Spin: Exercises - Part A*

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

Formal Methods Lab Class, April 11, 2018

UNIVERSITÀ DEGLI STUDI DI TRENTO

(compiled on 17/05/2018 at 12:52)

*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

Exercise 1: mutual exclusion [1/2]

Exercise: A solution to **mutual exclusion** for **N processes** is based on message passing instead of shared variables.

Idea: use a shared **buffered** message channel and synchronize by reading and writing from/onto this channel.

- the only shared global data structure can be a channel
- check with **ltl** that following properties hold for 3 processes:
  - mutual exclusion
  - progress
  - lockout-freedom

- **Q:** why is the fairness condition necessary for the **lockout-freedom** property to hold?
- **Q:** What changes if a **synchronous** channel is used instead? (why?)

Exercise 1: mutual exclusion [2/2]

Idea: replace the channel-based synchronization mechanism of **Exercise 1** with the famous **Test and Set** solution:

```
...
// enter critical section
do
  :: atomic {
     tmp = lock;
     lock = true;
     } ->
     if
        :: tmp;
        :: else -> break;
     fi;
od;
...
```

```
// global variable
bool lock = false
...
// exit critical section
lock = false;
...
```

Q: does the program still verify all the properties? (why?)

Exercise 2: factorial

Exercise: Model a process **factorial(n, c)** that **recursively** computes the factorial of a given value "n".

Hints & Tasks:

- use **channel** "c" to return the value to your parent process
- spawn the first factorial() process in the **init** block
- verify that fact(k) is greater than 2^k for $k > 3$. (e.g., try with $k = 10$)

Q:

- does the model always terminate, for any given value?
- if not, could you modify the solution so to be complete, whilst also performing all the computation in a recursive fashion? (why?)

Exercise 3: jumping array

Exercise: Model an array of **k** elements with **k-1** (random) memory locations initialized to 0 and **one** (random) location initialized to 1. Write an **algorithm** of your choice that searches the array for the memory location with value 1 and terminates only when it finds it. Each time that your algorithm reads **any** memory location, and before the next read, one of the following things must happen at random:

- the value 1 in location i jumps to location $(i+1)\%k$
- the value 1 in location i jumps to location $(i-1)\%k$
- the value 1 in location i does not move

Verify with **ltl** that the algorithm **always** terminates for **k=11**, use option "-mN" to control the **maximum depth** and "-i" for **breadth first** search.

- **Q:** is it possible to verify the correctness of your algorithm? why?
- **Q:** what is the most efficient algorithm (no cheats) for this problem?

Exercise 4: infinite monkey theorem

Exercise: Model a system of 26 *monkeys* and one human *reviewer*.

- Each **monkey** is given a button which, when pressed, sends a **unique** lower-case character (in the set a..z) to the reviewer. A monkey can press a button at any time, up until when the experiment is over.
- The **reviewer** checks the incoming sequence of characters, one by one, against a famous quote taken from the *Hamlet*: *"to be or not to be"* (spaces and punctuation marks are ignored). As soon as there is a match, the reviewer terminates the experiment.

Use a global, shared, channel `typewriter` to send characters from the *monkeys* to the human *reviewer*.
Write an *LTL* property s.t. the corresponding counter-example found by spin is an execution trace matching the sequence of characters `tobeornottobe`, and use *Spin* to find it.

Q: how can one guarantee correct termination for all processes?

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!