SE 3BB4. Sample solutions to the assignment 2.

Total of this assignment is 101 pts. Each assignment is worth 10% of total.

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.[35] The dining savages: A tribe of savages eats communal dinners from a large pot capable of holding *M* servings of stewed missionaries. When a savage wants to eat, he helps himself from the pot, unless it is empty, in which case he waits until the cook refills the pot. If the pot is empty, the cook refill the pot with *M* servings.
The behavior of the savages and the cook is described by:

```
SAVAGE = ( get_serving -> SAVAGE ) .
COOK = ( fill_pot -> COOK ) .
```

(a)[5] Model the behaviour of the pot and of the system as FSP processes.
(b)[5] Model the behaviour of the pot as and of the system as an Elementary Petri net (see Lecture Notes 3).
(c)[5] Model the behaviour of the pot as and of the system as an Place/Transition Petri net (see Lecture Notes 9, pages 21, 22).
(d)[5] Model the behaviour of the pot as and of the system as an Coloured Petri net (see Lecture Notes 9, pages 23-34).
(e)[5] Discuss the differences between the FSP and the various Petri Net solutions.
(f)[10] Implement the system Java program.

(a)

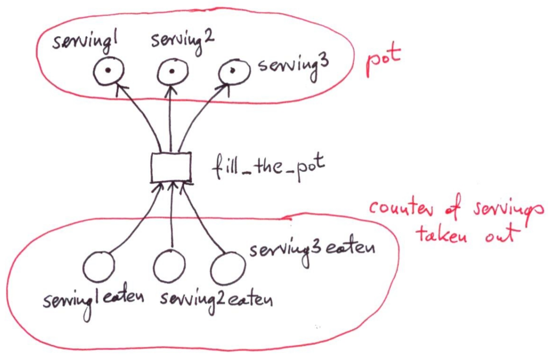
```
/* Question 2: The dining savages */

/* Process SAVAGE models the behavior of a savage where:
 *
 * (i) A savage is always eager to get a serving from the pot. */

SAVAGE = ( get_serving → SAVAGE ).

/* Process SAVAGES models a finite number of savages */

range Savages = 1..4

||SAVAGES = ( forall[i: Savages] savage[i]:SAVAGE ).

/* Process COOK models the behavior of a cook where:
 *
 * (i) a cook is always trying to fill the pot. */

COOK = ( fill_pot → COOK ).

/* Process POT models the behavior of a pot where:
 *
 * (i) The pot has an associated state  indicating the current number
 *   of stewed missionaries.
 * (ii) Initially it is assumed that the pot is empty and that
 * the maximum number of stewed missionaries is M.
 * (iii) action take_portion models a portion being taken from
 * the pot whenever possible.
 * (iv) action fill models the pot being filled. */

const M = 3
range Portions = 0..M

POT = POT[0],
POT[p: Portions] = (
 when ( p > 0 ) take_portion → POT[p-1]
 | fill → POT[M] ).

/* Dining savages */

||DINING_SAVAGES = ( SAVAGES || pot:POT || cook:COOK )
 /{
  savage[s: Savages].get_serving/pot.take_portion,
  cook.fill_pot/pot.fill
 }.
```

Comment on Petri nets solutions.

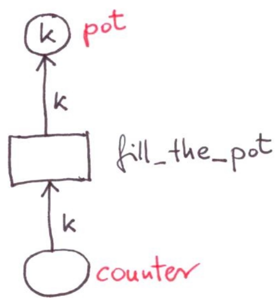
1. One obvious solution is just to mimic any FSP solution by replacing individual processes by their LTS and then merge common transitions. However such solution is not very readable and unnecessary complex (i.e. the resulting net is huge).

2. When modelling directly with Petri nets, the initial challenge could be how to model the fact that COOK can fill POT only when it is 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 cook does not look into the pot, he/she counts the servings taken out of the pot, he/she knows the pot capacity, so he/she can decide when the pot is empty by just counting. Of course this works only if initially the pot is full. And this can be modelled quite easily, as you can see below.

