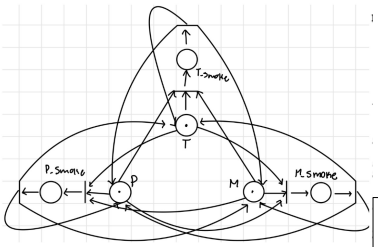
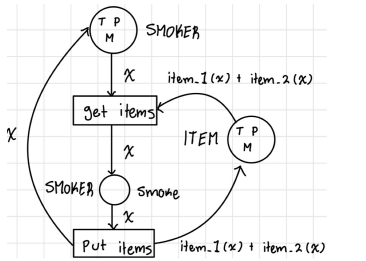
```
property CORRECT_PICKUP =
 (s_t.get_paper->s_t.get_match->CORRECT_PICKUP
 |s_p.get_tobacco->s_p.get_match->CORRECT_PICKUP
 |s_m.get_tobacco->s_m.get_paper->CORRECT_PICKUP).
SMOKER_T=( no_tobacco -> get_paper -> get_match->
        ↪ roll_cigarette ->
smoke_cigarrette -> SMOKER_T)
SMOKER_P=( no_paper -> get_tobacco -> get_match->
        ↪ roll_cigarette ->
smoke_cigarrette -> SMOKER_P)
SMOKER_M=( no_match -> get_tobacco -> get_paper->
        ↪ roll_cigarette ->
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.[14] Two workers W1 and W2 working separately need two different tools, say *drill* and *clamp* to do some work (say precise drill). In order to do the job, each worker needs get both tools. However, due to the nature of work, W1 needs to get drill first and clamp second, while W2 needs to get clamp first and drill second. We obviously want to avoid a situation when W1 grabs drill and waits for clamp, while W2 grabs clamp and waits for drill.

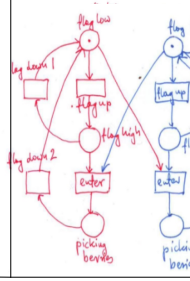
a.[7] Model the situation described above carefully avoiding deadlock with FSP.

b.[7] Model the situation described above carefully avoiding deadlock with Petri nets (any kind)

```
RESOURCE = (resourceTaken -> resourceAvailable -> RESOURCE).

LOCK = (lockAvailable -> releaseLock -> LOCK).

W1 = (getLock -> getDrill -> getClamp -> putDrill -> putClamp -> releaseLock -> W1).

W2 = (getLock -> getClamp -> getDrill -> putDrill -> putClamp -> releaseLock -> W2).

||W1W2 = (
  {a,b}:RESOURCE || {lock}:LOCK || w1:W1 || w2:W2
)
/{
  {w1,w2}.getLock / lock.lockAvailable,
  {w1,w2}.releaseLock / lock.releaseLock,
  {w1}.getDrill / a.resourceTaken,
  {w2}.getClamp / b.resourceTaken,
  {w1}.putDrill / a.resourceAvailable,
  {w1}.putClamp / b.resourceAvailable,
  {w2}.putDrill / a.resourceAvailable,
  {w2}.putClamp / b.resourceAvailable,
  {w1,w2}.releaseLock / lock.resourceAvailable
}.
```

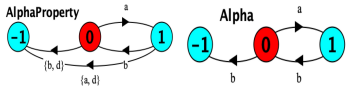
TWO WARRING NEIGHBOURS

```
const False = 0
const True = 1
range Bool = False..True
set BoolActions = {setTrue, setFalse,
    [False], [True]}
BOOLVAR = VAR[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).
progress ENTER1 = {n1.enter}
progress ENTER2 = {n2.enter}
||GREEDY = FIELD<<{n1,n2},{flag1,flag2}.setTrue}.|
```



PRODUCE LTS

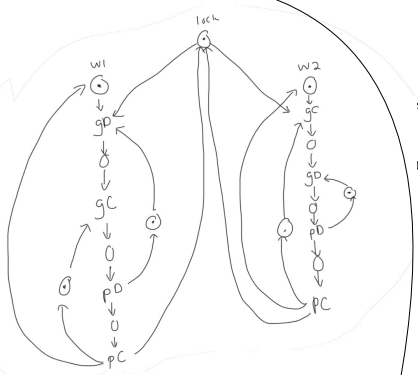
```
property Alpha = (a->b->Alpha | b->Beta)+{d}
         Beta  = (c->b->Alpha | b->Beta)

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



Alpha1 = (a→({a,d}→Beta | b→ Alpha1) | {b,d}→Beta).

