A cook puts burgers in a pot. A client checks if there is at least one burger in the pot, and if so, the client must take one. Assuming 2 clients:

POT ERROR CHECK

Fill.1 -> c1.check -> c2.check -> c1.get -> c2.get

POT MUTEX 2 clients cant choose same pot

range Burgers = 0..2 ↳ when check, must burger

CLIENT = (check -> get -> CLIENT).

POT = POT[0],
POT[p: Burgers] = (when p > 0 check -> POT[p]
| get -> POT[p-1]
| fill[n: Burgers] -> POT[n]).

LOCK = (acquire->check->release->LOCK).
||LOCKPOT = (LOCK || POT).

COOK = (fill[p: 1..2] -> COOK)+{fill[0]}.
||DS = (c1: CLIENT || c2: CLIENT || {c1,c2}::LOCKPOT || COOK)
/{ {c1, c2}.check/check, {c1, c2}.get/get }.

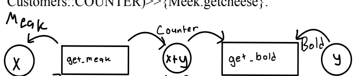
The cheese counter: There are bold customers who push their way to the front of the mob and demand services; and meek customers who wait patiently for service. Request for service is denoted by the action getcheese and service completion is signaled by the action cheese.

FSP of two bold customers and two meek:

set Bold = {bold[1..2]}
set Meek = {meek[1..2]}
set Customers = {Bold,Meek}
CUSTOMER = (getcheese->CUSTOMER).
COUNTER = (getcheese->COUNTER).
||CHEESE_COUNTER = (Customers:CUSTOMER || Customers::COUNTER).

Meek customers got lower priority than bold

||CHEESE_COUNTER = (Customers:CUSTOMER || Customers::COUNTER)>>{Meek.getcheese}.



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.

DINING SAVAGES MODEL

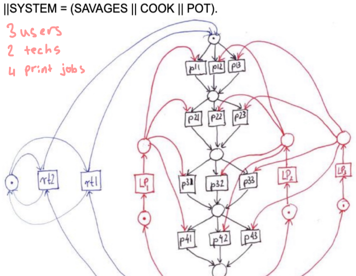
SAVAGE = (get_serving -> SAVAGE).
COOK = (fill_pot -> COOK).

const K = 3
range Savage = 1..K
||SAVAGES = (forall[i: Savage] savage[i]:SAVAGE).

const M = 3
range Servings = 0..M
POT = POT[0],
POT[s: Servings] = (when (s > 0) get_serving -> POT[s-1]
| when (s == 0) fill_pot -> POT[M]).

||SYSTEM = (SAVAGES || COOK || POT).

3 users
2 techs
4 print jobs



For Printer:
Savage = user
cook = technician
printer = pot
and fix rest

OPERATING SYSTEM BINARY SEMAPHORE

The user cannot access when the OS is off

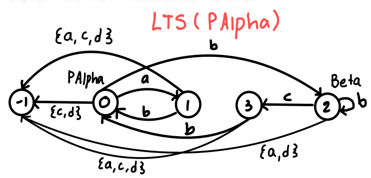
const Max = 1
range Int = 0..Max
SEMAPHORE(N=0) = SEMA[N],
SEMA[v:Int] = (up->SEMA[v+1]
|when(v>0) down->SEMA[v-1]),
SEMA[Max+1] = ERROR.

ACCESS = (down -> control_access -> up -> ACCESS).
||CONTROL = (user:ACCESS || system:ACCESS || {user,system}::SEMAPHORE(1)).

