Weryfikacja wspomagana komputerowo

2016-2017

Lab 3 – LTL and never claim

Vincent Penelle

Topics of this Lab

- Checking LTL properties over Promela programs (bis).
- Making a full (small) program.
- Understanding the never claim.

Summary:

For convenience, we recall here the syntax of LTL in promela.

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.

Exercise 1: 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). Make it so you have one process taking care of the even case and one of the odd case. For example 100 (but you are free to test other ones, e.g. 15).

2. State as LTL formula the following properties:

   (a) 1 will eventually be reached,

   (b) whenever a term is odd, the next is even,

   (c) whenever a term is even, the next is odd,

   (d) whenever a term is odd, an even term will be reached,

   (e) there will never be a term 10 times bigger than the initial term.

   and check them with SPIN for several values of the initial term.

3. Add a counter counting the number of steps before reaching 1 and stop your program when it reachs 1.

4. Check in LTL if the run takes more steps than the input value (find values for which it is false).

5. Bonus question, if you want to play (i.e., not now, later). Make a program running the Syracuse sequence until it reaches 1 for one integer after another (you run it for 2, then for 3, then for 4, etc.). Check the previous properties on it. You should check "iterative search for short trail" if you want to have a chance to get decent results. Note that Spin will not be good for that: this is a long linear program, and Spin is made for verifying concurrent systems...

Technical point:

An important feature in Promela is the concept of never claim. A never claim is a process that run in parallel with the other, with the constraint that it has to perform one step whenever any other process perform one step. If it reaches its end, the program stops and raise an error. If it cannot execute any step, the program backtracks to test other paths (without raising an error).

It is only used in the verification tab. Only one can be used at a time (by indicating its name like for the ltl formulæ).

It has two main uses:

- Restricting the space search,
- Raising an error if an undesirable state is reached.

For example, you can restrict the behaviour to executions in which the variable *step* is smaller than 10 with the following code:

```
never stepsmaller {
do
::  step <=10
od; }
```

You can check that a variable x is never bigger than 15 with the following code:

```
never xsmaller {
do
::  x>15 -> break;
od;
}
```

This code will raise an error if x is bigger than 15 at some point, but not restrict the execution space. You can mix both to obtain a never claim that restrict to executions in which step is smaller than 10 and check that in those case, x is never bigger than 15:

```
never xsmaller {
do
::  step <= 10 -> if
::  x>15 -> break;
fi;
od;
}
```

In never claims, you can use special labels starting with *accept*. If there is an execution passing infinitely often through an accept label, it raises an error.

For example, the following claim raises an error if there is a path for which x is always lower than 40:

```
never {
acceptblabla:  do
::  x<=40
::  else -> break;
od;
do
::  else
od;
}
```

Notice that in a never claim, you cannot use assignation or other instruction having direct effect on the values of the variables. Besides that, you can use every Promela instruction.

Never claims are the center of Spin verification. Actually, when you verify a LTL formula, Spin translates it into a never claim.

You can as well ask Spin to generate a never claim from a LTL formula, by using the command ≪spin -f 'formula'≫ in a terminal. This will generate a claim that will fail if and only if the formula is satisfied. For example, it is useful if you want an execution satisfying an existential formula.

Exercise 2: (Playing with never and Syracuse) Continue to work with your previous program.

1. Use Spin to generate never claims equivalent to the LTL formulæ you had, and test them.
2. Write a never claim that stop an execution stopping the first time that 1 appears (and use it to check how step have elapsed then).
3. Write a never claim exploring only the execution up to the step 20 and failing if x reaches 1 at some point. Find an input for which it does not fail (thus for which the process takes more than 20 step to reach 1).

Exercise 3: (Another (quite bad) example to play with never claims Download the code stepper.pml[1], and test it. Write the following properties in LTL and in never claims. Compare your results with the output of spin -f 'formula'

1. x is often not equal to 19 (equivalently, whenever x is equal to 19, in the future x is not equal to 19).
2. If $x == 1$, in the future $x == 9$.
3. Check the satisfaction of the following claim:
```
never {
accept:
do
::  x==19
::  else -> break
od
}
```
4. Change the initial value to 12, and inspiring from the previous case, write a never claim that raise an error if there is an execution which keeps $x < 15$.
5. Idem to check that $x > 4$ and $x < 15$.

Exercise 4: (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.

[1] `./stepper.pml`

- Use LTL formulæ to check the behaviour of your program and to test if there is a solution or not. Hint 1: With G=W=3 and L=2, there is a solution (but search for other). Hint 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?
- Use a never claim to find a solution to the program. Hint: you want a claim that fails when you have a solution.