System Validation 2014-1A
Homework Part 2 – JML

Stefan Blom, Marieke Huisman, and Wojciech Mostowski

Available: Tuesday October 14, 2014
Deadline: **23:59 CET, Tuesday October 28, 2014**

The assignment should be made in pairs.

The assignment has two parts:

- formal specifications from informal requirements of a lock system
- a thorough analysis of a small Java implementation of the Sokoban game with JML tools (OpenJML and JMLUnitNG).

All solutions should be uploaded via BlackBoard. You should hand in is a single ZIP (or TGZ) file with

- The report in PDF format as `report.pdf`. The report should be self contained, that is, we should be able to see from the report what you have done, without looking into your source files.
- The complete source with JML annotations for all the exercises of the Sokoban game along with the test data files and the test results from the last exercise.

In all files, and on the first page of the report, write your names and student numbers. The detailed overview of what needs to be handed in is given at the end in Section 6. **Please read that section carefully and take note of the files that you should not hand in!** Also, please read through the whole assignment text before you engage in your tasks.

Lock Requirements (10 pts)

Consider the requirements for the lock system, as established during the first lecture. Write formal specifications for the following requirements as possible using the most appropriate formalism (*i.e.*, use an appropriate temporal logic, JML).

- Always at least one door closed
- The doors may only be open when the water level on both sides is equal
- It should be possible to lower and raise the water level
- It should only be possible to raise/lower the water level when both doors are closed
- The doors should not close when there is a boat passing through

(−2 points for each incorrect formalisation)

You may assume the existence of appropriate methods (but indicate your assumptions clearly).

Sokoban

The Sokoban game implementation and any other necessary files are provided in the file `sokoban.tgz`. This is a TGZ-ed Eclipse project to be imported into your Eclipse environment, see next page.

The Sokoban assignment has four parts:

- Annotating the implementation with JML and performing hand guided (through playing the game) Runtime Assertion Checking (RAC),
- Writing down a functional specification for a particular piece of the implementation code and verifying it with the OpenJML static checker (ESC),
- Improving your specifications with the use of model fields,
- Providing suitable specifications for the test generation process (with JMLUnitNG), generating test cases and providing test data.

Important note: The OpenJML tool undergoes frequent updates to accommodate your needs and to make its quality better. This means that not everything always works smooth, but also that you can contribute to the this effort by collecting any problems you may run into with respect to tool quality and its bugs. Please send any such remarks, comments, and problems directly to Wojciech Mostowski at `w.mostowski@utwente.nl`, that is **outside** of the BlackBoard submission process, so that we can pass them on to the author of OpenJML.

Also, we urge you **not to update** the OpenJML plug-in in Eclipse to any newer version, even if Eclipse proposes so. Such an update will break the scripts in the last part of the assignment and will require some work on your side to make them work again. If you happened to update the plug-in anyway, the best (quickest) solution is probably to reinstall the System Validation image fresh from the download that we provided on BlackBoard (saving any files that you may have on the image first of course).

1 Warmup & Preparations

As a preliminary step get familiar with the game itself:

`http://en.wikipedia.org/wiki/Sokoban`

In this assignment we made one modification to the game rules, namely, we do not require the number of movable crates to be the same as the number of the marked spots on the board (though this would make for an interesting property to be specified in JML). The following explanations apply to course's SV Linux installation. The first part (up to an including Section 4) of the assignment is to be done within the Eclipse environment. The last part on test generation has to be done with command-line tools, but we provided ready-made scripts to maximally ease the process. These scripts are included in the project TGZ file and are explained in Section 5.

The first step is to import the Sokoban project into Eclipse. To do that, go to the *File* menu in Eclipse and choose the *Import...* item. There choose *General/Existing Projects into Workspace* and click *Next*. In the dialog choose the option *Select archive file* and browse to the location where you saved the `sokoban.tgz` file. Then click through with *Next* to import the project.

At this point, after selecting the Sokoban project in *Package Explorer*, you should be able just to run the game with the Eclipse run button (the one with the green play symbol). This will automatically create a run configuration for Sokoban, but you can also choose to create one manually. The game can be played both with the arrow keys and the mouse, try both.

Notes: Throughout the assignment ignore the class `GameGUI` entirely. It only provides a graphical interface to interact with the core of the game and, being implemented in Swing, it is very difficult to annotate or verify. **In particular, you are not allowed to change anything in this class to fix any problems that you encounter. It is the core of the game that should be correct, not the GUI.**

Finally, some of the methods and classes in the project are already annotated with JML tags to guide the RAC and ESC process. These tags are `@skipesc` and `@skiprac`. They switch off ESC or RAC for a given method or class. In particular, `toString` methods (we provided ready annotations for them) as well as constructors are a bit difficult to check statically, and the GUI class does not take Runtime Assertion Checking well (see also comment above, it is not core of the game and out of your interest for the assignment anyhow). You are allowed to included further `@skipesc` tags so that you only static check the method that you are interested in (see also Section 3). We, however, encourage you to statically check as much of the code as you can with in the given time frame (for example, the `Player` class, should be fully and easily static checkable with OpenJML). **You are not allowed to include additional `@skiprac` tags** – the whole code (excluding the `GameGUI` class) should be Runtime Assertion Checked. When you do put the `@skipesc` tags in your code make sure they occupy a single line on it owns, that is, look like:

```
//@ skipesc
... your code ...
```

(If you don't the test generation scripts in the last part of the assignment **will** break.)

