Concurrency, Parallelism and Distribution (CPD)

Exam Preparation

Joaquim Gabarro
(gabarro@cs.upc.edu)

Computer Science
BarcelonaTech, Universitat Politècnica de Catalunya

1. LTS+FSP
2. Models Problems and Solutions
3. Correctness of Concurrent Programs
4. Erlang
5. Erlang Gen Server

1. LTS+FSP

Class Exercise: three DAYS

You should do this exercise by hand first and then check using the LTSA tool.

- Draw the three `DAY` LTSs, representing the actions of some-one getting up and going to work:
  - `DAY1`: get up (action `up`), then have tea (action `tea`), then go to work (action `work`), then stop
  - `DAY2`: do DAY1 repeatedly
  - `DAY3`: do DAY2, but choose between tea and coffee
- Write the FSP process definitions for the above. You can check these using the LTSA tool.
- Extend DAY3 to DAY4 to include the effects of an `alarm` with a `snooze` button, so prior to the up action, an alarm action is performed. However instead of then doing up you may do a snooze action and go back to the start.

Class Exercise: SENSOR

You should do this exercise by hand first and then check using the LTSA tool.

A sensor measures the water level of a tank. The level (initially 5) is measured in units 0::9. The sensor outputs a `low` signal if the level is less than 2, a `high` signal if the level is greater than 8 and otherwise it outputs `normal`. Model the sensor as an FSP process, `SENSOR`.

Hint: The alphabet of SENSOR is

$$\{\texttt{level}[0 :: 9], \texttt{high}, \texttt{low}, \texttt{normal}\}$$

When the sensor receives a new level it should output low, normal or high as required. This can be done either via a choice, or by specifying that each level input is followed by the appropriate output.

Class Exercise: ||MICROWAVE

(1) Draw (at hand and check with the program) the LTS:

```
MICROWAVE = (put_food_in -> SETTINGS),
SETTINGS = (set_heat_level -> set_time -> COOK
            |set_time -> set_heat_level -> COOK),
COOK = (cook -> take_food_out -> MICROWAVE).
```

(2) Model again the `MICROWAVE` using parallel composition.
Hint: You will need to use handshaking with shared actions, so that it is not possible to produce silly action traces. eg to cook after take food out.

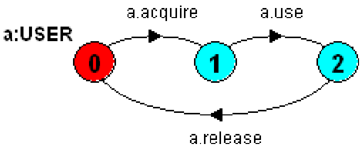
```
COOK = ( put_food_in -> .... ->  take_food_out ->COOK).
SET_HEAT = ( put_food_in -> ... -> cook -> SET_HEAT).
SET_TIME = ...
```

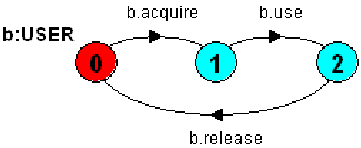
such that

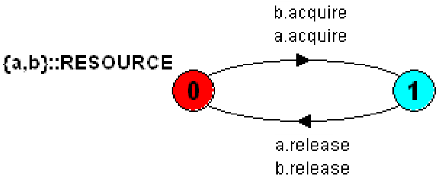
```
||MICROWAVE = ( COOK||SET_HEAT||SET_TIME).
```

Class Exercise: RESOURCE_SHARE

```
RESOURCE=(acquire->release->RESOURCE).
USER=(acquire->use->release->USER).
||RESOURCE_SHARE=(a:USER||b:USER||{a,b}::RESOURCE).
```







Give a picture of the LTS corresponding to RESOURCE_SHARE

2. Models Problems and Solutions

Class Exercise: MICRO_ACCOUNT

Consider a bank dealing with micro accounts.

- You can just deposit or withdraw one euro in each operation.
- There is a bound *M* to the maximum quantity of money .

```
const M=3
MICRO_ACCOUNT = MONEY[0],
MONEY[i:0..M] = (when i>0 withdrawn->MONEY[i-1]
                 |when i<M deposit ->MONEY[i+1]).
CLIENT=(withdrawn->CLIENT|deposit ->CLIENT).

||BANK = (alice:CLIENT||bob:CLIENT
          ||{alice, bob}::MICRO_ACCOUNT).
```

(1) Write a monitor for MICRO_ACCOUNT, complete:

```
public class MicroAccount {
   private int i=0;
   private int M;

   public MicroAccount(int bound) {
     M=bound;
   }
...
}
```

(2) Taking into account the implementation of `MicroAccount` reason about possible deadlocks.

```
public class  Client extends Thread{
   MicroAccount account; String name;
   public Client(String name, MicroAccount account){
      this.name=name; this.account=account;
   }
   public void run(){
     while(true){
       try{
        if(Math.random()<0.5) {
           account.deposit();
           System.out.println(name + " deposit one euro");
           } else {
             account.withdraw();
             System.out.println(name + " withdraw one euro");
           }
          Thread.sleep(3000);
          }catch (InterruptedException e){}
} } }
```

and

```
public class Bank {

   public static void main(String[] args){
      MicroAccount account= new MicroAccount(3);
      Client alice = new Client("Alice", account);
      Client bob = new Client("Bob", account);
      alice.start();
      bob.start();
   }
}
```

Class Exercise: RECURSIVE_LOCK

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 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.

