T–79.4301 **Spring 2006**
Parallel and Distributed Systems
Tutorial 4 – 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) [Unfortunately, there was an error in the exercise: the assertions were supposed to be used to check that the sequence formed of the incoming messages received so far

can always be *extended* into a sequence which belongs to the language of the given regular expression (equivalently, that the sequence of received messages belongs to the language of the regular expression $\big((0)^*\big) \cup \big((0)^*1(2)^*\big)$). The following solution is based on this interpretation.]

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 /* data == 0 */
     fi
  od
}
```

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

```
$ spin -a d.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 48 byte, depth reached 116, errors: 0
     151 states, stored
```

```
     105 states, matched
     256 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

2.622   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 15, "-end-"
        (1 of 15 states)
```

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 acknowledgments sent between the sender and the receiver, we change the direct communication between Sender and Receiver into communication with an additional process that models the behaviour of an unreliable transmission channel. Whenever this new process receives a message from the sender or an acknowledgment from the receiver, the process chooses nondeterministically whether the message should be passed on to the other

process. The model (without the Source and Sink processes, which remain unchanged) is as follows:

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

chan msgchan = [0] of { mtype, byte };
chan ackchan = [0] of { mtype };
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 LossyChannel()
{
  mtype msg;
  byte data;
  do
  :: msgchan?msg, data -> to_rcvr!msg, data
  :: msgchan?msg, data  /* lose message */
  :: ackchan?msg -> to_sndr!msg
  :: ackchan?msg        /* lose acknowledgment */
  od
}

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

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

(Actually, the effect of "losing" messages could be modelled without adding a new process to the model by adding nondeterminism directly into the Sender and Receiver processes—in effect, making the processes choose nondeterministically whether to actually send anything to the other process. This is a common technique that can be used for optimising the number of states in the model to be verified.)

g) The model with message loss has an error:

```
$ spin -a 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 31)
pan: wrote f.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 68 byte, depth reached 33, errors: 1
      23 states, stored
       1 states, matched
      24 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

2.622   memory usage (Mbyte)
```

Analyzing the error trail with Spin gives the following example of a scenario of events that leads to a deadlock:

```
$ spin -c -t f.pml
proc 0 = LossyChannel
proc 1 = Sender
proc 2 = Receiver
proc 3 = Source
proc 4 = Sink
q\p   0   1   2   3   4
  1   .   .   .   indata!0
  [...]
```

```
  1   .   .   .   indata!0
  1   .   indata?0
  2   .   msgchan!msg1,0
  2   msgchan?msg1,0
spin: trail ends after 32 steps
[...]
```

In the final four steps of this execution, the Source process generates a 0 and sends it to the sender process. The sender process sends the 0 to the unreliable channel, in which the message is lost. The Sender then ends up waiting for an acknowledgment to a message that did not reach the receiver process (which is still waiting for a message from the sender).

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 msgchan = [0] of { mtype, byte };
chan ackchan = [0] of { mtype };
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 LossyChannel()
{
  mtype msg;
  byte data;
  do
  :: msgchan?msg, data -> to_rcvr!msg, data
  :: msgchan?msg, data  /* lose message */
  :: ackchan?msg -> to_sndr!msg
  :: ackchan?msg        /* lose acknowledgment */
  od
}

active proctype Sender()
{
```

```
  byte data;
  do
  :: indata?data;
     msgchan!msg1, data;
     do
     :: to_sndr?ack1 -> break
     :: timeout -> msgchan!msg1, data
     od;
     indata?data;
     msgchan!msg0, data;
     do
     :: to_sndr?ack0 -> break
     :: timeout -> msgchan!msg0, data
     od
  od
}

active proctype Receiver()
{
  byte data;
  do
  :: do
     :: to_rcvr?msg0, data -> ackchan!ack0
     :: to_rcvr?msg1, data -> break
     od;
     ackchan!ack1;
     outdata!data;
     do
     :: to_rcvr?msg0, data -> break
     :: to_rcvr?msg1, data -> ackchan!ack1
     od;
     ackchan!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
     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 h.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 68 byte, depth reached 120, errors: 0
     387 states, stored
     229 states, matched
     616 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

2.622   memory usage (Mbyte)

unreached in proctype LossyChannel
        line 20, state 10, "-end-"
        (1 of 10 states)
unreached in proctype Sender
```

```
        line 39, state 22, "-end-"
        (1 of 22 states)
unreached in proctype Receiver
        line 58, state 22, "-end-"
        (1 of 22 states)
unreached in proctype Source
        line 69, state 10, "-end-"
        (1 of 10 states)
unreached in proctype Sink
        line 86, state 15, "-end-"
        (1 of 15 states)
```