2 JML Annotations and Runtime Assertion Checking (25 pts)

The first part of the assignment is about annotating the Sokoban source with JML and performing Runtime Assertion Checking (RAC).

Note: To compile your project with RAC enabled you normally press the RAC button on the Eclipse toolbar. This is tedious and easy to forget, and if forgotten Eclipse will compile your freshly changed files anyhow, but with a regular (non-RAC) compiler. This may result in not getting the expected complaints from RAC (because RAC checks are not compiled in). Thus, we strongly advise you to switch on RAC compiling to be the default Eclipse behaviour at this point. To do this, go to the *Window* menu and choose *Preferences*. There choose the OpenJML options, find the option *Enable Auto RAC Compiling (on file saves)* and activate it.

Exercises

i. Browse through all Java files and locate informal descriptions of code properties. These are marked with **`@informal`**. For every informal description provide a suitable JML annotation that formalises the requirement. Use class invariants (**`invariant`** keyword), methods contracts (**`behavior`**, **`requires`**, **`ensures`**, **`signals`**, . . .), and in-line assertions **`assert`** where you see fit. Provide all other annotations, like visibility or purity tags, you find necessary or useful. Describe your specifications in the report. **(15 pts)**

Exercise the run-time checker by playing the game and seeing if it can spot any problems.

It is very useful to periodically type check your annotations with the JML toolbar button. This will also clear up any previous warning tags in Eclipse that OpenJML is not very good in doing it by itself. In any case, you can always choose the *Delete JML markers* command in the *OpenJML* menu to clean up JML warnings.

ii. Fix any mistakes you find in the code and document them in the report. Also describe how exactly you found them, what exactly gave you evidence that there is a problem. Argue why you think you have examined all annotations, or why do you think you have not examined all of them. The game implementation has two problems that we expect you to identify. **(10 pts)**

Notes: For the Runtime Assertion Checking it is fine to leave the **`assignable`** clauses unspecified (they default to **`\everything`**). However, in many cases the **`assignable`** clauses are very obvious and can be easily specified.

You are likely to find some of the problems in the implementation even before you start with your annotations and Runtime Assertion Checking. Regardless of that, we still expect you to provide suitable specifications and show how they are violated by the RAC instrumented code.

3 Static Checking (25 pts)

Warning: This part is likely to take more time than you expect. Solving the entire part is not strictly necessary for the remaining parts of the assignment. Thus, you may do this part, especially point **iv**, out of order.

In this part of the assignment you will perform formal verification of one crucial method of the game. Make sure that your current state of the annotations from the previous part is fine (in terms of type checking) with OpenJML. Some of the specifications from the RAC part might be crucial for this part and may still have to be adapted to enable successful ESC. In particular, OpenJML is a bit peculiar when it comes to using pure method calls in specifications (like, for example, using the method `isOpen` in an invariant) when static checking is performed. Namely, in many cases such methods need to be annotated with the JML **`helper`** tag to avoid recursive static checking. The unpleasant consequence of using the the **`helper`** tag is that invariants are no longer assumed for helper methods and may have to be explicitly repeated in the **`requires`** clause.

Note: Only one small part of this assignment is hinted on in the source with the **`@informal`** tag, you need to come up with the rest. Also, you will be working with just one method in the `Game` class in this part, it is fine to annotate all the other methods in this class with **`@skipesc`** tag to only static check this one method.

Exercises

iii. Locate the method `wonGame` in the `Game` class. The `wonGame` method is responsible for checking the winning situation on the game board. Provide a top level specification for this method by iterating over the game board with a nested **`\forall`** quantifier, **(10 pts)**

iv. Verify your specification with the static checker (ESC). This will almost certainly fail on the first attempt. To complete the specification you need to: complete the specification of `wonGame` with loop invariant for the inner and the outer loop and verify the top level method. Obviously, you will need to iterate through the whole process back and forth to get the final OK from ESC. **(10 pts)**

In the report describe your specifications. If you have any remaining warnings from the static checker, try to explain why they are not an issue, and what could be done to get rid of them.

v. Also, in the report describe the general pattern for specifying such "checking" methods that iterate over data with a loop. That is, if after doing this assignment you would get a similar one, how would go about solving it, what would be the steps? **(5 pts)**

4 Abstracting Specifications (20 pts)

In this part you will use model fields to clean up your specification and provide new properties.

Note: Again, there are no `@informal` hints for this part of the assignement.

