CS 3SD3. Sample solutions to the assignment 2.

Total of this assignment is 122 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 during class, office hours, or just slip under the door to my office. The deadline for a complaint is 2 weeks after the assignment is marked and returned.

1.[40] Consider a group of *k* philosophers. Each philosopher either think or eats cookies (one serving at a time) or drinks cola (one bottle at a time). The cookie dispenser has a capacity of *M* servings, and cola dispenser has a capacity of *N* bottles. When a dispenser is empty a philosopher waits until a servant refills it. If any dispenser is empty, the servant refills it with either *M* servings of cookies or *N* bottles of cola, whichever the case. Refilling must be done as soon as possible, so they have priority over dispensing cookies and cola. There is no partial refilling of the dispensers. Assume that initially, both dispensers are full.

(a)[10] Model the behaviour of the system as FSP processes. Write a safety property that when composed with your system will check if no cookies are dispensed when there is no cola 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 the freedom to define them to fit your solution. This is a comment for points (b), (c) and (d).

(b)[5] Model the behaviour of the system as an Elementary Petri net (see Lecture Notes 3).

(c)[5] Model the behaviour of the system as a Place/Transition Petri net (see Lecture Notes 9, pages 21, 22).

(d)[5] 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.

(f)[10] Implement the system in Java program.

Solutions:

```
PHIL = (think -> PHIL | get_cookie -> eat_cookie -> PHIL
                | get_cola -> drink_cola -> PHIL ).

const K = 3
range Phil = 1..K

|| PHILS = ( forall[i: Phil] phil[i]:PHIL ).

SERVANT = (fill_cookies -> SERVANT | fill_cola -> SERVANT ).

const M = 3
range Cookies = 0..M

COOKIES = COOKIES[0],
COOKIES[s: Cookies] = ( when (s > 0) get_cookie -> COOKIES[s-1]
                                    | fill_cookies -> COOKIES[M] ).

const N = 2
range Cola = 0..N

COLA = COLA[0],
COLA[s: Cola] = ( when (s > 0) get_cola -> COLA[s-1] | fill_cola -> COLA[N] ).

property SCOLA = SCOLA[N],
SCOLA[s: Cola] = (when (s==0) fill_cola-> SCOLA[N] ).

property SCOOKIES = SCOOKIES[M],
SCOOKIES[s: Cookies] = (when (s==0) fill_cookies-> SCOOKIES[M] ).

||PHIL_COLA_COOKIE = (PHILS || COOKIES || COLA || SERVANT || SCOLA ||
                SCOOKIES).
```

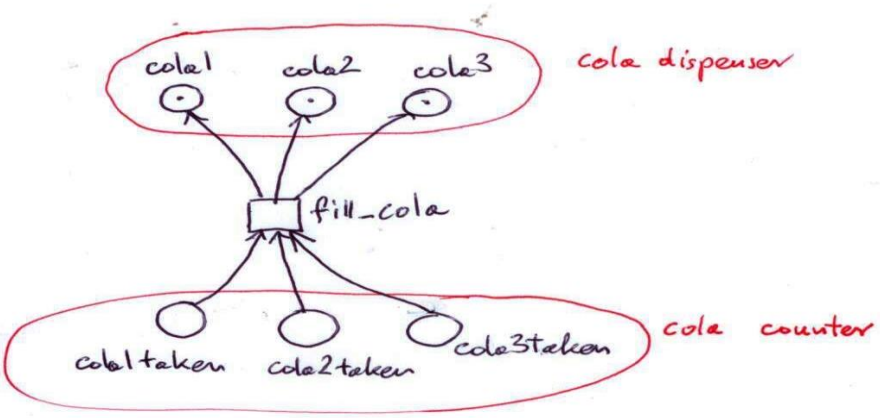
Comment on Petri nets solutions.

1. 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).

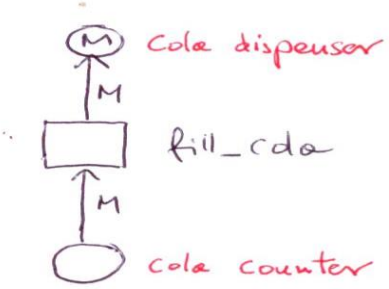
2. When modelling directly with Petri nets, the initial challenge could be how to model the fact that SERVANT can fill COOKIES and COLA only when they are empty. In principle this is a test for zero and standard Petri nets do not have it. Inhibitor nets have it, but they are equivalent to Turing Machines, 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 serving capacity of, say, COLA:



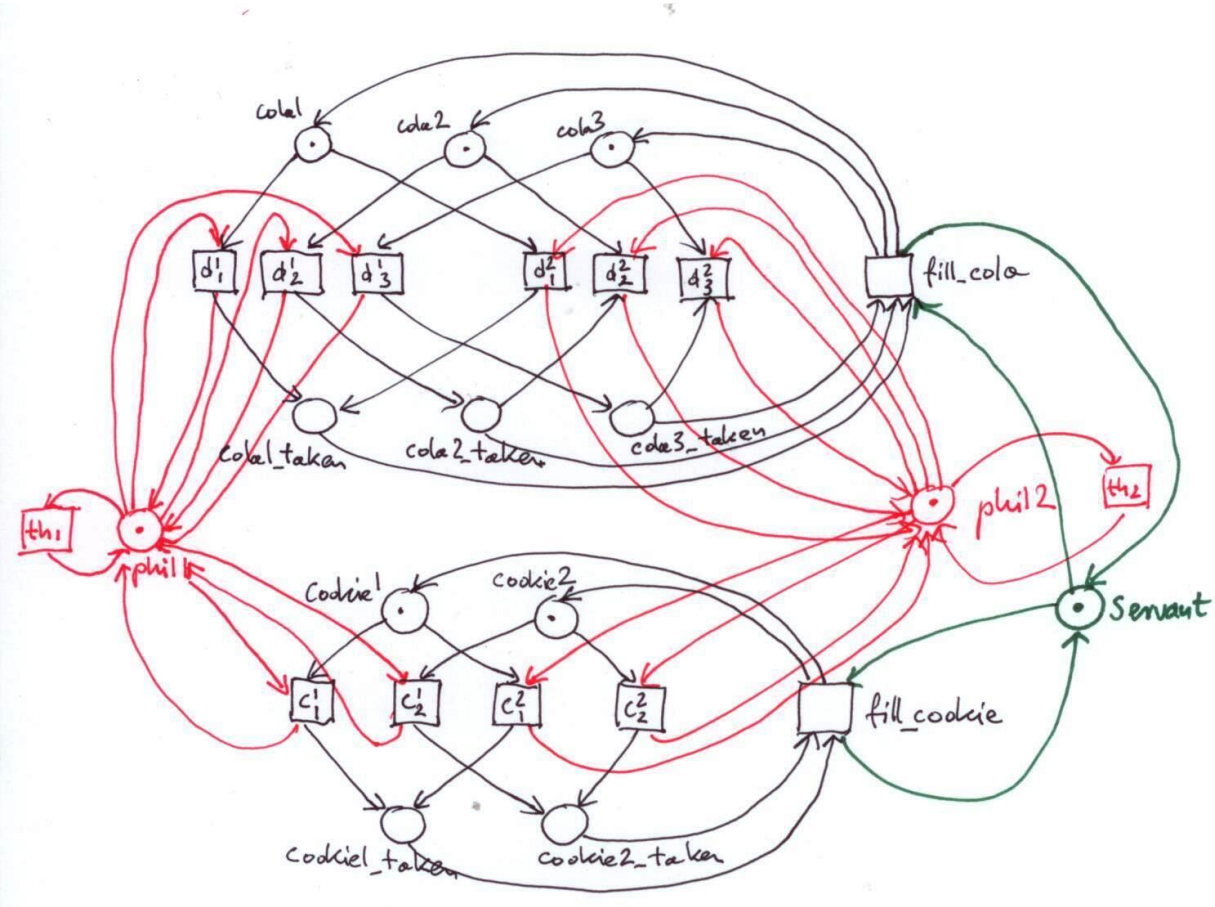
(ii) For Place/Transition nets it is even simpler and more intuitive. Let *M* be a number of servings in the cola dispenser.



There is a possibility of overfilling the pot 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.

