DD2452 Formal Methods
Introductory PROMELA Exercises

About the Exercises

You are recommended to do at least the first two exercises to get acquainted to PROMELA and SPIN/XSPIN before you start working with the lab assignment. Doing these two exercises will introduce you to most of what you need of PROMELA and SPIN/XSPIN in the assignment.

The Sieve of Eratosthenes

To get familiar with the syntax of PROMELA you should type in the following two process types for calculating prime numbers. You could either do this directly in XSPIN or in your ordinary text editor and then load the file into XSPIN. If you enter it directly in XSPIN you do not have to save it to be able to run simulations etc. However, it might be wise to save it once or twice, anyhow. To load a file into XSPIN you select *Load..* from the *File..* menu. If you are reloading the same file as before into XSPIN then you can use *File>ReOpen* instead.

Once you have entered the PROMELA model into XSPIN you should do some simulations. Click on the *Run..>Set Simulation Parameters..* button and chose between the different simulation styles. You can either run a *random simulation* which is most common, do a *guided simulation* that uses a error trace produced in a validation, or an *interactive simulation* that lets you step through your model and decide which path to take when there are alternative execution paths. Do a simulation of calculating the primes less or equal to 10. What will be the contents of the *Simulation Output*, *Time Sequence*, *Data Values*, and *Message Sequent Chart* windows? What will happen in the *Message Sequent Chart* window if you point at one of the boxes in that window? What will happen at the same time in the SPIN *control window*?

Once you are done with the simulations you should validate that there are no dead-locks in the model. Click on the *Run..>Set Verification Parameters..* button and state that you would like to verify the *State Properties*. If you find errors then you can do a guided simulation, which reads the error trace and reproduces the execution that lead to the error. In case SPIN reports errors, i.e., invalid end states, that you find are incorrect, i.e., a process has reached an

end state that actually is valid, then you have to add end-labels to your model. After adding the end-labels to the *correct* end states, redo the validation and check the new results. Continue to iterate until you are satisfied with the results.

```
mtype = { nextValue };

proctype Generator (int max)
{
        chan next = [1] of {mtype, int};
        int value = 2;

        run Sieve (next);

        do
        :: (value <= max) ->
                next!nextValue(value);
                value = value + 1
        :: (value > max) ->
                break
        od
}

proctype Sieve (chan in)
{
        chan next = [1] of {mtype, int};
        int value, myPrime;

        in?nextValue(myPrime);
        run Sieve (next);

        do
        :: in?nextValue(value) ->
                if
                :: (value % myPrime) != 0 ->
                        next!nextValue(value)
                :: else ->
                        skip
                fi
        od
}

init
{
        run Generator (10)
}
```

The Alternating Bit Protocol

In this exercise you should verify that the *Alternating Bit Protocol* behaves as you might expect. (How?) A PROMELA model of the Alternating Bit Protocol can be found at the course web-page. You should verify two models of the protocol: one which models the possibility to lose messages in the underlying communication medium, and one which does not, i.e., assumes a perfect communication medium.

Start with the model that does not lose any messages. First, do some simulations to check that the model behaves as you expected. Once you have convinced yourself that it is a proper model of the protocol, you should continue by checking if the model can loop forever without doing anything useful. Do this by verifying that there are or are not any non-progress cycles in the model. Remember that you can do a guided simulation based on an error trace to be able to follow what happens. Add progress labels to the model where something "useful" is done. Redo the verification. Are there still any non-progress cycles? Should there be any non-progress cycles in this model?

Continue with the model that may lose messages. Add progress labels to that model (Where?) and check if this model may loop forever without doing anything useful. Explain the result.

Another way to check if a model behaves correctly is to verify that it will perform some desired sequences. For example, in the Alternating Bit 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 *msg(0)* that is sent is (at some moment) followed by a *msg(1)*; another sequence is: each *msg(1)* that is sent is (at some moment) followed by a *msg(0)*. If the protocol can reach a state where it does not produce these sequences then it has either dead-locked or gone into an infinite loop not producing anything useful. (Why?)

Return to the first model of the protocol and check if it will perform these sequences. (What do you think the answer will be? Why?) Start by developing an accurate LTL proposition. (Once again, if your model is not correct, *including the proposition*, then the result does not state anything about the original specification or system.) Enter your proposition by clicking on the *Run..>LTL Property Manager..* button in XSPIN and type in your proposition using *variables*, $[]$, $<>$, $\rightarrow$, *etc.* Then, hit the *Generate* button and your property will be transformed into a *never-claim* in PROMELA. You can click on the *Help* button to receive some useful hints.

The Symbol Definition box lets you input definitions for predicates appearing in the formula. Useful predicate definitions might be:

```
#define msg0 (to_rcvr?[msg(0)]) /* Is a msg(0) sent to the receiver? */
#define msg1 (to_rcvr?[msg(1)]) /* Is a msg(1) sent to the receiver? */
```

Run a verification of the model and check if it fulfills your required sequences. What can be said about these results compared with the ones from verifying

the non-progress cycles? Do they verify the same thing? Where the results equivalent?

Redo the verification for the model which may lose messages. Explain the results.

The Museum Turnstile System

You are going to design an automatic turnstile system for the local museum. The system should consist of two turnstiles, one for entry and another for exit. Fire regulations do not permit more than N persons to be in the museum at the same time; operations of the input turnstile should therefore be disabled if there are N persons in the museum at any one time (the exact value of N is not known at the moment, and is subject to change). In order to assist in the orderly closing of the museum, the system should have a *Closing* switch. Turning this switch *on* should light a "closing" sign and at the same time disable the operation of the input turnstile. After the last person has left the museum, (assume that persons are bound to leave the museum within a finite time) the system should disable the output turnstile and display a sign "closed".

Design a turnstile system to meet these requirements, and model your design in PROMELA to perform a formal validation against the requirements. To do this, classify the requirements into state properties and path properties.

Questions that might help you to develop the model: What different activities are there? Should these be modelled as process types, transitions, or message communications? Which of these activities affects the properties that you would like to verify? What should you do with those that do not affect the chosen properties?

Hints: The signs might be modelled as variables that are set to *on* and *off*. Do not give N too high a value; 5 - 10 would probably be enough. (Why?)