CHEESE COUNTER

```
set Bold = {bold[1..2]}
set Meek = {meek[1..2]}
set Customers = {Bold, Meek}
CUSTOMER = (get_cheese -> CUSTOMER).
COUNTER = (cheese -> COUNTER).
||CC = (Customers:CUSTOMER
     || Customers::COUNTER).
```



Simplified Multidimensional Semaphores

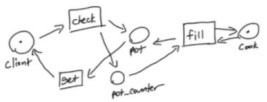
The extended primitives edown and eup are atomic (indivisible) and each operates on a set of semaphore variables which must be initiated with non-negative integer value. edown(S1,...,Sn): if for all i, $1 \le i \le n$, Si>0 then for all i, $1 \le i \le n$, $S_i := S_i - 1$ else block execution of calling processes eup(S1,...,Sn): if processes blocked on (S1,...,Sn) then awaken one of them else for all i, $1 \le i \le n$, $S_i := S_i + 1$

```
SEM(N=INITIAL_VALUE = SEMA[N],
SEMA[v:Int] = (when (v<=Max) up ->
        ↪ SEMA[v+1] | when (v>0)
        ↪ down -> SEMA[v-1]).
SEMS1S2(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/S1.down}.
```

Burger

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. a) Trace to error: fill[2], c2.check, fill[1], c1.check, c2.get, c1.get. Part b) change pot to be:

```
range Burgers = 0..2
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
        ↪ }.
```



Simple minded construction:

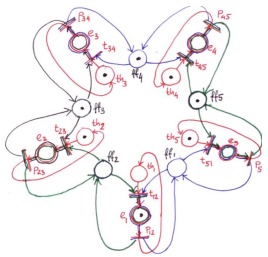
```
FORK = (get -> put -> FORK).
PHIL = (think -> right.get -> left.get -> eat ->
        ↪ right.put -> left.put -> PHIL).
||DINERS(N = 5) = forall[i : 1..N] (phil[i] : PHIL
        ↪ || {phil[i].right, phil[(i % 5) + 1].
        ↪ left} :: FORK
```

Solution 1 – Add asymmetry into the composition, where 1, 3, 5 always perform 'left.get -> right.get', while 2, 4 always perform 'right.get -> left.get'.

```
PHIL = (when(i=1|i=3|i=5) think -> left.get ->
        ↪ right.get -> eat -> left.put -> right.
        ↪ put -> PHIL
       |when(i=2|i=4) think -> right.get -> left.
        ↪ get -> eat -> right.put -> left.
        ↪ put -> PHIL).
```

Solution 2 – Use a butler to prevent more than 4 philosophers from sitting at the table.

```
PHIL = (think -> sitdown -> right.get -> left.get
        ↪ -> eat -> right.put -> left.put ->
        ↪ getup -> PHIL).
BUTLER(K=4) = COUNT[0],
COUNT[i:1..4] = (when(i<K) sitdown -> COUNT[i+1] |
        ↪ getup -> COUNT[i-1]).
||DINERS(N=5) ... Put all 5 lines from above
||{phil[i:1..N]}::BUTLER(N-1)).
```



Bounded Buffer

A buffer with a fixed number of slots

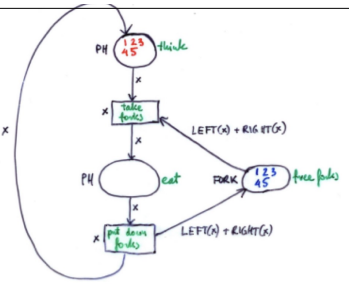
```
BUFFER(N=5) = COUNT[0],
COUNT[i:0..N]
    = (when (i<N) put->COUNT[i+1]
      |when (i>0) get->COUNT[i-1]
      ).
PRODUCER = (put->PRODUCER).
CONSUMER = (get->CONSUMER).
||BOUNDEDBUFFER = (PRODUCER||BUFFER(5)||CONSUMER).
```

