Formal Models for Parallel and Distributed Systems
Assignment 1 (April 28, 2025)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

The exercise is to be submitted by the deadline stated above via the Moodle interface as a single .zip or .tgz file containing

1. a single PDF file with a decent cover page (mentioning the title of the course, your full name and Matrikelnummer) with
   - nicely formatted listings of the commented model files,
   - the outputs of the RISCAL system executions/checks,
   - an explicit interpretation of the results (does the output indicate an error or not, how can it be be explained).
2. the sources of all RISCAL model files used in the exercise.

Email submissions are not accepted.

RISCAL Model of Lamport's Bakery Algorithm

Write a RISCAL model of a shared system that implements Leslie Lamport's *Bakery Algorithm*[1] to preserve mutual exclusion among $N \geq 2$ processes. In pseudo-code this model is as follows:

```
var e: Array[N,Bool] = Array[N,Bool](⊥);
var n: Array[N,ℕ[M]] = Array[N,ℕ[M]](0);
process Bakery(i:ℕ[N]) { // not actual RISCAL code
  loop {
    // compute ticket number n[i] = 1+max(n[0],..,n[N-1])
    0: e[i] := ⊤;
    1: m := 0; for j := 0; j < N; j := j+1; do { if n[j] > m then m := n[j] }
       n[i] := 1+m;
       e[i] := ⊥;

    // check that ticket number n[i] is less than the number n[j] of every process j
    2: for j := 0; j < N; j := j+1; do {
         // wait if process j currently computes its ticket number
         while e[j] { }
         // wait if process j has a smaller ticket number n[j]
         // break ties of equal ticket numbers by process number
         3: while n[j] > 0 ∧ (n[j] < n[i] ∨ (n[j] = n[i] ∧ j < i)) do { }
       }

    // enter critical region
    4: ...

    // leave critical region and reset ticket number
    n[i] := 0;
  }
}
```

The intuition for this algorithm is that of a "bakery" where the service of customers is regulated by issuing numbered "tickets"; whenever a customer enters the bakery, she receives a ticket whose number is greater than the number of the tickets of all other customers waiting in the bakery; the customer with the smallest ticket number is always the one to be served next.

In the actual algorithm, every process i has initially ticket number $n[i] = 0$ (indicating that it has no interest in the critical region). If process i wants to enter the critical region, it first sets the Boolean flag $e[i]$ to indicate that it is in the process of computing its ticket number $n[i]$ as 1 plus the maximum m of all ticket numbers; the process resets $e[i]$ to indicate that it is done with the computation.

Now, before process i enters the critical region, it waits until it has priority over every process j: if process j is currently in the course of computing its ticket number, as indicated by the flag $e[j]$, then process i waits until the computation has finished. Subsequently, process i waits, if process j has a ticket number $n[j]$ that is greater than 0 (indicating that process j wants to enter the critical region) but less than the ticket number $n[i]$ of process i (which indicates that process j is to be served before process i) or is equal to $n[i]$ (which represents a "tie" among the processes) but has process number j less than i (which breaks the tie in favor of process j).

Once process i has ensured its priority over all other processes, it enters the critical region. It leaves it by resetting its ticket number $n[i]$ to 0; this allows other processes to enter the critical region.

Now your tasks are as follows:

- First, using the accompanying file `bakery.txt` as a starting point, model this algorithm as a shared

[1] https://en.wikipedia.org/wiki/Lamport%27s_bakery_algorithm

system in the style of *Peterson's algorithm* presented in class (and described in Section 7.1 of the manuscript "Concrete Abstractions"):

  - Describe the system state by shared arrays e, n, j, and m that represent at every index i the values of above variables for process i. Also use an array pc of program counter values to indicate different steps of the algorithm, but use as few values as possible (above labels 0–4 roughly indicate the main steps of the algorithm that you may have to differentiate).
  - Ensure that every action $a(i)$ of process i reads *at most one* shared variable $v[j]$ owned by another process j (as above pseudo-code shows, process i never *writes* a variable of another process). Express the "execution logic" of the algorithm only by action preconditions; the bodies of the actions shall be just *plain assignments* to the variables owned by process i.

  The model requires one problem to be solved: the ticket numbers computed by the algorithm may become arbitrarily large while our variable domains only have finitely many values. In order to avoid "overflows", modify the algorithm such that, whenever process i detects that $m = M$ holds, it aborts the attempt to enter the critical region, i.e., it returns to its initial state by resetting $e[i]$ to "false" and $pc[i]$ to 0. Consequently, as soon as every process j with $n[j] = M$ has left the critical region and set $n[j]$ to 0 again, the value of m becomes less than M and another process may again enter the critical region with a ticket number less than equal M.

- Now check the execution of the algorithm for suitable values for $N \geq 2$ and $M \geq N$ (use $N = M = 2$ for debugging and interpreting counterexample runs, ultimately checking with $N = 3$ and $M = 5$ should be feasible in a reasonable amount of time):
  - Show that the modeled system indeed satisfies the *safety* property of "mutual exclusion" by formulating a corresponding invariant and checking it in all reachable states of the system (it is not necessary to make the invariant strong enough to show its correctness by checking the automatically generated verification conditions).
  - Show that the algorithm indeed requires the use of variable e by commenting out its use from the system model and deriving a counterexample run that demonstrates that the system violates mutual exclusion. Minimize the length of the example and informally explain the nature of problem exhibited by it.
  - Also show the mutual exclusion property by formulating a corresponding LTL formula (using the keyword `ltl`) and checking it. Remove variable e and demonstrate by the produced counterexample the violation of the property.
  - Formulate in LTL (using the keyword `ltl[fairness]`) the *liveness* property that every process infinitely often passes the loop position 0. Demonstrate that this property does not hold if we only assume weak fairness for all transitions (interpret the produced counterexample run correspondingly). Strengthen the annotations by *minimal* strong fairness requirements to make the property hold (hint: a single strong fairness requirement is sufficient). Explain why the weak fairness requirement was not sufficient but the strong fairness requirement is.
  - Formulate in LTL the property that some process infinitely often reaches the critical region. Again, show that this property does not hold with only weak fairness assumptions for all actions but does hold after a minimal strengthening of the requirements.
  - Formulate in LTL the property that for every process i it is always the case, that, if the process sets $e[i]$ to "true", it eventually enters the critical region. Demonstrate that this property does not even hold if we assume strong fairness for all transitions (interpret the produced counterexample run correspondingly).

Please give ample explanations of the results of the checks/executions and report any problems you have encountered.