Formal Methods in Software Development
Exercise 2 (November 15)

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.

Email submissions are *not* accepted.

Exercise 2: Deriving and Checking Verification Conditions

We are given an integer array a of length N that only contains elements 0 and 1. Let n be the number of occurrences of 0 in a (i.e., a has $N-n$ occurrences of 1). Our goal is to find an integer array b of length N such that

- b only contains elements 0 and 1,
- b has as many occurrences of 0 as a does (i.e., n ocurrences of 0)
- b has as many occurrences of 1 as a does (i.e., $N-n$ occurrences of 1),
- b has all occurrences of 0 before all occurrences of 1 (i.e., b holds 0 at all positions less than n and 1 at all other positions).

For instance, for input $a = [1, 0, 0, 1, 0]$, we expect output $b = [0, 0, 0, 1, 1]$.

We claim that this problem is solved by the following code fragment:

```
var b:array := a; var i:int := 0; var j:int := N-1;
while i < j do
{
  if b[i] = 0 then
    i := i+1;
  else if b[j] = 1 then
    j := j-1;
  else
  {
    var e:elem := b[i]; b[i] := b[j]; b[j] := e;
    i := i+1; j := j-1;
  }
}
```

The goal of this exercise is to verify this claim by deriving and checking the verification conditions whose validity implies the total correctness of this code with respect to the given specification.

1. Take file `arrange.txt` which embeds above code in a procedure `arrange` and equip this procedure with suitable preconditions (`requires`) and postconditions (`ensures`) that formalize above specification (see Exercise 1). You may use for this purpose the function `num(`a`,`e`)` defined in the file that denotes the number of occurrences of element e in array a. Check (for small values of N) that the procedure indeed satisfies the specification (be assured that for suitable pre- and post-conditions the procedure is indeed correct; you are not allowed to change the code in any way).
2. Select the operation button “Show/Hide Tasks” to display all tasks related to the specification of the procedure (“Execute specification”, “Validate specification” and “Verify specification preconditions”). Validate the specification by executing these tasks (run “Execute specification” with execution option “Silent” switched off to investigate the input/output pairs allowed by your specification; the checks of the other tasks which denote

theorems may be performed with option “Silent” switched on). If a task stays red, i.e., the corresponding theorem does not hold, you may choose the entry “Show Counterexample” from the popup menu of the task (which appears with a right-click) to get some further insight, which may help you to improve your procedure specification.

3. Annotate the loop with suitable invariants (`invariant`) and termination term (`decreases`). Rerun the procedure check to ensure that your annotations are not too strong (but they may be still too weak to carry the verification).

   Please note that RISCAL treats procedure parameters such as a as unmodifiable constants; thus all the knowledge from the precondition of the procedure is automatically inherited and does not have to specified in the invariants again. However, you have to express in the invariant the following information:

   - Knowledge about the range of the index variables i and j.
   - Knowledge about which elements may occur in b.
   - Knowledge about the number of occurrences of 0 and 1 in b.
   - Knowledge about the elements of b already traversed by i.
   - Knowledge about the elements of b already traversed by j.

4. Now demonstrate your knowledge of the Hoare calculus by *manually* deriving from the specification and the loop annotations the verification conditions whose validity implies the total correctness (partial correctness *and termination*) of the program.

   Hint: Hoare calculus reasoning yields six conditions: one for showing that the input condition of the loop (which is different from the input condition of the program!) implies the invariant, one for showing that the invariant and the negation of the loop condition implies the output condition, three for showing that the invariant is preserved and the value of the termination term is decreased for each of the three possible execution paths in the loop body, one for showing that the invariant implies that the value of the termination term does not become negative (if you apply weakest precondition reasoning within the loop body, only one condition is derived from the loop body).

   Do not only give the final verification conditions but show in detail their derivation by application of Hoare calculus (respectively the predicate transformer calculus). *Don’t try to “guess” the condition(s)!*

   Check these conditions with RISCAL, in the style of the verification of the “linear search” algorithm presented in class. For this purpose, define predicates `Input`, `Output`, and `Invariant` and a function `Termination`, where (as shown in class) `Invariant` and `Termination` should be parametrized over the program variables. Then define six theorems `A`, `T`, `B1`, `B2`, `B3`, `C` describing the verification conditions and check these. Do not forget to make the preconditions of the procedure also preconditions of these theorems. If a theorem does not hold, you may select the operation button “Show/Hide Tasks” and choose the menu entry “Show Counterexample” to get some further insight, which may help you to correct your annotations.

5. Finally, apply the capabilities of RISCAL to automatically generate the necessary verification conditions from the annotated program. For this select the operation button “Show/Hide Tasks” to display all tasks related to the implementation (“Verify correctness of result”, “Verify iteration and recursion”, and “Verify implementation preconditions”) and verify the implementation by checking these tasks. If your annotations are adequate (strong enough but not too strong), then all red tasks turn blue (as demonstrated in class for the “summation” example). Again, if a task stays red, i.e., the corresponding verification condition does not hold, you may choose the entry “Show Counterexample” to get some further insight, which may help you to correct your annotations.

The deliverables for this exercise consists of the following items:

1. a nicely formatted copy of the RISCAL specification (included as text, not as screenshots);
2. a detailed manual derivation of the verification conditions;
3. for each check of a verification condition a reasonable selection of the output (included as text, not as screenshots) with an explicit statement whether the check has succeeded; if a check failed, give a conjecture why the check failed.
4. screenshots of (part of) the RISCAL software illustrating the automatically generated tasks after checking (panel “Tasks”, all/most of these tasks should be blue);
5. an explicit statement of whether all tasks could be successfully checked or not; if some tasks could not be successfully checked, give screenshots of the RISCAL software after the task has been selected (indicating in the editor area those parts of the program related to the task) and your conjecture why this task failed;

If the check of a theorem fails, show the printed counterexample, and give corresponding explanations. You may also (but need not) attempt to visualize its evaluation (see Exercise 1).