T–79.4301 **Spring 2006**

Parallel and Distributed Systems

Home Exercise 2 - Deadline: 31st of March 2006 at 12:15 (strict deadline!)

Please return your answer to the exercise before the deadline by e-mail to `Heikki.Tauriainen@tkk.fi` (include the course code T-79.4301 in the subject). Include in your answer: your name, student id, e-mail address, short answers to the exercise questions (plain ASCII text, PDF, or PostScript accepted). Return also the requested Promela models as additional e-mail attachments, clearly indicating at the beginning of each file (using Promela comments) the number of the question to which the file gives the answer. (We also accept ".tar", ".tar.gz", ".tar.bz2", or ".zip" archives if that is more convenient for you.)

The home exercises are personal, no group work allowed! There are three rounds of home exercises of 10 points each. To pass the course, you need to earn ≥ 15 points (out of a maximum of 30) from the three exercises. A score of ≥ 24 points raises a passing exam grade (≥ 1) by 1 (no effect to exam grades 0 or 5).

1. We are designing a communication protocol with a unidirectional ring topology and want to model the initial design with Promela. The system consists of three nodes numbered 0, 1 and 2, and denoted `node(0)`, `node(1)`, and `node(2)`, respectively.

   The only means of communication in the system between nodes is provided by a unidirectional ring of message channels. The node i can directly send messages only to its following neighbour node $i+1$ (modulo 3). Thus the node `node(i)` will send messages to a message channel `in_chan[(i+1) % 3]` which are read by the node `node((i+1) % 3)`. All channels have a capacity of 1 message, and the send mechanism is such that sending to a full channel will block inside the sender procedure and prohibit the sending node from proceeding. In addition, there is no direct capability to check whether a send would block as this is not implemented in the underlying hardware system.

   Each node has a sending user attached to it which wishes to send messages consisting of two fields: `(data,to_node)`, where `data` denotes that the packet to be sent is a data packet destined to the receiving user attached to the node `to_node`. The actual data sent is abstracted away in the Promela model of the protocol. The nodes must now relay packets to other nodes in order to implement a usable data transmission path between all the sending and receiving users attached to the nodes.

First create a Promela model of a ring protocol containing no flow-control mechanism to avoid deadlocking of the system due to all the message channels being full.

Find the deadlock of your system using Spin and give a log of the verification run. Please return also the final Promela model in your answer. (4 p.)

2. In the second phase we add a distributed flow-control mechanism to the protocol working as follows. A new protocol message `(hole,0)` called a *hole* is added to the system and initially the message channels are initialized with two holes (see Promela code of the `init proctype` below). They are used to count the amount of free capacity available in the three channels in a distributed fashion.

   If a node does not want to send anything and receives a hole message, it forwards the hole to the next node like any other message not destined to it. If the node wants to send a new message from the user attached to it, it must wait until it receives a hole message. When a hole is received, it is not forwarded but instead the data message is sent in its place. A new hole is generated only upon the delivery of a message to the receiving user at the message's destination node.

   Model this ring protocol with the distributed flow-control using holes as described above in Promela. Prove that your system is deadlock-free using Spin and give a log of the verification run. Please return also the final Promela model in your answer. (5 p.)

   Hint: It is easy to overflow the capacity of Spin with this part. Be very careful in your modelling decisions (use as few extra variables as possible, use `atomic` sequences, etc.).

3. Consider the following generic question: Suppose you have some unidirectional ring protocol which you have model checked to be deadlock-free for $N = 3$ nodes. (Any protocol, not necessarily the one discussed above.) Is it safe to conclude that the protocol is also deadlock free for $N = 4$ nodes? (1 p.)

```
/* Message types used by the protocol. Do not change. */
mtype = {data, hole};

chan in_chan[3] = [1] of { mtype, byte };
chan from_user[3] = [0] of { mtype, byte };
chan to_user[3] = [0] of { mtype, byte };

/* A process modelling the sending user. Do not change. */

proctype sending_user(byte id) {
    /* Send a message randomly to one of the nodes. */
    do
        :: from_user[id] ! data, 0
        :: from_user[id] ! data, 1
        :: from_user[id] ! data, 2
    od
}

/* A process modelling the receiving user. Do not change. */

proctype receiving_user(byte id) {
    byte message_to;
    mtype message_type;
    do
        :: to_user[id] ? message_type, message_to;
           atomic {
               assert((message_type == data) && (message_to == id));
               message_to = 0;
               message_type = 0;
           }
    od
}

/* A simple ring protocol in Promela */

proctype node(byte id) {
    /* Add your Promela model of the node here. */

    /* Following line added to make a syntactically valid Promela */
    /* model. Remove it first.                                    */
    skip
}

/* The init process for starting all other processes. */

init {
    int id;
    atomic {
        id = 0;
```

```
        do
            :: (id < 3) ->
                   run node(id);
                   run receiving_user(id);
                   run sending_user(id);
                   id++
            :: (id == 3) -> break
        od;

/* Uncomment the following #define to generate the initial holes in the */
/* ring in question 2.                                                  */

/* #define QUESTION2 */
#ifdef QUESTION2
        /* Send 2 holes to the channels 0,1. */
        id = 0;
        do
            :: (id < 2) -> in_chan[id] ! hole, 0; id++
            :: (id == 2) -> break
        od
#endif
    }
}
```