   (i) Elementary nets and 3 serving capacity



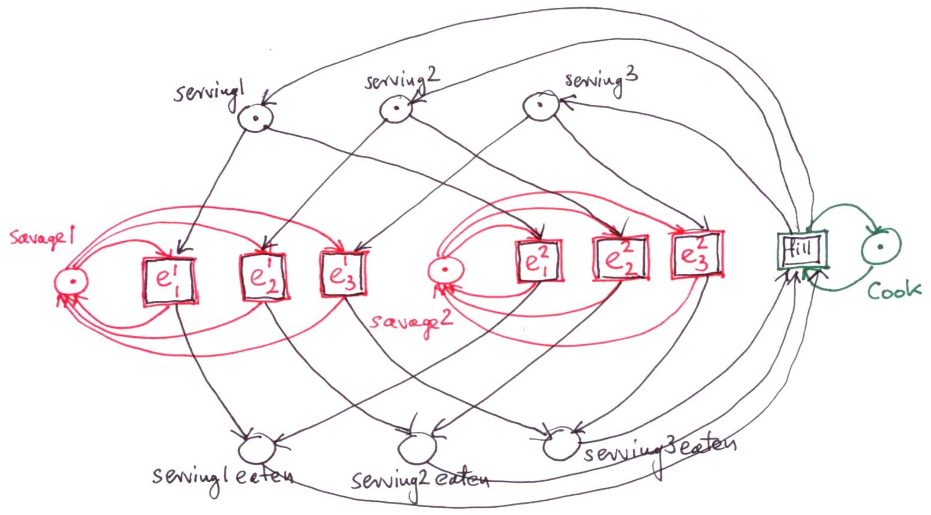
   (ii) For Place/Transition nets it is even simpler and more intuitive. Let *k* be a number of servings in the pot.



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.

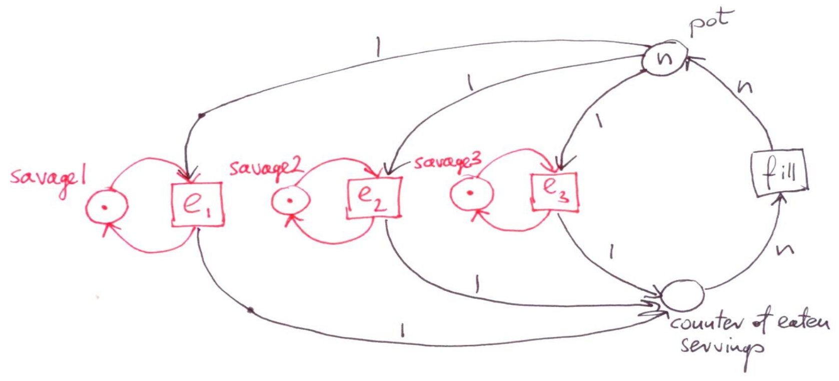
(b) A sample solution for 2 savages and 3 servings capacity of the pot. Savages are the red part of the net, the cook and the protocol are the black part of the net. If for some reasons the cook needs to be indicated separately, it is given by the green net.



e^i_j means *savage i* eats *serving 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.

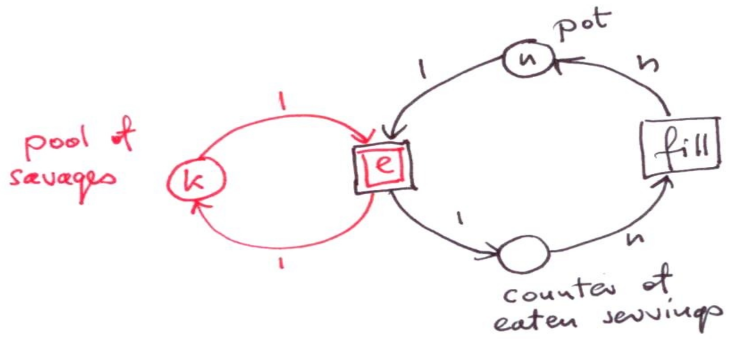
(c) A sample solution with Place/Transition nets for 3 savages and *n* servings capacity of the pot is the following:



e_i means *savage* i eats ***a serving.***

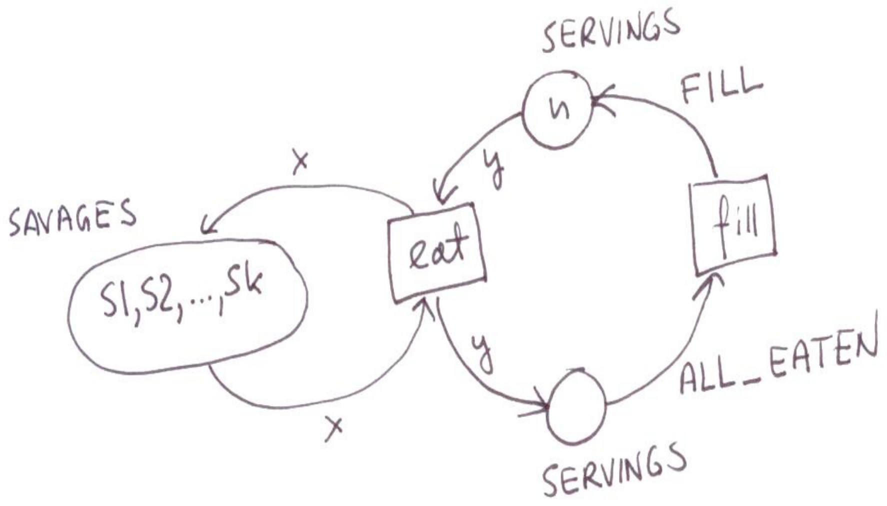
This solution is better as it makes servings not distinguishable, which I believe is the intention of the problem.

If for some reasons making a distinction between savages is not important, for instance only the behaviour of pot is what we are looking for, the following Place/Transition nets models k savages and n servings pot (simultaneous execution of e with itself is allowed here!)



(d) Coloured Petri nets.

colour SAVAGES = with s1 | s2 | ... | sk
colour SERVING = Integers
var x: SAVAGES
var y,z :SERVINGS
fun FILL z =n
fun ALL_EATEN z=n



(f) Java solutions will not be provided.

2.[16] 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.[6] 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 which implement each one of the above FSP processes.

(a)

