Formal Methods in Software Development
Exercise 6 (December 4)

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 6: Proving Program Correctness

Use the RISC ProgramExplorer to formally specify the following program, analyze its semantics, and verify its total correctness with respect to its specification:

```
// sort array a in ascending order using the "selection sort" algorithm
public static void sort(int[] a) {
  int n = a.length;
  int i = 0;
  while (i < n-1) {
    int j = minimum(a, i);
    if (i != j) { int e = a[i]; a[i] = a[j]; a[j] = e; }
    i = i+1;
  }
}

// returns position of smallest element in a[i..]
public static int minimum(int[] a, int i) {
  int n = a.length;
  int p = i;
  int m = a[p];
  int j = i+1;
  while (j < n) {
    if (a[j] < m) { p = j; m = a[p]; }
    j = j+1;
  }
  return p;
}
```

In detail, perform the following tasks:

1. (35P) For a first validation of specification and annotations, take the RISCAL specification file `sort.txt` which embeds an algorithmic version of above code and equip the procedures with suitable pre-conditions, post-conditions, invariants, and termination terms. Please note that here `sort` is a function that returns a sorted duplicate of its argument.

   For the purpose of this exercise, in the specification of `sort` it suffices to state that the result array is sorted; it is not necessary to establish a relationship between the elements of the input array and those of the result array.

   Validate (for values $N = 0$ and some $N > 0$) the annotations, i.e., check that the procedures satisfy the specification, the termination terms are adequate, and the invariants are 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 `sort` and check their validity (this validates that the loop invariant and the specification of `minimum` are indeed adequate, i.e., not too weak for the subsequent proof-based verification). Here you may apply the Hoare rule $\{P\}$ `j = minimum(a, i)` $\{Q[j/result]\}$ where P and Q are the precondition respectively postcondition of `minimum`.

Optional (10P bonus): complete the specification of `sort` by also claiming that the result array is a permutation of *a* and check it: for this you must specify the existence of a one-to-one mapping of indices from the original array to indices of the result array (such mappings are just arrays of integers).

2. (35P) Create a separate directory in which you place the file `Exercise6.java`, `cd` to this directory, and start `ProgramExplorer &` from there. The task directory `.PETASKS*` is then generated as a subdirectory of this directory.

   Derive a suitable specification of `sort` and `minimum` (clauses `requires`, `assignable`, `ensures`) and annotate the loop in the body of `sort` appropriately (clauses `invariant` and `decreases`). In the invariant of the loop, specify which part of the array has already been sorted and which has remained unchanged (along with all necessary minor conditions on *a*, *n* and *i*). Do not forget to specify the non-nullness status and the length of the array (in both pre-state and post-state). Based on these annotations analyze the semantics of `sort` (of the whole loop and of the method body) and verify the correctness of the method with respect to its specification.

3. (30P) Annotate the loop in the body of `minimum`, analyze the semantics of the method, and verify its correctness with respect to its specification.

The deliverables are for both `sort` and `minimum` the same that have been requested in Exercise 5 (if you cannot show all required verification conditions for Part 2 of the exercise you can nevertheless perform the proofs for Part 3).

Among all verification tasks, the only complicated one is to show in Part 2 that the invariant of the loop in `sort` is preserved by every iteration of the loop (which invokes `minimum`). If in this proof some goal contains a term of form `IF` *F* `THEN ... ELSE ... ENDIF`, perform a case distinction by executing `case` *F*.

Otherwise, all proofs may proceed by application of the commands `decompose`, `split`, `scatter`, `auto`, and `instantiate`.