Software Reliability Coursework 1
Using ESC/Java 2 and Daikon

Deadline: Friday 7 February (midnight)
Weight: 8% of module
To be undertaken in groups of up to three

1 Introduction

In this coursework you are provided with some simple Java classes. Your first task is to annotate the Java classes with method contracts and object invariants so that ESC/Java 2 (henceforth referred to as ESC/Java) is able to check that the application is free from various kinds of runtime error.[1]

Your second task is to write a main method which exercises these classes, and use the Daikon invariant generator to discover likely invariants from the behaviour of this main method.

Your final task is to apply Daikon to some additional Java code "in the wild". This could either be the code of an open source project, some Java software you have written in the past, or an application you develop specifically for this exercise. This final task is deliberately challenging and open-ended, for those students who want to aim for a really high mark (see the marking scheme at the end).

You will submit an archive of the annotated Java code from the first task, together with a report (maximum 5 pages, or 6 pages if you chose to undertake the third task, minimum font size 11pt) explaining various interesting aspects of the annotation and invariant discovery process associated with the first two tasks, as detailed below, and describing your experience applying Daikon to additional code in the third task.

It is recommended that you read the [ESC/Java] required reading paper before attempting the ESC/Java part of the coursework, and the [Daikon1] recommended

[1]As discussed in the lectures, ESC/Java makes some pragmatic assumptions which mean that, in general, it may not prove total absence of runtime errors. For brevity we shall say that ESC/Java successfully checks, or verifies, a Java application to mean that the tool finds no potential bugs.

reading paper before attempting the Daikon part of the coursework. These papers are available from the reading list on the course website.

Some useful ESC/Java annotation commands are provided at the end of this document. For further information on ESC/Java refer to the course lecture notes, the ESC/Java paper and the original ESC/Java manual:

```
ftp://gatekeeper.research.compaq.com/pub/DEC/SRC/technical-notes/SRC-2000-002.html
```

You might also find it useful to look at the Daikon user manual:

```
http://groups.csail.mit.edu/pag/daikon/download/doc/daikon.html
```

2 Getting started

You should work under Linux. In a `tcsh` shell, run the following command:

```
source /vol/lab/cs4/SoftwareReliability/coursework1.sh
```

This will set up various environment variables that are necessary to run the tools for this exercise. You will need to source this file whenever you start a new session and wish to work on this coursework.

Download `Bookings.tgz` from CATE and unpack it in your workspace. You will find a `src` directory containing a Java package, `bookings`, consisting of five Java classes:

- Seat.java – a class to represent seats in a theatre, where a seat is a row-number pair, such as B12
- Customer.java - an empty class which, in a full application, would represent details of customers
- SeatReservationManager.java – a class to manage the allocation of seats to customers
- ReservationException.java – an exception class
- SeatReservationDemo.java – contains an empty `main` method. In Section 4 you will write code to construct and manipulate instances of `Seat`, `Customer` and `SeatReservationManager`, and then run this code with Daikon for discovery of likely invariants

`SeatReservationManager.java` includes a couple of existing ESC/Java annotations which you should not change; these are related to the `toString` method, and for the purposes of this exercise can be ignored (though by all means try to understand them by searching the ESC/Java literature if you are interested).

Take a few minutes to familiarise yourself with the details of these classes. The classes are deliberately very simple, partly to appropriately restrict the scope of this first piece of coursework, and partly because, unfortunately, ESC/Java cannot easily be applied to large Java programs without significant annotation overhead.

Note: During the ESC/Java part of the exercise you should make *no* changes to the existing source code: you should *only* add annotations. During the Daikon part of the exercise you should modify `SeatReservationDemo.java` by adding code to the main method; you should not modify the other classes.

3 Checking the classes using ESC/Java

Navigate to the `src` folder and invoke ESC/Java as follows:

```
escjava2 -VCLimit 5000000 -NoCautions bookings/*.java
```

The `-NoCautions` argument suppresses the tool from complaining about some problems related to the installation of ESC/Java you are using. The -VCLimit 5000000 tells ESC/Java to try to check a method if the method's verification condition is no larger than 5M bytes. If at any point you see ESC/Java report "VC too big" then you will need to increase this limit.

You should find that, within a matter of seconds, ESC/Java 2 emits 31 warnings. For example:

```
bookings/SeatReservationManager.java:23: Warning: Possible negative
array index (IndexNegative)
                        [numberToIndex(s.getNumber())] = c;
                        ^
```

Add the following arguments to suppress all of these warnings:

```
-NoWarn IndexTooBig -NoWarn IndexNegative -NoWarn ArrayStore
-NoWarn Null -NoWarn NegSize -NoWarn Modifies
```

You should find that ESC/Java now reports no warnings on checking the classes.

