# Formal Methods in Software Development
# Assignment 8 (January 26)

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 file with the Promela model used in the exercise.

Email submissions are *not* accepted.

## Assignment 8: Model Checking Termination Detection in Spin

We consider a system of $n$ processes $p_0, \ldots, p_{n-1}$ which are linked by pairs of $n$ channels $l_0, \ldots, l_{n-1}$ and $r_0, \ldots, r_{n-1}$. Each process $p_i$ can receive messages from channels $l_i$ and $r_i$ and send messages to channels $l_{(i+1) \bmod n}$ and $r_{(i+n-1) \bmod n}$, i.e., the processes are organized in a bidirectional "ring".

Each process may be *active* or *passive*; initially all processes are active. An active process may send messages to its ring neighbors or it may become passive. Every process (active or passive) may receive messages from its ring neighbors; if it was passive, it thus becomes active again.

The system is *terminated* if all processes are passive and there are no messages in the channels. If the system is terminated, it stays so (termination is a *stable* property). Our goal is to model and check an algorithm (due to Dijkstra and Safra[1]) that allows Process 0 to detect whether the system has terminated.

File `DijkstraSafra.txt` contains a Promela model of the system described above. To limit its state space, every process contains a variable *counter* that denotes the difference of the number of messages that the process has sent and the number of messages it has received. A process is only allowed to send a new message, if this difference is less than $m$.

Your first task is to validate this system as follows:

1. Simulate for $n = 3$ and $m = 3$ a random execution (choose a random seed value such that the system runs a significant number of steps before termination).
2. Formulate for $n = 3$ the following properties:
   a) The system eventually terminates.
   b) The system does never terminate.
   c) Whenever the system terminates, it stays terminated forever.

   Please use sufficiently many parentheses to make the parsing of formulas unique (do e.g. not write `[]p->q` but write `([](p))->(q)` or write `[]((p)->(q))`); in particular always use parentheses for the bodies of temporal formulas (`[](x>0)` or `<>(x==y)`).

   Check these properties with Spin for $m = 1$. If a property is violated, visualize in the simulator the violating run. Analyze the results in detail and explain whether they indicate an error in the model or not.

   *Check the output of Spin carefully to determine whether an error has occurred during model checking (the message* `error:0` *may be even given, if the model checking has been prematurely aborted or not all of the state space has been explored). If the message "error: max search depth too small" appears, increase in the "Advanced Parameter Settings" the parameter "Maximum Search Depth". If the message "pan: reached -DMEMLIM bound" appears, increase the parameter "physical memory available".*

Your next task is to extend the Promela model by the termination detection algorithm described below. The Promela model already contains all the necessary variable declarations; all you have to do is to extend the `do` loop by additional clauses of form

[1] Shmuel Safra's Version of Termination Detection
https://www.cs.utexas.edu/users/EWD/ewd09xx/EWD998.PDF

```
:: atomic { guard -> stat; ...; stat; }
```

*You do not have to change any of the already given clauses.*

In detail the algorithm works as follows:

- Every process has a *color* which can be "black" or "white". Initially all processes are white; if a process receives a message, it becomes black.
- Process 0 uses a variable *termination* to recall the status of the termination detection. Initially its value is 0 (no termination detection is running); when Process 0 starts the detection, it sets its value to 1 (a termination detection is running). When the detection process is completed, Process 0 sets its value either back to 0 (termination detection has failed, another termination detection may be attempted later) or to 2 (termination detection has been successfully completed, i.e., termination has been detected).
- Process 0 may start at any time (if *termination* = 0), a termination detection by sending a "token" around the ring of processes; for this purpose, we assume additional $n$ channels $t_0, \ldots, t_{n-1}$ where process $i$ receives a token from channel $t_i$ and sends a token to channel $t_{(i+1) \bmod n}$. A token consists of two values: a "token color" and a "token counter". Process 0 starts the termination detection by sending a token with color "white" and counter 0 to Process 1; it also sets its own color to "white" (again) and *termination* to 1.
- If Process $i \neq 0$ is passive, it accepts a token from its predecessor in the ring and forwards a new token to its successor. The counter of the new token is the sum of the counter of the process and the counter of the received token. If the process is "black", the color of the new token is also black; otherwise, it is the color of the received token. In any case, the color of the process becomes "white" (again); its counter remains unchanged.
- If Process 0 receives a token from its predecessor, the token has traveled the whole ring and the termination detection attempt is completed. Now there are two possibilities:
  - If Process 0 is passive and its color is white and the color of the token is white and the sum of the process counter and of the counter of the received token is 0, then termination has been detected; in that case, Process 0 sets *termination* to 2.
  - Otherwise, termination has not been detected and Process 0 sets *termination* to 0.

