SPIN: Exercises on Message Channels*

Patrick Trentin
patrick.trentin@unitn.it
http://disi.unitn.it/trentin

Formal Methods Lab Class, March 17, 2017

Università degli Studi di Trento

*These slides are derived from those by Stefano Tonetta, Alberto Griggio, Silvia Tomasi, Thi Thieu Hoa Le, Alessandra Giordani, Patrick Trentin for FM lab 2005/16

Quiz #1

Q: are the following two pieces of code equivalent? (why?)

```
do
    :: if
        :: i == 0 -> printf("0");
        :: i == 1 -> printf("1");
       fi;
    :: else -> printf("not 0 1");
od;
```

```
do
    :: i == 0 -> printf("0");
    :: i == 1 -> printf("1");
    :: else -> printf("not 0 1");
od;
```

Quiz #2

Q: are the following two pieces of code equivalent? (why?)

```
do
    :: i <  10 -> v[i] = 0; i++;
    :: i <  10 -> v[i] = 1; i++;
    :: i >= 10 -> break;
od;
```

```
do
    :: i < 10 ->
        if
            :: v[i] = 0;
            :: v[i] = 1;
        fi;
        i++;
    :: else -> break;
od;
```

Quiz #3

Q: are the following two pieces of code equivalent? (why?)

```
do
    :: channel01?message(...);
    :: channel02?message(...);
od;
```

```
do
    :: true -> channel01?message(...);
    :: true -> channel02?message(...);
od;
```

Quiz #4

Q: what is the behaviour of the following program? (why?)

```
byte i;
do
    :: i < 10 ->
        i++;

    :: else ->
        break;

    assert(i != 5);
od;
```

Quiz #5

Q: what is the output of the following program? (why?)

```
chan c = [1] of { bit };

active proctype A ()
{
    bit i = 0;
    atomic {
        c!i ->
        printf("A: sent(%d)\n", i);
    };
    printf("A: waiting ...\n");
    atomic {
        c?i ->
        printf("A: recv(%d)\n", i);
    };
}
```

```
active proctype B ()
{
    bit i;
    atomic {
        c?i ->
        printf("B: recv(%d)\n", i);
    };
    i++;
    atomic {
        c!i ->
        printf("B: sent(%d)\n", i);
    };
}
```

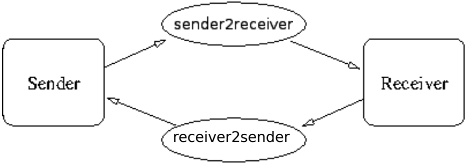
Contents

Reliable FIFO Communication

Goal: design a reliable FIFO communication over a non-reliable channel.

Alternating Bit Protocol:

- *Sender* and *Receiver* communicate over a couple of channels *sender2receiver* and *receiver2sender*
- the channels *sender2receiver* and *receiver2sender* are unreliable: messages might be lost or duplicated



Alternating Bit Protocol: Sender [1/2]

Sender specs:

- the *Sender* tags the messages with an alternating bit (e.g. it sends `(msg1, 0)`, `(msg2, 1)`, `(msg3, 0)`, ...).
- the *Sender* repeatedly sends a message with a tag value until it receives an acknowledgment from the *Receiver*.
- Suppose *Sender* has sent `(msg, out_bit)` and receives `in_bit` as acknowledgment:
  - if `in_bit` is equal to `out_bit`, then it means that *Receiver* has received the right message, so it sends a new message with a different value for `out_bit`.
  - otherwise it sends `(msg, out_bit)` again.
- the *Sender* attaches to each message a `sequence_number`, which is increased each time the tag value is changed.

Alternating Bit Protocol: Skeleton

```
mtype = { MESSAGE, ACK };

chan sender2receiver = [2] of { mtype, bit, int};
chan receiver2sender = [2] of { mtype, bit, int};

active proctype Sender () {
    ...
}

active proctype Receiver () {
    ...
}
```

Alternating Bit Protocol: Sender [2/2]

```
active proctype Sender () {
    bit in_bit, out_bit;
    int seq_no;

    do
        :: sender2receiver!MESSAGE(out_bit, seq_no) ->
            receiver2sender?ACK(in_bit, 0);
            if
                :: in_bit == out_bit ->
                    out_bit = 1 - out_bit;
                    seq_no++;
                :: else ->
                    skip
            fi
    od
}
```

Alternating Bit Protocol: Receiver [1/2]

Receiver specs:

- suppose *Receiver* receives `(msg, tag)`:
  - if `tag` is different from the last received bit, then it means that it is a new message;
  - otherwise, the message is old.
- When the *Receiver* receives a message, it sends the `tag` back to the *Sender* to communicate the correct message receipt.

