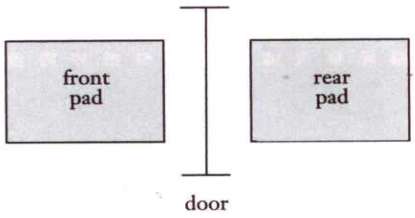
CS 3SD3 Midterm Test Sample Solutions

1.[5] Consider the controller for an automatic door. Often found at supermarkets entrances and exits, automatic doors swing open when the controller senses the person is approaching. An automatic door has a pad in front to detect the presence of the person about to walk through the doorway. Another pad is located to the rear of the doorway so that controller can hold the door open enough for the person to pass all the way through and also so that the door does not strike someone standing behind it as it opens. The configuration is shown in the following figure:



The controller is in either of two states: OPEN or CLOSED, representing the corresponding condition of the door, and there are four possible actions/conditions:
FRONT - the person is standing on the pad in front of the doorway,
REAR - the person is standing on the pad to the rear of the doorway,
BOTH - the people are standing on both pads,
NEITHER - no one is standing on either pad.

Model this controller as an *FSP* process, DOOR_CONTROLLER.

Solution 1:

```
DOOR_CONTROLLER = CLOSED,
CLOSED = (neither -> CLOSED
         |both -> CLOSED
         |rear -> OPEN
         |front -> OPEN),
OPEN = (neither -> CLOSED
         |both -> CLOSED %this is to prevent striking a person
         |rear -> OPEN
         |front -> OPEN).
```

Solution 2:

```
DOOR_CONTROLLER = CLOSED,
CLOSED = (neither -> CLOSED
          |both       -> CLOSED
          |rear    -> CLOSED        // avoid striking people standing on
the rear pad
          |front   -> OPEN),
OPEN = (neither -> CLOSED
         |both   -> OPEN
         |rear   -> OPEN            // avoid striking people standing on
the rear pad
         |front  -> OPEN).
```

2.[10] A small hotel cafeteria is used by several customers eating breakfast. The customers either drink tea of coffee. There is one Tea Machine and one Coffee Machine. When either Tea Machine or Coffee Machine runs out of tea or coffee, new tea or coffee is provided by cafeteria staff. Assume capacities of both machines are the same: N drinks. Initially both machines are full.

a.[5] Provide an FSP description of the above scenario. You must provide also a brief description of the intended behavior for each one of the processes you define.

Hint: Possible processes for the above scenario are CUSTOMER, TEA_MACHINE, COFFEE_MACHINE and STAFF_MEMBER.

b.[5] Provide a Petri nets (any kind) description of the above scenario.
Hint. Using Place/Transition nets is advised as it seems to be the most natural.