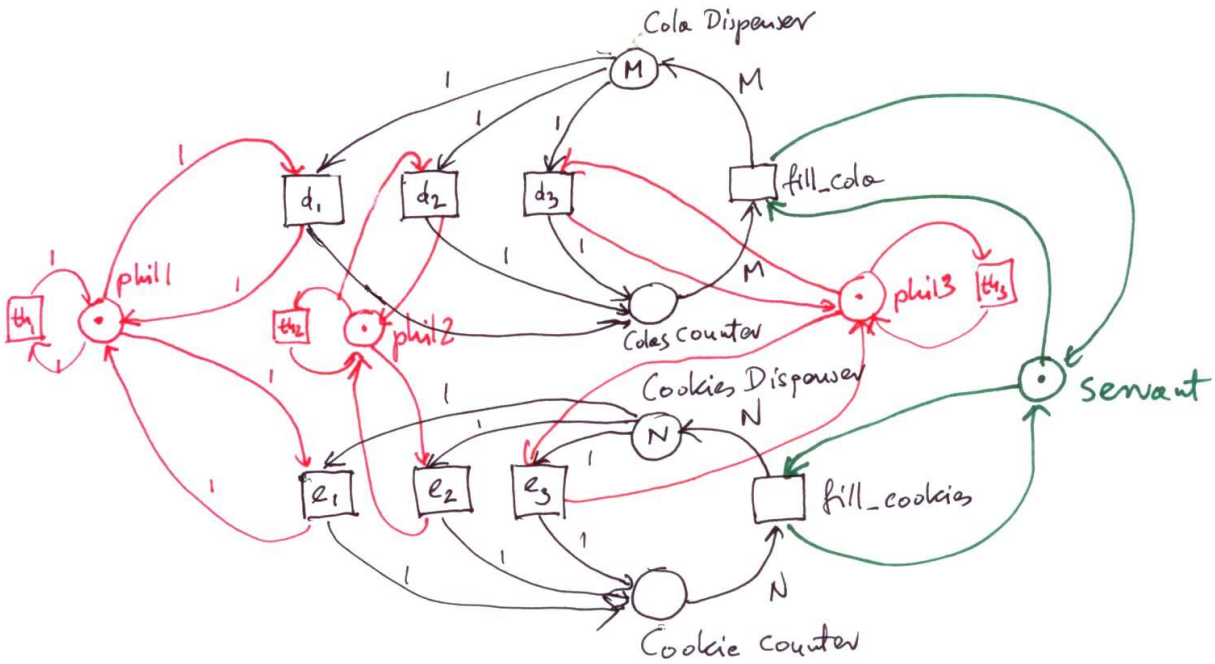
(a) A sample solution for 2 philosophers, 3 cola dispenser capacity and 2 cookies dispenser capacity. Philosophers are the red part of the net, servant is green.



d^i_j means *philosopher i* drinks *cola j*, e^i_j means *philosopher i* eats *cookie j*,

Note that servings numbers are *just names*, for example *serving*3 can be eaten by *savage*2 as the first step, also simultaneous eating, say *serving*1 by *savage*2 and *serving*3 by *savage*1 is allowed.

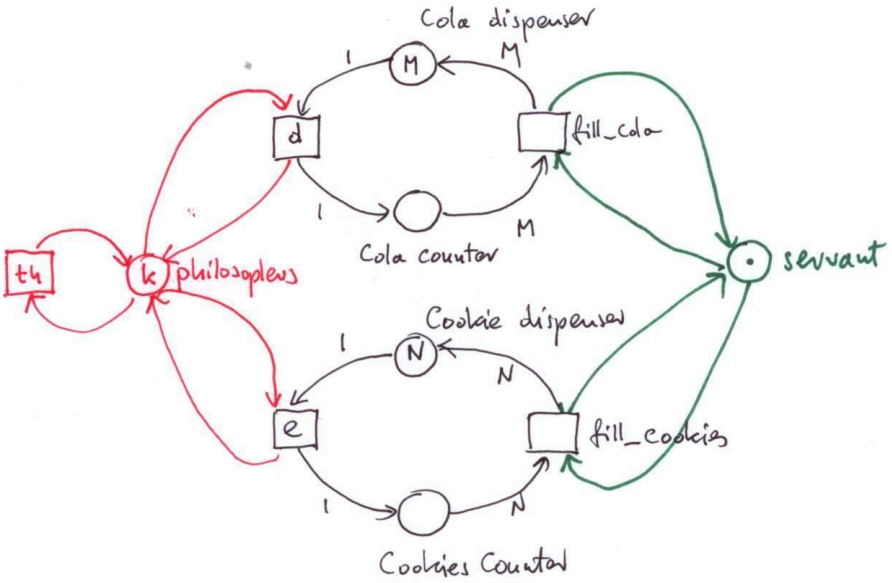
(c) A sample solution with Place/Transition nets for 3 philosophers, M cola dispenser capacity and N cookies dispenser capacity.



d_i means *philosopher* i drinks ***a cola***, e_i means *philosopher* i eats ***a cookie***, th_i means *philosopher* i is thinking.

