

CMPS 290G – Topics in Software Engineering
Winter 2004 – Software Validation and Defect Detection
Homework 5

Due: by email to cormac@cs.ucsc.edu before or on *18 March 2004*

This homework is in the form of a mini-project. This project pulls together ideas from testing, model checking, and theorem proving. Your contribution can be implemented with under 100 lines of code. The code on the following pages provides basic infrastructure for this project. (This code is also at <http://www.soe.ucsc.edu/classes/cmps290g/Winter04/code.>)

- The class `Bubblesort` contains a bubblesort algorithm and a test harness that applies the sorting algorithm to arrays of length 4. While we might normally sort `ints`, for this project `Bubblesort` sorts an array of `MyInts`. The class `MyInt` represents a symbolic integer.
- The supplied class `ConstraintDiff` expresses inequality constraints over `MyInts`, and the class `Prover` decides satisfiability of these constraints.
- Your task is to implement a class `ModelChecker`.

This class will contain the usual `public static void main(String[] args)` method, which in turn calls `Bubblesort.main()` multiple times. `Bubblesort.main()` creates `MyInts` and compares them to perform sorting, by calling `ModelChecker.leq(MyInt x, MyInt y)`.

You need to implement the class `ModelChecker` so that, using a finite (and minimal) number of test runs, it verifies that `Bubblesort.main()` never yields an assertion failure.

- Hint: When you run your model checker, it should require 24 test runs of `Bubblesort`. Please include the output of running the checker in your homework submission. The routine `Prover.constraintsToString()` may be helpful for producing useful output.
- Feel free to consult classmates and ask questions in class on this assignment.
- If necessary, you can collaborate in teams of size 2 for this project.
- Please let me know if you find any bugs in the supplied code.
- Once you have your model checker working, you should also use it to test some other program (a replacement for `Bubblesort`) for which you think your model checker would work well.

