T–79.4301 **Spring 2007**
Parallel and Distributed Systems
Tutorial 1 – Solutions

1. a) Dekker's algorithm can be modelled in Promela as follows (the line numbers are not strictly part of the code)

```
 1  bool flag[2] = false;
 2  bool turn = 0;
 3
 4  active [2] proctype mutex()
 5  {
 6    /* i is my index, j is the other process */
 7    pid i = _pid; pid j = 1 - _pid;
 8    /* Infinite loop */
 9  again:
10    /* [noncritical section] */
11    flag[i] = true;
12    /* [trying section] */
13    do
14    :: flag[j] ->
15         if
16         :: turn == j ->
17                        flag[i] = false;
18                        (turn != j);
19                        flag[i] = true;
20         :: else -> skip
21         fi;
22    :: else -> break
23    od;
24
25    /* [critical section] */
26    turn = j;
27    flag[i] = false;
28    goto again;
29  }
```

The simple pseudo-code translates into Promela in a very straightforward way. Line 1 initialises all elements of the `flag` array (shared between all processes) to `false`. Basically, line 4 instructs the model checker Spin to analyse two concurrent processes executing the code enclosed in the curly braces. The model checker Spin numbers these processes starting from zero; an individual process can use the special variable `_pid` to find its numeric identifier (line 7). The variables `i` and `j` defined at line 7 are local to each running process.

In the rest of the code, the comments, variable assignments, and tests on variables are replaced with their Promela equivalents (Promela uses C-style comments, assignments, and operators). Unlike C programs, however, where every statement reached by the program control can be "executed immediately", Promela processes may *block* on certain statements of the code. This feature is used at line 18 of the Promela code to implement the idle loop in the pseudo-code: a process that reaches this line of the Promela code will not proceed until the condition on the variables becomes true. To avoid similar blocking in the `do`-loop and the `if`-selection, we have to explicitly specify what the process should do when the condition at line 14 or 16 is false. The `skip` statement at line 20 is a no-op; a `break` statement can be used to break out of the innermost `do`-loop (line 22).

1. b) To check whether Dekker's algorithm solves the mutual exclusion problem correctly, we add to the program a new variable `count` (shared between the processes) that gives the number of processes that are currently in their critical section. A process increments and decrements the value of this variable whenever entering and leaving the critical section, respectively. The requirement that only one process should be in the critical section at any one time is represented by the assertion at line 27.

```
 1  bool flag[2] = false;
 2  bool turn = 0;
 3  byte count = 0;
 4
 5  active [2] proctype mutex()
 6  {
 7      /* i is my index, j is the other process */
 8      pid i = _pid; pid j = 1 - _pid;
 9      /* Infinite loop */
10  again:
11      /* [noncritical section] */
12      flag[i] = true;
13      /* [trying section] */
14      do
```

```
    :: flag[j] ->
         if
         :: turn == j ->
                        flag[i] = false;
                        (turn != j);
                        flag[i] = true;
         :: else -> skip
         fi;
    :: else -> break
    od;

    count++;
    assert(count == 1);
    /* [critical section] */
    count--;

    turn = j;
    flag[i] = false;
    goto again;
}
```

1. c) Suppose that the Promela code from b) is in the file `mutex.pml` in the current directory. We can now use the model checker Spin to generate an *analyser* for this model by giving the command

```
$ spin -a mutex.pml
```

This command generates a C source code file (`pan.c`, short for "protocol analyser") along with some auxiliary files in the current directory. We then compile the file and run the analyser:

```
$ cc -o pan pan.c
$ ./pan
hint: this search is more efficient if pan.c is compiled -DSAFETY
(Spin Version 4.2.6 -- 27 October 2005)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +
State-vector 20 byte, depth reached 49, errors: 0
     166 states, stored
     156 states, matched
     322 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

2.622   memory usage (Mbyte)

unreached in proctype mutex
        line 30, state 22, "-end-"
        (1 of 22 states)
```

Intuitively, the analyser generated by Spin checks whether there exists a way to interleave the invidivual statements of the code between the two processes competing for the permission to enter the critical section such that the assertion added to the code is violated. Such an interleaving of the statements exists if and only if it is possible for the two processes to simultaneously enter their critical section. The report given by the analyser shows that no such interleaving exists. Therefore, we can conclude that the Promela model for Dekker's algorithm correctly solves the mutual exclusion problem for two processes.