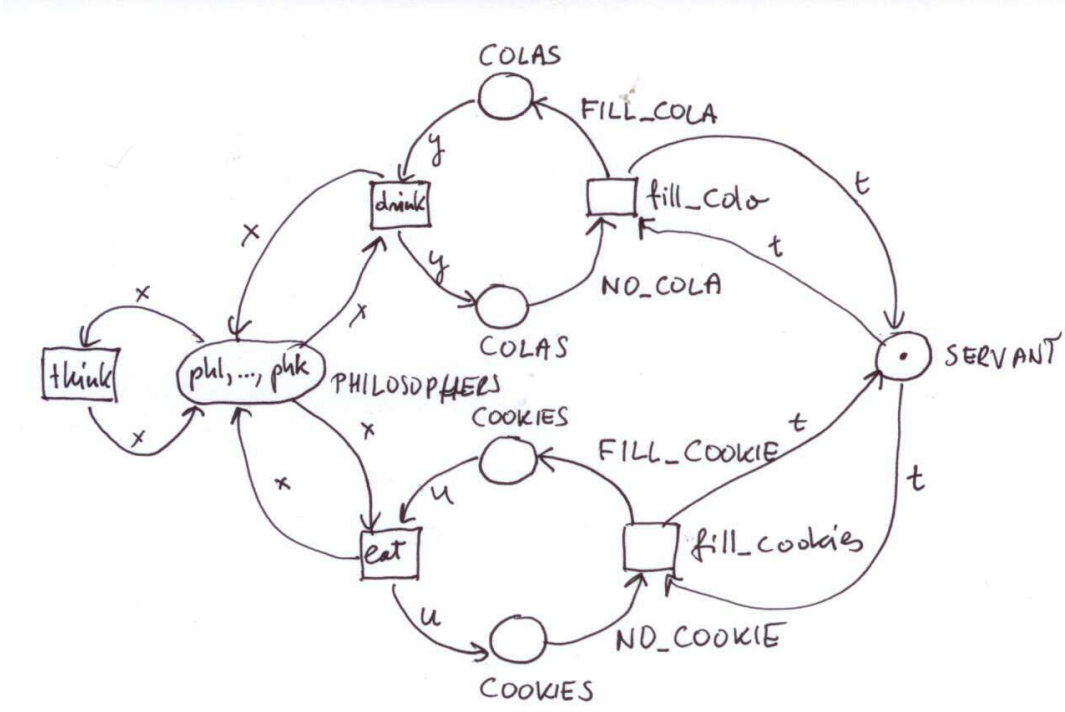
This solution is better as it makes servings not distinguishable, which I believe is the intention of the problem.

If for some reasons making a distinction between philosophers is not important, for instance only the behaviour of dispensers is what we are looking for, the following Place/Transition nets models k philosophers and M cola dispenser capacity and N cookies dispenser capacity (simultaneous execution of *e* with itself is allowed here, and the same for *d*!)



(d) Coloured Petri nets.

```
colour PHILOSOPHERS = with ph1 | ph2 | ... | phk
colour SERVANT = with serv
colour COLAS = Integers
colour COOKIES = Integers
var x: PHILOSOPHERS
var t: SERVANT
var y,z :COLAS
var u,v :COOKIES
fun FILL_COLA z =M
fun NO_COLA z=M
fun FILL_COOKIE z =M
fun NO_COOKIE z=M
```



(e) 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.

(f) Java solution is not provided.

2.[17] A museum allows visitors to enter through the east entrance and leave through its west exit. Arrivals and departures are signalled to the museum controlled 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.