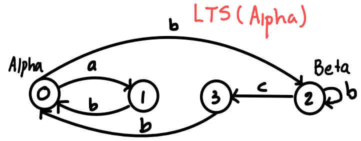
PROPERTY ALPHA AND ALPHA LTS

property Alpha = (a->b->Alpha | b->Beta)+{d}
Beta = (c->b->Alpha | b->Beta) AND
Alpha = (a->b->Alpha | b->Beta)+{d}
Beta = (c->b->Alpha | b->Beta)

LTS (PAlpha)



LTS (Alpha)

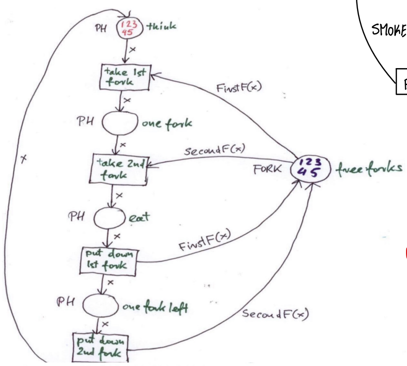


ALPHA1 = PROPERTY ALPHA FSP

ALPHA1 = (c->ERROR | d->ERROR | a->A1 | b->BETA),
A1 = (b->ALPHA1 | a->ERROR | c->ERROR | d->ERROR),
BETA = (b->BETA | c->B1 | a->ERROR | d->ERROR),
B1 = (b->ALPHA1 | a->ERROR | c->ERROR | d->ERROR).

DINING PHIL MODEL (SEQUENTIAL PICKING FORK)

FORK = (
reserve_right -> get_right -> put_right -> FORK
| reserve_left -> get_left -> put_left -> FORK).
PHIL = (think -> reserve_forks -> GET).
GET = (get_right -> get_left -> eat -> PUT),
PUT = (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))



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

Gas Station with N customers and M pumps

const N = 3
const M = 2
range C = 1..N
range P = 1..M
range A = 1..2 c
CUSTOMER = (prepay[a:A]->gas[x:A]->
if (x==a) then CUSTOMER else ERROR).
CASHIER = (customer[c:C].prepay[x:A]->start[P][c][x]->CASHIER). PUMP = (start[c:C][x:A] -> gas[c][x]->PUMP).
DELIVER = (gas[P][c:C][x:A]->customer[c].gas[x]->DELIVER).
||STATION = (CASHIER || pump[1..M]:PUMP || DELIVER) /
{pump[i:1..M].start/start[i],
pump[i:1..M].gas/gas[i]}.
||GASSTATION = (customer[1..N]:CUSTOMER ||STATION).

MODEL CHECKING

a i) φ = EG r: We have L(s0) = {r} and L(s2) = {q,r}. Clearly r ∊ s0, s2. So s0 |= φ HOLDS and s2 |= φ HOLDS

a ii) φ = G(r ∨ q): We have L(s0) = {r}, L(s1) = {p,t,r} L(s2) = {q,r} and L(s2) = {p,q}. Clearly r ∨ q ∊ s0, s2. Now s1 is reachable from s0, and s2 is reachable from s1, putting us in an infinite path where r ∨ q ∊ s1, s2. Furthermore, s3 is reachable from s0 (q ∊ s3), and s2 is reachable from s2, putting us in the same infinite path where r ∨ q ∊ s1, s2. So So s0 |= φ HOLDS and s2 |= φ HOLDS.

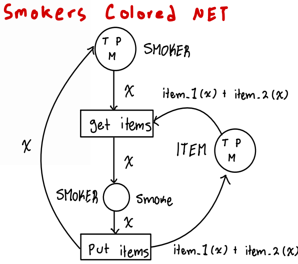
b)
"If the process is enabled infinitely often, then it runs infinitely often."
Let p:: "the process is enabled". q: "the process runs"
LTL: G(Fp ⇒ Fq)

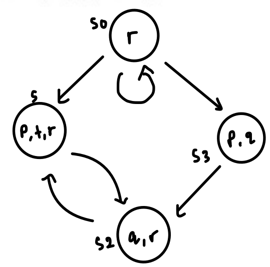
c)
"If the process is enabled infinitely often, then it runs infinitely often."
Let p:: "the process is enabled". q: "the process runs"
CTL: AG(EFp ⇒ EFq)

d)
"A passenger entering the elevator at 5th floor and pushing 2nd floor button, will never reach 6th floor, unless the 6th floor button is already lightened or somebody will push it, no matter if she/he entered an upwards or upward travelling elevator."
Atomic Predicates: predicates: floor=2, direction=up, direction=down, ButtonPres2, floor=6, etc.

