CS 3SD3. Sample solutions to the assignment 2.

Total of this assignment is155 pts. Each assignment is worth 11% of total. Most of solutions are not unique.

If you think your solution has been marked wrongly, write a short memo stating where marking in wrong and what you think is right, and resubmit to me via e-mail (as pdf). The deadline for a complaint is 2 weeks after the assignment is marked and returned.

1.[10] *The saving account problem.* A saving account is shared by several people. Each person may deposit or withdraw funds from the account subject to the constraint that the balance of the account must never become negative.

a.[5] Develop a model for the problem using FSP processes

b.[5] From the FSP model derive a Java implementation of a monitor for the saving account.

Solution:

a.[5]

```
const Max = 5
range Money = 0..Max

ACCOUNT = ACCOUNT[0],
ACCOUNT[balance:Money] = (when (balance>0)
                          withdraw[d:1..balance] -> ACCOUNT[balance-d]
                         |deposit[d:1..Max] -> ACCOUNT[balance+d]
                         ).
```

b.[5]

```
/* Java implementation

public class Account {
  protected double balance;

  public Account() {balance =0.0;}
  public Account(double initial) {balance = initial;}

  public synchronized void withdraw(d:double) throws
InterruptedException{
        while (balance < d) wait();
        balance -= d;
  }

  public synchronized void notify(d:double) {
        balance += d;
        notifyAll();
  }
}
```

2.[10] a.[5] (A single-slot buffer may be modelled by:

ONEBUF = (put -> get -> ONEBUF).

Program a Java class, OneBuf, that implements this one-slot buffer as a monitor.

b.[5] Replace the condition synchronization in your implementation of the one-slot buffer (question 2a above) by using semaphores. Given that Java defines assignment to scalar types (with the exception of long and double) and reference types to be atomic, does your revised implementation require to use of the monitor's mutual exclusion lock?

Solutions:

a.[5]

```
ONEBUF = (put -> get -> ONEBUF).

/* java Implementation

public class OneBuf {
    Object slot = null;

    public synchronized void put(Object o) throws InterruptedException
{
        while(slot != null) wait();
        slot = o;
        notifyAll();
    }

     public synchronized Object get () throws InterruptedException {
        while(slot == null) wait();
        Object o = slot;
        slot = null;
        notifyAll();
        return o;
    }
}
```

b.[5]

```
ONEBUF = (put -> get -> ONEBUF).

/* java Implementation using Semaphores
```

```
public class OneBuf {
    Object slot = null;
    Semaphore empty = new Semaphore(1);
    Semaphore full  = new Semaphore(0);

    public void put(Object o) throws InterruptedException {
        empty.down();
        slot = o;
        full.up();
    }
     public Object get () throws InterruptedException {
        full.down();
        Object o = slot;
        slot = null;
        empty.up();
        return o;
    }
}
```

3.[50] Consider machine shop with *K* identical machines. Due to heavy work each machine needs to have two parts, say *part1* and *part2* replaced from time to time by a technician. Each machine either works, or have *part1* replaced, or have *part2* replaced. The *part1* storage has capacity of *M1* parts, and *part2* storage has capacity of *M2* parts. When a storage is empty, a machine that needs this part waits until the technician refills it. If any storage is empty, the technician refills it with either *M1 parts1* or *M2 parts2*. There is only one technician. Refilling must be done as soon as possible, so they have priority over part replacing. There is no partial refilling of the storages. Assume that initially both storages are full.

a.[15] Model the behaviour of the system as FSP processes. Write a safety property that when composed with your system will check if no *part1* is replaced when *part1* storage is empty and vice versa. Compose this property with your system and verify if this it holds.

There is no standard model of priorities in Petri nets, so you have a freedom to define them to fit your solution. This is a comment for points (b), (c) and (d).

b.[10] Model the behaviour of the system as an Elementary Petri net (see Lecture Notes 3).

c.[10] Model the behaviour of the system as a Place/Transition Petri net (see Lecture Notes 9, pages 21, 22).

d.[10] Model the behaviour of the system as a Coloured Petri net (see Lecture Notes 9, pages 23-34).

e.[5] Discuss the differences between the FSP and various Petri Net solutions.

Solution
a.[15]

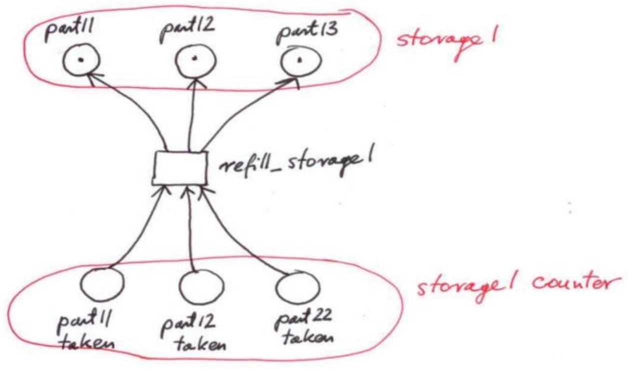
```
MACHINE = (work -> MACHINE | replace_p1 -> MACHINE | replace_p2 -> MACHINE)

const K = 4
range mach = 1..K

|| MACHINES = (forall[i:mach]machine[i]:MACHINE)

TECHNICIAN = ( replace_p1 -> TECHNICIAN | replace_p2 -> TECHNICIAN ) |
      refill_storage_1 -> TECHNICIAN | refill_storage_2 -> TECHNICIAN )

const M = 3
range part1 = 0..M
STORAGE_1 = STORAGE_1[M]
STORAGE_1[s:part1] = (when (s>0) replace_p1 -> STORAGE_1[s-1]
                     | when (s==0) refill_storage_1 -> STORAGE_1[M] )
const N = 3
range part2 = 0..N
STORAGE_2 = STORAGE_2[N]
STORAGE_2[s:part1] = (when (s>0) replace_p2 -> STORAGE_1[s-1]
                     | when (s==0) refill_storage_2 -> STORAGE_1[N]

property P_part1 = P_part1[M]
P_part1[s:part1] = (when (s==0) refill_storage_1 -> P_part1[M])

property P_part2 = P_part2[N]
P_part2[s:part2] = (when (s==0) refill_storage_2 -> P_part2[N])

||MACHINE_SHOP = ( MACHINES || STORAGE_1 || STORAGE_2 || TECHNICIAN ||
                  P_part1 || P_part2 ) << {refill_storage_1,refill_storage_2}
```

Comment on Petri nets solutions.