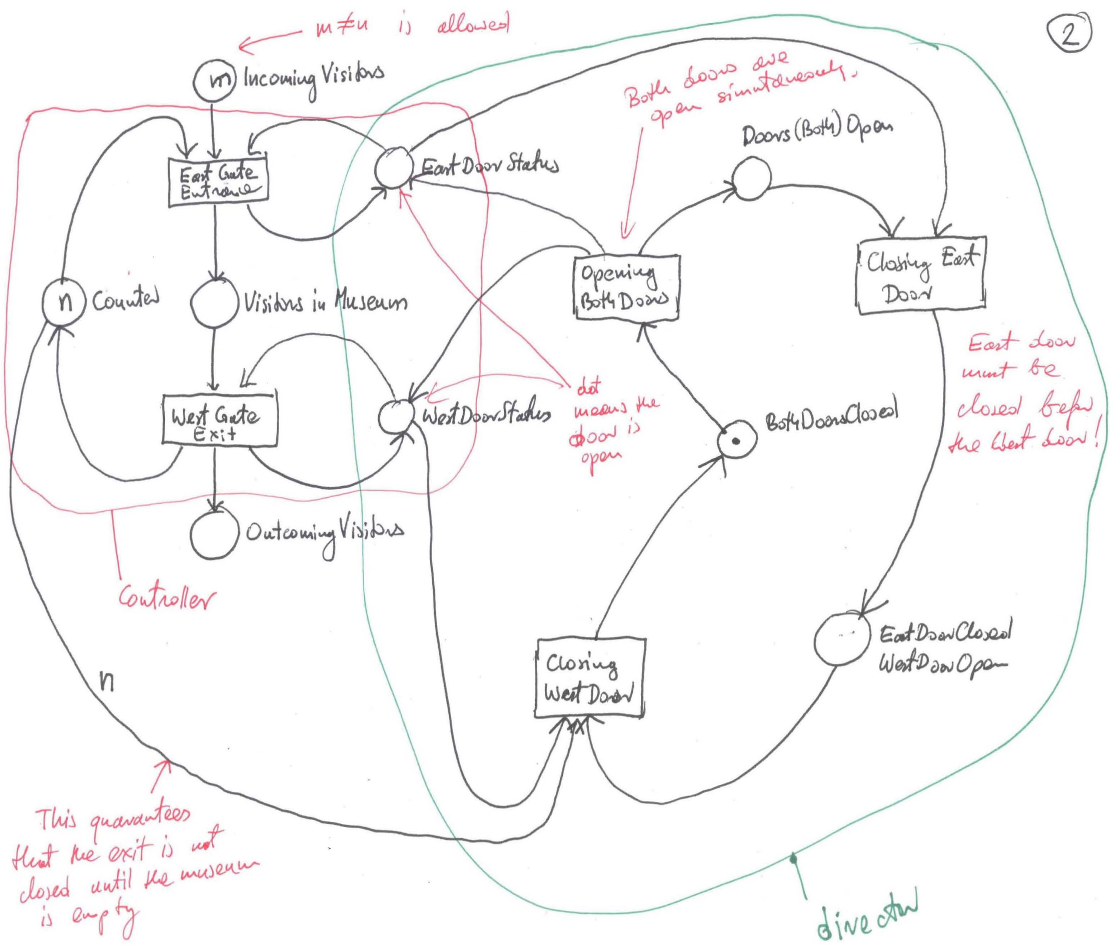
```
/* Question 4: Museum */

/* 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] A roller-coaster control system only permits its car to depart when it is full. Passengers arriving at the departure platform are registered with the roller-coaster controller by a turnstile. The controller signals the car to depart when there are enough passengers on the platform to fill the car to its maximum capacity of *M* passengers. Ignore the synchronization detail of passengers embarking from the platform and car departure. The roller-coaster consists of three processes: *TURNSTILE*, *CONTROL* and *CAR*. *TURNSTILE* and *CONTRO*L interact by the shared action *passenger* indicating an arrival and *CONTROL* and *CAR* interact by the shared action *depart* signalling the car departure.

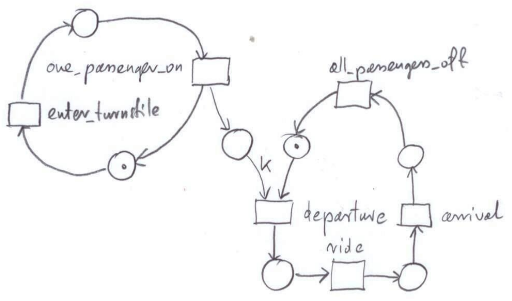
(a)[5] Provide FSP description for each process and the overall composition.

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

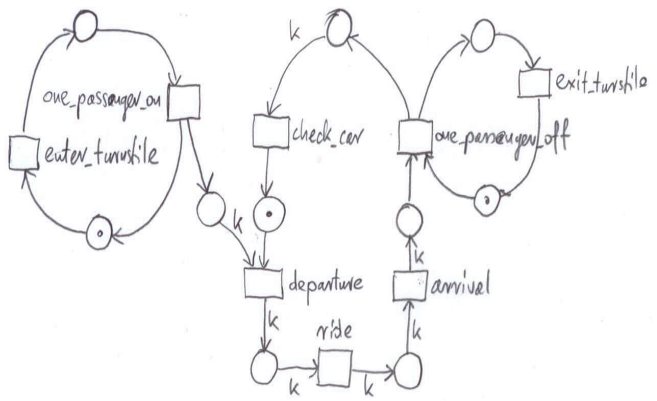
(a)

```
/* Question 7 */

/* The following processes model the behavior of a roller coaster. */

/* Process turnstile models the arrival of passengers to the roller coaster. */

TURNSTILE = (
 passenger → TURNSTILE ).