LTL G((ButtonPres2 ∧ floor = 5) ⇒(¬(floor = 6) ∨ ButtonPress6))
CTL: AG((ButtonPres2 ∧ floor = 5) ⇒ (A[¬(floor = 6) ∨ (ButtonPress6)]))

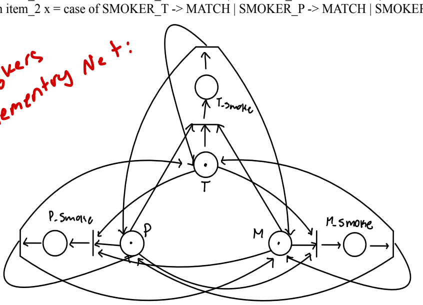
Smokers Colored NET





Colour SMOKER = with SMOKER_T | SMOKER_P | SMOKER_M
Colour ITEM = with TOBACCO | PAPER | MATCH
Var x: SMOKER
Fun item_1 x = case of SMOKER_T -> PAPER | SMOKER_P -> TOBACCO | SMOKER_M ->TOBACCO
Fun item_2 x = case of SMOKER_T -> MATCH | SMOKER_P -> MATCH | SMOKER_M ->PAPER

Smokers Elementry Net:



OFFICE PRINTER/TONER FSP

const J=3
range Jobs = 0..J

PRINTER = PRINTER [3],
PRINTER[j: Jobs] = (
when j==0 replace_toner->PRINTER[J]
|when j>0 print_job -> PRINTER[j-1]
).

USER = (print_job->USER).

const M = 2
range Users = 0..M

||USERS = (forall[i:Users] user [i]:USER).
TECHNICIAN=(replace_toner->TECHNICIAN).

||OFFICE=(USERS||PRINTER||TECHNICIAN)
/{user[Users].print_job/print_job}.

Description of intended behavior: any USER can print a job if the PRINTER has enough toner, if the printer is empty, then the TECHNICIAN comes to replace the toner.

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 to avoid the double-booking of seats while allowing clients to choose available seats freely.

```
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).
```

Midterm q5

Simplified Multidimensional Semaphores

Generic semaphore is defined in FSP's as:

SEM (N=INITIAL_CALUE) = SEMA[N]
SEMA[v:int] = (when (v<= MAX) up -> SEMA[v+1]
|when(v>0) down -> SEMA[v-1])

Then a Simplified Multidimensional Semaphores over variables S1, S2 and maximal values of S1,S2, equal to 3, can be modelled by FSPs as follows:

SEMS(INITIAL1=3,INITIAL2=3) = (S1:SEM(3) || S2:SEM(3))
/{S1.S2.up/S1.up, S1.S2.up/S2.up,
S1.S2.down/S1.down, S1.S2.down/S2.down}

SEMADOMO MULTIDIMENSIONAL SEMAPHORE

const Max = 3
range Int = 0..Max

SEMAPHORE(N=0) = SEMA[N],
SEMA[v:Int] = (up->SEMA[v+1]
|when(v>0) down->SEMA[v-1]
).

LOOP = (mutex.down -> critical -> mutex.up -> LOOP).

||SEMADEMO = (p[1..3]:LOOP ||
{p[1..3]}::mutex:SEMAPHORE(1)).

LTL

Φ ::= ⊥ | ⊤ | p | (¬Φ) | (Φ ∧ Φ) | (Φ ∨ Φ) | (Φ ⇒ Φ) |
(GΦ) | (FΦ) | (XΦ) | (Φ U Φ) | (Φ W Φ) | (Φ R Φ)

where p ranges over atomic formulas/descriptions.

