Formal Methods in Software Development
Exercise 5 (November 27)

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 RISCAL specification (`.txt`) file(s) used in the exercise;
3. the `.java`/`.theory` file(s) used in the exercise,
4. the task directory (`.PETASKS*`) generated by the RISC ProgramExplorer.

Email submissions are *not* accepted.

Exercise 5: Proving Program Correctness

Similar to Exercise 1, we consider the following problem: given an array a of length N that holds only non-negative integers, find the index p of the maximum of a. However, if $N = 0$, then p shall be set to -1. The goal of this exercise is to use the RISC ProgramExplorer to take the following Java program that solves this problem, equip it with specification and annotations, analyze its semantics, and verify its correctness with respect to its specification:

```
class Exercise5
{
  // returns index of maximum element in array a
  // of non-negative integers (-1, if a is empty)
  public static int maximum(int[] a)
  {
    int n = a.length;
    if (n == 0)
      return -1;
    else
    {
      int j = 0;
      int m = a[0];
      int i = 1;
      while (i < n)
      {
        if (a[i] > m)
        {
          j = i;
          m = a[j];
        }
        i = i+1;
      }
      return j;
    }
  }
}
```

In detail, perform the following tasks:

1. (35P) For a first validation of specification and invariants, take the RISCAL specification file `maximum.txt` which embeds an algorithmic version of above code in a procedure `maximumIndex` and equip this procedure with suitable pre-conditions, post-conditions, invariants, and termination term. Validate (for values $N = 0$ and some $N > 0$) the annotations, i.e., check that the procedure satisfies the specification, the termination term is adequate, and the invariant is not violated. These annotations shall then serve as the basis of the further proof-based verification.

   Optional (25P bonus): Derive (in the style of Exercise 2) verification conditions for the total correctness of the loop (for a suitable loop pre- and post-condition) and check their validity (this validates that the invariant is indeed adequate, i.e., not too weak for the subsequent proof-based verification).

2. (30P) Then create a separate directory in which you place the file `Exercise5.java` that contains above Java procedure, `cd` to this directory, and start `ProgramExplorer &` from there. The task directory `.PETASKS*` is then generated as a subdirectory of this directory. (If you use the virtual course machine, place the directory for this exercise into the home directory of the guest user; in particular, do not place it into the directory shared with the host computer).

   Specify the method by an appropriate contract (clauses `requires`, `assignable`, and `ensures`) and annotate the loop with an appropriate invariant and termination term (do not forget the non-null status of the array).

   Investigate the computed semantics of the method, in particular the transition relations and termination conditions of

   - the loop body
   - the loop itself,
   - the whole method.

   in order to judge the adequacy of your annotations. Give an informal interpretation of the semantics (and your detailed explanation whether respectively why it seems adequate).

   > Take the interpretation and explanation serious; not only will they be judged for the credit of this part of the exercise, they will also help you to detect errors that would be hard to find in the later proofs. In particular, a description in just two sentences will not do to get full credit.

3. (35P) Verify all (non-optional) tasks generated from the method. Only few of them should require interactive proofs; most of these can probably be performed just by application of `decompose`, `split`, `scatter` and `auto`.

   > The only more complex cases should be the proofs that the invariant is preserved and that the method body ensures the postcondition; here it is wise to first perform a `decompose` and then a `split` corresponding to the two branches in the method respectively the loop body (if you immediately perform a `scatter`, you have to make a `split` in each of the resulting proof obligations which considerably blows up the proof).
   >
   > Furthermore, if a proof results in the knowledge `executes_`(s) and `returns_`(s), this indicates that program state s is both the result of a normal execution and of executing the `return` statement, which is a contradiction; this contradiction can be exhibited and the the proof situation thus closed by executing the proof command `expand executes_, returns_;`.

   If the proof of some invariant-related verficiation condition fails, you may consider the bonus exercise for validating the adequacy of the loop invariant.

The deliverables of this exercise consist of

1. a nicely formatted copy of the RISCAL specification (included as text, not as screenshots);
2. the outputs of the checks (included as text, not as screenshots) with explicit statements whether the checks succeeded;
3. a (nicely formatted) copy of the `.java`/`.theory` file(s) used in the exercise,
4. a screenshot of the "Analysis" view of the RISC ProgramExplorer with the specification/implementation of the method and the (expanded) tree of all (non-optional) tasks generated from the method,
5. a screenshot of the corresponding "Semantics" view and an informal interpretation of the method semantics;
6. for each task generated by the RISC ProgramExplorer an explicit statement whether the goal of the task was achieved or not and, if yes, how (fully automatic proof, immediate completion after starting an interactive proof, complete or incomplete interactive proof),
7. for each truly interactive proof, a screenshot of the corresponding "Verify" view with the proof tree.