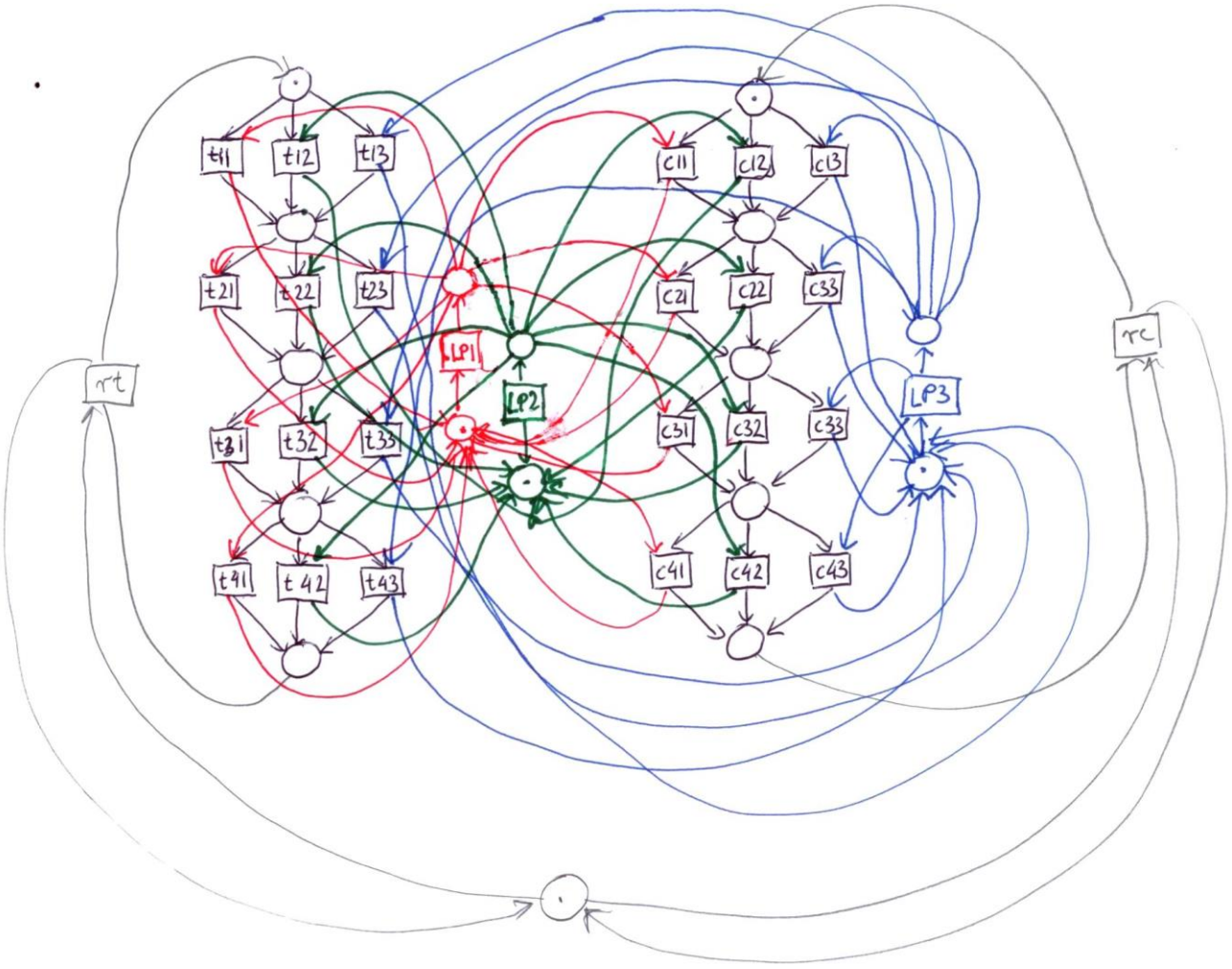
Solutions:
(a) Assume M customers

```
const N = 4
range Jobs = 0..N

TEA_MACHINE  = TEA_MACHINE[N],
TEA_MACHINE[j:Jobs] = (when j==0 replece_tea -> TEA_MACHINE[N]
                | when j>0 give_tea -> TEA_MACHINE[j-1])

COFFEE_MACHINE  = COFFEE_MACHINE[N],
COFFEE_MACHINE[j:Jobs] =
          (when j==0 replece_coffee -> COFFEE_MACHINE[N]
                | when j>0 give_coffee -> COFFEE_MACHINE[j-1])

CUSTOMER = ( drink_tea -> CUSTOMER | drink_coffee -> CUSTOMER )

Const M = 3
range Customers = 0..M
||CUSTOMERS = (forall[i:CUSTOMERS] customer[i]:CUSTOMER).

STAFF_MEMBER = (replace_tea -> STAFF_MEMBER
                | replace_coffee -> STAFF_MEMBER)

||CAFETERIA = (||CUSTOMERS || TEA_MACHINE || COFFEE_MACHINE
  || STAFF MEMBER)
     /{ customer[Customers].drink_tea/give_tea,
     customer[Customers].drink_coffee/give_coffee
     customer[Customers].replace_tea/replace_tea,
     customer[Customers].replace_coffee/replace_coffee}
```

(b) The solution below are very detailed. Such a level of details is not required, more sketchy solutions will be accepted.