Good luck!

```
/** The model checker skeleton. What you need to do is fill out this skeleton. */

public class ModelChecker {

    /** Called by Bubblesort application to compare symbolic integers. */

    public static boolean leq(MyInt x, MyInt y) {
        ... // Your job: write something clever here
    }

    /** Main routine: Calls Bubblesort.main() multiple times to test all possible paths. */

    public static void main(String[] args) {
        int numruns=0;
        while(...) {
            MyInt.reset();
            numruns++;
            Bubblesort.main();
        }
        System.out.println("Test runs: "+numruns);
    }
}
```

```
/** The code being model checked. */

public class Bubblesort {

    public static void main() {
        MyInt a[]=new MyInt[4];
        a[0] = MyInt.get();
        a[1] = MyInt.get();
        a[2] = MyInt.get();
        a[3] = MyInt.get();

        bubblesort(a);

        Assert.isTrue( ModelChecker.leq(a[0], a[1]));
        Assert.isTrue( ModelChecker.leq(a[1], a[2]));
        Assert.isTrue( ModelChecker.leq(a[2], a[3]));
        Assert.isTrue( ModelChecker.leq(a[0], a[2]));
        Assert.isTrue( ModelChecker.leq(a[1], a[3]));
        Assert.isTrue( ModelChecker.leq(a[0], a[3]));
    }

    static void bubblesort(MyInt[] a) {
        for(int i=0; i<a.length-1; i++) {
            for(int j=0; j<a.length-i-1; j++) {
                if ( !ModelChecker.leq(a[j],a[j+1])) {
                    MyInt t=a[j];
                    a[j] = a[j+1];
                    a[j+1] = t;
                }
            }
        }
    }
}
```

```
import java.util.*;

/** Represents a symbolic integer.
 * The class ConstraintDiff expresses constraints over MyInts, and the class Prover decides
 * satisfiability of these constraints.
 * The method get() yields the same sequence of MyInts on each run of Bubblesort,
 * provided reset() is called before each run of Bubblesort.
 */
public class MyInt extends Object {
    int id;
    static int numMyInt;

    static ArrayList myInts=new ArrayList();
    static int curNdx=0;

    static void reset() {
        curNdx=0;
    }

    private MyInt() {
        id=numMyInt++;
    }

    public String toString() {
        return "v"+id;
    }

    static MyInt get() {
        if (curNdx==myInts.size()) {
            myInts.add(new MyInt());
        }
        return (MyInt)myInts.get(curNdx++);
    }
}
```

```
public class Assert {
    public static void fail(String s) {
        throw new RuntimeException("Assertion failure: "+s);
    }
    public static void assertTrue(boolean b) {
        if(!b) fail("");
    }
}
```

```
public abstract class Constraint {
    public abstract Constraint negate();
}
```

```
/* Expresses constraints over MyInts of the form "x <= y + c", where x and y are MyInts.
 * The class Prover decides satisfiability of these constraints.
 * */
```

```
public class ConstraintDiff extends Constraint {
    int c;
    MyInt x,y;
    // x <= y+c

    ConstraintDiff(MyInt x, MyInt y,int c) {
        this.x=x;
        this.y=y;
        this.c=c;
    }

    public Constraint negate() {
        return new ConstraintDiff(y,x,-c-1);
    }

    public String toString() {
        return ""+x+"<="y+(c==0 ? "" : (c<0 ? ""+c : ""+c));
    }

    public boolean equals(Object o) {
        if (!(o instanceof ConstraintDiff)) return false;
        ConstraintDiff d = (ConstraintDiff)o;
        return d.x==x && d.y==y && d.c==c;
    }

    public int hashCode() {
        return x.hashCode()+y.hashCode()+c;
    }
}
```

```

import java.util.*;

/** You should need to understand only the interface of this class,
 * which is that the method satisfiable(ArrayList constraints)
 * checks satisfiability of the given constraint set.
 * There is (hoefully) no need to understand the implementation. */

public class Prover {

    private static boolean debug=false;

    public static String constraintsToString(ArrayList constraints) {
        return "s"+(new HashSet(constraints).toString());
    }

    /** Decides if the given collection of constraints is satisfiable. */
    public static boolean satisfiable(ArrayList constraints) {
        if(debug) System.out.println("Prover: "+constraints);
        HashMap stack=new HashMap();
        HashSet done = new HashSet();
        for(Iterator i=symInts(constraints).iterator(); i.hasNext();) {
            MyInt si=(MyInt)i.next();
            if (findFalseCycle(si,0,stack,done,constraints)) {
                if(debug) System.out.println("UNSAT: "+constraints);
                return false;
            }
        }
        if(debug) System.out.println("SAT: "+constraints);
        return true;
    }

    /** @param stack : SymInt -> Int
     * @param s: sum of edges from some root
     * */
    private static boolean findFalseCycle(MyInt si, int s, HashMap stack,
        HashSet done,ArrayList constraints) {
        if(debug) System.out.println("findFalseCycle: "+si+" "+s);
        Integer oi=(Integer)stack.get(si);
        if( oi != null ) {
            if(debug) System.out.println("oi.intValue(): "+oi.intValue()
                +" "+(oi.intValue() > s));
            return oi.intValue() > s;
        }
        if( done.contains(si)) return false;
        done.add(si);
        stack.put(si,new Integer(s));
        for(Iterator i=from(si,constraints); i.hasNext();) {
            ConstraintDiff cd=(ConstraintDiff)i.next();
            if( findFalseCycle(cd.y, s+cd.c, stack, done, constraints) ) {
                return true;
            }
        }
        stack.remove(si);
    }
}

```

```
        return false;
    }

    private static Iterator from(MyInt x,ArrayList constraints) {
        List r = new LinkedList();
        for(Iterator i=constraints.iterator(); i.hasNext();) {
            ConstraintDiff cd=(ConstraintDiff)i.next();
            if (cd.x==x) r.add(cd);
        }
        return r.iterator();
    }

    private static Set symInts(ArrayList constraints) {
        Set r = new LinkedHashSet();
        for(Iterator i=constraints.iterator(); i.hasNext();) {
            ConstraintDiff cd=(ConstraintDiff)i.next();
            r.add(cd.x);
            r.add(cd.y);
        }
        return r;
    }
}
```