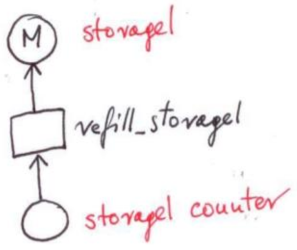
One obvious solution is just to mimic any FSP solution by replacing individual processes by their LTS and then merge common transitions. However such solution is not very readable and unnecessary complex (i.e. the resulting net is huge).

When modelling directly with Petri nets, the initial challenge could be how to model the fact that `TECHNICIAN` can fill `STORAGE_1` and `STORAGE_2` only when they are empty. In principle this is a test for zero and standard Petri nets do not have it. Inhibitor nets have it, but they are equivalent to Turing Machines, so many problems are undecidable, a headache for tool developers. However this can easily be modelled by the following scheme: the servant does not look into the dispensers, he/she counts the servings taken out of the dispensers, he/she knows the dispensers capacity, so he/she can decide when any dispenser is empty by just counting. Of course this works only if initially the dispensers are full. And this can be modelled quite easily, as you can see below.

(i) Elementary nets and 3 is the capacity of, say, storage 1:



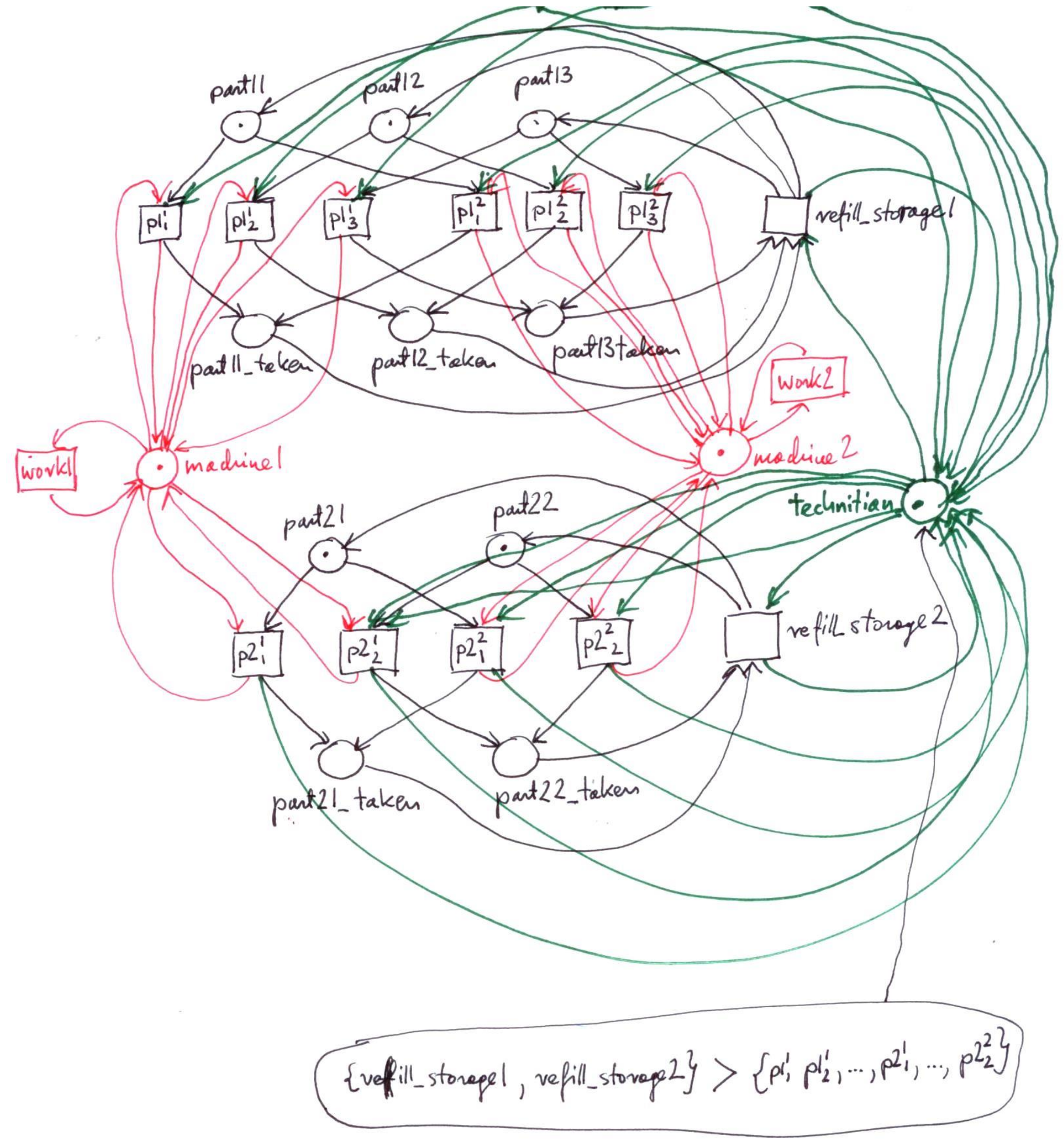
(ii) For Place/Transition nets it is even simpler and more intuitive. Let *M* be a capacity of storage 1.



There is a possibility of overfilling the storage here, but this can be taken care by the rest of the system.