Your task is to gradually remove these suppression flags, adding annotations that allow ESC/Java to establish freedom from each type of defect.

Suggestion: As detailed below, you must write a report detailing the various stages you go through when annotating the source code. Therefore you are advised to make thorough notes as you work through the exercise, and to make a backup (e.g., through version control) at each stage, so that you can remember what you did.

Note: You might apply annotations in a manner that will eliminate multiple classes of warning simultaneously. In this case you might find that when you remove a suppression flag no further warnings arise. If this happens then make sure you understand why, and explain this in your report.

Eliminating "null" warnings. Remove the `-NoWarn Null` argument, so that ESC/Java emits warnings related to null references.

Annotate the source code with method pre- and post-conditions and object invariants sufficient to eliminate these warnings.

As you do this, you may find it convenient to restrict ESC/Java to checking a single class by providing just the relevant `.java` file as an argument to the tool. You might also find it useful to restrict checking to a specific method, via the `-Routine <Name>` argument. E.g., pass `-Routine isReserved` to ask ESC/Java to check only the `isReserved` method of `SeatReservationManager`.

You may also find it useful to add `-NoTrace`, which stops ESC/Java from emitting an attempt at a counterexample trace illustrating a warning. Unfortunately these traces are not usually very easy to understand, and it can be simpler to do without them.

Among the warnings you deal with, and corresponding annotations you write in order to eliminate them, select the case you consider to be the most interesting. In your report, write a brief paragraph, with code snippet, explaining a) the cause of the warning, b) the annotations required to suppress this warning, including *why* the annotations are sufficient, and c) any knock-on effect that the new annotation has on further warnings raised by ESC/Java. It is very possible that you may find ESC/Java's output hard to understand, or counter-intuitive, at times. If this is the case, carefully explain why in your report.

Eliminating "negative length" warnings. Remove the `-NoWarn NegSize` argument. You should find that ESC/Java generates some warnings about the possibility of attempting to allocate an array with a negative size.

Add annotations (either in the form of pre- and post-conditions, or using other features of ESC/Java's language) to eliminate these warnings.

As above, select the most interesting case you encounter, and write a paragraph, with code snippet, explaining the warning, how you eliminated it, and any knock-on effect this had. Again, feel free to provide a discussion related to the usability of the tool.

Eliminating "negative array index" warnings. Now remove the `-NoWarnIndex Negative` argument. You should find that ESC/Java generates some warnings about the possibility of attempting to index into an array with a negative index.

Deal with these warnings through annotations, and again write about the most interesting case you encounter.

Eliminating "index too large" warnings. Do the same, and add to your report, for the `-NoWarn IndexTooBig` argument.

Eliminating "array element subtyping" warnings. Remove the `-NoWarn ArrayStore` argument. You should find that ESC/Java generates a warning indicating that it might be possible to write an element of the wrong type into an array. Add an object invariant to help ESC/Java eliminate this warning. Explain in your report the potential problem ESC/Java was warning you about, and describe your solution. **Hint:** this will involve using ESC/Java's `\typeof`, `\type` and `\elemtype` keywords.

Eliminating "violation of modifies clause" warnings. Remove the final warning suppression argument, `-NoWarn Modifies`. You should find that ESC/Java now complains about `SeatReservationManager`'s `toString` method, saying that various method calls may violate `toString`'s *modifies* clause. The problematic modifies clause is:

```
    assignable privateState;
```

which is an annotation of `toString` in `Object.spec`, ESC/Java's built-in formal specification of the `Object` class. This clause means that `toString` is limited to modifying only state which has been marked (using ESC/Java annotations which are beyond the scope of this course) as being part of a designated set called `privateState`. ESC/Java is concerned that methods which `toString` invokes may modify state outside this set.

Add annotations to satisfy ESC/Java that the modifies clause cannot be violated, and explain in your report what you had to do. Did you have to annotate as much as you expected here? If not, explain why not in your report.

Discussion of helper methods. If you marked any methods as `\helper` then write a short paragraph explaining why it made sense to do so, and discuss whether there are any disadvantages associated with this. If you did not mark any methods as `\helper` then think about whether you could have made your annotations any simpler by doing so, and discuss this in your report. You do not need to revisit your annotations to use `\helper` methods, though of course you can if you wish.

4 Discovering likely invariants using Daikon

In `SeatReservationDemo.java`, add code to `main` to create a `SeatReservationManager` object, and allocate a small number of individual seats. Compile all the Java classes in the `bookings` package using the `javac` command line tool.

Note: If you compile these classes using an IDE such as Eclipse you might run into problems: for technical reasons this coursework has been designed to use version 1.5 of the JDK; the `coursework1.sh` file sets the tools up for this. An IDE may compile your Java classes with a more recent Java version, leading to compatibility problems. If you wish, you may configure your IDE to use the JDK version 1.5, which is at `/vol/lab/cs4/SoftwareReliability/jdk1.5.0_22`.

