Software Security, Spring 2018

Project 2: Program verification with ESC/Java2

This exercise is intended to show how formal program verification can be used to certify properties of code. Use the program verification tool ESC/Java2 to verify (and debug) some pieces of Java code.

It is an individual exercise, required to pass the course.

Handing in the assignment

Prepare the annotated files in two attachements, called `BagYourName.java` and `AmountYourName.java`, where `YourName` is your full name, (and also put your name *inside* each Java file**).**

Background: The program verification tool ESC/Java2

ESC/Java2 is an automated program verification tool (aka extended static checker) for Java programs. It implements a Hoare logic for reasoning about Java programs that have been annotated with JML specifications. These specifications express, for instance, preconditions, postconditions, and invariants, and can be added as assertions to the program code. Essentially, ESC/Java2 computes weakest preconditions of methods, and then sends the resulting proof obligations to the automated theorem prover Simplify. The user never sees these proof obligations or the back-end theorem prover but receives the feedback from the tool over which assertions it could not prove.

ESC/Java2 is available for download here. The tool runs on Windows, Mac, Linux and UNIX, and comes with a lot of documentation.

The assignment

For the assignment, you should specify the two classes

1. Bag.java
2. Amount.java

with JML and run ESC/Java2 to verify these specifications. Run ESC/Java2 on these files and if the tool produces some warning,

- *either* add annotation to the code (e.g., to document invariants, preconditions, or postconditions)
- *or* fix the bugs in the code

to make the warning go away. You have to use your own best judgement to choose between these two options, but there are some deliberate bugs in the code for you to detect, with the help of the tool.

For Amount.java you must also try to formalise the informal invariant that is discussed in the file in JML, which should reveal some problems in the code.

In the end, ESC/Java2 should run without any complaints on the annotated code. ESC/Java2 complains (among other things) if it thinks a runtime exception may occur, say a NullPointerException, so if ESC/Java2 runs without complaints this means it has verified that no runtime exceptions can occur.

*More detailed instructions are given **in** the Java files. Read them!*
The ESC/Java2 tool has many command line options. The most useful ones are

- `escjava2 -suggest`
  For every warning the tool then tries to suggest an annotation that will suppress the warning.
- `escjava2 -routine`
  The tool then checks just the named routine; e.g. `escjava2 -routine arraycopy Bag.java` only checks method `arraycopy` in `Bag.java`

Some hints:

- Only look at the *first* warning that ESC/Java2 produces, and ignore the others, until you have removed the first one.
- The only JML keywords you need are `requires`, `invariant`, and `ensures`. If you want, you can also use `non_null` as an abbreviation and experiment with other features.
- Do **not** use the Java shorthand x+=10 for x = x+10.
- Whenever possible, split invariants than contain conjunctions into several smaller invariants. The tool then provides better feedback. For example, instead of writing
  ```
  //@ invariant A && B;
  ```
  write
  ```
  //@ invariant A;
  //@ invariant B;
  ```
  Do the same for preconditions (requires clauses).
- If the tool complains of invariant violations, it produces a lot of information that is hard to interpret. The crucial information to look at is
  - the line number saying *which* invariant is violated, and
  - the line number saying *where* this invariant is violated.

  NB beware that calling a method on some object may break the invariant of another object.

Points for Reflection (!)

Some questions to consider (and to answer)

1. In the end, do you think that you found ***all*** problems and that the code is correct?
2. Can you think of ways in which the tool or the specification language could be improved?
3. Instead of the tool we used, can you think of other ways (formal or informal, tool-supported or not) to find the problems that the tool found? If so, would these alternatives
   - find fewer problems, the same, or more?
   - find problems sooner or later than the current approach?
   - require more work or less work?
   - provide you with more confidence or less confidence that the code is correct?