
    }

    private void addNode(Node n) {
        nodes.add(n);
        Set<Node> nbrs = edges.get(n);
        if (nbrs == null) {
            nbrs = new HashSet<Node>();
            edges.put(n, nbrs);
        }
    }

    public void addEdge(int p1, Method m1, JavaClass c1, int p2, Method m2, JavaClass c2) {
        Node n1 = new Node(p1, m1, c1);
        Node n2 = new Node(p2, m2, c2);
        addNode(n1);
        addNode(n2);
        Set<Node> nbrs = edges.get(n1);
        nbrs.add(n2);
        edges.put(n1, nbrs);
    }

    public void addEdge(int p1, int p2, Method m, JavaClass c) {
        addEdge(p1, m, c, p2, m, c);
    }

    public String toString() {
        return nodes.size() + " nodes\n" + "nodes: " + nodes + '\n' + "edges: " + edges;
    }

    public boolean isReachable(String methodFrom, String clazzFrom,
                               String methodTo, String clazzTo) {
        // you will implement this method in Question 2.2
    }
}
```

A `CFG` object has a set of nodes that represent bytecode statements and a set of edges that represent the flow of control (branches) among statements. Each node contains:

- an integer that represents the position (bytecode line number) of the statement in the method.
- a reference to the method (an object of class `org.apache.bcel.classfile.Method`) containing the bytecode statement; and
- a reference to the class (an object of class `org.apache.bcel.classfile.JavaClass`) that defines the method.

The set of nodes is represented using a `java.util.HashSet` object, and the set of edges using a `java.util.HashMap` object, which maps a node to the set of its neighbors. The sets of nodes and edges have values that are consistent, i.e., for any edge, say from node a to node b, both a and b are in the set of nodes. Moreover, for any node, say n, the map maps n to a non-null set, which is empty if the node has no neighbors.

1 Generating a basic CFG [20 points]

Implement the class `GraphGenerator` that allows creation of control-flow graphs from bytecode programs. The following code snippet gives a partial implementation of `GraphGenerator`:

```
package pset3;
public class GraphGenerator {
    public CFG createCFG(String className) throws ClassNotFoundException {
        CFG cfg = new CFG();
        JavaClass jc = Repository.lookupClass(className);
        ClassGen cg = new ClassGen(jc);
        ConstantPoolGen cpg = cg.getConstantPool();

        for (Method m: cg.getMethods()) {
            MethodGen mg = new MethodGen(m, cg.getClassName(), cpg);
            InstructionList il = mg.getInstructionList();
            InstructionHandle[] handles = il.getInstructionHandles();
            for (InstructionHandle ih: handles) {
                int position = ih.getPosition();
                cfg.addNode(position, m, jc);
                Instruction inst = ih.getInstruction();
                // your code goes here
            }
        }
        return cfg;
    }

    public CFG createCFGWithMethodInvocation(String className) throws ClassNotFoundException {
        // your code goes here
    }

    public static void main(String[] a) throws ClassNotFoundException {
        GraphGenerator gg = new GraphGenerator();
        gg.createCFG("pset3.C"); // example invocation of createCFG
        gg.createCFGWithMethodInvocation("pset3.D"); // example invocation of createCFGWithMethodInovcation
    }
}
```

Complete the implementation of `GraphGenerator.createCFG`, which returns a `CFG` object that represents the control-flow graph for *all* the methods in the given class. For this part, ignore the edges that represent method invocations as well as `jsr[_w]` and `*switch` instructions.
Hint: The class `org.apache.bcel.generic.BranchInstruction` is a superclass of the classes that represent branching instructions.

2 CFGs with method invocations [20 points]

2.1 Core representation

Complete the implementation of `GraphGenerator.createCFGWithMethodInovcation` by extending your solution to the previous part, i.e., your implementation of `GraphGenerator.createCFG`, to provide generation of partial control flow graphs that *include* a representation of method invocations. You only need to support invocation of *class* methods (`INVOKESTATIC`). Assume (as before) the edges of the graphs are not labeled[4].

To illustrate, consider the following class `D`:

[4]While this assumption simplifies the generation of graphs, the resulting graphs may have paths that do not correspond to program paths!

```
package pset3;
public class D {
    public static void main(String[] a) {
        foo(a);
        bar(a);
    }

    static void foo(String[] a) {
        if (a == null) return;
        bar(a);
    }

    static void bar(String[] a) {}
}
```

and its bytecode representation:

```
Compiled from "D.java"
public class pset3.D extends java.lang.Object{
public pset3.D();
  Code:
   0:   aload_0
   1:   invokespecial   #8; //Method java/lang/Object."<init>":()V
   4:   return

public static void main(java.lang.String[]);
  Code:
   0:   aload_0
   1:   invokestatic    #16; //Method foo:([Ljava/lang/String;)V
   4:   aload_0
   5:   invokestatic    #19; //Method bar:([Ljava/lang/String;)V
   8:   return

static void foo(java.lang.String[]);
  Code:
   0:   aload_0
   1:   ifnonnull       5
   4:   return
   5:   aload_0
   6:   invokestatic    #19; //Method bar:([Ljava/lang/String;)V
   9:   return

static void bar(java.lang.String[]);
  Code:
   0:   return

}
```

Figure 2 illustrates the (partial) control flow graph your implementation should generate for class `D`.

2.2 Control-flow Graph Reachability

For this part of the question assume that (1) there is no method overriding; and (2) each method is invoked by at most one method (which may invoke it multiple times).

Implement the following method `isReachable` in class `CFG` to determine whether a method (directly or indirectly) invokes another method. String arguments `methodFrom` and `clazzFrom` represent the names of the caller method and its declaring class. String arguments `methodTo` and `clazzTo` represent the names of the callee method and its declaring class.

```
    public boolean isReachable(String methodFrom, String clazzFrom,
                               String methodTo, String clazzTo) {
        // ... your code goes here
    }
```

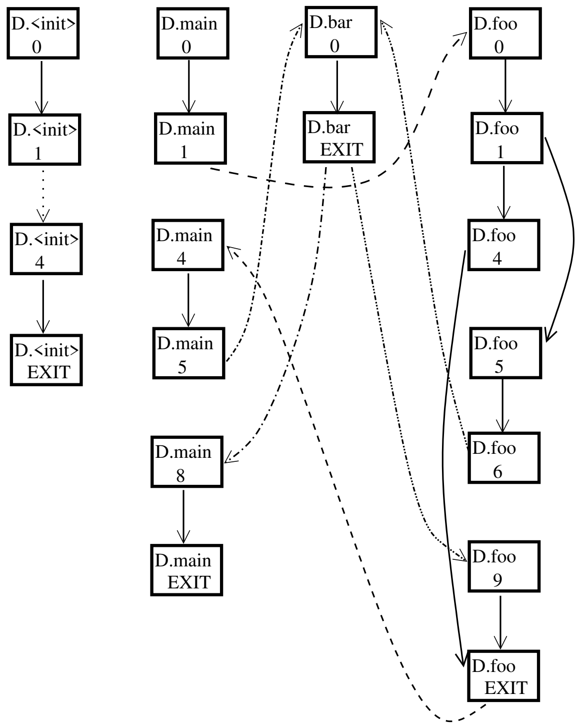

Figure 2: Partial control-flow graph for class `D`.

For example, in Figure 2, method `D.main` invokes method `D.foo` and method `D.bar`. Thus, invocations `isReachable("main", "pset3.D", "foo", "pset3.D")` and `isReachable("main", "pset3.D", "bar", "pset3.D")` both return true.