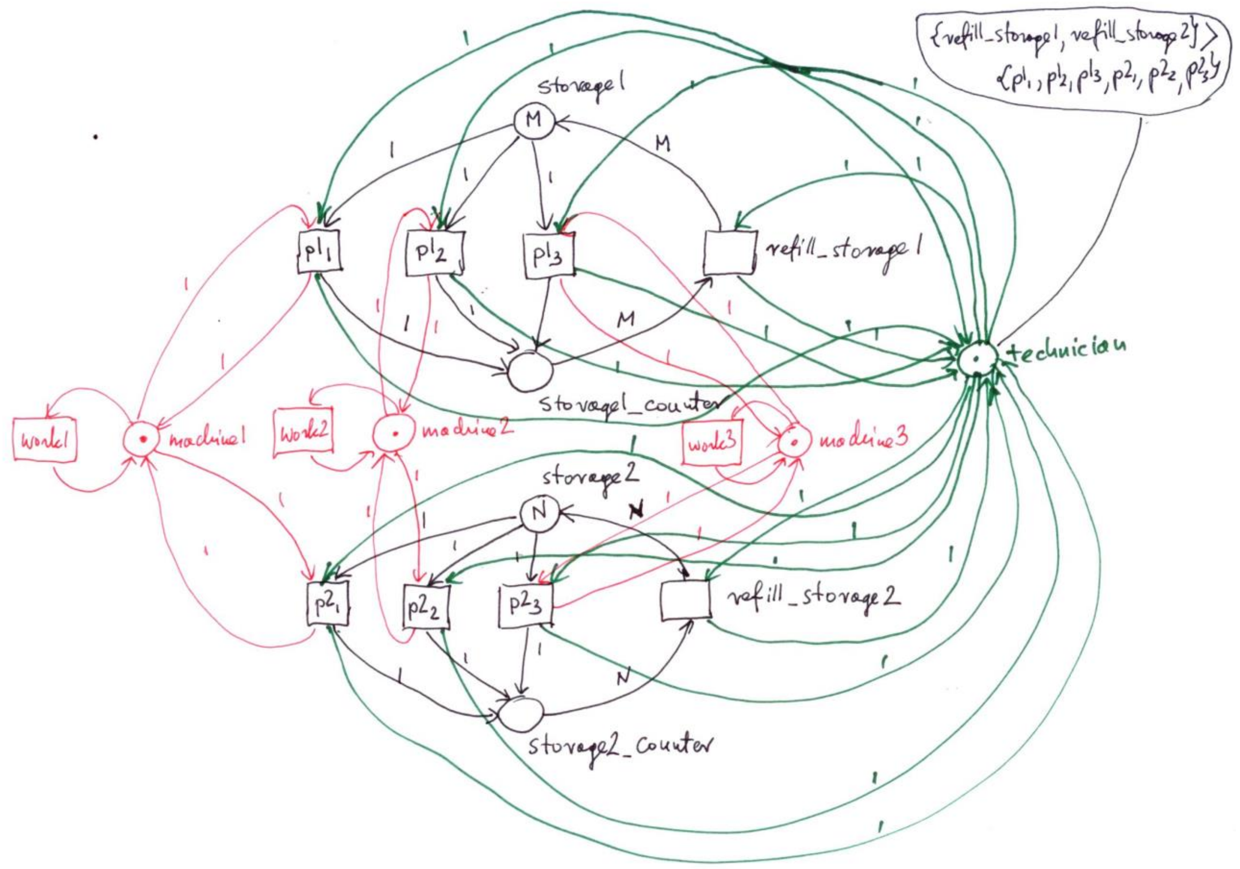
(iii) For Coloured Petri nets it is as (ii), only some syntactic difference.

b.[10] A sample solution with elementary nets for 2 machines, 3 storage 1 capacity and 2 storage 2 capacity. Machines are the red part of the net, technician is green.



$p1^i_j$ means that part1j is used as a replacement in machine i and $p2^i_j$ means that part2j is used as a replacement in machine i. The label attached to the technician defines priorities. Petri nets are very flexible in this matter, practically every method can be implemented.

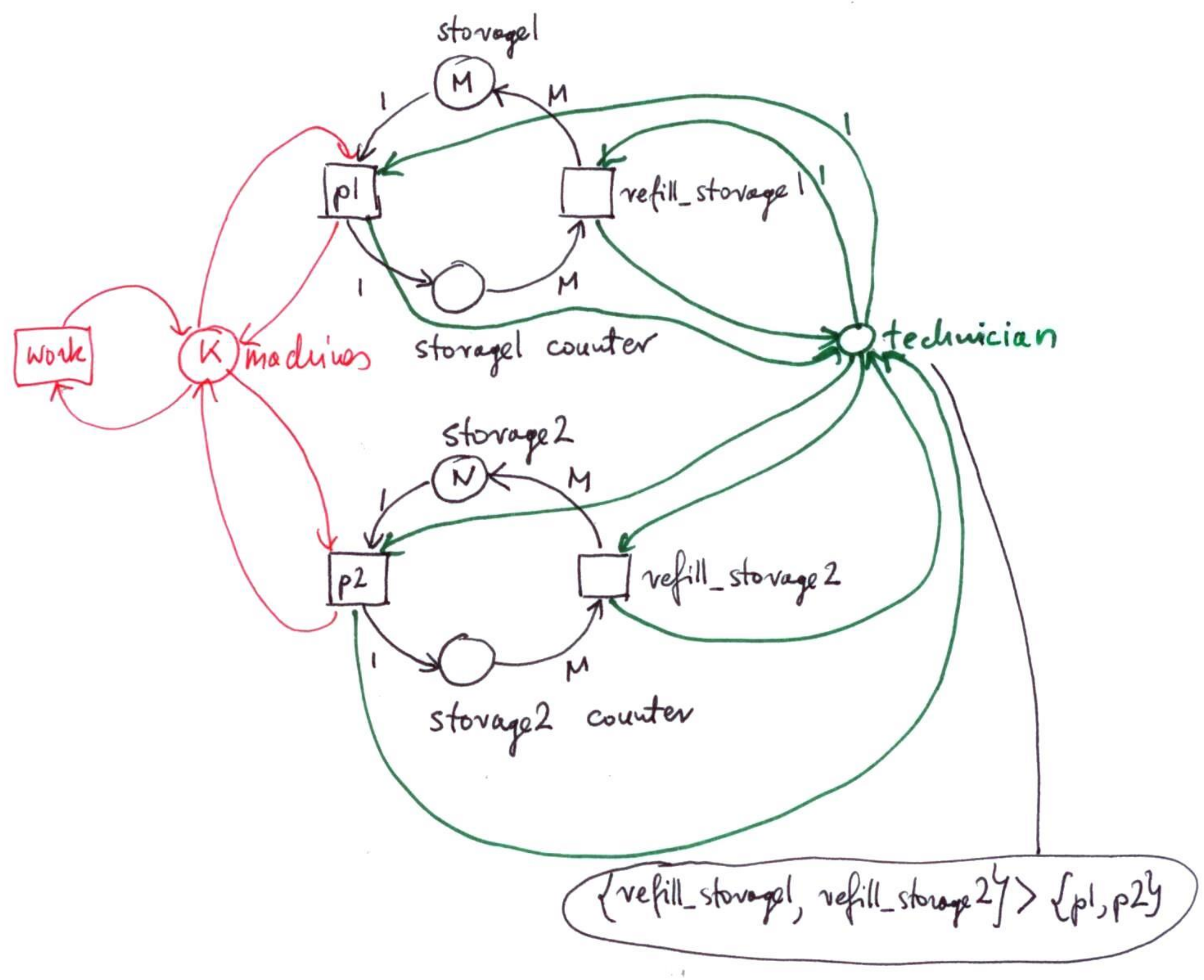
c.[10] A sample solution with Place/Transition nets for 3 machines, M storage 1 capacity and N storage 2 capacity. Machines are the red part of the net, technician is green.



p1$_i$ means machine i has part1 replaced and p2$_i$ means machine i has part2 replaced

This solution is better as it makes servings not distinguishable, which I believe is the intention of the problem.

