Verificação Formal de Software, 2019/20
FCUP

TP MC-4– Linear-time Properties and Simple programs in Promela (II)

Note: for Promela and Spin consult `http://spinroot.com/spin/Man/index.html`

1. (**Alternating bit protocol**) Consider the standard Alternating bit protocol (`https://en.wikipedia.org/wiki/Alternating_bit_protocol`) implemented in Promela.

```
mtype = { msg0, msg1, ack0, ack1 };
chan sender = [1] of { mtype };
chan receiver = [1] of { mtype };

active proctype Sender()
{
  do
  ::
    if
    :: receiver?msg0;
    :: skip
    fi;
    do
    :: sender?ack0 -> break
    :: sender?ack1
    :: timeout ->
        if
       ::
 send0: receiver!msg0;
        :: skip
        fi;
    od;
 ::
    if
    :: receiver?msg1;
    :: skip
    fi;
    do
    :: sender?ack1 -> break
    :: sender?ack0
    :: timeout ->
        if
        ::
send1: receiver!msg1;
        :: skip
        fi;
    od;
  od;
```

```
}

active proctype Receiver()
{
  do
  ::
    do
    :: receiver?msg0 ->
        sender!ack0; break
    :: receiver?msg1 ->
        sender!ack1
    od;
    do
    :: receiver?msg1 ->
        sender!ack1; break
    :: receiver?msg0 ->
        sender!ack0
    od
  od
}
```

(a) Identify the meaning of each variable. Check the automata for each process and convince yourself about its behaviour.

(b) Perform some simulations to check if the model behaves as expected. Use random and interactive simulations. If necessary you can change the model.

(c) Test for non-progress cycles, and if found insert progress labels in some adequate points.

(d) Now you should verify the intended behaviour of the protocol. The sender should send a sequence of bits to the receiver. Once the sender has received an acknowledgment that the receiver has received the bit that was sent, it will send the complement of the previous bit to the receiver, and so on. Therefore, one desired sequence is: each `msg0` that is sent is (at some moment) followed by a `msg1`; another sequence is: each `msg1` that is sent is (at some moment) followed by a `msg0`. Write LTL formulas to check this behaviour. You can use the following macros

```
#define smsg0 (receiver?[msg0])
#define smsg1 (receiver?[msg1])
```

Note with [] that the value of the expressions are true or false.

Do the properties always hold? Why? Compare with the results in the previous question.

(e) Write a LTL formula to verify the following property: if messages with bit 0 are send infinitely often they are acknowledged infinitely often. Test and conclude if it holds or not. As above use macros as:

```
#define rmsg0 (sender?[ack0])
#define rmsg1 (sender?[ack1])
```

(f) More difficult (but important) properties to test are the following ones:

1. Each message sent by the sender process is eventually accepted by the receiver process

2. If a message with a bit 0 is sent, then a message with a bit 0 is received
3. If a message with a bit 1 is sent, then a message with a bit 1 is received.

For that one needs to express that propositions φ and ψ stricly alternates, which can the written as the following LTL formula

$$\mathrm{G}(\varphi \rightarrow (\ \varphi \mathrm{U}(\neg\varphi \wedge (\neg\varphi \mathrm{U}\psi))))$$

Choose appropriated values for φ and ψ (or variants) to specify the above properties in LTL. Test if the model verifies them. You may consider (weak) fairness constraints. The following definitions can be used:

```
#define srmsg0 Sender@send0
#define srmsg1 Sender@send1
#define send (srmsg0 || srmsg1)
```

(g) (*) You may want to consider other implementations of the protocol: using variables for the alternating bit (shorting the code); consider that no message is lost or add data to your message (e.g sequences of integer values). In all these cases you should test the above properties