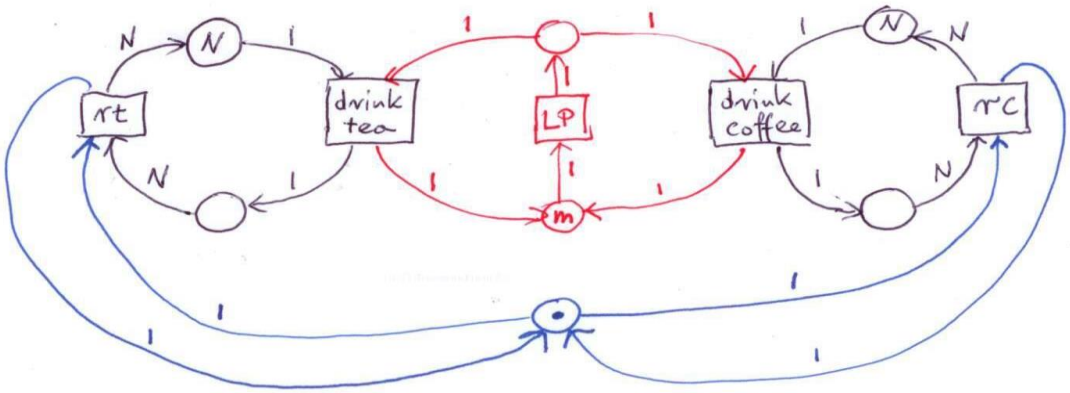
Elementary nets, 3 customers (1-red, 2-green, 3-blue), N=4 (capacity of machines)
tij - customer j drinks ith cup of tea
cij - customer j drinks ith cup of coffee
Li – local processing of customer i
rt – refill of tea
rc – refill of coffee

This is mimicking FSP solution, while easy, a little bit clumsy.



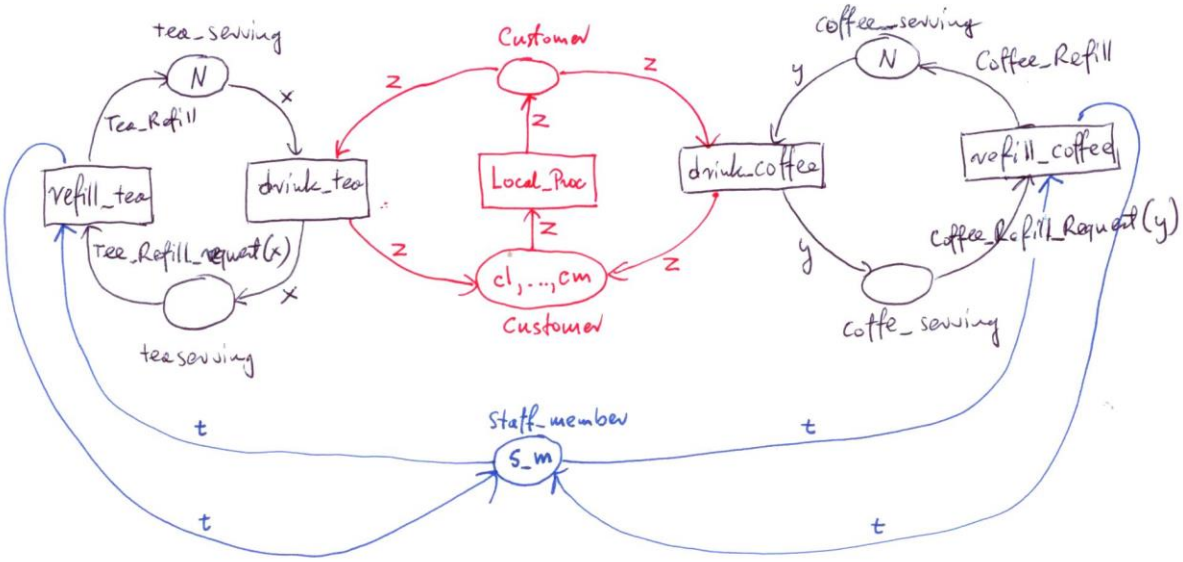
Place/Transition nets, *the best choice for this problem*. Assume *m* customers and *N* as the capacity of both machines.

