T–79.4301 **Autumn 2008**
Parallel and Distributed Systems
Tutorial 3 – Solutions

1. a) The `goto` statements in the model of the alternating bit protocol can be replaced, for example, with `do`-loops as follows:

```
mtype = { msg0, msg1, ack0, ack1 };

chan to_sndr = [2] of { mtype };
chan to_rcvr = [2] of { mtype };

active proctype Sender()
{
  do
    :: to_rcvr!msg1;
       to_sndr?ack1;
       to_rcvr!msg0;
       to_sndr?ack0
  od
}

active proctype Receiver()
{
  do
    :: to_rcvr?msg1;
       to_sndr!ack1;
       to_rcvr?msg0;
       to_sndr!ack0
  od
}
```

   b) To add data to the abstract messages sent by the Sender process to the Receiver process, we refine the message channel `to_rcvr` into a channel for transporting messages that consist of a "tag" of type `mtype` and the actual data (of type `byte`) associated with the message. (Because the receiver does not send any data back to the sender, the type of the `to_sndr` channel need not be modified.) Furthermore, we add the channels `indata` and `outdata` to model the interface via which the protocol communicates with its environment that actually generates and processes the data.

```
mtype = { msg0, msg1, ack0, ack1 };

chan to_sndr = [2] of { mtype };
chan to_rcvr = [2] of { mtype, byte };
chan indata = [0] of { byte };
chan outdata = [0] of { byte };

active proctype Sender()
{
  byte data;
  do
    :: indata?data;
       to_rcvr!msg1, data;
       to_sndr?ack1;
       indata?data;
       to_rcvr!msg0, data;
       to_sndr?ack0
  od
}

active proctype Receiver()
{
  byte data;
  do
    :: to_rcvr?msg1, data;
       to_sndr!ack1;
       outdata!data;
       to_rcvr?msg0, data;
       to_sndr!ack0;
       outdata!data
  od
}
```

c) Sequences of the requested form can be generated by the following process:

```
active proctype Source()
{
  do
    :: indata!0
    :: indata!1;
       do
         :: indata!2
       od
  od
}
```

d) The following process receives messages from the channel `outdata` and checks that every message received is either a 0 or a 1. After receiving a

1, the process enters an infinite loop and verifies that each subsequent message received is a 2. (Note that the `else` statement in the `if`-selection cannot be omitted; otherwise the process would block at the selection if the received data differed from 1.)

```
active proctype Sink()
{
  byte data;
  do
    :: outdata?data;
       assert(data == 0 || data == 1);
       if
         :: data == 1 ->
            do
              :: outdata?data;
                 assert(data == 2)
            od
         :: else -> skip
       fi
  od
}
```

e) Analyzing the model consisting of the processes defined in b), c) and d) reveals no errors:

```
$ spin -a solution-e.pml
$ cc -o pan pan.c
$ ./pan
hint: this search is more efficient if pan.c is compiled -DSAFETY

(Spin Version 5.1.3 -- 11 December 2007)
        + Partial Order Reduction

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

State-vector 48 byte, depth reached 122, errors: 0
      163 states, stored
      105 states, matched
      268 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

    2.501       memory usage (Mbyte)
```

```
unreached in proctype Sender
        line 19, state 10, "-end-"
        (1 of 10 states)
unreached in proctype Receiver
        line 32, state 10, "-end-"
        (1 of 10 states)
unreached in proctype Source
        line 43, state 10, "-end-"
        (1 of 10 states)
unreached in proctype Sink
        line 60, state 16, "-end-"
        (1 of 16 states)

pan: elapsed time 0.01 seconds
```

It is easy to see that the data transmission protocol is *data independent*, i.e., the behaviour of the processes Sender and Receiver does not depend on the actual data received via the channels `indata` and `to_rcvr`, respectively (both processes simply pass every data value received on to another channel without modifying it).

