Weryfikacja wspomagana komputerowo

2016-2017

Lab 6 – Hello NuXmv

Vincent Penelle

Topics of this Lab

- Discovering NuXmv
- Familiarising with the syntax
- Understanding the concept of declarative programming
- Starting verifying LTL formulæ

In this lab session, we are moving to a new tool: nuXmv. Contrary to Spin, this tool has no fancy interface, so you'll have to resign to use a command line to use it. However, it should not be that bad.

The idea of that tool is to describe a transition system. You have some variables which can take values in a finite set, and you describe the transition relation (possibly non-deterministic). nuXmv produces an abstraction of this transition system as a finite state machine (or automaton), and can check over it properties expressed in LTL or CTL. It can also check them in a bounded model checking fashion (we'll come to that in later lab sessions).

The manual, including an explanation of the syntax can be found here[1]. You can as well found there a quite complete tutorial whose this lab session has taken a lot of inspiration.

Exercise 1: (A first transition system)

1. Download this file[2], and watch it. It is composed of a module `main`, which contains two variables, state and request, that can take both two values. The model is thus composed of two formulæ: `INIT`, detailing what is true at the beginning, and `TRANS`, detailing what transition the system can make (if several configurations satisfy the formula, they all can be taken next).
2. Draw (on a paper) the automaton implied by this model.
3. Run nuXmv in interactive mode with the command `nuXmv -int first-example.smv`
4. Before doing anything, you have to ask nuXmv to build the model you gave him. To do so type `go`.

[1] `https://es-static.fbk.eu/tools/nuxmv/index.php?n=Documentation.Home`
[2] `./first-example.smv`

5. We will simulate the model, but first, you have to pick an initial state. This can be done with `pick_state -i`. The option `-i` stands for interactive. If you omit it, nuXmv will pick a state at random.

6. Type `simulate -v`. The option -v stands for verbose. If you omit it, it won't show you the simulation. Yet, you can see the last simulation with `show_traces`.

7. Type `pick_state` to reinitialise the computation, then type again `simulate -v`. Is the simulation different?

8. Do it again, but now with the option `-r` to simulate. Is the simulation different? (You can compare with `show_traces n` to see the simulation n). The option `t` stands for randomize and without it, the simulation is deterministic.

9. The command `goto_state` allow you to pick any state on any simulation to start a new simulation from it (whereas `pick_state` can only choose an initial state).

10. Finally, the command `reset` resets the whole program (if you want to use again the model, you will have to type `go` again). Quite useful if you modify the program and don't want to entirely quit it (which by the way is done with the command `quit`).

Exercise 2: (The same transition system) It can be quite tricky to write a transition system with a single formula. Hopefully, the nuXmv creators have thought to that as well.

1. Download this file[3] and watch it. It has again one module with the same variable. But now there is a bloc `ASSIGN` with one `init` and one `next` assignation for each variable (only one is authorised). The next assignments contain one case, which allow to separate several cases. They are evaluated in the order, so if several conditions are true, the first one is evaluated. Yet, it is possible to have non-determinacy by indicating several values between {}.

2. Check that this model is actually the same as previously.

Exercise 3: (Factorising writing) Some models can have very similar sub-parts, which can lead to very long models. Hopefully, it is possible to factor the similar parts with sub-modules.

1. Download this file[4], and watch it. It is composed of two modules. `counter_cell` contains one variable, takes one arguments, updates its variable value according to the previous one and its argument. It also compute at each step a value `carry_out` (this is done in the `DEFINE` section. That is a value which only depends on the current values). The module `main` is thus composed of three sub-modules `counter_cell`, and defines their relation. Notice that when you declare sub-modules, you can use the value they define, but you cannot modify them in the `ASSIGN` section (their transition is described in the sub-module).

2. What does this system is doing? Draw it on paper and check what you think with nuXmv.

[3] `./first-example-bis.smv`
[4] `./counter.smv`

Exercise 4: (LTL) As with Spin, nuXmv is able to check LTL formula over specification. You can find the (not unusual) syntax of LTL on the manual. In general you declare a LTL formula with a line of the form `LTLSPEC formula` (e.g `LTLSPEC G (i=0 -> X (i=1))`), and you can check it with nuXmv with the command `check_ltlspec`. If they are false, it will exhibit an execution violating them.

1. Take the program of the first exercise and write the following formulæ. Then check them.
   - Whenever a request is made, the state will become busy.
   - Whenever a request is made, the state is busy the next time step.
   - If the state is ready, then whenever a request is made, it becomes busy the next time step.
   - If there is no request, the state cannot become busy until a request is made.
   - Eventually, the state will be busy.
2. Check that these formulæ over the program of the second exercice *par acquis de conscience*.
3. On the program of the third exercise, write and check formulæ expressing that:
   - Eventually `bit2.carry_out` will be true.
   - Eventually `bit2.carry_out` will always be true.
   - Whenever `bit0.carry_out` is true, at the next time step `bit1.value` is true.
   - Whenever `bit1.carry_out` is true, at the next time step `bit2.value` changes of value (warning, quite tricky).

Exercise 5: (Writing your first own model)

Write a program that implements a counter as in exercise 3, but whenever n `carry_out` are on, it waits n steps before continuing incrementing the counter (after switching off the `carry_out` and updating the values).

Tool box: you may want to use the function `toint` which convert any data type to an integer. You can declare an integer variable that can take values between i and j with `name:i..j`. You are advised to make a submodule for checking if you add 1 or not.

Check the formulæ of the previous exercise on your new model. Did something change?

Exercise 6: (Processes (deprecated)) If we have time, we will quickly discuss of a feature which should disappear in a next version (announced since some years now though): processes. In nuXmv, it is not possible to have cycling dependency as it was the case in Spin. With processes, it used to be the case. Here[5] is an example of that behaviour, which should not be unfamiliar. Understand it and check it, but do not put to much time in this: this feature is not supposed to be used.

[5] `./semaphore.smv`