Weryfikacja wspomagana komputerowo

2016-2017

Lab 2 – Mutual Exclusion Problem and LTL

Vincent Penelle

Topics of this Lab

- Mutual Exclusion Problem.
- Checking LTL properties over Promela programs.

Exercice 1: (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[1]. Open the file Mutual-exclusion-1.pml[2] 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[3]. Verify it. Does it work? How can you fix it?

3. A more modern solution is the *Peterson's algorithm* (1981). Here[4] is its promela implementation. Observe how simpler it is. Verify it (add an assert at the relevant place).

[1] `http://en.wikipedia.org/wiki/Dekker's_algorithm`
[2] `./Mutual-exclusion-1.pml`
[3] `./Mutual-exclusion-2.pml`
[4] `./peterson.pml`

Technical point:

We will now manipulate LTL formulæ to check more complex properties over programs. LTL formula can express a property to be checked over one run of the program. Spin is able to check if a formula is falsified on one run, and give it to you (of course, if there is a too long run falsifying the formula, Spin may not see it).

In a Promela program, you declare a LTL formula as follows:

ltl *name* {*formula*}

The syntax of LTL in Promela is:

- [] for "always"
- <> for "eventually"
- ! for negation
- U for strong until
- W for weak until
- V for the dual of V ("p V q" is equivalent to "!(!p U !q)")
- && or /\ for the and
- || or \/ for the or
- $->$ for the implication
- $<->$ for the equivalence.

The atoms you can use to form your formulæ include any test expressible in Promela. You can also use as an atom the position of a process. To do that you can put label in your code. For example, let us say that you have a process with pid P containing a label *crit*, the atom *P@crit* is true whenever the process with pid P is in the line labelled by *crit*.

To use this plainly, you may want to declare active processes at the start of the program, as for example, if you started two instances of a process *dummy*, *dummy[0]* is the pid of the first one and *dummy[1]* the one of the second.

You can check one LTL formula at a time in the verification tab of Spin.

Exercice 2: Mutual exclusion problem and LTL Take back the program about the Peterson algorithm. Write in LTL, and check with SPIN the following properties:

- the critical section contains at most one process,
- every process enters the critical section,
- if a process has its flag[_pid] to 1, it will eventually enter the critical section,
- if a process p is waiting to enter the critical section, the other one will not enter it before p has entered it.
- if a process p is waiting to enter the critical section, the other one will not enter it twice before p has entered it.

Do the same exercice with the perterson algorithm for N processes.

Exercice 3: Peterson algorithm for N processes Propose a variant of the algorithm working for N concurrent processes. Adapt as well the LTL formula of the last exercise and check them.

Exercice 4: Syracuse Problem An infamous problem that holds since centuries is known as the Syracuse problem. It considers a sequence of integer u such that for every term u_i of the sequence if u_i is even, $u_{i+1} = u_i/2$, and if it is odd, $u_{i+1} = 3 \times u_i + 1$.

Collatz has conjectured that whatever u_0 you choose, 1 will appear in the sequence. It is not yet proved, yet as far as people have checked (and that goes to very high numbers) it has not been falsified.

1. Create a Promela program that construct the Syracuse sequence from a number (that you give in the code). For example 100 (but you are free to test other ones).

2. State as LTL formula the following properties:

   - 1 will eventually be reached,
   - whenever a term is odd, the next is even,
   - whenever a term is even, the next is odd,

   and check them with SPIN.

Exercice 5: (Goats and wolves)

Let us now conceive a program to solve a more consequent problem.

G goats and W wolves travelling together come across the Vistula and want to cross it. There is no bridge, but a boat is on the shores. It 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 LTL formulæ to check the behaviour of your program and to test if there is a solution or not. Clue 1: With G=W=3 and L=2, there is a solution (but search for other). Clue 2: You want to check the existence of a successful path, but Spin is able to check if a formula is satisfied for all path. How could you do?