EE382C: Verification and Validation of Software
Problem Set 3 – Using Java PathFinder

Out: March 15, 2016
Due: April 5, 2016 11:59pm

Instructions. Submit your Java code on Canvas. You need to download and install the Java PathFinder (JPF) model checker, which you can get from the following website: "`http://babelfish.arc.nasa.gov/trac/jpf`".

1 Loop-list implementation [20 points total; 5 points/part]

Recall from the last homework that a *loop-list* is a singly-linked list, which is either empty or its last node has a pointer back to that node itself.

Consider implementing a class to represent *loop-lists* of integers:

```
public class LList { // loop-list

    Node header;
    int size;

    static class Node {
        int elem;
        Node next;
    }
```

(a) Class invariant

Implement the following method `repOk`, which checks whether its input satisfies the class invariant as specified:

```
    public boolean repOk() {
        // returns true if and only if (1) this is a loop-list and
        //   (2) size is the number of nodes in this

        ...
    }
```

(b) `addFirst`

Implement the following method `addFirst` as specified:

```
    public void addFirst(int x) {
        // adds a new node with element x at the *head* of the list; all other list nodes
        //   remain unchanged

        ...
    }
```

(c) `addLast`

Implement the following method `addLast` as specified:

```
    public void addLast(int x) {
        // adds a new node with element x at the *tail* of the list; all other list nodes
        //   remain unchanged

        ...
    }
```

(d) `toString`

Implement the following method `toString` as specified:

```
    public String toString() {
        // returns a string representation of the list of elements in this, where
        //   consecutive elements are separated by a space

        ...
    }
}
```

2 Loop-list tester [20 points total]

Consider implementing the class `LListTester` to test `LList`:

```
public class LListTester {
```

(a) JUnit tests [6 points]

Write some JUnit tests such that each test makes exactly one invocation of `addLast` and running all the tests provides full statement coverage for the method `addLast`:

```
    @Test public void test0() {
        ...
    }

    @Test public void test1() {
        ...
    }

    ...
```

(b) JPF test generator [14 points]

Implement the following `main` method such that running it using the JPF JVM generates all method sequences of length up to `SEQUENCE_LENGTH`, where the first method in each sequence is a constructor call, which is followed by up to `SEQUENCE_LENGTH - 1` invocations of `addFirst` or `addLast`, and each invocation of `addFirst` and `addLast` uses only integers {`0, ..., ELEM_UPPER_BOUND`} as parameter values:

```
    public static void main(String[] a) {
        if (a.length != 2) throw new IllegalArgumentException();
        final int SEQUENCE_LENGTH = Integer.parseInt(a[0]);
        final int ELEM_UPPER_BOUND = Integer.parseInt(a[1]);
        ...
    }
}
```

To illustrate, executing your `main` method using JPF JVM for main arguments `["3", "1"]` should produce 21 JUnit tests, including:

```
@Test public void test0() {
    LList l = new LList();
}

@Test public void test1() {
    LList l = new LList();
    l.addLast(0);
}

@Test public void test2() {
    LList l = new LList();
    l.addLast(1);
}

@Test public void test3() {
    LList l = new LList();
    l.addFirst(0);
}

@Test public void test4() {
    LList l = new LList();
    l.addFirst(1);
}

@Test public void test5() {
    LList l = new LList();
    l.addLast(0);
    l.addLast(0);
}

@Test public void test6() {
    LList l = new LList();
    l.addLast(0);
    l.addLast(1);
}

@Test public void test7() {
    LList l = new LList();
    l.addLast(0);
    l.addFirst(0);
}

...
```

Hint: The JPF class `Verify` provides methods, such as `resetCounter`, `getCounter`, and `incrementCounter`, to implement a counter whose value is not reset during backtracking.