If for some reasons making a distinction between machines is not important, for instance only the behaviour of storages is what we are looking for, the following Place/Transition nets models k machines, M storage 1 capacity and N storage 2 capacity (simultaneous execution of *work* with itself is allowed).



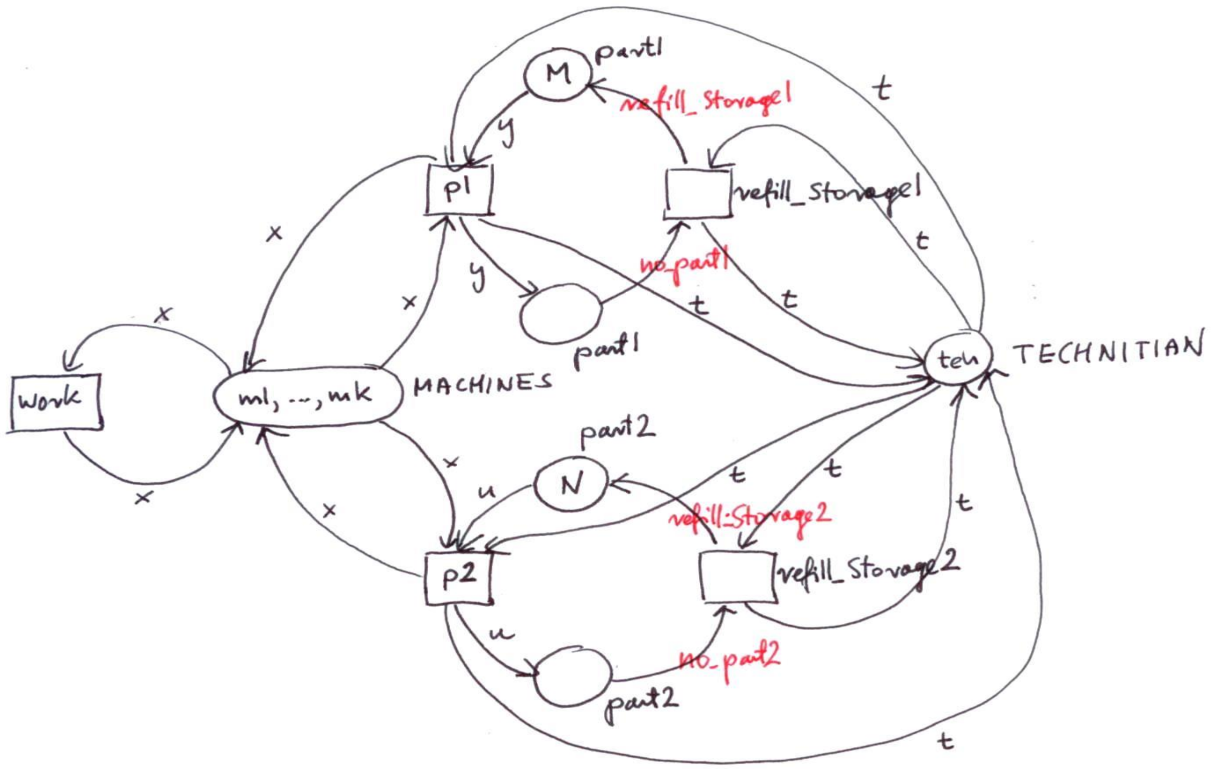
This is really very elegant solution.

d.[10] Coloured Petri nets.

color MACHINES = with m1 | m2 |…| mk
color TECHNICIAN = with techn
color part1 = Integers
color part2 = Integers
var x: MACHINES
var t: TECHNICIAN
var y,z: part1
var u,v: part2
fun refill_Storage1 z = M
fun no_part1 z =M
fun refill_Storage2 v = M
fun no_part2 v =M

/* defining priorities depends on a particular implementation of Coloured Petri Nets*/

priority {refill_Storage1,refill_Storage2}>{p1,p2}



e.[5] Any reasonable comments are acceptable. Elementary nets are the closest to FSP. Petri nets allow formal proofs, that is fairly difficult with FSP, where mainly checking can only be provided.

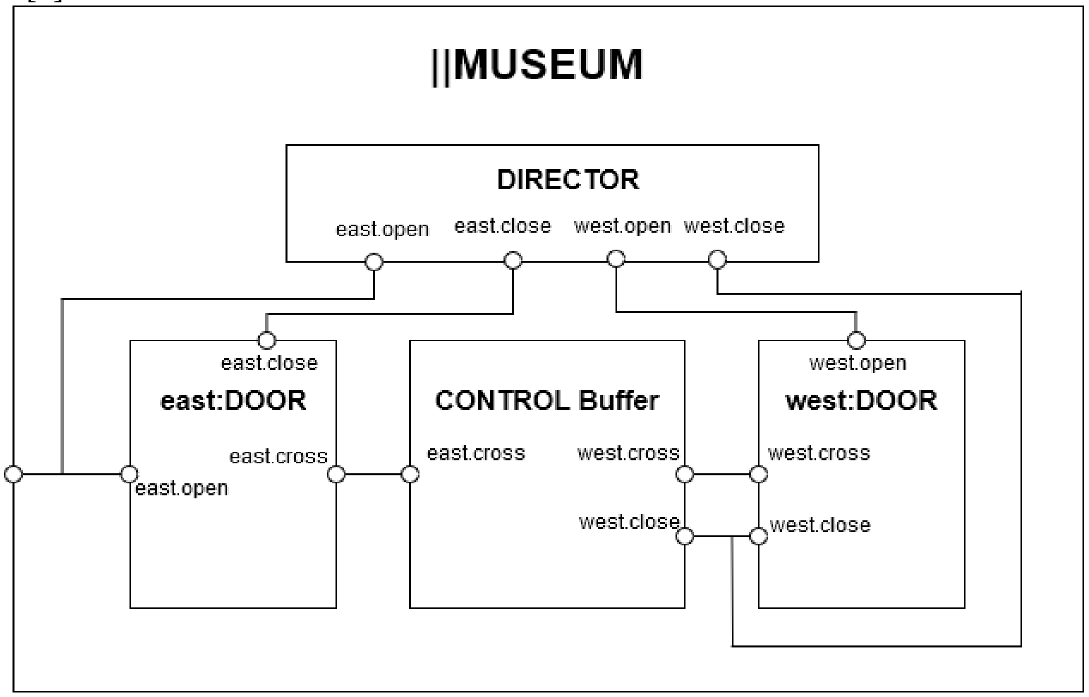
4.[35] A museum allows visitors to enter through the east entrance and leave through its west exit. Arrivals and departures are signalled to the museum controller by the turnstiles at the entrance and exit. At opening time, the museum director signals the controller that the museum is open and then the controller permits both arrivals and departures. At closing time, the director signals that the museum is closed, at which point only departures are permitted by the controller.

