ALBERT-LUDWIGS-
UNIVERSITÄT FREIBURG

Network Protocol Design and Evaluation

Exercise 5

Stefan Rührup

University of Freiburg
Computer Networks and Telematics

Summer 2009

Task 1

Task 1 *Dijkstra's Semaphore*

Consider the Promela model for Dijkstra's Semaphore from the lecture.

1. Warm-up ***(your homework!)***

- Make yourself familiar with SPIN. Run a simulation and generate a sequence chart using XSPIN or spin with the command line parameters -c or -M.
- Build a verifier and compile it using the cc parameters -DNOREDUCE and -DNP
- Check for non-progress cycles.

The (binary) Semaphore

```
mtype {p,v}

chan sema = [0] of {mtype}

active proctype Semaphore() {
  do
  :: sema!p -> sema?v
  od
}
```

```
active [3] proctype user() {
  do
  :: sema?p;  /* enter critical section */
     skip;    /* critical section */
     sema!v;  /* leave critical section */
  od
}
```

semaphore.pml

Task 1.2

Task 1.2

Insert a correctness claim stating that at most one process can enter its critical section at any time. What do you use, assertions, meta-labels or a never-claim? Check the correctness with SPIN.

We use a never claim stating that it can never happen that

- user 1 and user 2 are at the same time in a critical section, or
- user 1 and user 3 are at the same ...
- user 2 and user 3 ...

Task 1.2

```
mtype {p,v}

chan sema = [0] of {mtype}

active proctype Semaphore() {
end:        do
            :: sema!p ->
progress:      sema?v
            od
}

active [3] proctype user() {
           do
           :: sema?p;  /* enter critical section */
critical:     skip;    /* critical section */
              sema!v;  /* leave critical section */
           od
}

never {
        do
        :: user[1]@critical && user[2]@critical -> break
        :: user[2]@critical && user[3]@critical -> break
        :: user[1]@critical && user[3]@critical -> break
        :: else
        od
}
```

Task 1.3

Task 1.3

Extend the semaphore such that an arbitrary fixed number of processes can enter their critical section at any time.

We use a counter to store the number of processes that are allowed to enter (number of permits)

Task 1.3

for a correctness check we can use an assertion

```
mtype {p,v}

chan sema = [0] of {mtype}

active proctype Semaphore() {
     byte count = 1;
end: do
     :: (count >= 1) -> sema!p;
                        assert(count >= 1);
                        count = count - 1;
     :: (count == 0) -> sema?v;
                        count = count + 1;
     od }

active [3] proctype user() {
           do
           :: sema?p;  /* enter critical section */
critical:     skip;    /* critical section */
              sema!v;  /* leave critical section */
           od }

never {
        do
        :: user[1]@critical && user[2]@critical -> break
        :: user[2]@critical && user[3]@critical -> break
        :: user[1]@critical && user[3]@critical -> break
        :: else
        od }
```

the never claim is still tailored to the binary (mutex) version!

Task 1.3

- The correctness property is specified by the assertion
  ```
  assert(count >= 1);
  ```
  when setting the semaphore

- When using an initial counter > 1, the old never claim is violated (it was tailored to the mutex variant)

```
>spin -a semaphore2.pml
>cc pan.c -o pan -DNOREDUCE
>./pan

pan: claim violated! (at depth 11)
pan: wrote semaphore2.pml.trail
```

- Otherwise SPIN reports no errors

Task 1.4

Task 1.4

Implement a semaphore without a semaphore process by using only a channel. Validate your model.

Idea: whenever a process enters its critical section, it posts a message on a channel and receives (removes) the message upon leaving the critical section. Other processes will be blocked as long as the channel is full.

Task 1.4

```
mtype {p}

chan sema = [1] of {mtype}


active [3] proctype user() {
           do
           :: sema!p;  /* enter critical section */
critical:     skip;    /* critical section */
              sema?p;  /* leave critical section */
           od
}

never {
        do
        :: user[1]@critical && user[2]@critical -> break
        :: user[2]@critical && user[3]@critical -> break
        :: user[1]@critical && user[3]@critical -> break
        :: else
        od
}
```

Task 2

Task 2

1. Change the Promela model of the lower layer such that messages might get lost.
2. Extend the communication protocol such that messages are retransmitted after they were lost. Simulate and verify your protocol.
3. Extend your protocol such that it also works when messages are delivered out of order (introduce sequence numbers). Extend the channel model. Simulate and verify your protocol.

