Concurrency, Parallelism and Distributed Systems (CPDS)
Module I: Concurrency
Facultat d'Informàtica de Barcelona
Final Exam
January 17, 2020

Answer the questions concisely and precisely
Answer each problem in a separate page (remember to put your name)
Closed-book exam. Duration: 2 hour

Exercise 1 (3 Points) LTS
The following 3 processes describe the actions that students undertake. Use the parallel compositionv operator to create a process `SYS` modelling the concurrent execution of these different processes.

- Modify the individual processes (using handshaking with shared action called `donework` and `doneplay`) so that it is not possible to produce silly action sequences, for example that allow a student to go to lectures undressed, or, that allow a student to perform PLAY actions before WORK actions have completed.

```
PLAY = ( pub -> PLAY | gig -> PLAY| clubbing -> PLAY).

WORK = ( lectures -> WORK | laborartory -> WORK
        | library -> WORK | assessments ->  WORK).

DAY = ( wake -> eat -> dress -> undress -> sleep -> DAY).
||STUDENT = (DAY || WORK || PLAY).
```

Hint. Complete the following `LTS`

```
PLAY = ( donework -> TOPLAY),
TOPLAY = ( pub -> TOPLAY | gig -> TOPLAY | clubbing -> TOPLAY
          | ... -> PLAY).

WORK = ( dress -> READY),
READY = (lectures -> READY| ... | ... | assessments -> READY
         | ...  -> WORK).

DAY = ( wake -> eat -> dress -> ... -> undress -> sleep -> DAY).

||STUDENT = (DAY || WORK || PLAY).
```

- Make a picture of the `LTS` corresponding to `||STUDENT`

Exercise 2 (2 Points) Safety

- Draw the LTS for the `SAFETY` process below.

```
property SAFETY
  = (a -> (b -> SAFETY | a -> SAFETY)  | b  -> a -> SAFETY).
```

- What trace violates the safety property?

Exercise 3 (3 Points) Recursive Locking in Java
Once a thread has acquired the lock on an object by executing a synchronized method, that method may itself call another synchronized method from the same object (directly or indirectly) without having to wait to acquire the lock again. The lock counts how many times it has been acquired by the same thread and does not allow another thread to access the object until there has been an equivalent number of releases. This locking strategy is sometimes termed recursive locking since it permits recursive synchronized methods. For example:

```
public synchronized void increment(int n) {
   if (n>0) {
      ++value;
       increment(n-1);
   } else return;
}
```

This is a rather unlikely recursive version of a method which increments value by `n`. If locking in Java was not recursive, it would cause a calling thread to block resulting in a deadlock.
Given the following declarations:

```
const N = 3
range P = 1..2  //thread identities range
C = 0..N  //counter range for lock
```

Model a Java recursive lock as the FSP process `RECURSIVE_LOCK` with the alphabet

`{acquire[p:P],release[p:P]}.`

The action `acquire[p]` acquires the lock for thread p.
`Hint` Complete the following `LST` shema:

```
const N = 3
range P = 1..2
range C = 0..N

RECURSIVE_LOCK   = (acquire[p:P] -> LOCKED[p][0]),
LOCKED[p:P][c:C] = (acquire[p]   -> LOCKED[p][c+1]
                   |when (c==0) ... -> ...
                   |...
                   ).
```

Exercise 5 (2 Points) Erlang
Given the module

```
-module(tut14).
-export([start/0, say_something/2]).

say_something(What, 0) -> done;
say_something(What, Times) ->
    io:format("~p~n", [What]),
    say_something(What, Times - 1).

start() ->
    spawn(tut14, say_something, [hello, 3]),
    spawn(tut14, say_something, [goodbye, 3]).
```

First, how do we compile such a module? Second, what is printed when we execute:

```
6> tut14:say_something(hello, 3).
```