EE360T: Software Testing
Problem Set 1

Out: Feb 17, 2011; **Due: Feb 27, 2011 11:59pm**
Submission: tarball via blackboard
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`:

```
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 pset1.C extends java.lang.Object{
public pset1.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

}
```

[1] This assignment ignores the labels that are traditionally annotated on nodes and edges, as well as the edges that correspond to method invocations, or `jsr[_w]` or `*switch` bytecodes.

[2] `http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html`

[3] http://jakarta.apache.org/bcel/

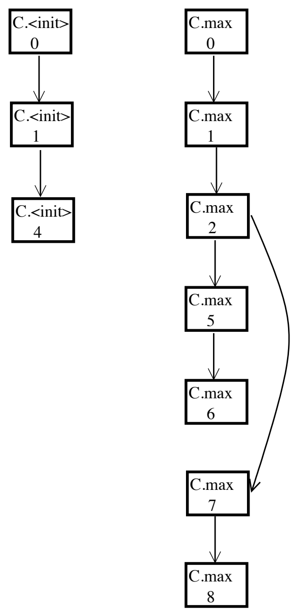

Figure 1: Partial control-flow graph.

Figure 1 illustrates the corresponding (partial) control-flow graph.

1 Graph representation of control-flow

Implement the class `CFG` to model control-flow in a Java bytecode program. 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.

Represent the set of nodes 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. Ensure 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, ensure that for any node, say n, the map maps n to a non-null set, which is empty if the node has no neighbors.

The following code snippet gives a partial implementation of `CFG`:

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

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

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

        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) { ... }

    public void addEdge(int p1, Method m1, JavaClass c1,
                        int p2, Method m2, JavaClass c2) {
        ...
    }

    public boolean isReachable(int p1, Method m1, JavaClass c1,
                               int p2, Method m2, JavaClass c2) {
        ...
    }
}
```

1.1 Adding a node [5 points]

Implement the method `CFG.addNode` such that it creates a new node with the given values and adds it to `nodes` as well as initializes `edges` to map the node to an empty set. If the graph already contains a node that is `.equals` to the new node, `addNode` does not modify the graph.

graph - refers to nodes and edges..?

1.2 Adding an edge [5 points]

Implement the method `CFG.addEdge` such that it adds an edge from the node `(p1, m1, c1)` to the node `(p2, m2, c2)`. Your implementation should update `nodes` to maintain its consistency with `edges` as needed.

1.3 Reachability [10 points]

Implement the method `CFG.isReachable` such that it traverses the control-flow graph starting at the node represented by `(p1, m1, c1)` and ending at the node represented by `(p2, m2, c2)` to determine if there exists any path from the given start node to the given end node. If the start node or the end node are not in the graph, the method returns `false`.

2 Generating a 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`:

```
public class GraphGenerator {
    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();
                ...
            }
        }
        return cfg;
    }

    public static void main(String[] a) throws ClassNotFoundException {
        new GraphGenerator().createCFG("pset1.C");
    }
}
```

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. You can 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.