Now run the Daikon tool to discover likely invariants associated with the execution of your `main` method. This is done in two stages. First run Chicory, Daikon's Java front-end:

```
java daikon.Chicory bookings.SeatReservationDemo
```

This should create `SeatReservationDemo.dtrace.gz`, which contains details of your program's execution trace.

Then run Daikon itself, which analyses these trace details:

```
java daikon.Daikon SeatReservationDemo.dtrace.gz
```

You should find that Daikon prints a lot of output. For each class in the application, this output includes: facts that were found to hold for all objects at method boundaries, and thus which may be object invariants (OBJECT in the Daikon output); for each method, facts that were found to hold on method entry, and thus may correspond to pre-conditions (ENTER in the output); and facts that were found to hold on method entry, and thus may correspond to post-conditions (EXIT in the output).

Use your intuition and common sense, as well as referring to the Daikon manual, to understand what this output means.

You should find that, because your `main` method does not exercise the application's classes very much, many of the invariants suggested by Daikon are spurious. Write a section in your report giving examples of this, and explaining why Daikon reports the spurious invariants.

Now rewrite `main` so that it thoroughly exercises the `Seat` and `SeatReservationManager` classes. Run Chicory and Daikon and examine the suggested invariants. If you still find invariants that look spurious then think how you could adapt `main` so that Daikon no longer suggests them. Repeat this process as you see fit.

In your report, comment on:

- the extent to which Daikon suggested invariants that you required when annotating your source code for ESC/Java
- the way you designed your `main` method to attempt to avoid the generation of spurious invariants

- any spurious invariants suggested by Daikon which you did not manage to suppress by extending `main` (if relevant)

5 Applying Daikon "in the wild"

Note: This part of the assignment is deliberately challenging and open-ended. It is there for students who wish to get a really high mark (see the marking scheme below).

Your final task is to apply Daikon so some additional code of your choosing. You could, for example, apply Daikon to an open source Java project by running the project's test suite. Alternatively, you could apply Daikon to some Java software you have written previously, perhaps for a past coursework or lab. Or you could write a piece of Java software specifically for this exercise. Explain the nature of the Java code you decided to use, and why you made this decision.

If it is not straightforward to apply Daikon to the code you have selected then explain why, and think about what changes you need to make to the code base so that it can be analysed using Daikon. If it is feasible to do so, make these changes to the code base.

If you do manage to get Daikon to work on code base, write up a brief analysis of the likely invariants that Daikon reports and the effectiveness of Daikon in doing so. What you choose to dwell on in your analysis is up to you. But you might, for instance, comment on whether you could understand the invariants, whether you believe they are real invariants vs. spurious invariants, and whether Daikon exhibited good scalability when analysing the software.

6 Submission and Marking

Note: Your report should be 6 pages maximum, but should be no more than 5 pages if you decide *not* to undertake the final part of the assignment (Section 5). These are *upper* limits: if you can express the interesting issues you encountered clearly and concisely in fewer pages that is fine.

Make an archive, `BookingsAnnotated.tgz`, containing:

- your annotated Java files, including the final version of the `main` method in `SeatReservationDemo.java`
- your report, in PDF format, minimum font size 11pt

You should *not* submit Java code for the application studied in the final part of the assignment. Instead, write about the salient features of this code, in relation to analysis using Daikon, in your report.

Upload the archive via CATE.

Marking scheme:

- Annotated source code (Section 3) and `main` method (Section 4): 20
- Discussion in the report of the annotation process associated with using ESC/Java (Section 3): 30
- Discussion in the report of your experience applying Daikon to the `bookings` code base (Section 4): 25
- Discussion of your approach to, and experience of, applying Daikon "in the wild" (Section 5): 25

Total marks available: 100

Useful ESC/Java constructs

- Single line comment annotation: `//@ <annotations>`
- Multi-line comment annotation:
  `/*@ <annotations, possibly spanning many lines @*/`
- Method precondition: **`requires`**
- Method postcondition: **`ensures`**
- Object invariant: `invariant`
- Mark parameter as non-null: `non_null`
- Refer to the result of a method call: `\result`
- Refer to the value a field had at the start of a method call: `\old(<field>)`
- Mark a method as "helper": `helper`
- Implication: `P ==> Q`
- Universal quantification over all values x of a type:
  `(\forall <type> x; <formula that may talk about x>)`
  Note that a `\forall` construct *must* be enclosed in parentheses

- Dynamic type of expression: `\typeof(e)`
- Refer to a type T in an annotation: `\type(`T`)` (e.g., `\type(Seat)`, not just `Seat`)
- Get the element type of an array type in an annotation: `\elemtype(<array type>)`