Formal Methods in Software Development
Exercise 6 (December 8)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

September 8, 2014

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 (nicely formatted) copy of the *.java/.theory* file(s) used in the exercise,
   - the deliverables requested in the description of the exercise,
   - for each program method, 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,
   - for each program method, a screenshot of the corresponding "Semantics" view and an informal interpretation of the method semantics;
   - for each task 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),
   - for each truly interactive proof, a screenshot of the corresponding "Verify" view with the proof tree,
   - optionally any explanations or comments you would like to make;
2. the *.java/.theory* file(s) used in the exercise,
3. the task directory (*.PETASKS**) generated by the RISC ProgramExplorer.

Email submissions are *not* accepted.

Exercise 6: Merging Sorted Arrays

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

```
class Exercise6
{
  // merges two sorted arrays a and b into a sorted result array c
  // ("sorted" means "sorted in ascending order")
  public static int[] merge(int[] a, int [] b)
  {
    int n = a.length+b.length;
    int[] c = new int[n];
    int i = 0;
    int j = 0;
    int k = 0;
    while (k < n)
    {
      boolean aisnext = append(a, b, c, i, j, k);
      if (aisnext)
        i = i+1;
      else
        j = j+1;
      k = k+1;
    }
    return c;
  }

  // writes into c[k] either a[i] or b[j], whatever is smaller
  // returns true iff a[i] was written
  public static boolean append(int[] a, int[] b, int[] c, int i, int j, int k)
  {
  }
}
```

First, 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.

Then perform the following tasks:

1. (35P) Derive a suitable specification of `merge` (clauses `requires`, `assignable`, `ensures`) and annotate the loop in the body of the method appropriately (clauses `invariant` and `decreases`). Based on these annotations analyze the semantics of `merge` (of the whole loop and of the method body) and verify the correctness of the method with respect to its specification.

   Since `append` has not yet been specified, the semantics of the loop body cannot

yet be analyzed and it cannot yet be verified that the loop invariant is preserved by every loop iteration; however, all other tasks can be solved.

For the purpose of this exercise, in the specification of `merge` it suffices to state that the result array is sorted; it is not necessary to establish the *exact* relationship between the elements of c and a resp. b (but you have to specify *some* relationship, see the remarks at the end). Also do not forget to specify the non-nullness status and the length of the post-state array and the usual range conditions on the index variables.

2. (35P) Develop a suitable specification of `append` and complete the analysis of the semantics and the verification of all tasks of `merge`.

   The specification of `append` must describe its return value and the post-state value of c (along with all necessary minor conditions that concern the non-nullness status and the lengths of all arrays and the range of i, j, k). Do not forget appropriate specifications of the boundary cases $i = a.length$ and $j = b.length$.

3. (30P) Provide a suitable implementation of `append`, analyze its semantics, and verify its correctness with respect to its specification.

By above tasks, you ultimately deliver for `merge` and `append` all results that have been requested in Exercise 5 (if you cannot show all required verification conditions for one part of the exercise, you may nevertheless continue with the subsequent parts).

The proofs of most verification tasks may proceed by application of the commands `decompose`, `split`, `instantiate`, `scatter`, and `auto`.

The only complicated verification task is to show in Part 2 that the invariant of the loop in `merge` is preserved by every iteration of the loop (which invokes `append`): the core is to show that if array c was sorted at the entry of the loop body up to position $k - 1$, then it is sorted at the end of the loop body up to position k; this involves a comparison of $c[k]$ with its predecessor(s); since $c[k]$ is by the execution of `append` either $a[i]$ or $b[j]$, this involves a case distinction and invariant knowledge about the relative order of the elements in c before position k and the elements of a (resp. b) at or after position i (resp. j). The corresponding proofs require multiple explicit instantiations of universally quantified formulas.