/* Process control models a control system for the roller coaster.
 *
 * The idea is that control will keep track of the number of passengers
 * in the roller coaster car. Its behavior is described as follows:
 *
 * (i) action 'arrive' initializes to 0 the internal counter for control.
 * Intuitively it models the arrival of an empty car to the platform.
 *
 * (ii) action 'passenger' signals a passenger jumping into the car.
 *
 * (iii) action 'depart' signals the departure of the roller coaster car is. The
 * condition on this actions implies that the car will depart only when full. */

const N = 3

CONTROL = CONTROL[0],

 CONTROL[n: 0..N] = (
  when n < N passenger → CONTROL[n+1]
  | when n == N depart → CONTROL[n]
  | arrive → CONTROL ).

/* Process car models the behavior of a roller coaster car.
 *
 * (i) action 'departs' signals the departure of the car from the platform
 * and the begining of the ride.
 *
 * (ii) action 'ride' models the roller coasters trip.
 *
 * (iii) action 'arrive' signals the arrival of the car to the platform. */

CAR = ( depart → ride → arrive → CAR ).

/* It is assumed that there is only one car in the roller coaster.
 *
 * Moreover, it is assumed that intially there is a car waiting for
 * passengers and ready to start the ride.
 *
 * Passengers getting off the roller coaster car is modeled as a single action. */

||ROLLER_COASTER = ( TURNSTILE || CONTROL || CAR ).
```

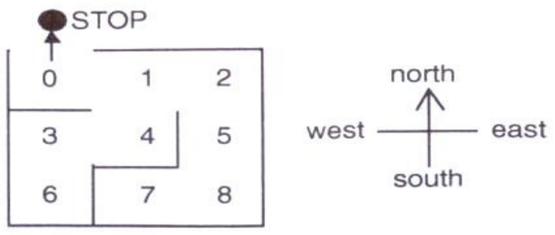
(b) Place/Transition nets are the best options, however just a plain translation of FSP solution is also OK. If we can model getting off from the car as one action the below simple P/T-net is a nice solution (k is the car capacity):



If we want to model getting off the car one by one, the net below is a possible solution:



4.[5] The figure below depict a maze.



Write a description of the maze in FSP, which using deadlock analysis provided by LTSA, finds the shortest path out of the maze starting at any square.
(*Hint*) At each numbered square in the maze, a directional action can be used to indicate an allowed path to another square.

Solution:

```
MAZE(Start=8) = P[Start],
P[0] = (north->STOP|east->P[1]),
P[1] = (east ->P[2]|south->P[4]|west->P[0]),
P[2] = (south->P[5]|west ->P[1]),
P[3] = (east ->P[4]|south->P[6]),
P[4] = (north->P[1]|west ->P[3]),
P[5] = (north->P[2]|south->P[8]),
P[6] = (north->P[3]),
P[7] = (east ->P[8]),
P[8] = (north->P[5]|west->P[7]).