For Process Algebra models (as FSP), the museum system consists of processes EAST, WEST, CONTROL and DIRECTOR.

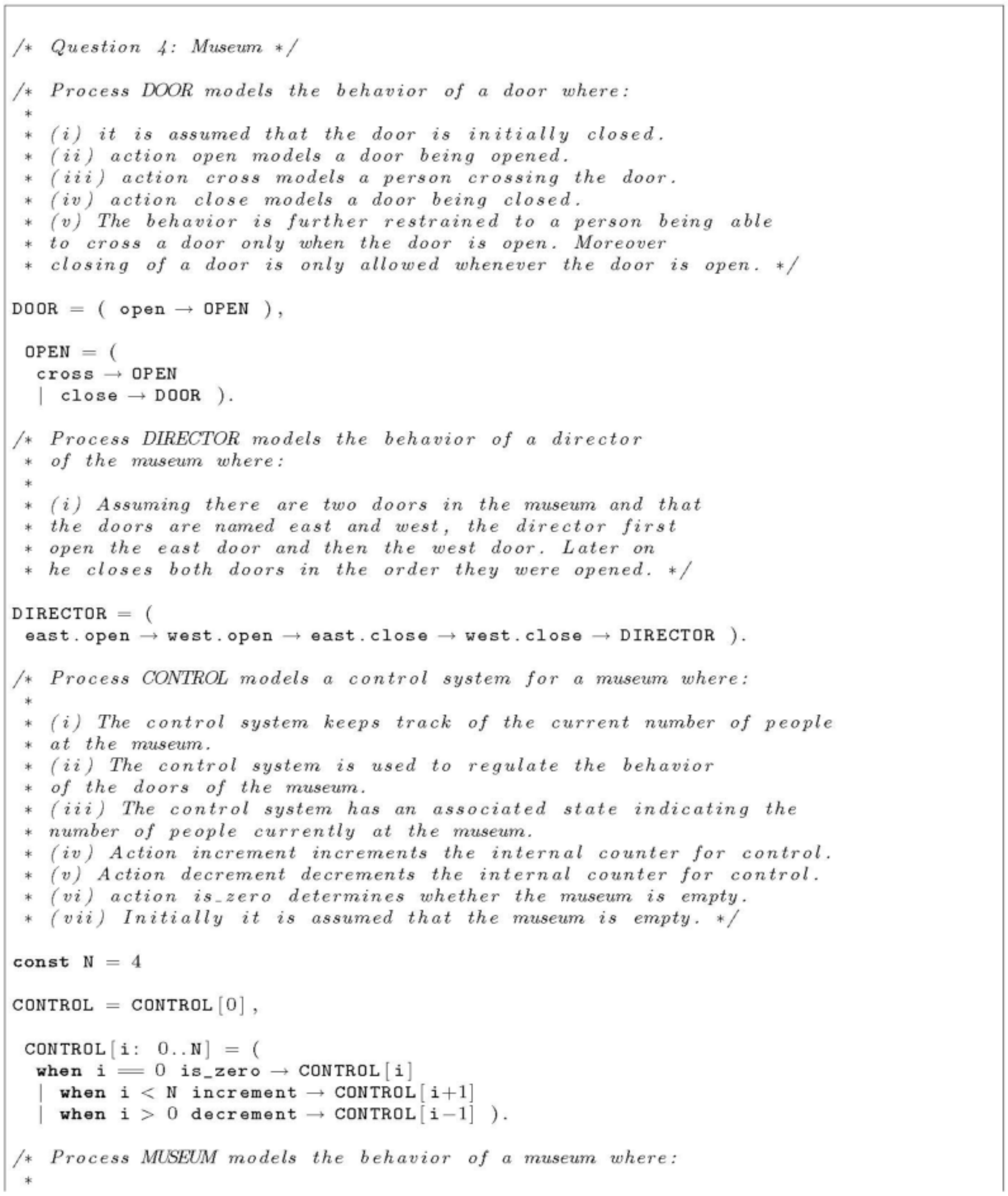
a.[5] Draw the structure diagram for the museum and
b.[10] Provide an FSP description for each of the processes and the overall composition.
c.[10] Model the above scenario with Petri nets (any kind, your choice)
d.[10] Provide Java classes which implement each one of the above FSP processes.

Solutions.

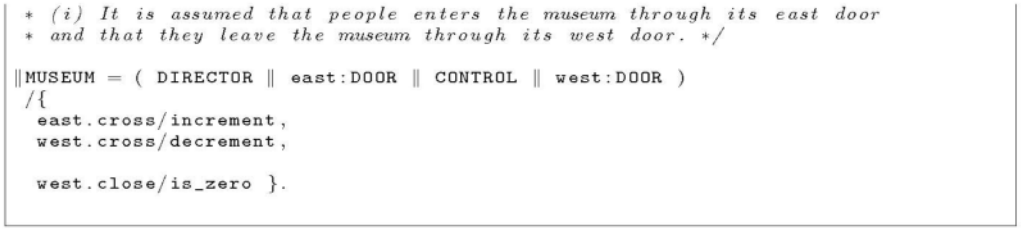
a.[5]



b.[5]



```
/* Question 4: Museum */

/* Process DOOR models the behavior of a door where:
 *
 * (i) it is assumed that the door is initially closed.
 * (ii) action open models a door being opened.
 * (iii) action cross models a person crossing the door.
 * (iv) action close models a door being closed.
 * (v) The behavior is further restrained to a person being able
 * to cross a door only when the door is open. Moreover
 * closing of a door is only allowed whenever the door is open. */

DOOR = ( open → OPEN ),

 OPEN = (
  cross → OPEN
  | close → DOOR ).

/* Process DIRECTOR models the behavior of a director
 * of the museum where:
 *
 * (i) Assuming there are two doors in the museum and that
 * the doors are named east and west, the director first
 * open the east door and then the west door. Later on
 * he closes both doors in the order they were opened. */

DIRECTOR = (
 east.open → west.open → east.close → west.close → DIRECTOR ).

/* Process CONTROL models a control system for a museum where:
 *
 * (i) The control system keeps track of the current number of people
 * at the museum.
 * (ii) The control system is used to regulate the behavior
 * of the doors of the museum.
 * (iii) The control system has an associated state indicating the
 * number of people currently at the museum.
 * (iv) Action increment increments the internal counter for control.
 * (v) Action decrement decrements the internal counter for control.
 * (vi) action is_zero determines whether the museum is empty.
 * (vii) Initially it is assumed that the museum is empty. */

const N = 4

CONTROL = CONTROL[0],

 CONTROL[i: 0..N] = (
  when i == 0 is_zero → CONTROL[i]
  | when i < N increment → CONTROL[i+1]
  | when i > 0 decrement → CONTROL[i-1] ).

/* Process MUSEUM models the behavior of a museum where:
 *
```