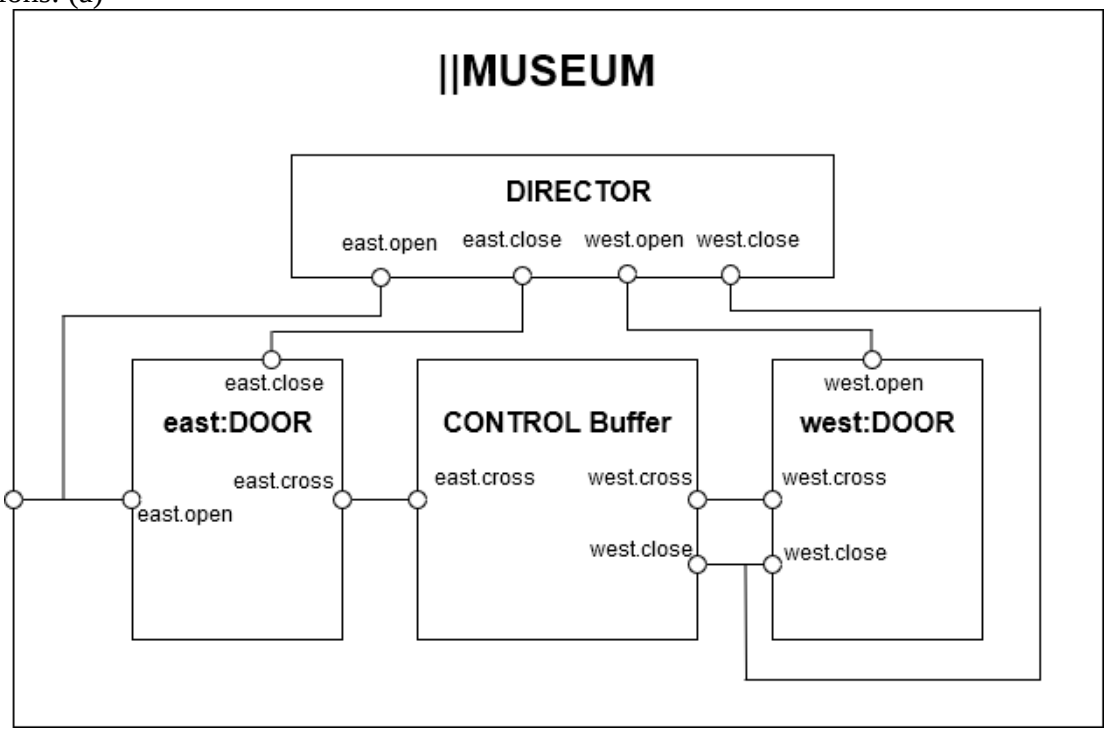
For Process Algebra models (as FSP), the museum system consists of processes EAST, WEST, CONTROL and DIRECTOR.

(a)[7] Draw the structure diagram for the museum and provide an FSP description for each of the processes and the overall composition.

(b)[5] Model the above scenario with Petri nets (any kind, your choice)

(c)[5] Provide Java classes that implement each one of the above FSP processes.

Solutions: (a)



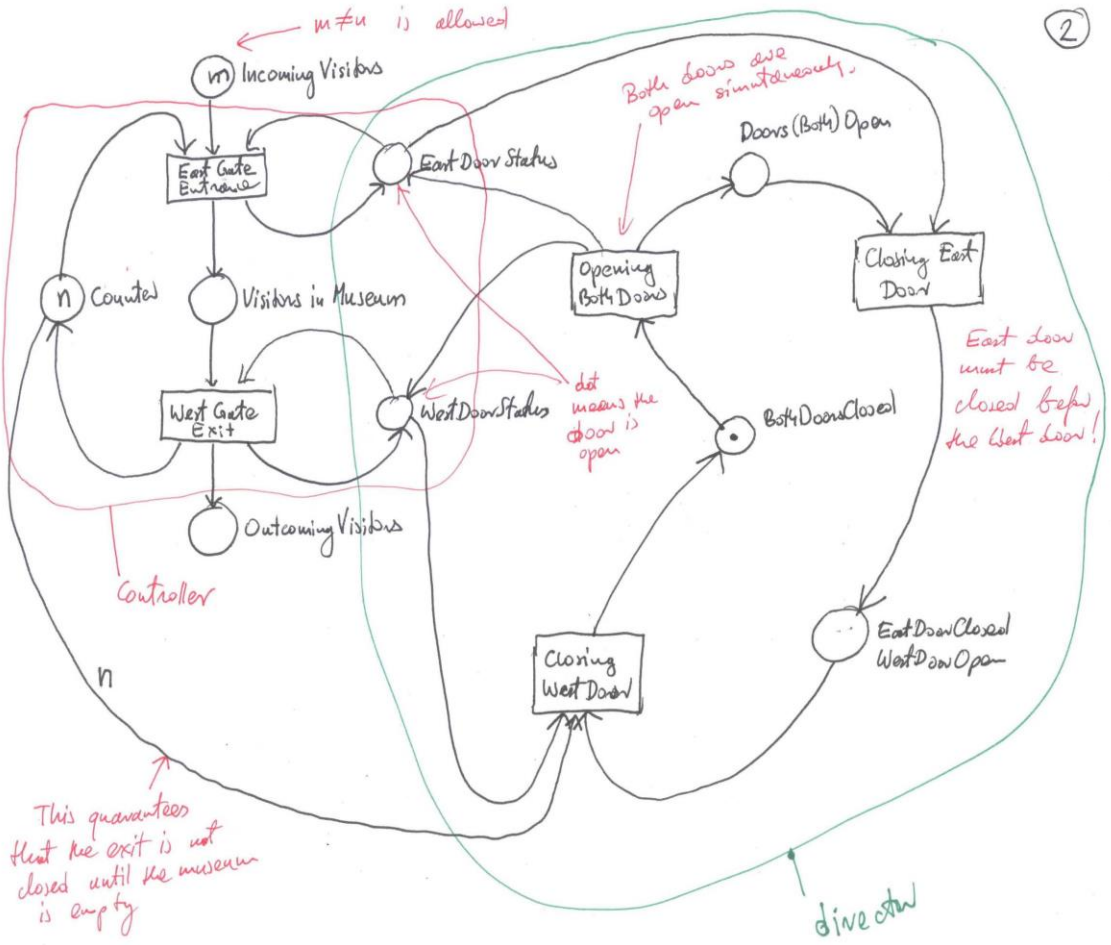
```
/* Question 4: Museum */

/* 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 }.
```