Alternating Bit Protocol: Receiver [2/2]

```
active proctype Receiver () {
    bit in_bit, old_bit;
    int seq_no;

    do
        :: sender2receiver?MESSAGE(in_bit, seq_no) ->
            if
                :: in_bit != old_bit ->
                    printf("received: %d\n", seq_no);
                    old_bit = in_bit;
                :: else ->
                    skip
            fi
            receiver2sender!ACK(in_bit, 0);
    od
}
```

Alternating Bit Protocol: Unreliability

```
inline unreliable_send(channel, type, tag, seqno) {
    bool loss = false;
    bool duplicate = false;
    if
        :: channel!type(tag, seqno);
            if
                :: channel!type(tag, seqno); duplicate = true;
                :: skip;
            fi
        :: loss = true;
    fi;
}

// + modify Sender and Receiver to use this function
```

Q: what happens with the unreliable channel? (why?)

Alternating Bit Protocol: Unreliability

```
inline unreliable_send(channel, type, tag, seqno) {
    bool loss = false;
    bool duplicate = false;
    if
        :: channel!type(tag, seqno);
            if
                :: channel!type(tag, seqno); duplicate = true;
                :: skip;
            fi
        :: loss = true;
    fi;
}

// + modify Sender and Receiver to use this function
```

Q: what happens with the unreliable channel? (why?) deadlock, ...

Exercise 1: Reliable FIFO Communication

- configure *Sender* and *Receiver* to use `unreliable_send()`.
- fix the *Alternating Bit Protocol* so that there is no more **deadlock** and the input specification is still respected.

Contents

Process-FIFO

Goal: design a process `fifo(chan in, out)` that behaves like a FIFO.

- for simplicity (!), it uses an array of `bytes` for internal storage (of size `FIFO_SIZE`)
- the following **commands** can be received through the `in` channel:
  - `PUSH`: add `byte` to fifo, return `true` if successful
  - `POP`: remove and return oldest `byte` from fifo, returns `true` on success

    $\Longrightarrow$ push/pop failure: free choice among *blocking* and `false` return
  - `IS_EMPTY`: return `true` if empty, `false` otherwise
  - `IS_FULL`: return `true` if full, `false` otherwise
- messages through the `out` channel should be of type `RETURN` only
- *call simulation*: a process `sends` a command to the `fifo`, and **waits** for an answer
- `in`/`out` contain an `mtype` encoding the *command*, a `byte` encoding the *pushed/popped* value (if any), a `bit` encoding the `Boolean` outcome of a command request and a `byte` used as *UID* for the process that is using the *fifo*.

Exercise 2: Process-FIFO

- implement a process that behaves like a `fifo` (see previous slide)
- test the implementation by adding a pair of *producer* / *consumer* processes:
  - *producer*: infinitely adds some random `0..16` value to the `fifo`, if it is not full
  - *consumer*: infinitely pops a value from the `fifo`, if it is not empty

Disclaimer:

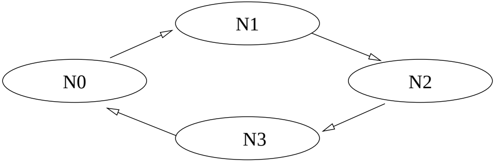
- next week you will be asked to *formally verify* the `fifo`
- some might **rightly** call bad design modeling an object with a process

  $\Longrightarrow$ still, it is a good exercise

Contents

Leader Election Problem

- N processes are the nodes of a unidirectional ring network: each process can send messages to its clockwise neighbor and receive messages from its counterclockwise neighbor.
- The requirement is that, eventually, **only one** process will output that it is the **leader**.
- We assume that every process has a **unique id**.
- The leader must have the **highest id**.



Le Lann, Chang, Roberts (LCR) solution

The algorithm:

- Initially, every process passes its identifier to its successor.
- When a process receives an identifier from its predecessor, then:
  - if it is greater than its own, it keeps passing on the identifier.
  - if it is smaller than its own, it discards the identifier.
  - if it is equal to its own identifier, it declares itself leader:
    - the leader communicates to its successor that now it is the leader.
    - after a process relayed the message with the leader id, it exits.

Complexity: at worst, n^2 messages.

Peterson/Dolev, Klawe, Rodeh solution

The algorithm:

- If a process is "active", it compares its identifier with the two counter-clockwise predecessors:
  - if the highest of the three is the counter-clock neighbor, the process proposes the neighbor as leader,
  - otherwise, it becomes a "relay".
- If the process is in "relay" mode, it keeps passing whatever incoming message.

Complexity: at worst, $n \cdot log(n)$ messages.

Exercise 3: Leader Election

```
mtype = { candidate, leader };
chan c[N] = [BUFSIZE] of { mtype, byte };

proctype node(chan prev, next; byte id) { ... }

init {
    byte proc, i;
    atomic {
        // TODO: set i random in [0,N]
        ...
        do
            :: proc < N ->
                run node(c[proc], c[(proc+1)%N], (N+i-proc)%N);
                proc++
            :: else ->
                break
        od
    }
}
```

- Implement a `leader` election algorithm of your choice.
- Verify that there is at most one leader.

$\longrightarrow$ strong solution hint!

Exercises Solutions

- will be uploaded on course website later this week
- send me an email if you need help or you just want to propose your own solution for a review

- learning programming languages requires practice: try to come up with your own solutions first!