- ⊥ - false, ⊤ - true
- GΦ, FΦ, XΦ, Φ U Φ, Φ W Φ, Φ R Φ are **temporal connections**.
- X means "neXt moment in time"
- F means "some Future moments"
- G means "all future moments (Globally)"
- U means "Until"
- W means "Weak-until"
- R means "Release"

CTL

Φ ::= ⊥ | ⊤ | p | (¬Φ) | (Φ ∧ Φ) | (Φ ∨ Φ) | (Φ ⇒ Φ) |
AXΦ | EXΦ | A[ΦUΦ] | E[ΦUΦ] |
AGΦ | EGΦ | AFΦ | EFΦ

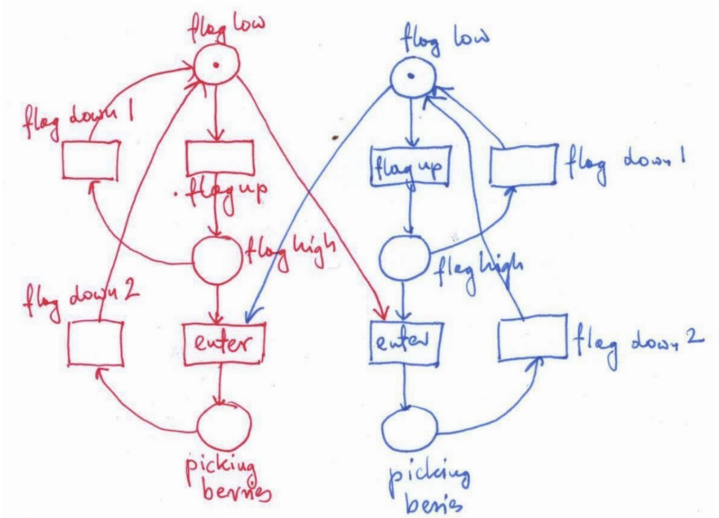
where p ranges over atomic formulas/descriptions.

- ⊥ - false, ⊤ - true
- AX, EX, AG, EG, AU, EU, AF, EF are **temporal connections**.
  all pairs, each starts with either A or E
- A means "along All paths" (inevitably)
- E means "along at least (there Exists) one path" (possibly)
- X means "neXt state"
- F means "some Future state"
- G means "all future states (Globally)"
- U means "Until"
- X, F, G, U cannot occur without being preceded by A or E.
- every A or E must have one of X, F, G, U to accompany it.

Two warring neighbours are separated by a field with wild berries. They agree to permit each other to enter the field to pick berries, but also need to ensure that only one of them is ever in the field at a time. After negotiation, they agree to the following protocol.
When a one neighbour wants to enter the field, he raises a flag. If he sees his neighbour's flag, he does not enter but lowers his flag and tries again. If he does not see his neighbour's flag, he enters the field and picks berries. He lowers his flag after leaving the field.
Model this algorithm for two neighbours n1 and n2. Specified the required safety properties for the field and check that it does indeed ensure mutually exclusive access. Specify the required progress properties for the neighbours such that they both get to pick berries given a fair scheduling strategy. Are any adverse circumstances in which neighbours would not make progress? What if the neighbours are greedy?

```
const False = 0
const True = 1
range Bool = False..True
set BoolActions = {setTrue, setFalse, [False], [True]}
BOOLVAR = VAL[False],
VAL[v:Bool] = (setTrue -> VAL[True]
          |setFalse -> VAL[False]
          |[v] -> VAL[v]
 ).
||FLAGS = (flag1:BOOLVAR || flag2:BOOLVAR).
NEIGHBOUR1 = (flag1.setTrue -> TEST),
TEST = (flag2[b:Bool] ->
        if(b) then
           (flag1.setFalse -> NEIGHBOUR1)
        else
           (enter -> exit -> flag1.setFalse -> NEIGHBOUR1)
      )+{{flag1,flag2}.BoolActions}.
NEIGHBOUR2 = (flag2.setTrue -> TEST),
TEST = (flag1[b:Bool] ->
         if(b) then
           (flag2.setFalse -> NEIGHBOUR2)
         else
           (enter -> exit-> flag2.setFalse -> NEIGHBOUR2)
      )+{{flag1,flag2}.BoolActions}.
property SAFETY = (n1.enter -> n1.exit -> SAFETY
                  |n2.enter -> n2.exit-> SAFETY). -
||FIELD = (n1:NEIGHBOUR1 || n2:NEIGHBOUR2
          ||{n1,n2}::FLAGS ||SAFETY).
progress ENTER1 = {n1.enter} //NEIGBOUR 1 always gets to enter
progress ENTER2 = {n2.enter} //NEIGHBOUR 2 always gets to enter
||GREEDY = FIELD << {{n1,n2}.{flag1,flag2}.setTrue}.
```

