Concurrency, Parallelism and Distributed Systems (CPDS)
Module I: Concurrency
Facultat d'Informàtica de Barcelona
Final Exam
April 19, 2017

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 2 *LIBRARY* (2 Points) Draw the `LTS` for the `LIBRARY FSP` process below.

```
LIBRARY = (take_book_to_desk -> give_details -> take_book->OPTIONS),
OPTIONS
   = ( renew -> OPTIONS | return_on_time -> LIBRARY | return_late -> FINE),
FINE = ( pay_fine -> LIBRARY | pay_fine -> never_go_back -> STOP).
```

Exercise 2 *Hot Drinks Machine* (5 Points) The composite process `HOT_DRINKS` models interactions between a hot drinks machine, a client and a replenishment worker. The drink machine allows the user to chose between coffee and tea unless one of the drinks is exhausted. If this is the case the exhausted product is not available. The machine has two storages that allows to store N coffee portions and N tea portions. The replenishment worker fills in the two storages from time to time. At the beginning —initial state— both storages are empty.

```
const N = some number

CLIENT = (coffee -> CLIENT | tea -> CLIENT).

WORKER = (fill ->WORKER).

MACHINE = STORAGE[0][0],
........
........
........

||HOT_DRINKS = (CLIENT||WORKER||MACHINE).
```

1. Complete the FSP description of `MACHINE`. Alphabet actions are `coffee`, `tea` and `fill`.
2. Draw the LTS of `HOT_DRINKS`. Here (and only here) you can assume `N` is 1.
3. Write a Java monitor class for `MACHINE`.

4. Write a safety property to check that traces of `HOT_DRINKS` never have a chain of consecutive `coffee` actions of size bigger than `N`. Simpler property definitions will have higher scores.

5. Consider the stressed system:

   ```
   ||STRESS_HOT_DRINKS = HOT_DRINKS << {coffee}
   ```

   Discuss the progress properties of this system. Has any action progress problems?

Exercise 3 *Good and Bad Workers* (3 Points). We ask to complete the following `goog_bad` module.

1. (1 Point) First we ask to complete the `bad_worked` to deal with `V1+V2`, `V1-V2`, `V1*V2`, `V1*V1`. In any other case shoudl print `"I can't work on this!~n"`.

   ```
   -module(good_bad).

   -export([good_worker/0,bad_worker/0]).

   bad_worker()->
       receive
          {add,V1,V2}->
              io:format("Result  is ~p ~n", [V1+V2]);
          {sub,V1,V2} ->
              io:format("Result  is ~p ~n", [V1-V2]);

          ...
   end.
   ```

2. (1 Point) Please explain why we get:

   ```
   25> c(good_bad).
   {ok,good_bad}
   26> P_Bad = spawn(good_bad,bad_worker,[]),
   26> P_Bad!{add,2,3}.
   Result is 5
   {add,2,3}
   27> P_Bad!{sub,3,2}.
   {sub,3,2}
   28>
   ```

   and not:

   ```
   25> c(good_bad).
   {ok,good_bad}
   26> P_Bad = spawn(good_bad,bad_worker,[]),
   26> P_Bad!{add,2,3}.
   Result is 5
   {add,2,3}
   27> P_Bad!{sub,3,2}.
   Result is 1
   {sub,3,2}
   28>
   ```

3. (1 Points) Desing a `good_worker` such that a behaviour like the following one is possible:

```
30> c(good_bad).
{ok,good_bad}
31> P_Good = spawn(good_bad,good_worker,[]).
<0.113.0>
32> P_Good!{add,3,2}.
Result is is 5
{add,3,2}
33> P_Good!{sub,3,2}.
Result is 1
{sub,3,2}
34> P_Good!{mul,3,2}.
Result is is 6
{mul,3,2}
35> P_Good!{square,4}.
Result is 16
{square,4}
36> P_Good!something_stupid.
I can't work on this but I will wait for something else!
something_stupid
37>
```