nuXmv: Exercises - Part A*

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

Formal Methods Lab Class, May 25, 2018

Università degli Studi di Trento

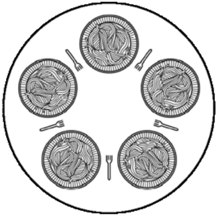
(compiled on 18/05/2018 at 10:18)

*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: 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: 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

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)

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!

Contents

Optional Exercise: Needham-Schroeder Protocol [1/3]

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

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

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

Optional Exercise: Needham-Schroeder Protocol [2/3]

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**)
- $A \Longrightarrow B : \{N_B\}_{K_B}$

Optional Exercise: Needham-Schroeder Protocol [3/3]

Goals:

- 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.
- 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.