Task 2.1

- To model message loss we can simply add a skip statement after receiving a message in the lower layer

```
proctype lower_layer(chan fromS, toS, fromR, toR)
{   byte d; bit b;

    do
    ::fromS?data(d,b) ->
        if
        ::toR!data(d,b)  /* correct */
        ::toR!error(0,0) /* distorted */
        ::skip           /* lost */
        fi
    ::fromR?ack(b) ->
        if
        ::toS!ack(b)
        ::toS!error(0)
        ::skip
        fi
    od
}
```

Task 2.2

- In the sender process, we retransmit a message after timeout:

```
proctype Sender(chan in, out)
{   byte mt;  /* message data */
    bit at;   /* alternation bit transmitted */
    bit ar;   /* alternation bit received */

    FETCH;              /* get a new message */
    out!data(mt,at);    /* send it */
    do
    ::in?ack(ar) -> /* await response */
        if
        ::(ar == at) -> /* correct send */
            FETCH;      /* get a new message */
            at=1-at     /* toggle bit */
        ::else ->       /* there was a send error */
            skip        /* don’t fetch */
        fi;
        out!data(mt,at)
    ::in?error(ar) ->   /* recv error */
        out!data(mt,at)
    ::timeout ->            /* no ack received */
        out!data(mt,at)     /* -> retransmit */
    od
}
```

