DD2452 Formal Methods ESC/Java2 Assignment

Spring 2009

Due: 19 February 2009

The purpose of this assignment is to investigate the potential advantages of using an extended static checker for the verification of code written in a high-level programming language. Moreover, the assignment is meant to help you catch a glimpse of how state-of-the-art technologies may be applied to the production of high-quality software.

1 Requirements

The assignment should be done individually or in groups of two students. Each group should present its solution at a workstation (or own portable machine) and at the same time submit a *report* documenting the annotated code and results.

The report should contain:

- Name and e-mail address of each of the participants in the group.
- The annotated ESC/Java2 code along with the explanation of your preferred specifications.

2 ESC/Java2 Installation

1. Check if you already have ESC/Java2 installed on your system.
2. If you do not have ESC/Java2 already installed on your system, get ESC/Java2 from:

   `http://kind.ucd.ie/products/opensource/ESCJava2/download.html`.
3. Using the same link above, download and install ESC/Java2 specifications for the standard API classes and interfaces.
4. Read the documentation accompanying ESC/Java2 and test some of the simple examples included in the distribution and the Bag example provided on the course web page.

3 Tasks

1. Choose a simple implementation of a *data structure* in Java. Please get your choice approved by the course leader. Your code should satisfy the following minimal requirements:

   - The code is written in Java,
   - it includes at least a class definition, and
   - at least three/four meaningful methods, with at least one containing a loop construct.

   If you are out of ideas (but only in this case), you can pick the implementation of *priority queues* presented in Appendix A.

2. Run the tool over your (unannotated) code to check that it can access all needed class specifications. You may have to provide a class specification yourself if it is not available within the standard API specs.

3. Provide *pre-* and *post-condition* specifications for each method in your code. These should first of all capture *what* a method is supposed to do and how it is to be used, and be meaningful without knowledge of the body of the method.

4. Provide *class invariants* for your code. These should capture the essential properties of the data structure that should be preserved by the (public) methods of the class (as for instance the *shape property* and the *heap property* in the case of binary heaps – see for example Wikipedia for definitions of these). However, private *helper* methods may violate the invariants; you can use the `helper` modifier to designate these.

5. Provide *loop invariants* for all loops in your code.

6. Check your specification using ESC/Java2 using the `-loopsafe` switch.

7. Repeat Tasks 3 through 6, iteratively enriching the speicification, until ESC/Java2 reports no warnings, or until you are confident that all warnings are *spurious* (you may still add assumptions to investigate how much help ESC/Java2 needs to prove your program correct).

Do not forget to discuss all your *specification decisions* in the report. Please summarise your experiences with the tool.

A Priority Queues over Binary Heaps

This appendix contains a sample Java implementation of priority queues over binary heaps. The code is also available from the course web page.

```
 /************************************************************************
 *  Compilation:  javac PQ.java
 *  Execution:    java PQ
 *
 *  Priority queue (of integers) implementation with binary heap.
 *
 * Acknowledgement:
 * A modified version of code originally by
 * Robert Sedgewick and Kevin Wayne, responsibles for the COS 126
 * course at Princeton University
 * url: http://www.cs.princeton.edu/introcs/home/
 *
 *************************************************************************/

class PQ {
    private int[] pq;          // store elements at index 1 to N
    private int N;             // number of elements

    // set initial size of heap to hold size elements
    public PQ(int size) {
        pq = new int[size + 1];
        N = 0;
    }

    // set initial size of heap to hold 0 elements
    public PQ() { this(0); }

    boolean isEmpty() { return N == 0; }
    int size()        { return N;      }

    void insert(int item) {
        // double size of array if necessary
        if (N >= pq.length - 1) {
            int[] pq = new int[2*(N+1)];
            System.arraycopy(this.pq, 0, pq, 0, N + 1);
            this.pq = pq;
        }

        pq[++N] = item;
        swim(N);
    }
```

```
int delMax() {
    exch(1, N);
    sink(1, N-1);
    return pq[N--];
}

private void swim(int k) {
    while (k > 1 && less(k/2, k)) {
        exch(k, k/2);
        k = k/2;
    }
}

private void sink(int k, int N) {
    while (2*k <= N) {
        int j = 2*k;
        if (j < N && less(j, j+1)) j++;
        if (!less(k, j)) break;
        exch(k, j);
        k = j;
    }
}

/***********************************************************************
 * Helper functions.
 **********************************************************************/
private boolean less(int i, int j) {
    return (pq[i] < pq[j]);
}

private void exch(int i, int j) {
    int swap = pq[i];
    pq[i] = pq[j];
    pq[j] = swap;
}
```

```
    /***********************************************************************
     * Test routine.
     **********************************************************************/
    public static void main(String[] args) {
        PQ pq = new PQ();
        pq.insert(2);
        pq.insert(3);
        pq.insert(7);
        pq.insert(5);
        while (!pq.isEmpty())
            System.out.println(pq.delMax());
    }

}
```