Suppose that the data transmission protocol could lose or duplicate a message such that the sequence of messages received by the Sink process differs from the sequence generated by the Source process. Because of data independence, we may assume that the protocol loses or duplicates a 1 that is generated by the data source process. But then the Sink process would receive a sequence of messages conforming to one of the regular expressions $(0)^*(2)^*$ or $(0)^*11(2)^*$; however, one of the assertions added to the Sink process would fail in such a case. Because the assertions never fail, it follows that the model of the data transmission protocol cannot lose or duplicate messages sent from the sender to the receiver.

f) To model the possibility of losing messages or acknowledgements sent between the sender and the receiver, we add two inline macros that handle the sending of messages and acknowledgements. The macros nondeterministically decide whether to send the message or quietly drop it. These macros are then used for communication instead of directly sending to the channels. The model (without the Source and Sink processes, which remain unchanged) is as follows:

```
mtype = { msg0, msg1, ack0, ack1 };

chan to_sndr = [2] of { mtype };
chan to_rcvr = [2] of { mtype, byte };
chan indata = [0] of { byte };
chan outdata = [0] of { byte };

inline send(m,d) {
  if
    :: skip -> to_rcvr!m,d
    :: skip
  fi
}

inline ack(a) {
  if
    :: skip -> to_sndr!a
    :: skip
  fi
}

active proctype Sender()
{
  byte data;
  do
    :: indata?data;
       send(msg1,data);
       to_sndr?ack1;
       indata?data;
       send(msg0,data);
       to_sndr?ack0
  od
}

active proctype Receiver()
{
  byte data;
  do
    :: to_rcvr?msg1, data;
       ack(ack1);
       outdata!data;
       to_rcvr?msg0, data;
       ack(ack0);
       outdata!data
  od
}
```

Note that the sending branches in the `send` and `ack` macros start with the `skip`-statement. This allows the execution of the branch even if

the queue is full. Without it, the other branch would be the only option in such a case. This could potentially lead to a situation where a real deadlock is not detected, because the system would be forced to drop every message if the receiving end did not read them. Now it is possible to execute the sending branch and then block on the actual sending command.

g) The model with message loss has an error:

```
$ spin -a solution-f.pml
$ cc -o pan pan.c
$ ./pan
hint: this search is more efficient if pan.c is compiled -DSAFETY
pan: invalid end state (at depth 80)
pan: wrote solution-f.pml.trail

(Spin Version 5.1.3 -- 11 December 2007)
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 48 byte, depth reached 82, errors: 1
       75 states, stored
       50 states, matched
      125 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

    2.501       memory usage (Mbyte)


pan: elapsed time 0.01 seconds
```

The deadlock is reached only after 80 steps, which makes it somewhat difficult to analyze what actually went wrong. The `-i` option can be given to `pan` to find the shortest path to the error, but for it to work accurately it needs to be compiled with the `-DREACH` option:

```
$ cc -DREACH -o pan pan.c
$ ./pan -i
hint: this search is more efficient if pan.c is compiled -DSAFETY
```

```
pan: invalid end state (at depth 80)
pan: wrote solution-f.pml.trail
pan: reducing search depth to 79
pan: wrote solution-f.pml.trail

... some output omitted ...

pan: reducing search depth to 9
pan: wrote solution-f.pml.trail
pan: reducing search depth to 1

(Spin Version 5.1.3 -- 11 December 2007)
        + Partial Order Reduction

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

State-vector 48 byte, depth reached 82, errors: 13
      117 states, stored
      121 states, matched
      238 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

    2.501       memory usage (Mbyte)

unreached in proctype Sender
        line 33, state 20, "-end-"
        (1 of 20 states)
unreached in proctype Receiver
        line 46, state 20, "-end-"
        (1 of 20 states)
unreached in proctype Source
        line 57, state 10, "-end-"
        (1 of 10 states)
unreached in proctype Sink
        line 69, state 5, "assert((data==2))"
        line 67, state 6, "outdata?data"
        line 74, state 16, "-end-"
        (3 of 16 states)

pan: elapsed time 0.03 seconds
pan: rate      3900 states/second
```

