NUXMV: Exercises*

Patrick Trentin
patrick.trentin@unitn.it
http://disi.unitn.it/trentin

Formal Methods Lab Class, May 19, 2017

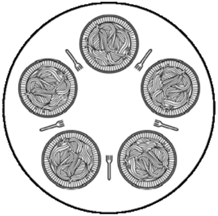
Università degli Studi di Trento

*These slides are derived from those by Stefano Tonetta, Alberto Griggio, Silvia Tomasi, Thi Thieu Hoa Le, Alessandra Giordani, Patrick Trentin for FM lab 2005/16

Contents

Exercise: Dining Philosophers [1/2]

Five philosophers sit around a circular table and spend their life alternatively thinking and eating. Each philosopher has a large plate of noodles and a fork on either side of the plate. The right fork of each philosopher is the left fork of his neighbor. Noodles are so slippery that **a philosopher needs two forks to eat it**. When a philosopher gets hungry, he tries to **pick up his left and right fork, one at a time**. If successful in acquiring two forks, he **eats for a while** (preventing both of his neighbors from eating), then **puts down the forks, and continues to think**.



Exercise: Dining Philosophers [2/2]

Exercise:

1. Implement in SMV a system that encodes the philosophers problem. Assume that when a philosopher gets hungry, he tries to pick up his left fork first and then the right one.
   Hint: you might consider an **altruist** philosopher, which can resign his fork in a deadlock situation.

2. Verify the correctness of the system, by specifiying and checking the following properties:
   - Never two neighboring philosophers eat at the same time.
   - No more than two philosophers can eat at the same time.
   - Somebody eats infinitely often.
   - If every philosopher holds his left fork, sooner or later somebody will get the opportunity to eat.

Contents

Exercise:

- encode the following code in NUXMV:

```
    void isort(arr) {
      // init: i = 1, j = 1;
l1:   while (i < 5) {
l2:     j = i;
l3:     while (j > 0 & array[j] < array[j-1]) {
l4:       swap(array[j], array[j-1]);
l5:       j--;
        }
l6:     i++;
      }
l7:   // done!
    }
```

- set `arr` equal to { `9, 7, 5, 3, 1` }
- **verify** the following properties:
  - the algorithm always terminates
  - eventually in the future, the array will be sorted forever
  - the algorithm is not done (`pc = l7`) until the array is sorted

Exercise: Insertion Sort [2/2]

Hints:

- use `pc` to keep track of the possible state values `{ l1, l2, l3, l4, l5, l6, l7 }`
- declare `i` in 1..5, initialize 1
- declare `j` in 0..4, initialize 1
- ensure that the content of 'arr' does never change when 'pc != l4'
- ensure that the content of 'arr' that is **not** involved in a 'swap' operation does not change even when 'pc = l4'
- *(easier?)* encode the constraints over `arr` with constrained-style modelling
- *(easier?)* encode the evolution of `pc`, `i` and `j` with assignment-style modelling

Contents

Exercise: Cleaning Robot [1/3]

Exercise: model a rechargeable cleaning **robot** which task is to move around a 10×10 room and clean it.

The robot state is so composed:

- variables "x" and "y", ranging from 0 to 9, which keeps track of the robot's position
- variable "state", with values in { MOVE, CHECK, CHARGE, CLEAN, OFF }, which keeps track of the next action taken by the robot
- variable "budget" in { 0..100 } which signals the remaining power
- output variable "pos", *defined* to be equal $y \cdot 10 + x$

At the beginning, the robot is in state "CHECK" and all other *vars* are 0.

The budget is decreased by a single unit each time the robot is in state "MOVE" or "CLEAN" (and $budget > 0$), and restored to 100 if the robot is in "CHARGE" state. Otherwise, the budget doesn't change.

Exercise: Cleaning Robot [2/3]

The robot changes state according to this **ordered** set of rules:

- if the robot is in "pos" 0 and the budget is smaller than 100, then the next state is "CHARGE"
- if the budget is 0, then the next state is "OFF"
- if the robot is in state "CHARGE" or "MOVE", then the next state is "CHECK"
- if the robot is in state "CHECK", then the next state is either "CLEAN" or "MOVE"
- otherwise, the next state is "MOVE".

Encode, using the **constraint-style** (**easier!**), the following constraints:

- if the state is different than "MOVE", then the position of the robot never changes.
- if the state is equal to "MOVE", then the robot moves by a single square in one of the cardinal directions: it increases or decreases either "x" or "y", but not both at the same time.

Exercise: Cleaning Robot [3/3]

Encode and verify the following properties:

- in all possible executions, the robot changes position infinitely many times (false)
- it's definitely the case that sooner or later the robot exhausts its budget, turns OFF and stops moving (false)
- it is never the case that the robot's action is either "MOVE" or "CLEAN" and the available budget is zero (false)
- if the robot charges infinitely often, then it changes position infinitely many times (true)
- there exists an execution in which the robot cleans every cell that it visits (true)
- if the robot is in "pos" 0, then it is necessarily always the case that in the future it will occupy a different position (true)
- the robot does not move along the diagonals (true)

Contents

Exercise: Elevator [1/5]

Exercise:

- Given the `model` of an elevator system for a **4-floors** building, including the complete description of:
  - reservation buttons
  - cabin
  - door
  - controller
- Enrich the model with **properties** encoding the **requirements** that must be met by each component of the system, and **verify** that such requirements are satisfied.

Exercise: Elevator - Button [2/5]

For each floor there is a **button** to request service, that can be pressed. A pressed button stays pressed unless reset by the controller. A button that is not pressed can become pressed nondeterministically.

Requirements:

- The controller must not reset a button that is not pressed.