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.

In order to model in FSP a Java recursive lock process, complete the following LST shema:

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

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

Class Exercise: `Eating in Rounds`

Consider three friends, Alice, Bob and Mary eating in rounds

```
a_eat, b_eat, m_eat, a_eat, b_eat,
m_eat, a_eat, b_eat, m_eat, ...
```

`LUNCH` process is defined as:

```
ALICE =(a_eat->ALICE).
BOB =(b_eat->BOB).
MARY = (m_eat->MARY).

FIRST_CONTROL =(a_eat->b_eat->m_eat->FIRST_CONTROL).

||LUNCH =(ALICE||BOB||MARY||FIRST_CONTROL).
```

Remember the M&K approach to design a monitor:

```
FSP: when (cond)  act-> NEWSTAT

JAVA: public synchronized void act()
              throws InterruptedException{
          while.....
          .....
      }
```

- First question. We ask you to redesign FIRST_CONTROL into a SECOND_CONTROL with explicit when guards. Complete the following LTS:

```
SECOND_CONTROL = TURN[1],
TURN[i: 1..3]
   = (when (...) a_eat->TURN[...]
      |  ...
      |...
      ).
```

such that

$$||\mathrm{OTHER\_LUNCH} = (\mathrm{ALICE}||\mathrm{BOB}||\mathrm{MARY}||\mathrm{SECOND\_CONTROL})$$

works correctly.

- Second question, design a monitor `SecondControl` corresponding to `SECOND_CONTROL`. Please follow the M&K schema:

```
class SecondControl{
    protected int turn = 1;
    public ... a_eat()
        throws InterruptedException{...}
    public ... b_eat( )
        throws InterruptedException{...}
    public ... m_eat()
        throws InterruptedException{...}
```

3. Correctness of Concurrent Programs

Class Exercise: `SUPERMARKET`

The recent launch of the drink *Sugarola* has been a success.

- `SUPERMARKET` below models a supermarket where the number of Sugarola bottles that a customer can buy is limited by the number of available bottles on the shelf.
- For instance, action `get[2]` means buying 2 bottles of Sugarola in a purchase.
- The process `WORKER` refills the shelf when Sugarola bottles are scarce (smaller or equal than `Min`).
- At any time, the maximum numbers of bottles in the shelf is `Max`.

```
const Min = 1 //defines the threshold for filling
const Max = 3 //shelf capacity

SHELF = BOT[0],
BOT[i:0..Max]
   = (when (i > 0) get[k:1..i] -> BOT[i - k]
      |when (i<= Min) fill -> BOT[Max]
      ).

WORKER = (fill->WORKER).

CLIENT = (get[1..Max]->CLIENT).

||SUPERMARKET = (SHELF || WORKER || CLIENT).
```

- Draw the `SUPERMARKET` for `Max = 3` and `Min = 1`.
- Define a safety property `NOFILLCHAINED` to show that there are no traces of `SUPERMARKET` issuing two chained fill actions.

```
property NOFILLCHAINED
  = (get[...] -> ...
     |...
     ).
```

- Check the corrrectness with:

```
||CHECK = (SUPERMARKET || NOFILLCHAINED).
```

Class Exercise

A new definition of SUPERMARKET models two types of client, one of them greedy

```
||NEW_MARKET = (SHELF || WORKER || {a,b}:CLIENT)
               /{{a,b}.get/get}<<{a.get}.
```

- Draw NEW_MARKET when Max = 3 and Min = 1.
- Think about the following progress properties concerning NEW_MARKET. Are they true?
  1. progress FILL = {fill}
  2. progress B_GET = {b.get[1..Max]}

Class Exercise: Safe LIFT

A lift has a maximum capacity of `N` people. In a model of the lift control system, passengers entering the lift are signalled by an `enter` action and passengers leaving the lift are signalled by an `exit` action. Specify a safety property in `FSP`, which when composed with the lift model, will check that the system never allows the lift that it controls to have more than `N` occupants.

Complete the following code:

```
const N = ...
property LIFTCAPACITY = LIFT[0],
LIFT[...] = (enter -> LIFT[i+1]
                |when(i>0) ....
                |...).
```

4. Erlang

Class Exercise: `pmax(L)`

The following program find the max of a list

```
my_max([H|T]) -> my_max(T, H).

my_max([H|T], Max) when H > Max -> my_max(T, H);
my_max([_|T], Max)              -> my_max(T, Max);
my_max([],    Max)              -> Max.
```

Desing a `pmax(L)` such that.