||GETOUT = MAZE(7).
```

5.[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)

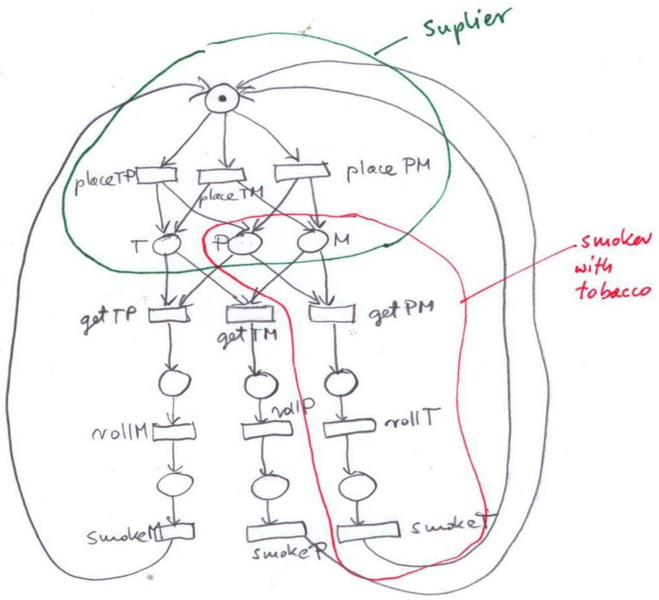
```
SMOKER_T=( no_tobacco -> get_paper -> get_match->roll_cigarrette ->
      smoke_cigarrette -> SMOKER_T)
SMOKER_P=( no_paper -> get_tobacco -> get_match->roll_cigarrette ->
      smoke_cigarrette -> SMOKER_P)
SMOKER_M=( no_match -> get_tobacco -> get_paper->roll_cigarrette ->
      smoke_cigarrette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

6.[10] Pages 16-18 of Lecture Notes 9, provide a solution to the Dining Philosophers problem with Elementary Petri Nets assuming that simultaneous picking forks (i.e. resources) is allowed, while pages 23-24 provide a solution to the same problem using Coloured Petri Nets, also assuming simultaneous picking forks (i.e. resources is allowed).

a.[5] Provide a solution to Smokers problem with Elementary Petri Nets and assuming that simultaneous picking (and delivery) resources is allowed (just mimic the solution for Dining Philosophers).

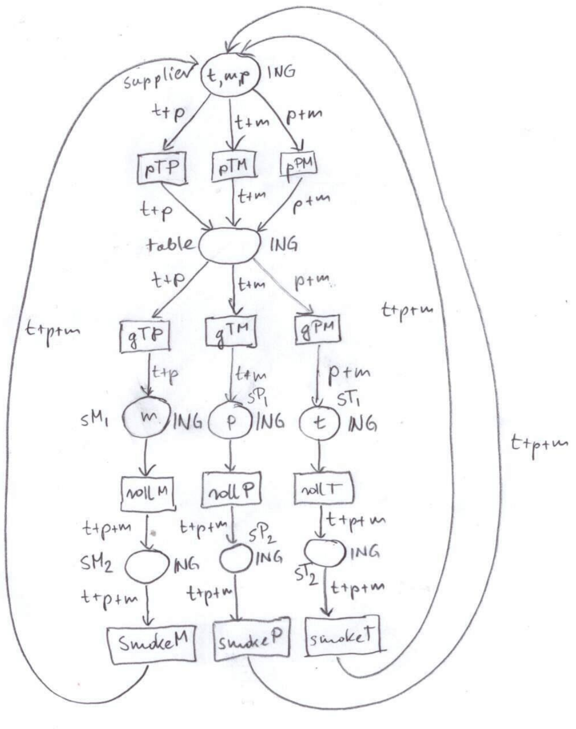
b.[5] Provide a solution to Smokers problem with Coloured Petri Nets and assuming that simultaneous picking (and delivery) resources is allowed (just mimic the solution for Dining Philosophers).

(a) Probably the simplest solution:



(b) This is one of the cases where Colour Petri Nets do not give too much simplification:

colour ING = with t | p | m


supplier
t,m,p
ING
t+p
t+m
p+m
pTP
pTM
pPM
table
ING
gTP
gTM
gPM
t+p+m
sM1
m
ING
SP1
p
ST1
t
rollM
rollP
rollT
SM2
SP2
ST2
smokeM
smokeP
smokeT