Exercises

vi. In previous part point **iii** you specified the winning condition of the game. To be able to reuse this condition make it a model field of the `Game` class called `wonGame`, define its representation, and use it to specify the `wonGame()` method instead. For consistency, recheck that the static checker still verifies the method to be correct, **(7 pts)**

vii. The game won is one of the possible states of the game board. Another possible state is when the game is still theoretically in play (not yet all the crates are on their marked squares), but there are no more meaningful moves possible, because all the crates are located in such positions that they cannot be pushed any more. Define another model field `gameStuck` and give it a definition, **(7 pts)**

viii. Using these two model fields specify an invariant for the `Game` class, stating that the game is never stuck, unless it is already won, **(2 pts)**

The invariant that you just specified obviously does not hold. RAC-compile and RAC-execute the game and arrange the crates on the board such that the game crashes because of an assertion failure,

ix. In the report describe in natural language what is the simplest check you can think of to establish that the crates cannot be moved any more. Provide the screen-shot of the game board documenting the state of the board when the game crashed because of the assertion failure mentioned above. **(4 pts)**

Note: If you run into probles taking screen-shots – make the SV Linux go into VirtualBox windowed mode and then the *PrintScr* button should work in your host OS.

5 Test Generation (20 pts)

In this part you will provide suitable specifications for specification-based test generation, generate test cases with JMLUnitNG, and provide test data for the generated tests.

Help note: For this assignment we provided a handful of scripts that will help you in (probably repetitive) generation, compilation, and running of tests. This part of the assignment has to be done with command-line tools. We also provided an XML file for the test running script that helps in getting the tests results pretty printed in a browser window. Please do not edit any of these files (scripts or the XML file).

Note: Yet again, there are no official `@informal` hints for this part of the assignment.

Exercises

- The method of interest for this exercise is `movePlayer` from the `Game` class. The test generation process is the following:
  - ○ First open the Terminal and go to the Sokoban project directory:

    ```
    cd workspace/sokoban
    ```

    (This is assuming you did not change the default workspace location in Eclipse or settings when importing the project into Eclipse.) You can keep Eclipse running at the same time to edit your JML specifications, there is no problem with that (however, if you edit the Java files outside of Eclipse with Eclipse still running you need to press F5 to make Eclipse refresh the files in its windows).
  - ○ Then generate tests for Sokoban with:

    ```
    ./generatetests
    ```

    This will put the generated tests in the `tests` directory. (If you are not familiar with Linux – the "`./`" bit in the command line call is important.)
  - ○ Compile the annotated source with RAC compiler: `./compilesrc`
  - ○ Compile the test suite with Java compiler and necessary libraries: `./compiletests` (The compiled files from the last two steps end up in the `testbin` directory.)
  - ○ Run the tests for the `movePlayer` method: `./runtests`
    This prints the test results on the console but also produces an HTML file. You can view the results by viewing this file in the browser (keep this file loaded and just refresh it doing the exercise):

    ```
    firefox "test-output/System Validation 2014 Homework 2/movePlayer.html"
    ```

  Make sure that this works for you with the current state of your JML annotations, whatever they are.

x. The tests results that you got are probably far from interesting. The method `movePlayer` have many interesting execution branches that may not have been tested. To fix this do the following: **(20 pts)**

  - ○ Construct different sets of preconditions to provide different JML specification cases, so that you have at least one specification case for every possible outcome of the method. For each specification case state what the return result should be and how the position of the player changes. Document these specifications in the report,

  - Provide the test data in the template files generated by JMLUnitNG so that the test run can exercise your specifications. The bare minimum that you need for this is the `Game_instanceStrategy` template to define suitable test game setups (board and player) and the strategy for providing the **`newPosition`** parameter for the method `movePlayer`. It is sufficient to provide minimal test data, but large enough so that a move can be made or at least checked by the test. Save the files you edited in this step for the hand-in. In the report, describe what test data you had to provide.

It is fine to have skipped (yellow) tests in the report after this (JMLUnitNG will still insist to run tests with **`null`** parameters), but you have to produce enough passed (green) tests to convince yourself that every specification case is checked at least once. Failed (red) tests should not be present in the test results.

In the report put the results of the run of the generated tests from the browser. Simple screen-shots are fine, but a Firefox generated PDF is even better (use the *Print* option).

6 Summary of the hand-in material

- The report file in PDF format should contain the descriptions and data you were asked to provide in the different parts of this assignments. In particular, the JML specification you wrote should be included,
- The source Java files of the Sokoban game should contain JML annotations for all parts (that you completed) of the assignment. There is no need to create a separate copy for the different parts, the JML annotations should be compatible throughout with every assignment step. Provide the whole source in one ZIP (or TGZ) file, the archive should also include test results from the last part,
- The JMLUnitNG template files that you edited in the last part to provide the test data,
- **Do not hand in the compiled classes or the files generated by JMLUnitNG other than the ones you edited!**

  Because this is important for us for the correction work, we repeat this again: **Do not hand in the compiled classes or the files generated by JMLUnitNG other than the ones you edited!** :-)