Karlsruher Institut für Technologie
Institut für Theoretische Informatik
Prof. Dr. B. Beckert
Dr. Mattias Ulbrich

Formale Systeme II: Anwendung, SS 2019

Model Checking with Spin

The course homepage `https://formal.iti.kit.edu/teaching/FormSys2SoSe2019/` *lists all relevant resources.*

Assignment 1

Find the sources for this assignment in file `21.pml` *on the course homepage.*

(a) Do a random simulation of the following PROMELA Programm. What are the sources of nondeterminism in it? Do an interactive simulation in the webfrontend and become a "Winner".

```
int a = 0;
active proctype P() {
    do
        :: a < 21 -> a = a + 2
        :: a < 21 -> a = a + 5
        :: a > 21 -> printf("Loser.\n"); break
        :: true -> a = 0
        :: a == 21 -> printf("Winner.\n"); break
    od
}
```

(b) Now look at the LTL properties `prop1` to `prop3` at the end of the file. Which of them are valid[1]?

```
ltl prop1 { [] (a >= 0)  }
ltl prop2 { <> (a == 0) }
ltl prop3 { <> (a == 21) }
```

(c) Formulate LTL property `prop4` which encodes:
"If `a` reaches 21 at all, then `a` is always less than 21 before reaching it."

Assignment 2

Find the sources for this assignment in file `critical.pml` *on the course homepage.*

Two processes need to share an access-restricted resource. The relevant part accessing the resource is confined to a critical section. A flag indicates that the process has entered the critical section.

The following piece of code models this situation in PROMELA.

```
bool flag[2];    // initialized to 0
int critical;    // # of threads in CS, initialized to 0

active[2] proctype P() {
  // int _pid;      is an implicit variable holding the process' id

  // wait for other process to not be in CS
```

[1] and what does "valid mean"?

```
  do
  :: flag[1 - _pid] == 0 -> break;
  od;

  // since the other process is now not in CS, I can set my flag.
  flag[_pid] = 1;

  // enter CS
  critical ++;
  printf("%d␣is␣now␣in␣critical␣section.␣(%d)\n", _pid, critical);
  critical --;
  flag[_pid] = 0;
  // leave CS
}

// ltl { *** to do *** }
```

(a) Why is this problematic?

(b) Add an according LTL specification whose verification fails and thus exposes the problem.

(c) Adjust the promela code using an `atomic` block. Prove that mutual exclusion for the critical section is thus ensured. What *assumptions* about the system did you make by changing the model?

(d) Resolve the problem *without* the atomic block by implementing *Peterson's algorithm*. Prove that mutual exclusion for the critical section is ensured.

Assignment 3

Find the sources for this assignment in file `skeleton.pml` *on the course homepage.*

This sorting algorithm, developed for use on parallel processors, compares all odd-indexed list elements with their immediate successors in the list and, if a pair is in the wrong order (i.e., if the first is larger than the second) swaps the elements. The next step repeats this for even-indexed list elements (and their successors). The algorithms iterates between these two steps until the list is sorted.

On $\frac{n}{2}$ parallel processors that have random access to the array of n elements to be sorted, the processors all concurrently do a compareexchange operation with their neighbours, alternating between oddeven and evenodd pairings in each step. The algorithm has linear runtime as comparisons can be performed in parallel. The skeleton of a PROMELA implementation that uses shared memory for synchronisation is presented in the following. The driver code spawns $\frac{n}{2}$ processes.

```
#define N 5
#define M 5

byte array[N];
bit state[N];

init {

  // fill the array with random numbers between 0 and M

  // start the processes
  i = 1;
  do
  :: i < N -> run sort(i); i = i + 2;
  :: i == N -> break;
  od;
```

```
}

proctype sort(byte id) {
  byte i = 0;
  byte tmp;

  // while i < N do
  //   if i is even and state[id-1] is 0 then
  //      sort id-1 and id in array
  //      state[id-1] = 1
  //   if i is odd and state[id+1] is 1 then
  //      sort id and id+1 in array
  //      state[id+1] = 0
  //   i++

  // Local sortedness
  assert(array[id-1] <= array[id] && array[id] < array[id+1]);
}
```

(a) Implement the sorting algorithm following this skeleton code.

(b) Verify that the result of the algorithm is a sorted array. Can the proposed assertions be used for this purpose?

(c) *Challenge:* Verify that the result of thie algorithm is a permutation of the original array.

(d) Tweak your model to obtain higher numbers for N and M.