EE379K: Software Testing
Problem Set 2

Out: March 10 2011; **Due: March 28, 2011 11:59pm**
Submission: via blackboard
Maximum points: 40 (20 for each part)

Java Classfiles, Control Flow Graphs, and Input Generation

This assignment has two parts. In the first part, you are to build on your solution to Problem Set 1 (PS1) to handle method invocations. In the second part you are to write a simple test generator.

1 CFGs with method invocations

1.1 Core representation

Extend your solution to PS1 to provide generation of partial control flow graphs (as defined in PS1) that include a representation of method invocations. You only need to support invocation of class methods (`INVOKESTATIC`). Assume the edges of the graphs are not labeled[1].

Hint

Consider introducing 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. To illustrate, recall the class `C` from PS1:

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

and its bytecode representation:

[1] While this assumption simplifies the generation of graphs, the resulting graphs may have paths that do not correspond to program paths! Question 1.2 asks you to identify such a path.

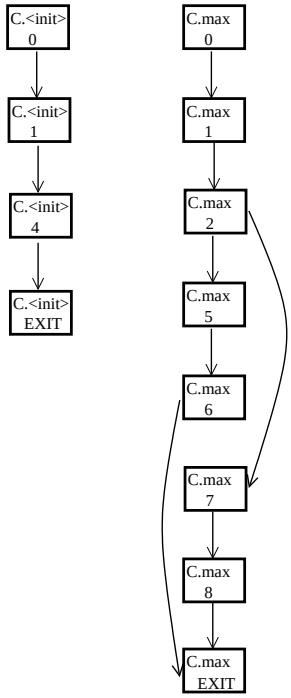

Figure 1: Partial control-flow graph with unique exit points.

```
Compiled from "C.java"
public class ee379k.pset1.C extends java.lang.Object{
public C();
  Code:
   0:   aload_0
   1:   invokespecial   #8; //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 unique exit points.

Behavior illustration

To illustrate, consider the following class `D`:

```
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 ee379k.pset2.D extends java.lang.Object{
public ee379k.pset2.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`.

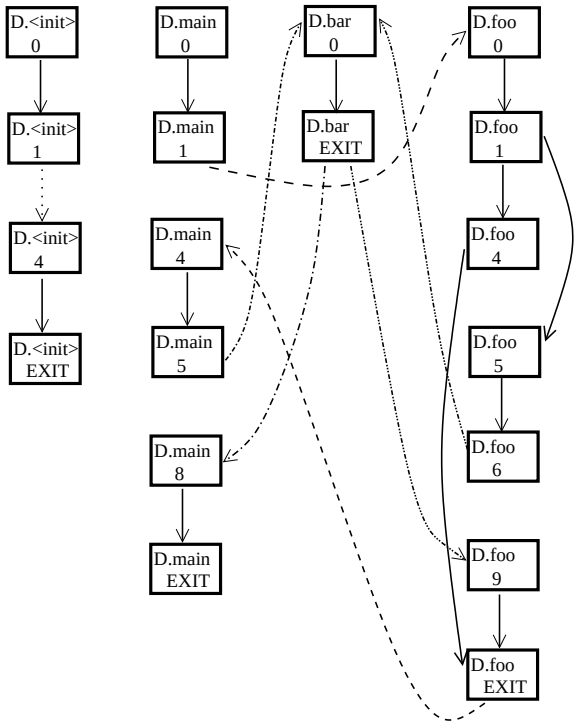

Figure 2: Partial control-flow graph for class `D`.

1.2 Limitation

Identify a path in the CFG shown in Figure 2 that does not exist in the original program. Your path may use any edges from the graph.

2 Test Generation

Implement the class `TestGenerator` that provides generation of tests for all class (i.e., `static`) methods in a given class:

```
public class TestGenerator {
    public String createTests(String className) throws ClassNotFoundException {
        JavaClass jc = Repository.lookupClass(className);
        ClassGen cg = new ClassGen(jc);
        ConstantPoolGen cpg = cg.getConstantPool();

        for (Method m: cg.getMethods()) {
            ...
        }
        ...
    }
}
```

The output of the method `createTests` is a `String` representation of JUnit tests that for each method in the given class, check the method against all combinations of previously defined default values for the argument types of the method.

Formally, let C be the given class, and m be a static method in C. Let the argument types of m be $T_1, \ldots, T_k$. Let the set of default values for type T_i be S_i $(1 \leq i \leq k)$. Then for method m, your implementation should generate $|S_1| \times \ldots \times |S_k|$ tests, one for each combination of the default values.

Assume you are provided the following class `Domain`, which represents the default values for the types that may appear as method argument types:

```
public class Domain {
    public static final int[] INT = new int[]{ -1, 0, 1 };
    public static final boolean[] BOOLEAN = new boolean[]{ false, true };

    public static final String[] OBJECT = new String[]{ "null", "new Object()" };

    public static final String[] STRING =
        new String[]{ null, "", "0", "Hello" };
}
```

Behavior illustration

To illustrate, assume you are given the following class:

```
public class Demo {
    static void m(int x) {
        ...
    }

    static void n(boolean b, String s) {
        ...
    }
```

For the class `Demo`, your implementation should generate 3 tests for method `m`, e.g.:

```
    @Test public void test_m_1() { m(-1); }

    @Test public void test_m_2() { m(0); }

    @Test public void test_m_3() { m(1); }
```

and 8 tests for the method `n`, e.g.:

```
    @Test public void test_n_1() { n(false, null); }

    @Test public void test_n_2() { n(false, ""); }

    @Test public void test_n_3() { n(false, "0"); }

    @Test public void test_n_4() { n(false, "Hello"); }
```

```
@Test public void test_n_5() { n(true, null); }

@Test public void test_n_6() { n(true, ""); }

@Test public void test_n_7() { n(true, "0"); }

@Test public void test_n_8() { n(true, "Hello"); }
```