Lab 2: Specification and Verification of Programs using KeY

SEFM Course Team

General Remarks

Please read these remarks carefully.

Please check first that all your settings are correct and chosen as shown below:

> Attention: When using KeY, please check the following settings:
>
> **Strategy Tab (lower left pane)**
>
> | | |
> |---|---|
> | JavaDL | Selected |
> | Goal Choser, Stop At | Default |
> | Logical Splitting | Normal |
> | Loop treat. | None |
> | Method treat., Query treat. | Expand |
> | Arithmetic treat. | DefOps |
> | Quantifier treat. | No Splits with Progs |
>
> **Menu > Options > Default Taclet Options** (*To change these settings you have to load a problem first.*)
>
> | | |
> |---|---|
> | intRules | arithmeticSemanticsIgnoringOF |
> | programRules | Java |
> | initialisation | disableStaticInitialisation |
> | rtsj | off |

A few words on the specification, implementation and verification:

- All fields are marked as `spec_public`. Please access them directly in your specifications and not via queries (pure methods).
- You are not allowed (and it is not necessary) to introduce additional methods for specification-only purposes.
- If you are asked to provide a specification, stay as close and complete as possible to the assignment text *and* the comments in the source code. If you think a comment is ambiguous, choose the most sensible interpretation and document your choice in the comments.
- If you are asked to provide an implementation keep it as simple as possible, avoid the use of any Java API as well as floats and doubles.
- If you are asked for proofs of methods with several specification cases, please provide one saved proof for each specification case in isolation. Do not combine contracts.
- When KeY stops keep in mind that it can have several reasons: some proofs are just big and you need to press the Start/Continue button several times (depending on the maximal steps you have set), the proof might not be closable and you need to find the bug in your implementation resp. specification or the automation cannot find a proof and you need to perform some steps by hand.

In this lab assignment we extend the Maze example from the exercise session. For a short introduction into the example and additional explanations see the website of the exercise session.

For this lab you will need the skeleton classes as provided on the lab assignment website. Please download the files, save them to your harddisk and unpack them. After unpacking the files, you should see a directory called `MazeLab` with a subdirectory called `src`.

Please upload for each of the three assignments an archive file (zip or tgz) called `assignment`Nr`.zip`
The archive file should contain:

- all source files containing your specifications and implementations;
- all proofs, choose the following naming scheme:

  $$\texttt{assignment}\langle Nr \rangle \langle ProofObligation \rangle \langle MethodName \rangle\texttt{.proof}$$

  for instance: `assignment2EnsuresPostIsPossible` (similar for proof obligations `PreservesInv` and `RespectModifies`). If there is more then one specification case please number them by adding e.g., `NormalBehavior`Nr or `ExceptionalBehavior`Nr.
  If when you submit you haven't been able to close a proof, save the unclosed proof and send it anyway! In this way you will receive effective feedback.
- where appropriate a text file `report.txt` with your answers to the questions (please reference them clearly) and or additional comments.

Assignment 1 (Preliminary). Check if you can load the Maze example as provided on the webpage. You can do so by starting KeY(the version linked to from the SEFM web page), choosing Load in menu File. Navigate to directory `MazeLab` select the `src` directory and click OK. KeY should then load the source code and the Proof Obligation Browser window should come up.

Assignment 2 (The Maze class).

1. Complete the specification of method `move` by specifying its exceptional behavior.
2. Look at the specifications of methods `won` and `isPossible` and provide a corresponding implementation for these methods.
3. Verify with KeY that the methods `won`, `isPossible` and `move` ensure their respective postconditions (EnsuresPost) for all specification cases and their assignable clauses (RespectModifies) for all normal behavior specification cases.
4. For which method(s) does it make sense to verify that the invariants are preserved? Why not for the other(s)?
5. Which invariants may be violated by the methods? Justify your answer.
6. Prove for the method(s) identified at item 4 that they preserve the invariants they may potentially violate (PreservesInv). *Hint: You may need to assume more invariants than to ensure.* Provide for each of potentially violated invariants a separate proof.

In case of problems try to understand the proof situation and to find out where the problem or bug lies.

Assignment 3 (Adding a high score list).
We extend our example now by a adding a high score list which keeps information about the players' score. The high score class is called Highscore which stores the achieved high scores in an array of objects of type Record.

1. Read the class and field comments of class Highscore and class Record carefully and formalise them as JML invariants.
2. Now we look into the methods of class Highscore.
   (a) Identify those methods that are side-effect free and add the appropriate JML modifiers.
   (b) Verify that method `minScore` ensures its postcondition. If your proof does not close, identify the problem (either in the specification or implementation), fix it (in such a way that the original intent of the method is still preserved) and verify the method afterwards again.
   (c) Complete now the specification of the `add` method. The final method specification should consist of three normal behavior specification cases: i) capacity not yet exceeded, ii) capacity exceeded, record is smaller or equal than the minimum, and iii) capacity exceeded, new record is greater than a current minimal element.

   The specification case i) is already given, you need only two provide the specification cases ii) and iii).

   The specification should be as complete and precise as possible with the following exception, you need only to specify that the element is afterwards in the array you do not need to specify that all previous elements are still in the array.
   (d) We want now to verify that the method `add` is correct with respect to the specification cases of the previous item.

   For the moment we look *only* at the specification cases i) and ii). Verify that method `add`

- ensures its post-conditions (*EnsuresPost*). You might find that the specification case i) cannot be proven correct. Where is the problem (implementation or spec)? Provide a natural language explanation of the problem, fix it and verify the fixed version.
- changes at most those locations specified in the assignable clause (*RespectModifies*).

(e) We turn now towards specification case iii). Your task is to verify that method `add` ensures the post-condition of the specification case.

Note: To determine the new minimum another method `findNewMin()` is called. The verification of the third specification case involves proving a loop. The prover will stop at some point in front of the loop, you need then to select the loop invariant rule with variant and apply it. The code is already annotated with the necessary JML statements (loop invariant, assignable, and decreases clauses) for the verification of the loop and KeY should pull them in by itself during the application of the loop invariant rule. What you may also try doing to automate proof search further is to switch the loop treatment option in the proof options pane from *None* to *Invariant*, in which case the loop invariant rules will be chosen automatically, provided the necessary JML annotations are present at the loop entry. If you cannot close the specification case then there is most likely something wrong with your specification of the third case (rather than the specification of the loop).

(f) Verify that method `add` preserves the invariants of class Highscore. The note from the previous item applies here as well.

Assignment 4 (The class HighscoreSorted).The class HighscoreSorted is a subclass of Highscore. In contrast to class Highscore, HighscoreSorted keeps its list sorted. Your task is now as follows:

1. Implement its method `max` runtime efficiently, specify explicitly that it returns the maximal element and verify the method.

2. Complete the specification of method `insertAt()` by adding the missing exceptional specification cases and verify *EnsuresPost* for the exceptional behavioral cases only.

3. Uncomment and complete the loop invariant, the decreases, and assignable expressions in `insertAt()`. Verify then *EnsuresPost* for the normal behavior specification case.