Notation: LP – customers do local processing
rt – refill tea
ct – refill coffee



Coloured Petri nets, practically a copy of the solution with P/T nets above.

```
colour Customer = with c1 | … | cm
colour Staff_member = with s_m
colour tea_serving = Integer
colour coffee_serving = Integer
var x: tea_serving
var y: coffee_serving
var z : Customer
var t : Staff_member
fun Tea_Refill_Request x = if x==N then N
fun Coffee_Refill_Request y = if y==N then N
fun Tea_Refill N
fun Coffee_Refill N
```



3.[10] Consider the following problem. Three processes P, Q and R continuously perform a certain task separately. In order to achieve that task, each process needs to obtain two resources, say, *a* and *b*. Obtaining and releasing a resource requires to perform actions *get* and *put* in this order. With FSP this is modelled by:

RESOURCE = (get −> put −> RESOURCE) .

To work properly, process P needs to get *a* before *b*, process Q needs to get *b* before *a*, while for process R the order does not matter.

a.[5] Model the situation described above carefully avoiding deadlock with FSP.

b.[5] Model the situation described above carefully avoiding deadlock with Petri nets (any kind).

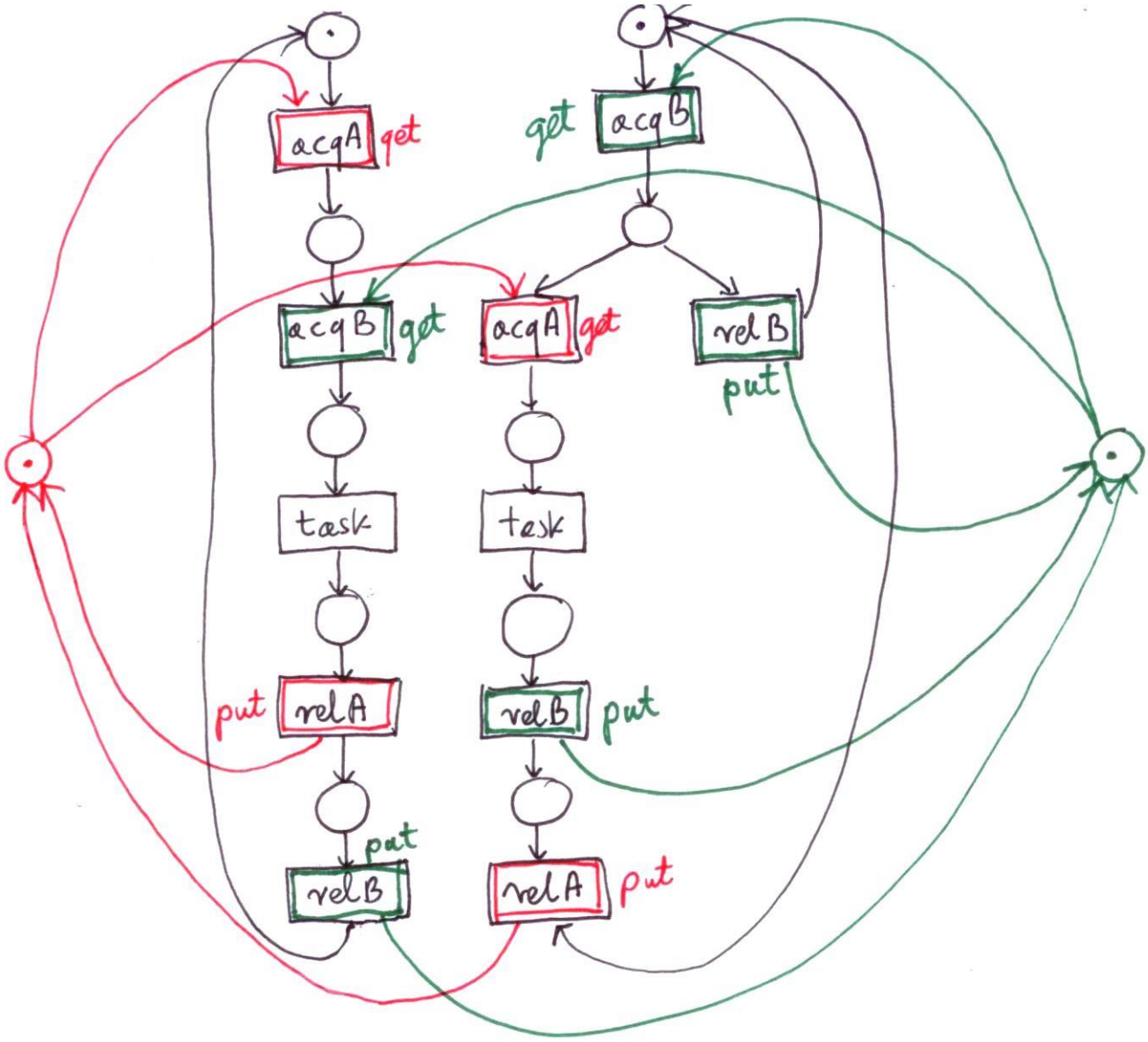
Solutions:
(a)