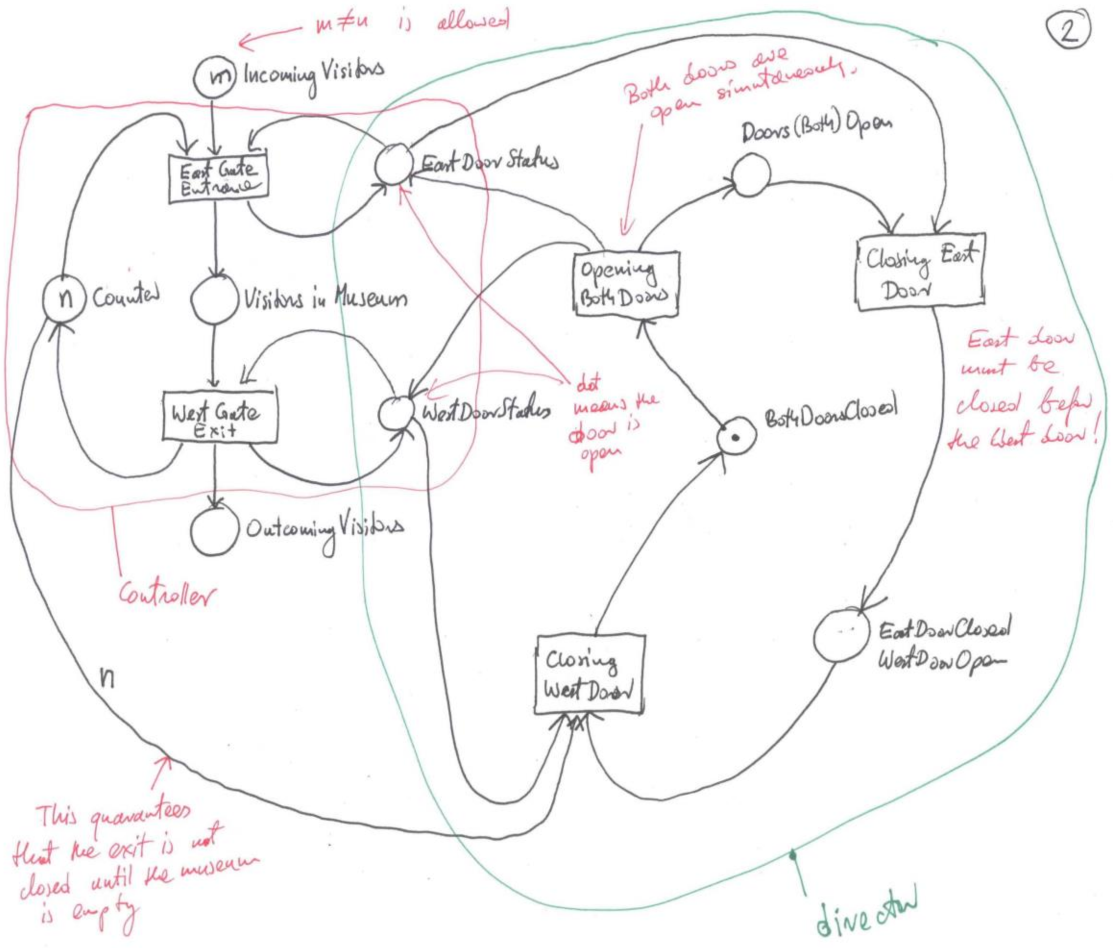
```
 * (i) It is assumed that people enters the museum through its east door
 * and that they leave the museum through its west door. */

||MUSEUM = ( DIRECTOR || east:DOOR || CONTROL || west:DOOR )
 /{
  east.cross/increment,
  west.cross/decrement,

  west.close/is_zero }.
```

c.[7] There are many solutions. Probably the simplest one is that what follows (disregard 2 in a circle in the right top corner). Some important points.

(i) We assume that m≠n, where n is the museum capacity and m is the number of potential visitors. Assume that the entrance is open if the museum is not full and the East door status is 'open'.

(ii) Both doors can be (and actually are in this solution) opened simultaneously, but closing must be in the order East → West.

(iii) Exit cannot be closed if there is a visitor in the museum.

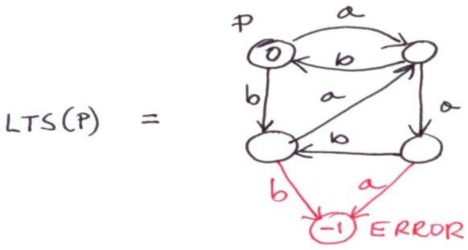


d.[5] Java solutions are not provided.

5.[5] Consider the safety property:

```
property P = ( a -> P1 | b -> a -> P1)
         P1 = ( a -> b -> a -> P1 | b -> P )
```

Provide LTS generated by the property *P* and a standard process *SP* such that LTS(*P*)=LTS(*SP*).
Solution:



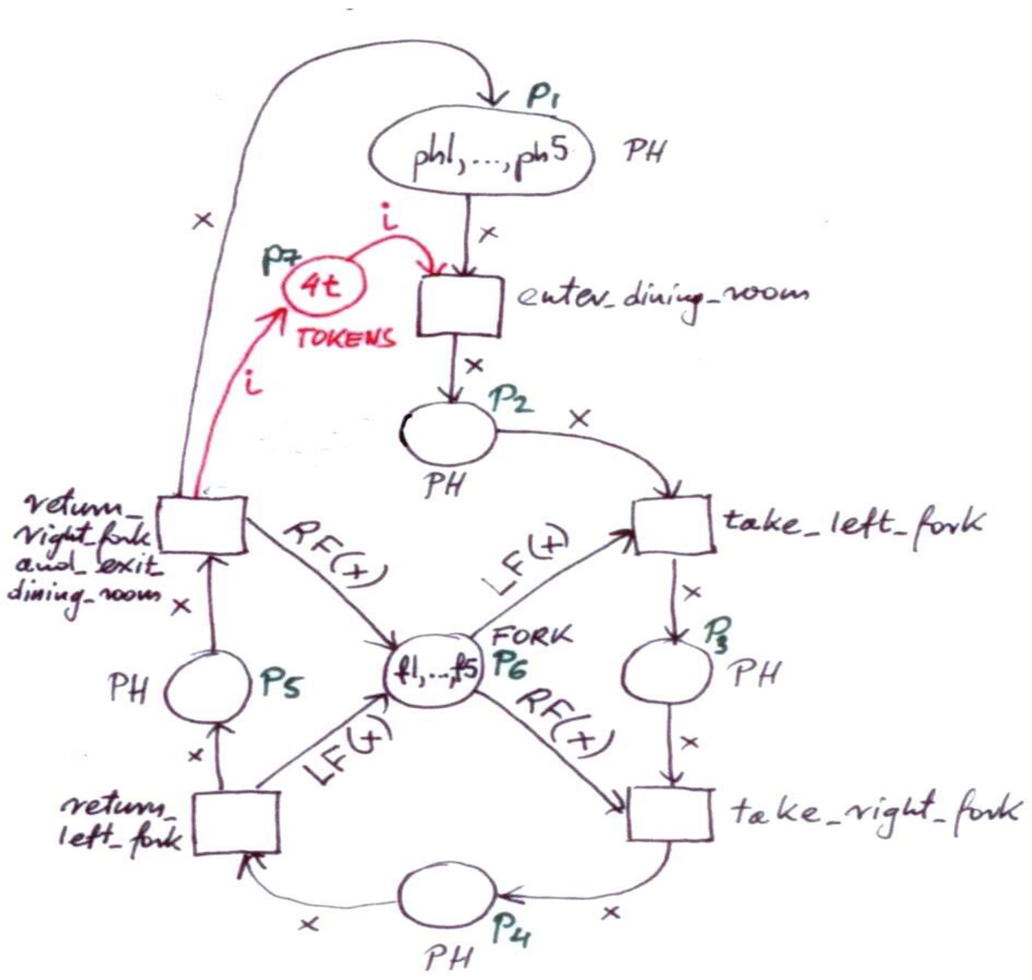
```
SP = ( a -> P1 | b -> ( a -> P1 | b -> ERROR))
P1 = ( a -> ( b -> (a -> P1| b -> ERROR) | a -> ERROR) | b -> P)
```

6.[10] Provide a Petri nets solution to the Dining Philosophers with a Butler (Lecture Notes 9, pages 14-15). Any kind of Petri net is allowed, however a Coloured Petri Nets give probably the most elegant and simplest solution.
Solutions:

One solution is just to translate FSP solution from Lecture Notes 9 into Elementary Petri Nets. But the result is a rather crowded net. The most elegant solution is when Coloured Petri Nets are used. The solution is just an adaptation (adding 'butler' or 'counter') of the solution from Lecture Notes 9, page 23. 'Butler' is just counter to zero from four (4, 3, 2, 1, 0), and zero blocks the entrance to the dining room.

The 'Butler' is implemented by red part. The place p_7 initially contains 4 tokens and one execution of the transition *enter_dining_room* delete on token from p_7, while ane execution of *return_right_fork_and_exit_dining_room* adds one token to p_7. When p_7 is empty the transition *enter_dining_room* cannot be executed, so only 4 philosophers can be in the dining room.

colour PH = with ph1 | ph2 | ph3 | ph4 | ph
colour FORK = with f1 | f2 | f3 | f4 | f5
colour TOKENS = with t
var x : PH
var i: TOKENS
fun LF x = case of ph1 ⇒ f2 | ph2 ⇒ f3 | ph3 ⇒ f4 | ph4 ⇒ f5 | ph5 ⇒ f1
fun RF x = case of ph1 ⇒ f1 | ph2 ⇒ f2 | ph3 ⇒ f3 | ph4 ⇒ f4 | ph5 ⇒ f5



Interpretation of places:
p_1 - thinking room
p_2 - philosophers without forks in the dining room
p_3 - philosophers with left forks in the dining room
p_4 - philosophers that are eating
p_5 - philosophers that finished eating and still with right forks   in the dining room
p_6 - unused forks
p_7 - butler or counter

7.[10] Specify a *safety property* for the car park problem of Lecture Notes 7 or Chapter 5 of the textbook, which asserts that the car park does not overflow.
Also specify a *progress property* which asserts that cars eventually enter the car park.
If car departure is *lower priority* than car arrival, does starvation occur?

Solution:

```
CARPARKCONTROL(N=4) = SPACES[N],
SPACES[i:0..N] = (when(i>0) arrive->SPACES[i-1]
            |when(i<N) depart->SPACES[i+1]
            ).

ARRIVALS   = (arrive->ARRIVALS).
DEPARTURES = (depart->DEPARTURES).

||CARPARK = (ARRIVALS||CARPARKCONTROL(4)||DEPARTURES).

property OVERFLOW(N=4) = OVERFLOW[0],
OVERFLOW[i:0..N] = (arrive -> OVERFLOW[i+1]
                   |depart -> OVERFLOW[i-1]
                   ).

||CHECK_CARPARK = (OVERFLOW(4) || CARPARK).

/* try safety check with OVERFLOW(3) */

progress ENTER = {arrive}

||LIVE_CARPARK = CARPARK >>{depart}.
```

8.[25] Consider the formulation of Smokers' Problem in plain English given in Lecture Notes 10, pages 5-7. The formulation of Dining Philosophers in the same style is in Lecture Notes 9 on page 7. A straightforward FSP model of Dining Philosophers is presented in Lecture Notes 9 on page 9 ('Hungry Simple Minded Philosophers')

a.[10] Provide a straightforward FSP model of Smokers similar to that of 'Hungry Simple Minded Philosophers'. In principle add supplier to the processes described on page 6 and represent the system using FSP. Use both the compact FSP notation (as upper part of page 9 of LN 9, above the horizontal line) and it expanded version (as lower part of page 9 of LN 9, below the horizontal line). The smoker with for example tobacco could be modelled by the process (but other solutions are also possible):

```
SMOKER_T=( get_paper -> get_match->roll_cigarrette ->
smoke_cigarrette ->SMOKER_T)
```

The resource ‘tobacco’ could be modelled for example by the process:

```
TOBACCO = ( delivered -> picked -> TOBACCO)
```

etc. If your solution deadlock, provide the shortest trace that lead to the deadlock, if not, provide some arguments why not.

b.[5] Write (safety) *property* process (syntax `property CORRECT_PICKUP = ...`) that verifies correct sequences of picking resources, i.e. picking up the paper by the smoker with tobacco must be followed by picking up match by the same smoker, picking up the tobacco by the process with paper must be followed by picking up match by the same smoker, and picking up tobacco by the smoker with matches must be followed by picking the paper by the same smoker.