(b) 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.



(c) Java solutions are not provided.

3.[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}.
```

Starvation won't occur when car departure has lower priority than car arrival.

4.[15] For 'The Dining Philosophers Problem', simultaneous picking up of both forks is an abstraction of a general rule that 'the act of picking up both forks is atomic'. In other words, an order 'pick right', 'pick left' is arbitrary but once the process of picking starts, it cannot be interrupted. In practice quite often *conceptual simultaneity* is implemented as *atomicity*.
Provide a solution with FSP for Dining Philosophers with 'atomic act of (sequential) picking up both forks'.

Solution. A possible idea is to use a concept of 'reservation' ('ask first do later'), and use the classical net solution from page 18 of LN9 and FSP from page 19 of LN9 as a guidance. This is not the only solution, but probably the simplest one.

```
FORK = ( reserve_right -> take_right -> put_right -> FORK
       | reserve_left -> take_left -> put_left -> FORK )

PHIL = (think -> reserve_forks -> USE_FORKS)
USE_FORKS = ( take_right -> take_left -> eat -> PUT_FORKS
            | take_left -> take_right -> eat -> PUT_FORKS )
PUT_FORKS = ( put_left -> put_right -> PHIL
            | put_right -> put_left -> PHIL )


||DINERS(N=5) = ( forall[i:1..N]
  ( phil[i]:PHIL || {phil[i].right,phil[(i+1)%N].left}::FORK) )
/{
 reserve_forks.1/reserve_right.1,reserve_forks.1/reserve_left.2,
  reserve_forks.2/reserve_right.2,reserve_forks.2/reserve_left.3,
 reserve_forks.3/reserve_right.3,reserve_forks.3/reserve_left.4,
  reserve_forks.4/reserve_right.4,reserve_forks.4/reserve_left.5,
 reserve_forks.5/reserve_right.5,reserve_forks.5/reserve_left.1}
```

5.[5] Provide a Petri Nets solution (any kind of nets can be used) to asymmetric Dining Philosophers discussed in Lecture Notes 9 page 13 (or Chapter 6.2.2 of the textbook).

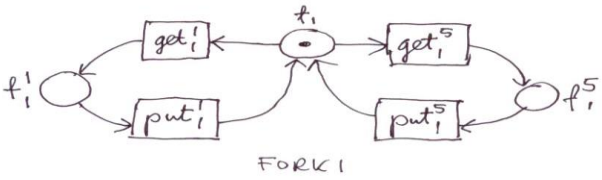
Solution.
A solution with Elementary Petri Nets is simple but the resulting graph is rather big. We may start with FSP solution in explicit extended form as the one for 'Hungry, Simple Minded Philosophers' on page 9 of LN9.

