Concurrency, Parallelism and Distributed Systems (CPDS)
Module I: Concurrency
Facultat d'Informàtica de Barcelona
Final On-Line Exam
June 8, 2020

Solutions

Answer the questions concisely and precisely.
Duration: 2 hour
A solution of the Exam will be publish on the Racó on June 10, Wednesday
A preliminary grading will be publish on the Racó on June 15, Monday

Exercise 1 (3 Points) FSP & LTS
Let us consider a small vending system.

1. (1 Point) Design the `TWO_DRINKS` process working as follows:
   - Initailly the machine is empty and needs to be filled (execute the action `fill`) before deliver a drink)
   - The machine can deliver just two drinks before to be filled again (execute action `fill`).
   - The drinks delivered can be `coffee` or `tea`.
   - When the machine is empty, it is refilled again (the action `fill` is executed).
2. (1/2 Point) Define a WORKER just executing the action `fill` forever.
3. (1/2 Point) Define a CLIENT as follows: first it chooses between `coffee` and `tea`, later on it takes a `tea` for sure. Starts again. That is, choose between `coffee` and `tea`, later take `tea`, later choose between `coffee` and `tea`,...
4. (1 Point) Given the system `||HOT_DRINKS = (CLIENT||WORKER||TWO_DRINKS)` give discription of `HOT_DRINKS`, called `STRAIGHT`, just using prefixing and recursion (with no parallel composition).

Solution Exercise 1

```
const N = 2

// --- (1 Point )---
TWO_DRINKS = STORAGE[0],
STORAGE[i:0..N] = (when (i == 0) fill -> STORAGE[N]
                          |when(i > 0) tea -> STORAGE[i-1]
                          |when (i > 0) coffee -> STORAGE[i-1]
                          ).

// --- (1/2 Point )---
```

```
WORKER = (fill -> fill -> WORKER).

// --- (1/2 Point )---
CLIENT = (coffee ->  FIRST|tea -> FIRST),
FIRST = (tea -> CLIENT).


// --- (1 Point )---
STRAIT   = (fill -> OPTION),
OPTION = (coffe -> tea -> STRAIT | tea -> tea -> STRAIT).
```

Exercise 2 (2 Points) Stressed Systems.
Let us give high priority to `coffee` over `tea`.

1. (1 Point) First, given the system

   ```
   ||BETTER_COFFE_HOT_DRINKS = (CLIENT||WORKER||TWO_DRINKS)<<{coffee}.
   ```

   Give a description of it, called `BETTER_COFFE_ONE` without parallel composition. Explain intuitivelly the behaviour of LTS.

2. (1 Point) Given `||LIKE_COFFEE = TWO_DRINKS << {coffee}` consider the system

   ```
   ||OTHER_NEW_HOT_DRINKS = (CLIENT||WORKER||LIKE_COFFEE).
   ```

   Give a descriction of the LTS a `OTHER_NEW_HOT_DRINKS` called `BETTER_COFFE_TWO` without parallel composition describing such a system. Explain shortly the behaviour of `OTHER_NEW_HOT_DRINKS`.

Solution Exercise 2

(1 Point) Note that

```
||BETTER_COFFE_HOT_DRINKS = STRAIT<<{coffe}.
```

In state `OPTION` of `STRAIT` there is a choice between `coffee` and `tea`. As we have the operator << only `coffe` will be taken and

```
BETTER_COFFE_HOT_DRINKS = (fill -> coffee -> tea -> BETTER_COFFE_HOT_DRINKS).
```

(1 Point) let us consider `TWO_DRINKS` when `coffee` has high priority. After the `fill` action there is a choice between `coffee` and `tea`, under priority only `coffe` will be choosen the

```
LIKE_COFFEE =  (fill -> coffee -> coffee -> LIKE_COFFEE).
```

Asking for safety in `OTHER_NEW_HOT_DRINKS` e get

```
...
Trace to DEADLOCK:
fill
coffee
```

There is clearly a conflict between the second election of the `CLIENT`, that is `tea` and the possibility offered by the machine, just `coffee`.

Exercise 3 (1 Points) Safety & Liveness properties.