A possible model for the above scenario can be defined as

```
RESOURCE = ( get → put → RESOURCE ).

/* P performs the acquisition of resources normally */
P = ( acquireA → acquireB → task → releaseA → releaseB → P ).

/* The possible idea for avoiding deadlock is that process
 * Q try to acquire normally the first resource needed to
 * perform its task. Then, it 'attempts' to acquire the second one
 * if possible, otherwise it releases the acquired resource. */

Q = ( acquireB → GET_A ),
GET_A = (
  acquireA → task → releaseB → releaseA → Q
  | releaseB → Q ).

/* Processes interaction */

||PQ = ( p:P || a:RESOURCE || b:RESOURCE || q:Q )
  /{
    {p,q}.acquireA/a.get,
    {p,q}.releaseA/a.put,

    {p,q}.acquireB/b.get,
    {p,q}.releaseB/b.put
  }.
```

Clearly the above model is deadlock free as if the process Q cannot obtain both of the resources to perform its task, it will release the acquired resources allowing process P to continue.

b. (1) Elementary Petri Nets. *Probably the best choice for this problem.*



(2) Place Transitions Nets. Because resources are distinguishable, they cannot be represent by identical tokens, so P/T-net solution is exactly the same as Elementary nets solution.

(3) Coloured Petri Nets. We can introduce colour Resource with two elements a and b, and join red and green graphs into one. But it does not simplify a picture significantly (black part, i.e. processes P and Q must be unchanged) and requires defining appropriate functions attached to arrows (to specify correct choice). Altogether it is not worth to do it.

4.[10] A central computer, connected to remote terminals via communication links, is used to automate seat reservations for a concert hall. A booking clerk can display the current state of seat reservations on the terminal screen. In order to book a seat, a client chooses a free seat and then the clerk enters the number of the chosen seat at the terminal and issues a ticket. A system is required which avoids the double-booking of seats while allowing clients free choice of the available seats.

Construct a model and prove that the model does not permit double-booking.

Hints: For the LTSA model *possible* processes are SEAT, CLERK (or CLIENT). Moreover, it is only necessary to model a small number of them.