Now the execution path can be analyzed for example with:

```
$ spin -c -s -r -p -t solution-f.pml
Starting Sender with pid 0
proc 0 = Sender
Starting Receiver with pid 1
proc 1 = Receiver
Starting Source with pid 2
proc 2 = Source
Starting Sink with pid 3
proc 3 = Sink
q\p   0   1   2   3
  1   .   .   indata!0
  1:    proc  2 (Source) line  51 "solution-f.pml" (state 1)    [indata!0]
  1   indata?0
  2:    proc  0 (Sender) line  26 "solution-f.pml" (state 1)    [indata?data]
  3:    proc  0 (Sender) line  11 "solution-f.pml" (state 4)    [(1)]
spin: trail ends after 3 steps
-------------
final state:
-------------
#processes: 4
  3:    proc  3 (Sink) line  62 "solution-f.pml" (state 13)
  3:    proc  2 (Source) line  50 "solution-f.pml" (state 7)
  3:    proc  1 (Receiver) line  38 "solution-f.pml" (state 17)
  3:    proc  0 (Sender) line  28 "solution-f.pml" (state 8)
4 processes created
```

This prints out the sent and received messages in columns, the states that each process goes through, and the final state where all processes are waiting for something to happen. For more details about the options accepted by spin and the generated analyzer, see `http://spinroot.com`. The trace shows that the sender sends a message which is then lost, and because the receiver has not seen it, no acknowledgement is sent, and no process can continue.

h) The error in the model can be fixed by making Sender retransmit a message if a timeout occurs. (The `timeout` construct is a special Promela statement that becomes enabled if there is no other way for any of the processes in the model to proceed.) Similarly, the Receiver is made to resend the acknowledgment to the last message it received if it receives a message with an incorrect tag (corresponding to a situation in which the original acknowledgment was lost in transmission). The final model with all processes is as follows:

```
mtype = { msg0, msg1, ack0, ack1 };

chan to_sndr = [2] of { mtype };
chan to_rcvr = [2] of { mtype, byte };
chan indata = [0] of { byte };
chan outdata = [0] of { byte };

inline send(m,d) {
  if
    :: skip -> to_rcvr!m,d
    :: skip
  fi
}

inline ack(a) {
  if
    :: skip -> to_sndr!a
    :: skip
  fi
}

active proctype Sender()
{
  byte data;
  do
    :: indata?data;
       send(msg1,data);
       do
         :: to_sndr?ack1 -> break
         :: timeout -> send(msg1,data)
       od;
       indata?data;
       send(msg0,data);
       do
         :: to_sndr?ack0 -> break
         :: timeout -> send(msg0,data)
       od
  od
}

active proctype Receiver()
{
  byte data;
  do
    :: do
         :: to_rcvr?msg0, data -> ack(ack0)
         :: to_rcvr?msg1, data -> break
       od;
       ack(ack1);
```

```
        outdata!data;
        do
          :: to_rcvr?msg1, data -> ack(ack1)
          :: to_rcvr?msg0, data -> break
        od;
        ack(ack0);
        outdata!data
  od
}

active proctype Source()
{
  do
    :: indata!0
    :: indata!1;
       do
         :: indata!2
       od
  od
}

active proctype Sink()
{
  byte data;
  do
    :: outdata?data;
       assert(data == 0 || data == 1);
       if
         :: data == 1 ->
            do
              :: outdata?data;
                 assert(data == 2)
            od
         :: else -> skip
       fi
  od
}
```

The Spin-generated verifier now confirms that this model of the protocol works as expected. The model has no deadlocks, and, by the same argument as in step d), no message generated by the data source process is lost or duplicated in the sequence of messages "seen" by the data sink process.

```
$ spin -a solution-h.pml
$ cc -o pan pan.c
$ ./pan
```

```
hint: this search is more efficient if pan.c is compiled -DSAFETY

(Spin Version 5.1.3 -- 11 December 2007)
        + Partial Order Reduction

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

State-vector 48 byte, depth reached 156, errors: 0
      389 states, stored
      249 states, matched
      638 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

    2.501       memory usage (Mbyte)

unreached in proctype Sender
        line 39, state 42, "-end-"
        (1 of 42 states)
unreached in proctype Receiver
        line 58, state 42, "-end-"
        (1 of 42 states)
unreached in proctype Source
        line 69, state 10, "-end-"
        (1 of 10 states)
unreached in proctype Sink
        line 86, state 16, "-end-"
        (1 of 16 states)

pan: elapsed time 0.02 seconds
pan: rate     19450 states/second
```