neighbour

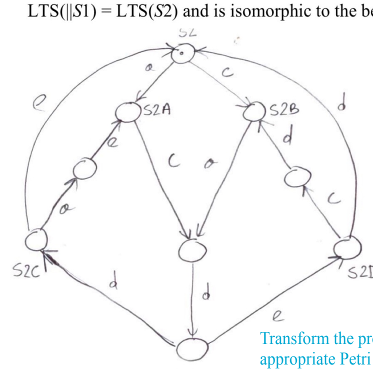


Show that processes ||S1 and S2 generate the same Labelled Transition Systems, i.e. LTS(||S1) = LTS(S2) (or equivalently, they generate the same behaviour)

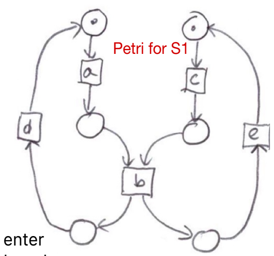
P = (a -> b -> d -> P)
Q = (c -> b -> e -> Q)
|| S1 = (P || Q)

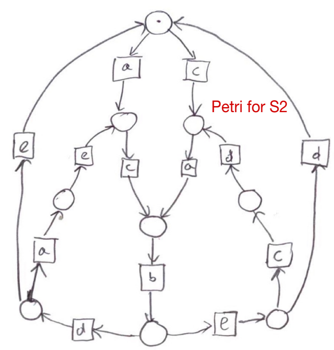
S2 = (a -> S2A | c -> S2B)
S2A = (c -> b -> d -> S2C | c -> b -> e -> S2D)
S2B = (a -> b -> d -> S2C | a -> b -> e -> S2D)
S2C = (e -> S2 | a -> e -> S2A)
S2D = (d -> S2 | c -> d -> S2B)

LTS(||S1) = LTS(S2) and is isomorphic to the below diagram:



Transform the process S1|| and S2|| into appropriate Petri nets. Are these nets identical? Explain the difference. Which one allows simultaneity.

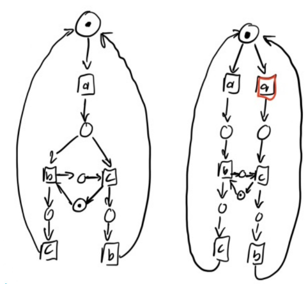




MIDTERM Q7
LTS for P1 and P2 have the same traces, Traces(P1) = Traces(P2) = prefix((a(bc U cb))*). In petri net,there is a deadlock in ||P1Q. So, ||P1Q deadlocks while ||P2Q does not.

7.[6] Consider the following processes:
P1 = (a → b → c → P1 | a → c → b → P1)
P2 = (a → (b → c → P2 | c → b → P2))
Q = (b → c → Q)
and
||P1Q = (P1 || Q)
||P2Q = (P2 || Q)

But ||P1Q and ||P2Q are not equivalent because ||P1Q will deadlock. See below:



P2 P1

'Between the events q and r, p is never true but t is always true'

LTL: G(F q ∧ F r ∧ (q ⇒ (¬ p U r) ∧ (q ⇒ (Ft U r)))
CTL: AG(AF q ∧ AF r) ∧ AG(q ⇒ A(¬ p U r))

Express in LTL and CTL: 'Φ is true infinitely often along every paths starting at s'. What about LTL for this statement?
CTL: s |= AG(AF Φ)
LTL: s |= G(F Φ)

Express in LTL and CTL: 'Whenever p is followed by q (after some finite amount of steps), then the system enters an 'interval' in which no r occurs until t'.