Solution #2. Here we have only SEAT and LOCK processes:

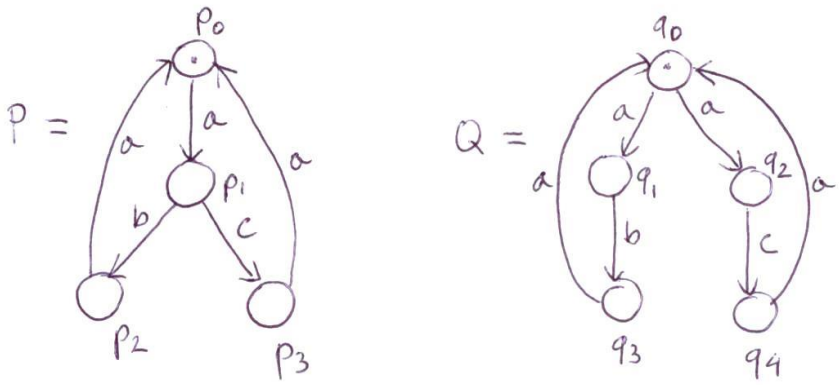
```
const False = 0
const True  = 1
range Bool =  False..True

SEAT = SEAT[False],
SEAT[reserved:Bool]
     = ( reserve    -> SEAT[True]
       | query[reserved] -> SEAT[reserved]
       | when (reserved) reserve -> ERROR   //error of reserved twice
       ).

range Seats = 0..1
||SEATS = (seat[Seats]:SEAT).

LOCK = (acquire -> release -> LOCK).

TERMINAL = (choose[s:Seats] -> acquire
           -> seat[s].query[reserved:Bool]
           -> (when(!reserved)seat[s].reserve -> release-> TERMINAL
              |when(reserved) release -> TERMINAL)
           ).

set Terminals = {a,b}
||CONCERT = (Terminals:TERMINAL || Terminals::SEATS || Terminals::LOCK).
```

Solution #2.
We will only have SEAT and CLERK processes as a behaviour of possible processes CLIENT are entirely defined by actions of SEATs and CLERKs the same as CLERK.

```
/* Question 1: Booking of seats for a concert */

range Bool = 0..1
const False = 0
const True = 1

/* Process SEAT models the behavior of a seat where:
 *
 * (i) A seat has an associated state indicating if it is available or not.
 * (ii) Action take allows to take an available seat,
 * (iii) Action free displays an available seat.
 * (iv) Action busy displays a busy seat. */

SEAT = SEAT[True],
SEAT[a: Bool] = (
 when ( a == True ) take → SEAT[False]
 | when ( a == True ) free → SEAT[a]
 | when ( a == False ) busy → SEAT[a] ).


/* Process SEATS models a finite number of seats. */

range Seats = 1..4

||SEATS = ( forall[i: Seats] seat[i]:SEAT ).


/* Process CLERK models the behavior of a clerk where:
 *
 * (i) Action display models the displaying of available seats.
 * (ii) Action book models the booking  of available seats.
 * (iii) Action ticket models the issueing of tickets. */

CLERK = (
 display[s: Seats] → CLERK
 | book[s: Seats] → ticket[s] → CLERK ).
```

```
/* Process CLERKS models a finite number of clerks. */

range Clerks = 1..2

||CLERKS = ( forall[i: Clerks] clerk[i]:CLERK ).

/* Booking of seats */

||CONCERT = ( CLERKS || SEATS )
 /{
  clerk[i: Clerks].display[s: Seats]/seat[s].free,
  clerk[i: Clerks].book[s: Seats]/seat[s].take
 }.
```

5.[5] Consider to Labelled Transition Systems P and Q given below. Are they *bisimilar*, or not? Prove your answer.



Solution:

No they are not. Consider the trace *a*. In *P* it leads to the state p_1, in *Q* it leads to the states q_1 or q_2. However from p_1 we can execute either *b* or *c*, while from q_1 only *b* and from q_2 only *c*. Hence neither (p_1, q_1) nor (p_1, q_2) are bisimilar, so *P* and *Q* are not either.