Weryfikacja wspomagana komputerowo

2016-2017

Lab 1 – Introduction to Promela and Spin

Vincent Penelle

Topics of this Lab

- Language Promela
- Spin and iSpin
- Modelisation of first programs
- Assert

The goal of this lab session is to introduce you to the SPIN model checker, along with its associated language, Promela. Promela is a language allowing to model the behaviour processes acting concurrently on a shared set of resources. Spin allows to run this language and to verify some properties, e.g. expressed in LTL, over these programs.

If you are interested, you can refer to the basic spin manual[1], or the book The SPIN model checker[2].

On your machine, you will have access to spin and a graphical interface called ispin. You can use the one you prefer. That said, the graphical interface is more convenient to have a global view of what you are doing, and this exercise session will suppose you are using it. Feel free to use spin directly from the terminal if you like, but in that case, refer yourself to the SPIN roadmap[3] to have the needed syntax.

Exercice 1: (Basic Promela Examples)

1. Run ispin and explore the interface. Focus on the first three tabs.
2. Open the file **step1-init.pml**[4] and observe the code. init is the process launched at the launch of the program (equivalent to main in C). Run the code in the tab Simulate/Replay, and do so in the Verification tab. Observe the results.
3. Open the file **step2-proctype.pml**[5] and observe the code. Here, init creates a child process H, which run concurrently to it. Execute the program in guided mode and observe in which order you can display the two messages. Add **active [3]** before the keyword **proctype**. What happens?

[1] `http://spinroot.com/spin/Man/Manual.html`
[2] `http://spinroot.com/spin/Doc/Book_extras/index.html`
[3] `http://spinroot.com/spin/Man/Roadmap.html`
[4] `./step1-init.pml`
[5] `./step2-proctype.pml`

4. Open the file **step3-params.pml**[6] and observe the code. The process H takes an argument. Run it in guided mode and observe which order you can obtain on the executions and what is the effect of the argument.

5. Open the file **step4-guards.pml**[7] and observe the code. The code is similar to the previous one, except that there is a global variable and two supplementary instruction manipulating it in H. Can you guess what it changes for the execution of the program? Run it in guided mode and test your claim.

6. Open the file **step5-if.pml**[8] and observe the code. This code introduces the instruction if – notice its syntax. Run it in guided mode, and observe the possible execution. What is the behaviour of the instruction if?

7. Open the file **step6-do.pml**[9] and observe the code. It introduces the instruction do. Run it in guided mode and describes its behaviour.

8. Compare the last program with **step6-do-niedet.pml**[10].

Summary:

Summary of elements seen so far:

- Promela is C-like in its basic syntax.
- Promela run processes in parallel. At the beginning of the program, the process **init** is launched.
- To call a proccess, use **run H(arguments);**
- To have already existing processes at the beginning of the program, use the keyword **active**, followed by the number of processes you want to run in [].
- You can use conditions as single instructions. This blocks the process until the statement is true.
- The **if** statement behave differently from C: if two conditions are true, the process can take any of them. You can as well put the **else** keyword in the if block, it will be taken only if all other conditions are false.
- The do statement behave similarly to the if statement, except that it repeats itself. The instruction **break** allows you to leave a do loop. Notice that the **else** keyword also exists for a do.
- Types are **bit , bool , byte , pid , short , int , unsigned**.
- You can (and should) use global variables.
- The instruction **goto** is also present (no example given above).

[6] `./step3-params.pml`
[7] `./step4-guards.pml`
[8] `./step5-if.pml`
[9] `./step6-do.pml`
[10] `./step6-do-niedet.pml`

- You can also declare arrays (like in C).

Exercice 2: (Channels)

One features of Promela is the existence of channel to pass messages between processes. A channel is basically a queue. Any process can write to a channel if it is not full and then put its message at the end of the channel. Any process can read in a channel if it is not empty and then take its first element (erasing it in the process). Like the guard instructions, if these instructions are not fireable when a process reads them, it is blocked until it can fire them.

1. Open the file **step7-chan.pml**[11] and observe its code, and namely the syntax of the channels. Run it in guided mode and observe the behaviour of the program.