1. d) To check whether the algorithm remains correct when the `do`-loop in the model is changed to an `if`-selection, we model the modified algorithm in Promela:

```
bool flag[2] = false;
bool turn = 0;
byte count = 0;

active [2] proctype mutex()
{
    /* i is my index, j is the other process */
    pid i = _pid; pid j = 1 - _pid;
    /* Infinite loop */
again:
    /* [noncritical section] */
    flag[i] = true;
    /* [trying section] */
    if
    :: flag[j] ->
         if
         :: turn == j ->
                        flag[i] = false;
                        (turn != j);
                        flag[i] = true;
         :: else -> skip
         fi;
    :: else -> skip
    fi;

    count++;
    assert(count == 1);
    /* [critical section] */
    count--;

    turn = j;
    flag[i] = false;
    goto again;
}
```

Repeating the analysis for this model (`mutex_2.pml`) gives the following result:

```
$ spin -a mutex_2.pml
$ cc -o pan pan.c
$ ./pan
hint: this search is more efficient if pan.c is compiled -DSAFETY
pan: assertion violated (count==1) (at depth 104)
pan: wrote mutex_2.pml.trail
(Spin Version 4.2.6 -- 27 October 2005)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +
State-vector 20 byte, depth reached 115, errors: 1
     162 states, stored
      81 states, matched
     243 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

2.622   memory usage (Mbyte)
```

In this case the assertion is violated. Indeed, both processes can enter the critical section at the same time in this case. A simple scenario in which this occurs proceeds as follows:

- Process 1 begins its execution and sets `flag[1]` to `true` (line 12). Because `flag[0]` was initialised to `false`, only the `else` branch of the outer `if`-selection is executable. Therefore, process 1 increments `count` to 1 (line 26) and proceeds to the critical section.
- While process 1 is still in its critical section, process 0 begins its execution. After setting `flag[0]` to `true` (line 12), it finds `flag[1]` `true` at line 15 and proceeds to the inner `if`-selection. Because process 1 has not left its critical section, however, the variable `turn` still has its initial value 0 at this point. Therefore also process 0 is allowed to enter its critical section, and the assertion at line 27 is violated.

(In practice, counterexamples such as these can be generated automatically by the analyser. These features of the tool will be demonstrated later during the course.)

1. e) The Promela model of Peterson's algorithm (augmented with an assertion to check its correctness) is as follows:

```
/* Peterson's mutex algorithm, two parallel processes 0 and 1 */
bool flag[2] = false;
bool turn = 0;
byte count = 0;

/* flag is initialized to all false, */
/* and turn has the initial value 0  */
```

```
active [2] proctype peterson()
{
    pid i = _pid; pid j = 1 - _pid;
    /* Infinite loop */
again:
    /* [noncritical section] */

    flag[i] = true;
    /* [trying section] */
    turn = i;
    (flag[j] == false || turn != i);

    count++;
    assert(count == 1);
    /* [critical section] */
    count--;

    flag[i] = false;
    goto again
}
```

We again analyse the model using the Spin model checker:

```
$ spin -a peterson.pml
$ cc -o pan pan.c
$ ./pan
hint: this search is more efficient if pan.c is compiled -DSAFETY
(Spin Version 4.2.6 -- 27 October 2005)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 20 byte, depth reached 22, errors: 0
      38 states, stored
      45 states, matched
      83 transitions (= stored+matched)
       0 atomic steps
```

```
hash conflicts: 0 (resolved)

2.622   memory usage (Mbyte)

unreached in proctype peterson
        line 27, state 9, "-end-"
        (1 of 9 states)
```

No errors are reported by the analyser. Therefore, the model of Peterson's algorithm correctly solves the mutual exclusion problem for two processes.