Exercise: Elevator - Cabin [3/5]

The **cabin** can be at any **floor** between 1 and 4. It is equipped with an engine that has a **direction** of motion, that can be either `standing`, `up` or `down`. The engine can receive one of the following commands: `nop`, in which case it does not change status; `stop`, in which case it becomes standing; `up` (`down`), in which case it goes up (down).

Requirements:

- The cabin can receive a stop command only if the direction is up or down.
- The cabin can receive a move command only if the direction is standing.
- The cabin can move up only if the floor is not 4.
- The cabin can move down only if the floor is not 1.

Exercise: Elevator - Door [4/5]

The cabin is also equipped with a **door** (kept in a separate module in the SMV program), that can be either `open` or `closed`. The door can receive either `open`, `close` or `nop` commands from the controller, and it responds opening, closing, or preserving the current state.

Requirements:

- The door can receive an open command only if the door is closed.
- The door can receive a close command only if the door is open.

The **controller** takes in input (as sensory signals) the `floor` and the `direction` of motion of the cabin, the `status` of the door, and the `status` of the four buttons. It decides the controls to the engine, to the door and to the buttons.

Requirements:

- no button can reach a state where it remains pressed forever.
- no pressed button can be reset until the cabin stops at the corresponding floor and opens the door.
- a button must be reset as soon as the cabin stops at the corresponding floor with the door open.
- the cabin can move only when the door is closed.
- if no button is pressed, the controller must issue no commands and the cabin must be standing.

Contents

Exercise: Odd/Even Counter [1/2]

Implement a 5-bit counter that alternates counting all odd numbers from 31 to 1 *(e.g. 31, 29, 27, ..., 3, 1)* and counting all even numbers from 30 to 0 *(e.g. 30, 28, 26, 2, 0)*. Use a variable "**out**" to represent the output of the counter. Use five Boolean variables "**b0**", "**b1**", "**b2**", "**b3**", "**b4**" to represent the bits of the counter, from the least-significative to the most-significative ones. Initially, all bits are set to TRUE. The transition relation is described as follows:

- "**b0**" changes value only when all other bits are FALSE
- "**b1**" changes value at each transition
- "**b2**" changes value only when "**b1**" is FALSE
- "**b3**" changes value only when both "**b1**" and "**b2**" are FALSE
- "**b4**" changes value only when "**b1**", "**b2**" and "**b3**" are all FALSE

Exercise: Odd/Even Counter [2/2]

Model the 5-bit counter, express the following properties, and check with nuXmv that all properties are verified.

- it is necessarily always the case that, if out is 1, then at the next step the value of the counter is 30
- it is necessarily always the case that if out = 31 then in 5 iterations out will evaluate to 21
- it is always the case that b1 changes value at each iteration
- it is always the case that, if b1, b2 and b3 are all FALSE, then the next value of b4 is !b4
- infinitely often out is 0
- if out=30 then eventually in the future out=20

Contents

Exercise: Overflow Counter [1/3]

Implement a 3-bit counter which counts the number of times an input boolean variable "**bin**" changes value from `FALSE` to TRUE. Use three boolean variables "**b0**", "**b1**", "**b2**" to represent the bits of the counter, from the least-significant to the most-significant one. Use an output variable "**out**" to represent the value of the counter. Use a variable "**overflow**", with values in the set {`NO`, `YES`}, to keep track of a counter overflow event. Use a variable "**obin**" to keep track of the previous value of the input variable "**bin**", and an output variable "**rise**" to express the fact that "**bin**" changed value from `FALSE` to TRUE in the current step. Use an input boolean variable "**reset**" to reset the value of "**b0**", "**b1**", "**b2**" and "**obin**" to their initial value. Initially, "**b0**", "**b1**", "**b2**", "**bin**" and "**obin**" should be set to `FALSE`, while "**overflow**" should evaluate '`NO`'.

Exercise: Overflow Counter [2/3]

Implement, using the assign-syntax, the following transitions:

- "**obin**" is set to `FALSE` if "**reset**" is TRUE, and to "**bin**" otherwise
- "**b0**" is set to `FALSE` if "**reset**" is TRUE, it is set to "**!b0**" if "**rise**" is TRUE, and keeps its value otherwise
- "**b1**" is set to `FALSE` if "**reset**" is TRUE, it is set to "**!b1**" if "**rise & b0**" is TRUE, and keeps its value otherwise
- "**b2**" is set to `FALSE` if "**reset**" is TRUE, it is set to "**!b2**" if "**rise & b0 & b1**" is TRUE, and keeps its value otherwise
- "**overflow**" is set to `'NO'` if "**reset**" is TRUE, it is set to `'YES'` if "**rise & b0 & b1 & b2**" is TRUE, and keeps its value otherwise

Manually verify that the simulation works as intended.

Exercise: Overflow Counter [3/3]

Express the following properties, and have NUXMV verify that all properties are `FALSE`.

- CTL: it is necessarily always the case that infinitely often the counter is 0
- CTL: it is necessarily always the case that eventually the counter is always different than 0
- CTL: it is necessarily always the case that , if "**overflow**" is 'YES' in a given state then it also holds that "**overflow**" is 'YES' until "**reset**"
- CTL: it is necessarily always the case that when "**b0**", "**b1**" and "**b2**" are TRUE then from the next state eventually the value of counter will go back to 0
- LTL: if "**rise**" is TRUE infinitely often, then "**overflow**" is 'YES' infinitely often as well
- **Bonus Point**: explain why the latter formula is verified if CTL is used instead of LTL.

Exercises Solutions

- will be uploaded on course website within a couple of days
- send me an email if you need help or you just want to propose your own solution for a review

- learning programming languages requires practice: try to come up with your own solutions first!