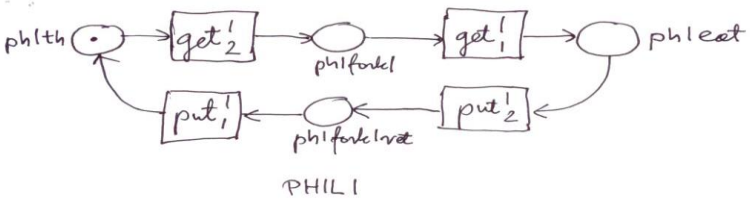
Assuming that for philosopher #i the right fork is fork #i and the left fork is fork #(i⊕1) we have:

- Notation: for get_j^i, put_j^i, i - philosopher number, j - fork number

$$FORK_1 = (get_1^1 \rightarrow put_1^1 \rightarrow FORK_1 \mid get_1^5 \rightarrow put_1^5 \rightarrow FORK_1)$$
$$FORK_2 = (get_2^2 \rightarrow put_2^2 \rightarrow FORK_2 \mid get_2^1 \rightarrow put_2^1 \rightarrow FORK_2)$$
$$FORK_3 = (get_3^3 \rightarrow put_3^3 \rightarrow FORK_3 \mid get_3^2 \rightarrow put_3^2 \rightarrow FORK_3)$$
$$FORK_4 = (get_4^4 \rightarrow put_4^4 \rightarrow FORK_4 \mid get_4^3 \rightarrow put_4^3 \rightarrow FORK_4)$$
$$FORK_5 = (get_5^5 \rightarrow put_5^5 \rightarrow FORK_5 \mid get_5^4 \rightarrow put_5^4 \rightarrow FORK_5)$$
$$PHIL_1 = (think_1 \rightarrow get_2^1 \rightarrow get_1^1 \rightarrow eat_1 \rightarrow put_2^1 \rightarrow put_1^1 \rightarrow PHIL_1)$$
$$PHIL_2 = (think_2 \rightarrow get_2^2 \rightarrow get_3^2 \rightarrow eat_2 \rightarrow put_2^2 \rightarrow put_3^2 \rightarrow PHIL_2)$$
$$PHIL_3 = (think_3 \rightarrow get_4^3 \rightarrow get_3^3 \rightarrow eat_3 \rightarrow put_4^3 \rightarrow put_3^3 \rightarrow PHIL_3)$$
$$PHIL_4 = (think_4 \rightarrow get_4^4 \rightarrow get_5^4 \rightarrow eat_4 \rightarrow put_4^4 \rightarrow put_5^4 \rightarrow PHIL_4)$$
$$PHIL_5 = (think_5 \rightarrow get_1^5 \rightarrow get_5^5 \rightarrow eat_5 \rightarrow put_1^5 \rightarrow put_5^5 \rightarrow PHIL_5)$$
$$\| DINERS = (FORK_1 \| \ldots \| FORK_5 \| PHIL_1 \| \ldots \| PHIL_5)$$

Now we may transform each individual FSP into an appropriate Elementary Petri Net. To simplify net solution (and make it more in 'net spirit'), we may model 'think' and 'eat' by places instead of transitions. For example the nets corresponding to $FORK_1$ and $PHIL_1$ may look as follows:

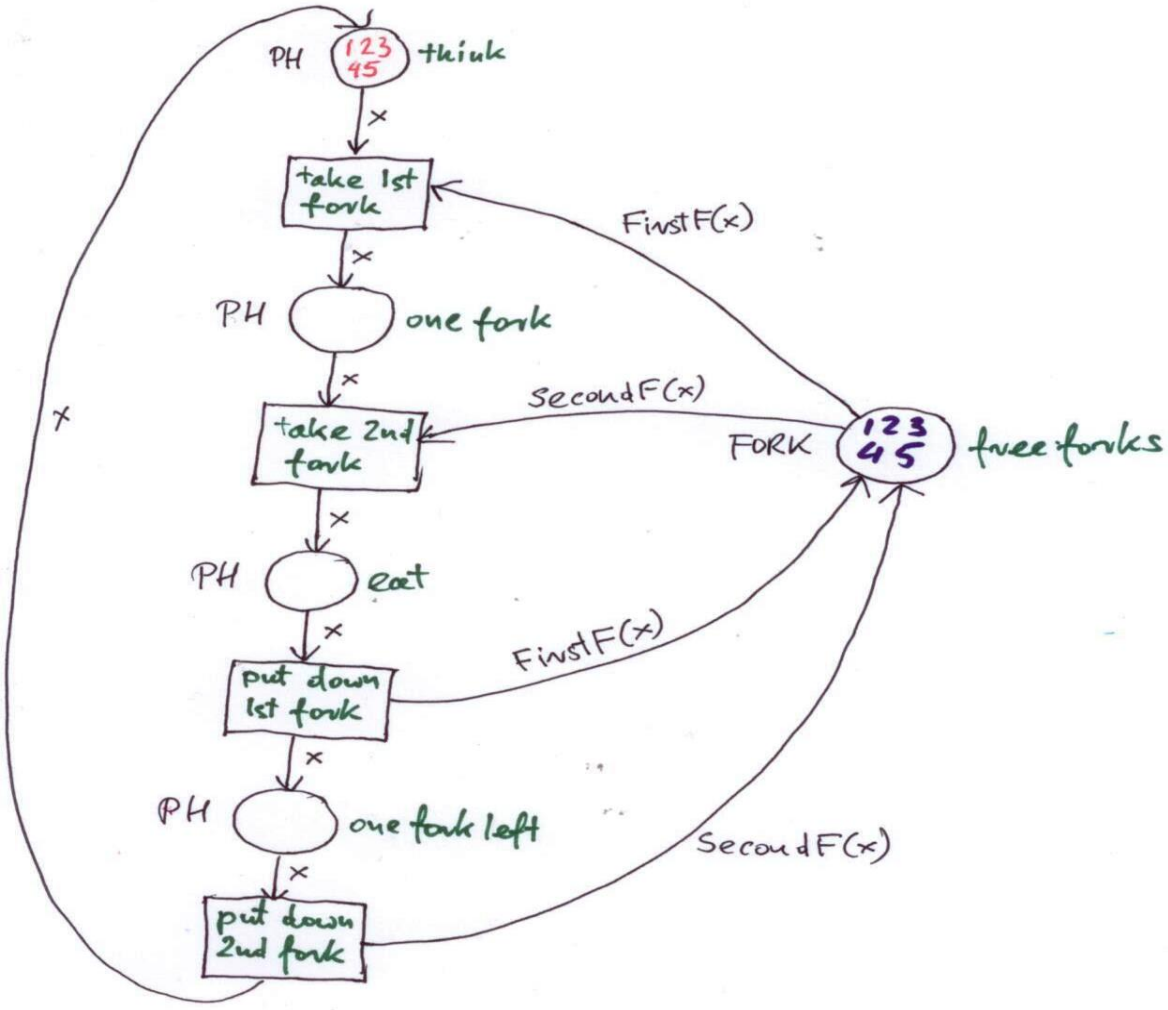




Now we just need to compose the nets for $FORK_1$, … , $FORK_5$, $PHIL_1$, …, $PHIL_5$, by gluing together the same actions. The solution fits one page but barely ☺