1. (1/2 Point) Define the safety property `NEVER_TWO_FILL` assuring that never one action `fill` is immediatlly followed by other action `fill`. Explain shortly, how do you test that `HOT_DRINKS` verifies the safety property.

2. (1/2 Point) Define the liveness property that: It is always possible to take a tea. Do you think that process BETTER_COFFE_HOT_DRINKS verifies this property? Justify the answer.

Solution Exercise 3

(1/2 Point) Define the property NEVER_TWO_FILL ...

```
property NEVER_TWO_FILL
   = (fill-> (coffee -> NEVER_TWO_FILL | tea-> NEVER_TWO_FILL)
      |coffee-> NEVER_TWO_FILL
      |tea -> NEVER_TWO_FILL).
```

(1/2 Point) Explain shortly....
Just composing HOT_DRINKS in parallel with the property,

```
||SAVE_HOT_DRINKS = (CLIENT||WORKER||TWO_DRINKS||NEVER_TWO_FILL).
```

If the HOT_DRINKS verifies the property, in SAVE_HOT_DRINKS state -1 will never be reached, otherwise -1 will be reached. In our case, running for Safety we get

```
No deadlocks/errors
```

Intuitively, we can "see visually" that property holds because in the LTS corresponding to HOT_DRINKS a fill is never immediate followed by another fill.

(1/2 Point) We have

```
progress TEA = {tea}
BETTER_COFFE_HOT_DRINKS = (fill -> coffee -> tea -> BETTER_COFFE_HOT_DRINKS).
```

As BETTER_COFFE_HOT_DRINKS has only one terminal set of states S = { fill, coffee, tea} and tea $\in$ S the property is verified.

Exercise 4 (2 Points) Java
Define JAVA monitor corresponding to TWO_DRINKS follow M&K approach.

Solution Exercise 4

Straightforward from M&K translation . Given

```
// FSP:
when cond act -> NEWSTAT
```

Translate into:

```
// Java:
public synchronized void act() throws InterruptedException{
      while (!cond) wait();
      // modify monitor data
     notifyAll()
}
```

In our case

```
class Two_Drinks {
    protected int i = 0;
    protected int N = 2;

    public synchronized void fill() throws InterruptedException{
```

```
      while (i != 0) wait();
      i = N;
      notifyAll()
    }

   public synchronized void coffee throws InterruptedException{
      while (i == 0) wait();
      i = i-1;
     notifyAll()
   }

   public synchronized void tea throws InterruptedException{
      while ( i == 0) wait();
      i = i-1;
     notifyAll()
}
```

Exercise 5 (2 Points) Erlang
Remind the `server5` given in the Exam Preparation class:

```
-module(server5).
-export([start/0, rpc/2]).

start() -> spawn(fun() -> wait() end).

wait() ->
   receive
     {become, F} -> F()
   end.

rpc(Pid, Q) ->
    Pid ! {self(), Q},
    receive
        {Pid, Reply} -> Reply
    end.
```

started with

```
Pid=server5:start().
```

Suppose that `server5` is currently running as a factorial server. Imagine that you need it to become a quicksort server.

- (1 Point) Design a module `my_quicksort_server` to do the job.
- (1/2 Point) Complete the following instruction in order to update the server.

  ```
  Pid!{... , ...}
  ```

- (1/2 Point) Write the instruction (or instructions) needed to ask the `server5` to sort the $L = [5.0, 1.0, 10.0, 2.0, 7.0, 6.0]$.

Solution Exercise 5

(1 Point) Following `my_quicksort_server.erl`

```
-module(my_quicksort_server).
-export([loop/0]).

loop() ->
```

```
    receive
{From, {qs, L}} ->
    From ! {self(), qs(L)},
    loop();
{become, Something} ->
    Something()
    end.

%% sequential quicksort

qs([]) -> [];
qs([H|T]) ->
    LT = [X || X <- T, X < H],
    GE = [X || X <- T, X >= H],
    qs(LT) ++ [H] ++ qs(GE).
```

(1/2 Point) Complete ...

```
Pid!{become, fun my_quicksort_server:loop/0}.
```

(1/2 Point) Write the instruction...

```
L = [5.0, 1.0, 10.0, 2.0,  7.0, 6.0 ].
server5:rpc(Pid, {qs, L}).
```
