# nuXmv: Exercises - Part B*

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

Formal Methods Lab Class, June 01, 2018

Università degli Studi di Trento

(compiled on 18/05/2018 at 10:20)

*These slides are derived from those by Stefano Tonetta, Alberto Griggio, Silvia Tomasi, Thi Thieu Hoa Le, Alessandra Giordani, Patrick Trentin for FM lab 2005/18

# Contents

# Exercise: Alarm System [1/3]

**Exercise:** Model a simple **alarm** system installed in the **safe** of a bank.

The **alarm** system can be activated and deactivated using a `pin`. After being activated, the **alarm** system enters a `waiting` period of 10 seconds, time that allows users to evacuate the **safe**, after which the **alarm** is `armed`. The **alarm** detects and `intrusion` when someone is inside the **safe** and the alarm is `armed`, after which it enters a `waiting` period of 5 seconds to allow the intruder to deactivate the alarm using the `pin`. If the alarm is not deactivated after an intrusion is detected, it will `fire` and remain `fired` until deactivation.

The alarm system is comprised by a `state` variable, with domain { `OFF`, `EVACUATE`, `ARMED`, `INTRUSION`, `FIRED` }, and a `s_clock` variable, with domain equal to 0..59. Initially, `state` is `OFF` and `s_clock` is 0.

The alarm system has two `boolean` inputs: `sensor` –`true` iff a person is detected inside the safe– and `use_pin` –`true` iff the pin is being used–. Express the fact that a person must be inside the safe to use the pin as an *invariant* of the inputs.

# Exercise: Alarm System [2/3]

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

- if the state is `OFF` and the pin is used, then the next state is `EVACUATE`
- if the pin is used, then the next state is `OFF`
- if the state is `EVACUATE` and the internal clock is 0, then the next state is `ARMED`
- if the state is `ARMED` and a person is detected in the safe, then the next state is `INTRUSION`
- if the state is `INTRUSION` and the internal clock is 0, then the next state is `FIRED`
- otherwise, the state does not change

The value of `s_clock` is set to 10 when the `state` value changes from `OFF` to `EVACUATE`, and it is set to 5 when the `state` value changes from `ARMED` to `INTRUSION`. Otherwise, its value is decreased by one unit at each transition until it reaches 0.

# Exercise: Alarm System [3/3]

Encode the following *LTL* properties, and verify with *NuSMV* that they are `true`:

- if the input pin is never used, then the alarm state is always `OFF`
- it is always true that, whenever an intrusion is detected then sooner or later the alarm state will be either `OFF` or `FIRED`
- it is always true that "if the alarm is armed in a certain state $s_k$, but the pin is never used starting from $s_k$ onward, then it is necessarily the case that either the sensor won't detect any intruder (starting from $s_k$ onward) or the alarm will eventually fire"
- if the state of the alarm is infinitely often equal to `EVACUATE`, then someone must enter the safe infinitely often

# Contents

# Exercise: Gnome Sort [1/2]

**Exercise:**

- Model the following code as a **module** in NuSMV or nuXmv:

```
    procedure gnomeSort(arr, len):
l0:     pos := 0
l1:     while (pos < len):
l2:         if (pos == 0 or arr[pos] >= arr[pos - 1]):
l3:             pos := pos + 1
            else:
l4:             swap(arr[pos], arr[pos - 1])
                pos := pos - 1
l5:     return                      # self-loop here!
    }
```

- Declare, inside the **main** module, the following variables:
  - `arr`, an array initialised to { `9, 7, 5, 3, 1` }
  - `sorter`, an instance of `gnomeSort(arr, 5)`
- **Verify** the following properties:
  - the algorithm always terminates
  - eventually in the future, the array will be sorted forever
  - eventually the array is sorted, and the algorithm is not done until the array is sorted

# Exercise: Gnome Sort [2/2]

**Hints:**

- use ‘`pc`’ to keep track of the possible state values { `10, 11, 12, 13, 14, 15` }
- declare ‘`pos`’ in 0..len, initialize 0
- 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 constraint-style modelling
- *(easier?)* encode the evolution of ‘`pc`’ and ‘`pos`’ with assignment-style modelling

# 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 non-deterministically.

**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.

# Exercise: Elevator - Controller [5/5]

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.

# 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!