Formal Methods in Software Engineering, 2022
Assignment on Java PathFinder
Due date: April 11th, 2022, 11.59 pm. Please upload into the Teams Assignment a zip file that contains all the files mentioned below. Any delayed submissions or submissions via other mechanisms such as email or chat will not be accepted.

Problem 1. The objective of this problem is to write a sorting routine and test it exhaustively using JPF. Write a program *Sort.java*. The program should have three methods: `main`, `sort`, and `checkSorted`. All three should be *static* methods.

`sort` should take an integer array as argument and sort it in place. `checkSorted` should take an integer array as argument and *check* if the array is sorted. It should throw an exception (via an *assert* statement) if the array is not sorted.

`main` should first allocate an integer array of a fixed size. Let five be this size. `main` should populate each element of the array with a symbol. The symbols can be created using *Debug.makeSymbolicInteger* (make sure to use a different name for each symbol). `main` should then pass this array to `sort`, and then check that the array is sorted by calling `checkSorted`.

Create a file *Sort.jpf*. It can be modeled after *TestZ3.jpf* on the course web page. Methods `sort` and `checkSorted` should have their argument marked as "con".

Make sure that when one runs the "run-JPF-symbc" run configuration on your .jpf file, assertion violations do not occur. At the same time, if one wilfully introduces any errors in your sorting code, then assertion violations should be thrown.

Things you need to submit:

1. Your .java file and .jpf files, and a file *Problem1.out* containing the entire output from the JPF run.

2. An image of the *symbolic execution tree*, corresponding to the output from the JPF run mentioned above. The symbolic execution tree should be similar to what is shown in the course web site; it should show the updates to the program state, the path-condition at each branch, and all choice points and backtracking. For this run, you can prune the size of the array being sorted to 2 (instead of 5). The image can be a photo of a neat, hand-drawn picture (with no striking out, etc.). The file name should be *Problem1.jpg*.

3. A text file *Problem1.txt* containing your explanation of why the exercise above can be considered to be an exhaustive testing of the `sort` method (for the chosen array size).

Problem 2. Consider the method below:

```
public static void test5(NodeSimple n) {
1:    int len = 0;
2:    NodeSimple curr = n;
3:    while( curr != null && curr.next != null && curr !=  curr.next) {
4:         len++;
5:         if (len == 2) {
6:             System.out.println("length at least 2");
7:             break;
8:         }
9:         curr = curr.next;
10:   }
}
```

`NodeSimple` is a class with a `next` field. Place a call to the method above from main(), passing some arbitrary object to the method. Put the code above in a file *LinkedList.java*, and create a file *LinkedList.jpf*, indicating `test5` in the *symbolic.method* option with the parameter 'n' being symbolic. The rest of the options can be similar to the ones in NodeSimple.jpf (see our course web page), with appropriate modifications. Run the run-JPF-symbc configuration. From the output, draw a symbolic execution tree as mentioned in the previous problem. Also mention against each node in the tree the corresponding line number in the code fragment above.

1. Submit the .java file, the .jpf file, as well as your image of the tree (named *Problem2.jpg*). Also submit a file *Problem2.out* containing the entire output from the JPF run.

2. If one removes Lines 5-8 in the the method `test5` above, the symbolic execution does not terminate. Submit a text file *Problem2.txt* in which you explain why the JPF algorithm does not terminate in this setting.