Formal Methods in Software Development
Assignment 6 (January 5)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

The result is to be submitted by the deadline stated above *via the Moodle interface* of the course as a `.zip` or `.tgz` file which contains

1. a PDF file with
   - a cover page with the course title, your name, Matrikelnummer, and email address,
   - a section for each part of the exercise with the requested deliverables and optionally any explanations or comments you would like to make;
2. the JML-annotated `.java` file(s) used in the exercise.

Email submissions are *not* accepted.

Assignment 6: JML Verification with OpenJML and KeY

Take from Assignment 5 the JML-annotated program functions `minimumPosition`, `minimumElement1`, `minimumElement2`, `overwrite`, `replace`, and `add1` (not `add2`). Perform for each of these functions the tasks described below (you may use a single Java class file for all functions, but do not include any test code or `main` functions in this file).

Annotate every loop in the function with an appropriate invariant (`loop_invariant`) and termination measure (`decreases`) and check these with `escjava2` and `openjmlesc` (potentially also `openjml8esc`). Please note that `escjava2` only checks whether the invariants hold after some iterations, i.e., only if an invariant is too *strong*, it reports an error. On the other hand, `openjmlesc` inded tries to prove the verification conditions generated from the invariants; if it reports an error, this may indicate that an invariant is too *weak*. However, the `openjmlesc` prover is not complete; it may also fail to prove a valid verification condition.

It is recommended to use multiple `loop_invariant` statements for each conjunct of the invariant; then it is easier to determine which part of an invariant failed. In the case of a `for` loop, do not forget to add the range condition for the loop variable to the invariant. If an array is modified, do not forget to specify which part of the array has remained unchanged so far.

When you are confident about these annotations, provide the loop also with an `assignable` clause (which is not standard JML but nevertheless needed by the KeY verifier) that lists all variables/array contents changed in the loop; in the case of a `for` loop, also add the loop variable to this clause (Key should actually automatically add all local variables changed by the loop body to the `assignable` clause, but actually not all proofs work without doing this explicitly). Then verify the method with KeY.

If your annotations are correct and sufficiently strong, the proofs should run through automatically with a few invocations of the KeY prover (be sure that in tab "Proof Search Strategy" the "Defaults" options are selected; you may want to reduce the "Max. Rule Applications" to speed up the proof search)[1]. After each proof search, you may also attempt to apply an SMT Solver (I recommend CVC5 or Z3) to close some proof obligations. If you cannot complete the proof, investigate the proof tree to find out what went wrong and reconsider your annotations (they may be wrong, i.e, too strong, or too weak); for this purpose, you may unselect the option "Hide intermediate proof steps" in the context menu of the proof tree in order to see all simplification steps performed by the prover. If you cannot complete the proof, explain in detail which part of the verification failed and what you believe is the reason for the failure.

Optional: you may validate your specifications/loop invariants by translating the Java functions to RISCAL procedures, equip them with specifications and loop annotations, and additionally

[1] The verification of `minimumElement1` does not automatically run through in a situation where the KeY prover does not find the correct instantiation of a universally quantified goal with the variable `result_`X denoting the result of `minimumPosition`. In that situation, you may left-click on the `forall` formula to be instantiated, select from the popup menu the first rule `allLeftHide`, and then enter in the popup window in the tab "Variable Instantiations" in the empty field to the right of field `t (G term)` the instantiation term `result_`X (check the proof situation for a corresponding variable to determine the actual value of X). Then press "Apply" and you will see the formula has been correspondingly instantiated. Press the green arrow and the proof runs through.

check/prove these (this is recommended, if an OpenJML/KeY proof fails). For each such RISCAL specification/check/proof, you get 10P bonus. Please note that the RISCAL versions of `overwrite` and `add1` cannot modify their argument arrays but have to return a corresponding result array (in addition to the result of the Java function, i.e., the RISCAL procedure may return a `Tuple` or `Record` value).

The deliverables of this exercise consist of

- a nicely formatted copy of the JML-annotated Java code (the version with the `assignable` clauses used for running KeY),
- the output of `jml -Q` or `openjml` on the class,
- the output of `escjava2 -NoCautions` and `openjmlesc` on the class,
- for each function, an explicit statement where you say whether you could complete the verification or not (and how many proof branches have remained open)
- for each function, a screenshot of the KeY prover when the proof has been completed (or got stuck) illustrating the generated proof tree (without the intermediate steps) with emphasis of the still open proof branches (if any),
- for each open proof branch a screenshot of the proof obligation, an explanation of the role of this obligation in the overall verification, and your conjecture why the proof failed.

Please also report any observations or insights you have gained.