Then compose `CORRECT_PICKUP` with your solution to (a) above and use the system provided by the textbook to verify if this safety property is violated.

c.[5] An elegant deadlock free solution to the Smokers can be constructed by applying ‘ask first, do later’ paradigm. Assume that the supplier informs smokers *explicitly* about the ingredient that is *not* supplied, for example it supplies paper, matches and a sign ‘no tobacco’. Each smoker reads the sign first and then start picking ingredients only if he has the ingredient that is mentioned on the sign.
Provide this solution using FSPs.

d.[5] Compose your solution from (c) with the property `CORRECT_PICKUP` from (b) and use the system provided by the textbook to verify if this safety property is *not* violated.

a) A possible solution that does not semaphores explicitly. Direct translation of page 7 from LN10 is also a feasible solution.

```
SMOKER_T=( get_paper -> get_match->roll_cigarrette -> smoke_cigarrette ->
      SMOKER_T)
SMOKER_P=( get_tobacco -> get_match->roll_cigarrette -> smoke_cigarrette ->
      SMOKER_P)
SMOKER_M=( get_tobacco -> get_paper->roll_cigarrette -> smoke_cigarrette ->
      SMOKER_T)

TOBACCO = ( delivered -> picked -> TOBACCO )
PAPER = ( delivered -> picked -> PAPER )
MATCH = ( delivered -> picked -> MATCH )

AGENT_T = (can_deliver -> deliver_paper -> deliver_match -> AGENT_T )
AGENT_P = (can_deliver -> deliver_match -> deliver_tobacco -> AGENT_P )
AGENT_M = (can_deliver -> deliver_tobacco -> deliver_paper -> AGENT_M )
```

```
RULE = (can_deliver -> smoking_completed -> RULE )
```

```
SMOKERS = s_t:SMOKER_T || s_p:SMOKER_P || s_m:SMOKER_M
RESOURCES = {s_m,s_p}::TOBACCO || {s_t,s_m}::PAPER || {s_t,s_p}::MATCH
AGENT_RULE = {s_m,s_p,s_t}::RULE || {s_m,s_p}::AGENT_T || {s_m,s_t}::AGENT_P
            || {s_t,s_p}::AGENT_M
```

```
CIG_SMOKERS = (SMOKERS || RESOURCES || AGENT_RULE)/
      { s_t.get_paper/s_t.picked,s_m.get_paper/s_m.picked,
      s_p.get_paper/s_p.picked,
      s_t.deliver_paper/s_t.delivered,s_m.deliver_paper/s_m.delivered,
      s_p.deliver_paper/s_p.delivered,
        s_t.smoking_completed/s_t.smoke_cigarrette,
        s_m.smoking_completed/s_m.smoke_cigarrette,
        s_p.smoking_completed/s_p.smoke_cigarrette}
```

This is not the only solution. For example the processes TOBACCO, PAPER and MATCH can also be modelled as one RESOURCE, etc.

(b) The details of safety property depend on how `CIG_SMOKERS` has been defined, for our solution from the above, the simplest could look as follws:

```
property CORRECT_PICKUP = ( s_t.get_paper -> s_t.get_match -> CORRECT_PICKUP
                          |s_p.get_tobacco -> s_p.get_match -> CORRECT_PICKUP
                          |s_m.get_tobacco -> s_m.get_paper -> CORRECT_PICKUP)

FULL_CIG_SMOKERS = (SMOKERS || RESOURCES || AGENT_RULE || CORRECT_PICKUP)/
      { s_t.get_paper/s_t.picked,s_m.get_paper/s_m.picked,
      s_p.get_paper/s_p.picked,
      s_t.deliver_paper/s_t.delivered,s_m.deliver_paper/s_m.delivered,
      s_p.deliver_paper/s_p.delivered,
        s_t.smoking_completed/s_t.smoke_cigarrette,
        s_m.smoking_completed/s_m.smoke_cigarrette,
        s_p.smoking_completed/s_p.smoke_cigarrette}
```

(c)

```
SMOKER_T=( no_tobacco -> get_paper -> get_match->roll_cigarrette ->
      smoke_cigarrette -> SMOKER_T)
SMOKER_P=( no_paper -> get_tobacco -> get_match->roll_cigarrette ->
      smoke_cigarrette -> SMOKER_P)
SMOKER_M=( no_match -> get_tobacco -> get_paper->roll_cigarrette ->
      smoke_cigarrette -> SMOKER_T)

TOBACCO = ( delivered -> picked -> TOBACCO )
PAPER = ( delivered -> picked -> PAPER )
MATCH = ( delivered -> picked -> MATCH )

AGENT_T = (can_deliver -> no_tobacco ->deliver_paper->deliver_match->
AGENT_T)
AGENT_P = (can_deliver -> no_paper -> deliver_match->deliver_tobacco-
>AGENT_P)
```

```
AGENT_M = (can_deliver -> no_match ->  deliver_tobacco->deliver_paper-
>AGENT_M)

RULE = (can_deliver -> smoking_completed -> RULE )
___________________________________________________________________________

SMOKERS = s_t:SMOKER_T || s_p:SMOKER_P || s_m:SMOKER_M
RESOURCES = {s_m,s_p}::TOBACCO || {s_t,s_m}::PAPER || {s_t,s_p}::MATCH
AGENT_RULE = {s_m,s_p,s_t}::RULE || {s_m,s_p}::AGENT_T || {s_m,s_t}::AGENT_P
            || {s_t,s_p}::AGENT_M
___________________________________________________________________________

CIG_SMOKERS = (SMOKERS || RESOURCES || AGENT_RULE)/
      { s_t.get_paper/s_t.picked,s_m.get_paper/s_m.picked,
      s_p.get_paper/s_p.picked,
      s_t.deliver_paper/s_t.delivered,s_m.deliver_paper/s_m.delivered,
      s_p.deliver_paper/s_p.delivered,
        s_t.smoking_completed/s_t.smoke_cigarrette,
        s_m.smoking_completed/s_m.smoke_cigarrette,
        s_p.smoking_completed/s_p.smoke_cigarrette}
_____________________________________________________________________________
```

(d) This simple solution is not provided.