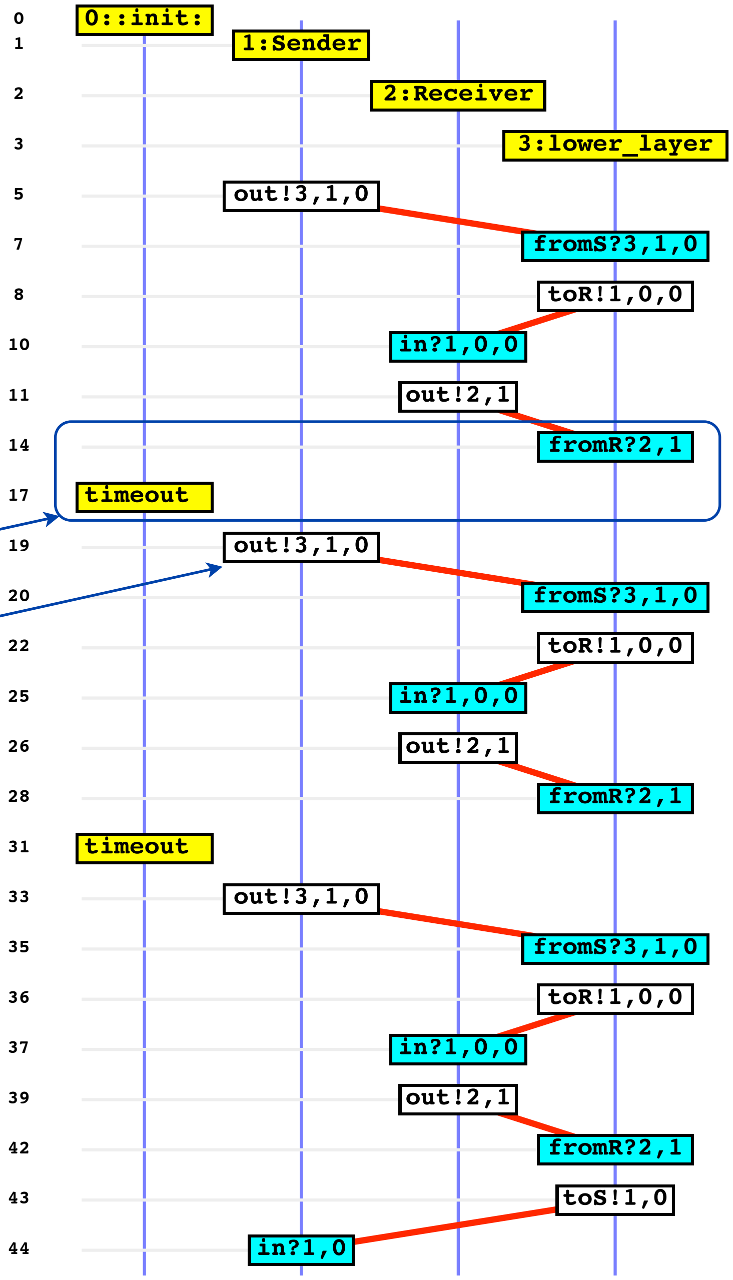
Task 2.2

Simulation trace

generated with
`spin -M -u45 alternating2.pml`

lost message

retransmission



Task 2.2

Validation: check for assertion violations

```
>spin -a alternating2.pml
>cc pan.c -o pan -DNOREDUCE -DSAFETY
>./pan -n

(Spin Version 5.1.7 -- 23 December 2008)

Full statespace search for:
     never claim              - (none specified)
     assertion violations     +
     cycle checks        - (disabled by -DSAFETY)
     invalid end states  +

State-vector 88 byte, depth reached 205, errors: 0
     2256 states, stored
     1323 states, matched
     3579 transitions (= stored+matched)
        2 atomic steps
hash conflicts:         2 (resolved)

    2.658 memory usage (Mbyte)


pan: elapsed time 0 seconds
>
```

Task 2.3

- Channel with reordering (changes to alternating.pml):

```
proctype lower_layer(chan fromS, toS, fromR, toR)
{   byte d; byte s;

    do
    ::fromS?data(d,s) ->
        if
        ::toR!data(d,s)   /* correct */
        ::skip            /* lost */
        ::fromS!data(d,s) /* reorder */
        fi
    ::fromR?ack(s) ->
        if
        ::toS!ack(s)
        ::skip
        ::fromR!ack(s)
        fi
    od
}
```

Task 2.3

- Retransmission after timeout: Promela's global timeout applies if no statement is executable, including receive statements, i.e. if all queues are empty. This maintains the order of the messages.

```
proctype Sender(chan in, out) {
    ...
    do
    ::in?ack(..) -> ...
    ::timeout -> out!data(mt,at)     <-- global timeout!
    od
}
```

- We model timer expiry by an unconditional retransmission. (Remember, that there is no timing in promela models)

```
proctype Sender(chan in, out) {
    ...
    do
    ::in?ack(..) -> ...
    ::true -> out!data(mt,at)
    od
}
```

Task 2.3

The sender process:

```
proctype Sender(chan in, out)
{   byte mt;    /* message data */
    byte st=1;  /* sequence number transmitted */
    byte sr;    /* sequence number received */

    FETCH;            /* get a new message */
    out!data(mt,st);  /* send it */
    do
    ::in?ack(sr) ->   /* await response */
        if
        ::(sr == st) ->          /* correct send */
            FETCH;               /* get a new message */
            st = (st+1)%MAXSN    /* increase sequence number */
        ::else ->                /* there was a send error */
            skip                 /* don’t fetch */
        fi;
        out!data(mt,st)
    ::true ->
        out!data(mt,st)
    od
}
```

Task 2.3

The receiver process:

```
proctype Receiver(chan in, out)
{   byte mr;          /* message data received */
    byte last_mr;     /* mr of last error-free msg */
    byte sr;          /* sequence number received */
    byte last_sr=0;   /* sr of last error-free msg */

    do
    ::in?data(mr,sr) ->
        out!ack(sr);
        if
        ::(sr != (last_sr+1)%MAXSN) ->
            skip
        ::(sr == (last_sr+1)%MAXSN) ->
            ACCEPT;
            last_sr=sr;
            last_mr=mr
        fi
    od
}
```

Task 2.3

Some definitions

```
#define N   4
#define MAX 16
#define MAXSN 16
#define FETCH   mt = (mt+1)%MAX
#define ACCEPT  printf("ACCEPT %d\n", mr); assert(mr==(last_mr+1)%MAX)

mtype = {data, ack}
```

... and the channel declarations:

```
    chan fromS = [N] of { byte, byte, byte };
    chan toR   = [N] of { byte, byte, byte };
    chan fromR = [N] of { byte, byte };
    chan toS   = [N] of { byte, byte };
```

(The rest remains the same)

Task 2.3

- Simulation gives no errors
- Unfortunately the model is too big to do a full state space search

```
>spin -a alternating3.pml
>cc pan.c -o pan -DSAFETY
>./pan -nE
error: max search depth too small
Depth=    9999 States=    1e+06 Transitions= 3.05e+06 Memory=
113.243   t=    2.3 R=   4e+05
Depth=    9999 States=    2e+06 Transitions= 6.22e+06 Memory=
223.887   t=   5.03 R=   4e+05
pan: resizing hashtable to -w21..  done
Depth=    9999 States=    3e+06 Transitions= 9.41e+06 Memory=
342.141   t=   7.88 R=   4e+05
...
```

- We have to restrict the number of resent messages after timeout. Furthermore we can reduce the channel capacity, the sequence number range, etc.