2. Open the file **step8-chan2.pml**[12], observe its code and run it. What is the difference with the previous program? What means "<eval(i)>"? What happens if you remove the "< >"?

Summary:

Channels are used to stock and exchange messages between processes. Formally, they are queues. A process can put a value v in a channel ch with the instruction **ch!v**, and can extract a value from ch and put it into the variable i with the instruction **ch?i**. A process reading this instruction is blocked until it can perform it (i.e. it can actually extract or put something in it).

Exercice 3: (Some promela tricks)

1. Open the code **step9-atomic.pml**[13] and run it. What behaviour can you get?

2. Now, put the two instructions **printf** into a single instruction **atomic**$\{\cdots\}$. What happens?

Summary:

atomic is an instruction which forces a process to do a block of instructions in a row. That is useful to restrict the interleaving of processes in simulation. That can be crucial to verify some properties by restricting the interleaving to section you want to verify.

[11] `./step7-chan.pml`
[12] `./step8-chan2.pml`
[13] `./step9-atomic.pml`

Exercice 4: (Assert)

We now turn to a program a bit harder to understand by hand. Open **increment-par.pml**[14]. What do you think it does? We will now begin to use the verification tool to try to understand what happens without having to investigate by hand.

1. Add the instruction **assert(n<=20)**, after the instruction **finish==2;**, and run the verification tool under the verification tab.
2. Replace it with **assert(n>=10)**, and verify it.
3. Which one fail? Observe the trace making it fail.
4. Can you use the **atomic** to have both assertions satisfied?

Summary:

assert allows you to specify some properties you want to program to satisfy, and to have the verification tool tell you if it is satisfied or not, and if not, gives you a counter-example.

Exercice 5: (Mutual Exclusion and Peterson's Algorithm)

When making concurent programs, a crucial point is avoid undesirable side-effects between processes that could introduce undesirable behaviour, by having two processes acting at the same time on the same variable. For example, if a process A tries to read a variable x and add 1 to its value, if between the read and the write, a process B reads x and writes back the double of its value, we do not have an expected behaviour. To solve this, we should ensure that A and B cannot act on x while the other process is currently using it.

In the modelling of a program, we will identify so-called *critical sections* which are the part of the code in which a process access to the shared variables and must not be interfered with to ensure a correct behaviour. The *mutual exclusion problem* consist to determine if it is impossible for two processes to be simultaneously in their critical section.

1. The first algorithm ensuring that two properties has been given by Dekker in 1965[15]. Open the file Mutual-exclusion-1.pml[16] and observe the modelisation of this algorithm. Verify it using spin (add an *assert* testing a relevant condition).
2. This algorithm is complex. An idea to simplify it could be to use the following code[17]. Verify it. Does it work? How can you fix it?
3. A more modern solution is the *Peterson's algorithm* (1981). Here[18] is its promela implementation. Observe how simpler it is. Verify it (add an assert at the relevant place).
4. Propose (and test) a variant of the algorithm working for N concurrent processes.

[14] `./increment-par.pml`
[15] `http://en.wikipedia.org/wiki/Dekker's_algorithm`
[16] `./Mutual-exclusion-1.pml`
[17] `./Mutual-exclusion-2.pml`
[18] `./peterson.pml`

Exercice 6: (Goats and wolves)

We now ask you to conceive a little program in Promela, and to use the verification by Spin to know if it can yied a solution.

You have a river. On one side you have G goats and W wolves, who want to go to the other side. There is a boat in the river, that can welcome L animals at the same time (who can drive the boat, yes. You are not require to explain how they do that). The trouble is that if at any time in the boat or one of the banks there are strictly more wolves than goats, the predator nature of these otherwise nice animals will take over. And we don't want any casualty.

- Create a promela program modelling this problem. You are strongly encouraged to use several process, for example one to load animals to the boat, one to unload animals, one to move the boat, etc. And leave the non-determinism take care of the rest. You should as well use atomic to prevent too much strange and useless behaviours.
- Use assert conditions to test if there is a solution or not. Clue 1: With G=W=3 and L=2, there is a solution. Clue 2: using an assert condition which fails only if the animals can cross the river may be easier than the contrary.