To model this algorithm in Promela, three clauses are sufficient (one for Process 0 initiating the termination detection, one for Process $i \neq 0$ accepting and forwarding a token, one for Process 0 receiving the token and completing the detection attempt (successfully or unsuccessfully).

By above algorithm, the system blocks (no transition is enabled any more) if and only if the system has terminated and *termination* = 2. After a system has terminated, typically two detection rounds have to be performed to detect termination: in the first round the passing of the token colors all black processes white (but the token itself becomes black); in the second round, the token remains white (which indicates that no more message was sent since the last round). When this token is received by Process 0 (which also indicates that all other processes have been passive), it carries as its counter the sum of the counters of all other processes; if sum of this counter and the counter of Process 0 is 0, there are no more messages in the network. If Process 0 itself is passive and white, we may thus conclude termination.

Your final task is to validate and verify this algorithm as follows:

1. Simulate for $n = 3$ and $m = 3$ a random execution of the system (choose the random seed value such that the system runs a significant number of steps before termination) and check whether the algorithm behaves as expected and the system terminates in a state with *termination* = 2.
2. Formulate for $n = 3$ the following properties:
   a) Always, if termination has been detected, the system is indeed terminated.
   b) Whenever the system terminates, this will be eventually detected.
   c) If a termination detection is started, it will be eventually (successfully or unsuccessfully) completed.

   In all these formulations you only need refer to the global variables already declared in the given Promela model (not to the local variables).

   Check these properties with Spin for some (the largest possible) $m \geq 1$. If a property is violated, visualize in the simulator the violating run. Analyze the results in detail and explain whether they indicate an error in the model or not.

   *Again use sufficiently many parentheses in your formulas and check the output of Spin carefully to determine whether an error has occurred.*

The deliverables of the exercise consist of

- The completed Promela model.
- Screenshots of (the final parts of) the simulation runs.
- The LTL properties.
- The output of Spin for each model check of a property.
- Screenshots of counterexample simulations (if any).
- For each model check, an interpretation (did the requested property hold or not and why)?

**Alternative:** Alternatively to the assignment given above, you may also use RISCAL to write a `shared system` model (in the style given in the lecture), take file `DijkstraSafra2.txt` as a starting point (it already contains 5 transitions, 3 more are needed). Run the system to elaborate its state space and make sure that the execution does not crash (show the output of the execution). Perform the checks, analyze the results, and interpret them as described above. If some fairness constraint is required, it shall not be encoded in LTL but by annotating the corresponding transition with a suitable `fairness` clause and declaring the LTL formula as `ltl[fairness]`.

Some hints/reminders on Promela are given below:

- The Promela version of `if (E) C1 else C2` is

  ```
  if
  :: E -> C1
  :: else -> C2
  fi
  ```

  The Promela version of `if (E) C` is

  ```
  if
  :: E -> C
  :: else -> skip
  fi
  ```

  The Promela version of `while (E) C` is

  ```
  do
  :: E -> C
  :: else -> break
  od
  ```

  In all cases, if you forget the `else` branch, the system will *block* in a state that is not allowed by all the conditions in the other branches.

- The expression *c* `?` `[` *M* `]` is true if and only if a channel *c* holds a message of type *M*. The statement *c* `?` *M* will then remove the message. A typical application is in

  ```
  do/if
  :: cond && c ? [ M ] ->
     c ? M;
     ...
  :: ...
  od/fi;
  ```

  where in a certain situation only a certain kind of message may be accepted.

- In the attached Promela model, the processes receive identifiers 1,2,3,. . . i.e.

  ```
  p[1]@label
  ```

  indicates that `p(0)` is at the position indicated by `label` (see also the simulations for the identifiers of the individual processes).