LTL: $G(p \Rightarrow XG(\neg q \ \vee \neg r \, U \, t))$,

CTL: $AG(p \Rightarrow AX\, AG(\neg q \ \vee A \neg r \, U \, t\,]))$,

Express in LTL and CTL: 'Between the events q and r, p is never true'.

LTL: G(F q ∧ F r ∧ (¬ q ∨ (¬ p U r))
CTL: AG(AF q ∧ AF r) ∧ AG(q ⇒ A(¬ p U r))

Specify a safety property for the car park problem 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?

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

```
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]
                   |when (i>0) depart -> OVERFLOW[i-1]
                   ),
OVERFLOW [N+1]=ERROR.

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

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

progress ENTER = {arrive}

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

CTL:

An upwards traveling elevator at the second floor does not change its direction when it has passengers wishing to go to the fifth floor:

$AG(floor = 2 \wedge direction = up \wedge ButtonPressed5 \Rightarrow$
$A[direction = up \ U \ floor = 5])$

The elevator can remain idle on the third floor with its doors closed:

$AG((floor = 3 \wedge idle \wedge door = closed) \Rightarrow$
$EG(floor = 3 \wedge idle \wedge door = closed))$

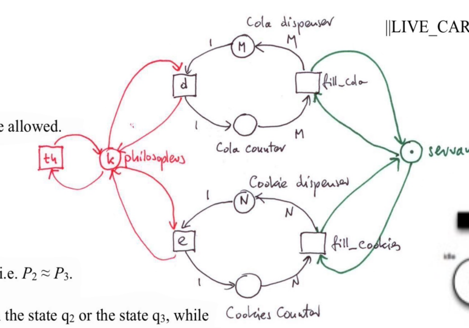
'$floor = 2$', '$direction = up$', $ButtonPressed5$', '$door = closed$', etc. are names of atomic formulas.

Bisimilarity

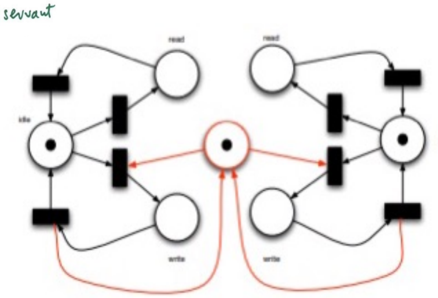
p_0 and s_0 are bisimilar as in both cases actions a and b are allowed.
p_1 and s_1 are bisimilar as they both allow only a.
p_2 and s_2 allow a, b and c, so they are bisimilar.
p_3 and s_3 are bisimilar as they both allow a and c.
p_4 and s_4 are bisimilar as they both allow a and b.
p_5 and s_5 are bisimilar as they both allow only c, and
p_5 and s_6 are also bisimilar as they also allow only c.
We have exhausted all cases, so P_2 and P_3 are bisimilar, i.e. $P_2 \approx P_3$.

After trace aa the labeled transition system P_1 is either in the state q_2 or the state q_3, while P_2 is in the state p_2. In the state p_2 the actions a, b and c are allowed, in the state q_2 the actions a and c are allowed, while in the state q_3 the actions a and b are allowed. Hence both pairs (q_2,p_2) and (q_3,p_2) are *not* bisimilar, i.e. $P_1 \not\approx P_2$.

After trace aa the labeled transition system P_1 is either in the state q_2 or the state q_3, while P_3 is in the state s_2. In the state s_2 the actions a, b and c are allowed, in the state q_2 the actions a and c are allowed, while in the state q_3 the actions a and b are allowed. Hence both pairs (q_2,s_2) and (q_3,s_2) are *not* bisimilar, i.e. $P_1 \not\approx P_3$.



General lock for 2 entities



Add a lock to ensure mutual exclusion