Solutions with Place/Transition Nets are the same as with Elementary Nets, still big.

With Coloured Petri Nets we may get a very simple and elegant solution in the style of the solution from page 23 of LN9.



colour PH = with ph1 | ph2 | ph3 | ph4 | ph5
colour Fork = with f1 | f2 | f3 | f4 | f5
FirstF : PH → FORK, SecondF : PH → FORK
FirstFR : PH → FORK, SecondFR : PH → FORK
var x: PH
'for philosophers 1, 3 and 5, left fork is first, for philosophers 2 and 4, right fork is first'
fun FirstF x = case of ph1 ⇒ f2 | ph2 ⇒ f2 | ph3 ⇒ f4 | ph4 ⇒ f5 | ph5 ⇒ f5
fun SecondF x = case of ph1 ⇒ f1 | ph2 ⇒ f3 | ph3 ⇒ f3 | ph4 ⇒ f3 | ph5 ⇒ f1
fun FirstFR x = case of ph1 ⇒ f2 | ph2 ⇒ f2 | ph3 ⇒ f4 | ph4 ⇒ f5 | ph5 ⇒ f5
fun SecondFR x = case of ph1 ⇒ f1 | ph2 ⇒ f3 | ph3 ⇒ f3 | ph4 ⇒ f3 | ph5 ⇒ f1

6.[5] A lift has a maximum capacity of ten people. In the model of the lift control system, passengers entering a 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 will check that the system never allows the lift that it controls to have more than ten occupants.

Solution.

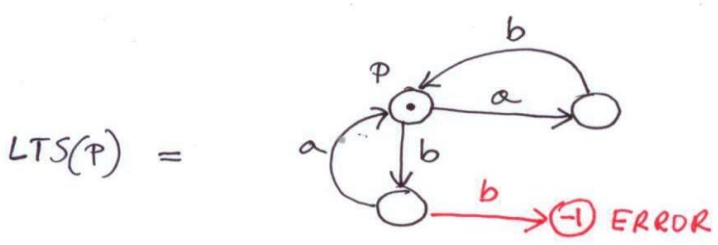
```
const N = 10

property LIFTCAPACITY = LIFT[0],
LIFT[i:0..10] = (when (i<10) enter ->LIFT[i+1]
              |when(i>0) exit  -> LIFT[i-1]).
```

7.[5] Consider the safety property:

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

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

Solution:




```
SP = (a -> (b -> SP | a -> SP) | b -> (a -> SP | b -> ERROR)).
```

8.[25] Consider the formulation of Smokers' A 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 a 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 leads 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 the 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 smokers are not so hungry for smoking and look on the table first, before making any movement. Then each smoker starts picking ingredients only if he has the ingredient that he does not see on the table. Otherwise, he waits patiently.
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.

Solutions.

(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) Simple solution is not provided.