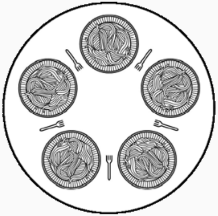
nuXmv exercises

Enrico Magnago

University of Trento,
Fondazione Bruno Kessler

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.

Exercise: Insertion Sort [1/2]

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 constraint-style modelling
- *(easier?)* encode the evolution of `pc`, `i` and `j` with assignment-style modelling

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, keep track of the robot's position;
- variable "state", with values in `MOVE, CHECK, CHARGE, CLEAN, OFF`, 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 $y \cdot 10 + x$.

Exercise: Cleaning Robot [2/5]

- 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 $budget > 0$)
- The budget is restored to 100 if the robot is in “CHARGE” state.
- Otherwise, the budget doesn’t change.

Exercise: Cleaning Robot [3/5]

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".

Exercise: Cleaning Robot [4/5]

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 [5/5]

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)

Exercise: Alarm System [1/4]

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 this amount of time the *alarm* is `armed`.
- The *alarm* detects an `intrusion` when someone is inside the *safe* and the alarm is `armed`.
- When an intruder is detected the *alarm* 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`. The *alarm* remains `fired` until deactivation.

Exercise: Alarm System [2/4]

The alarm system is comprised by:

- `state` variable, with domain { `OFF`, `EVACUATE`, `ARMED`, `INTRUSION`, `FIRED` };
- `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
- `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 [3/4]

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.

Encode the following *LTL* properties, and verify with nuXmv 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

Exercise: Gnome Sort [1/3]

Exercise
Model the following code as a **module** in SMV:

```
    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`: array initialised to { `9, 7, 5, 3, 1` }
- `sorter`: instance of `gnomeSort(arr, 5)`

Exercise: Gnome Sort [2/3]

Verify

- 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 [3/3]

Hints

- use `pc` to keep track of the possible state values `{ l0, l1, l2, l3, l4, l5 } ;`
- declare `pos` in 0..len, initialize to 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.

Exercise: Elevator [1/7]

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 [2/7]

Button

- 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.

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

Cabin

- The cabin can be at any floor between 1 and 4.
- The cabin 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).

Exercise: Elevator - Cabin [4/7]

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.

Door

- The cabin is equipped with a door, that can be either `open` or `closed`.
- The door can receive either `open`, `close` or `nop` commands from the controller, and it responds by 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 [6/7]

Controller

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

Exercise: Elevator - Controller [7/7]

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.

Exercise: Needham-Schroeder Protocol [1/5]

Exercise
Consider the following, simplified, public-key **Needham-Schroeder** protocol:

- **A** initiates the protocol by sending a nonce N_A and its identity I_A (both encrypted with *B*'s public key) to *B*.
- **B** deciphers the message and retrieves *A*'s identity, using its private key.
- **B** sends his nonce N_B and *A*'s nonce N_A (both encrypted with *A*'s public key) back to *A*.
- **A** decodes the message and checks that its nonce is returned, using its private key.
- **A** returns *B*'s nonce N_B (encrypted with *B*'s public key) back to *B*.
- **B** decodes the message and checks that its nonce is returned, using its private key.

Exercise: Needham-Schroeder Protocol [2/5]

In this protocol, the sequence of messages being exchanged is:

- $A \Longrightarrow B : \{N_A, I_A\}_{K_B}$
- $B \Longrightarrow A : \{N_A, N_B\}_{K_A}$
- $A \Longrightarrow B : \{N_B\}_{K_B}$

Exercise: Needham-Schroeder Protocol [3/5]

A known **man-in-the-middle attack** exists for this protocol:

- $A \Longrightarrow E : \{N_A, I_A\}_{K_E}$ (**A** wants to talk with **E**);
- $E \Longrightarrow B : \{N_A, I_A\}_{K_B}$ (**E** wants to convince **B** that it is **A**);
- $B \Longrightarrow E : \{N_A, N_B\}_{K_A}$ (**B** returns nonces encrypted by K_A);
- $E \Longrightarrow A : \{N_A, N_B\}_{K_A}$ (**E** forwards the encrypted message to **A**);
- $A \Longrightarrow E : \{N_B\}_{K_E}$ (**A** confirms it is talking to **E**);
- $E \Longrightarrow B : \{N_B\}_{K_B}$ (**E** returns **B**'s nonce back).

To prevent this attack, the original protocol was patched as follows:

- $A \Longrightarrow B : \{N_A, I_A\}_{K_B}$;
- $B \Longrightarrow A : \{N_A, N_B, I_B\}_{K_A}$ (**B** also sends its identity back to **A**);
- $A \Longrightarrow B : \{N_B\}_{K_B}$.

Exercise: Needham-Schroeder Protocol [4/5]

Goals [1/2]

- Model an instance of the *Needham-Schroeder* protocol in which *Alice* initiates communication with *Bob* and the protocol is successfully completed.
- Write a CTL property s.t. its counterexample is an execution trace which witnesses this successful attempt.
- Extend the previous model with the addition of a malicious user, namely *Eve*, which implements a modified version of the protocol so as to perform the *man-in-the-middle attack*.
- Write a CTL property s.t. its counterexample is an execution trace which witnesses this successful attack.

Exercise: Needham-Schroeder Protocol [5/5]

Goals [2/2]

- Extend the previous model with the suggested patch for the *Needham-Schroeder* protocol.
- Write a CTL property which verifies that the *man-in-the-middle attack* can no longer be successfully performed, plus an additional CTL property s.t. its counterexample is a failed attack attempt.