- When `L` has less than 10 elements it calls `my_max`, otherwise:
- It halves `L` into `L1`, `L2`.
- It creates two processes `P1` and `P2`. The list `L1` goes to `P1` and `L2` goes to `P2`. Process `P1` uses `my_max` to find the max and send back this value. Process `P2` do the same with `L2`.
- Suppose that `Max1` and `Max2` store the values received from `P1` and `P2`, use `my_max` to compute and return the max.

Describe shortly the (possible) advantages or disadvantages of `pmax(L)` in relation to `my_max`. In fact which program is faster in your opinion `my_max` or `pmax`?

Class Exercise: Two Words Translator

Please write a module containing a simple translation program that gets a word in Spanish and prints an English translation. The process should run in a loop, waiting for words to translate.

At the beginning just translates the words `casa` and `blanca`.

Following a trace of a possible execution:

```
28> Pid = spawn(fun translate:loop/0).
<0.156.0>
29> Pid ! "casa".
house
"casa"
30> Pid ! "blanca".
white
"blanca"
31> Pid ! "rosada".
I do not understand
"rosada"
```

Class Exercise: my_counter

Complete the following code,

```
-module(my_counter).
-export([start/0,loop/1,increment/1,value/1,stop/1]).

%% (1) The interface functions.
start() -> spawn(...).
increment(Counter) -> Counter ! increment.
value(Counter) ->  ... ! {self(),value},
                   receive
                        {..., ...} -> ....
                   end.
 stop(Counter) -> Counter ! stop.

%% (2) The counter loop.
loop(Val) -> receive
                 increment -> loop(...);
                 {From,value} -> From ! {self(),Val}, ...;
                 stop -> true;
                  _ -> % All other messages, recursive call
                         ...
             end.
```

in order to have the following behaviour,

```
20> c(my_counter).
{ok,my_counter}
21> Counter = my_counter:start().
<0.89.0>
22>
22> my_counter:value(Counter).
0
23> my_counter:increment(Counter).
increment
24> my_counter:value(Counter).
1
25> my_counter:increment(Counter).
increment
26> my_counter:value(Counter).
2
27> my_counter:stop(Counter).
stop
28>
```

5. Erlang Gen Server

Class Exercise: Matrix Product & Server

Let us develop a `matrix_product` module to compute the matrix product.

- (1) Remind that given two vectors `X` and `Y`, for instance

```
2> X= [1.0, 2.0, 3.0, 4.0].
[1.0,2.0,3.0,4.0]
4> Y= [1.0, 4.8, 9.8, 16.0].
[1.0,4.8,9.8,16.0]
```

the dot product is `dot_prod(X,Y) = 1.0*1.0 +...+4.0*16.0`. Define a function

```
% Pre: both input lists have the same length
dot_prod([], []) -> 0;
dot_prod([... | ...], ...) -> ....
```

such that

```
6> D= matrix_product:dot_prod(X,Y).
104.0
```

- (2) In Erlang we give matrix row by row

```
7> M=[[1.0,2.0,3.0,4.0],
      [1.0,4.0,9.0,16.0],
      [1.0,8.0,27.0,64.0]].
```

Corresponds to the matrix

$$M = \begin{bmatrix} 1.0 & 2.0 & 3.0 & 4.0 \\ 1.0 & 4.0 & 9.0 & 16.0 \\ 1.0 & 8.0 & 27.0 & 64.0 \end{bmatrix}$$

Given `M` the `transpose(M)` changes rows into columns:

$$\texttt{transpose}(M) = \begin{bmatrix} 1.0 & 1.0 & 1.0 \\ 2.0 & 4.0 & 8.0 \\ 3.0 & 9.0 & 27.0 \\ 4.0 & 16.0 & 64.0 \end{bmatrix}$$

In Erlang

```
11> matrix_product:transpose(M).
[[1.0,1.0,1.0],[2.0,4.0,8.0],[3.0,9.0,27.0],[4.0,16.0,64.0]]
```

Complete the following `transpose(M)` function

```
transpose([]) -> [];
transpose([[]|_]) -> [];
transpose(M) ->
  [ [H || [...| ...] <- M ] | transpose([ ... || ... ]) ].
```

- (3) Finally complete the following function to compute the matrix product

```
mult(A, B) ->
   BT = transpose(...),
   [[ dot_prod(RowA, ...) || ...] || RowA <- ...].
```

such that (shematically):

```
11> M=[[1.0,2.0,1.0],[2.0,0.0,3.0]].
12> N=[[1.0,2.0,3.0],[2.0,1.0,2.0],[3.0,2.0,1.0]].
13> c(matrix_product).
14> P=matrix_product:mult(M, N).
[[8.0,6.0,8.0],[11.0,10.0,9.0]]
```

- (4) Remind the `server5` given in the section 16.1 of Armstrong book.

```
-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 (as in the book). Imagine that you need it to become a matrix product server.

- Design a module `my_mult_server` to do the job.
- Complete the following instruction in order to update the server.

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