Dining Philosophers with 'atomic act of picking up both forks'

```
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/right.reserve_right,
reserve_forks/left.reserve_left,
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
```

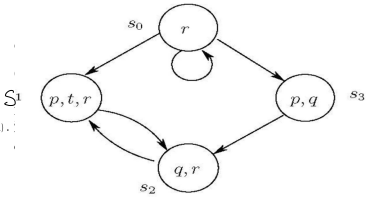
Hotel Reservation

```
HOTEL = (request -> REQUESTED),
        ↪ REQUESTED = (available ->
        ↪  confirmation_sent ->
        ↪ CONFIRMED | not_available
        ↪  -> put_on_list ->
        ↪ ON_WAITING_LIST),
CONFIRMED = (moves_in -> pays ->
        ↪ ARCHIVED),
ON_WAITING_LIST = (room_available
        ↪ -> CONFIRMED | giving_up
        ↪ -> CANCELED),
ARCHIVED = (record_transactions ->
        ↪ STOP), CANCELED = (
        ↪ record_transactions ->
        ↪ STOP)
```



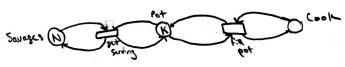

```
colour PH = with ph1 | ph2 | ph3 | ph4 | ph5
colour Fork = with f1 | f2 | f3 | f4 | f5
LEFT : PH -> FORK, RIGHT : PH -> FORK
var x : PH
fun LEFT x = case of ph1 => f2 | ph2 => f3 | ph3
        ↪ => f4 | ph4 => f5 | ph5 => f1
fun RIGHT x = case of ph1 => f1 | ph2 => f2 | ph3
        ↪ => f3 | ph4 => f4 | ph5 => f5
```

IODEL CHECKING



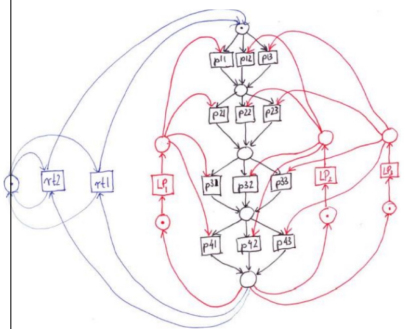
DINING SAVAGES

```
const M = 5
SAVAGE = (get_serving -> SAVAGE).
COOK = (fill_pot -> COOK).
POT = SERVINGS[0]
SERVINGS[i:0..M] =
    (when (i==0) fill_pot -> SERVINGS[M]
    |when (i>0) get_serving -> SERVINGS[i-1]).
||SAVAGES = (SAVAGE || COOK || POT).
```



OFFICE PRINTER PROBLEM

```
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}.
```



i) EG r
We have L(s0) = {r} and L(s2) = {q,r}. Clearly r s0, s2. So s0 |= HOLDS and φ = ∈ φ s2 |= HOLDS
ii) G (r v q)
"If the process is enabled infinitely often, then it runs infinitely often."
p: "the process is enabled" q: "the process runs"
Express in LTL: G(Fp→Fq)
Express in CTL: AG(EFp→EFq)
"A passenger entering the elevator at 5th floor and pushing 2nd floor button, will never reach 6th floor, unless 6th floor button is already lighten or somebody will push it, no matter if she/he entered an upwards or upward travelling elevator."
LTL: G((floor = 5 ∧ ButtonPress = 2 ∧ ((F floor = 6 ∧ F ¬ButtonPress = 6)) ∧ (F floor = 2 ∧ F direction = up))) ∧ (F(floor = 6 → ButtonPress = 6)))
CTL: AG(floor = 5 ∧ ButtonPress = 2) ∧ AG(EF floor = 6 ∧ EF ¬ ButtonPress = 6) ∧ AG(EF floor = 2 ∧ EF direction = up) ∧ AG(EF(floor = 6 → ButtonPress = 6))

Saving Account

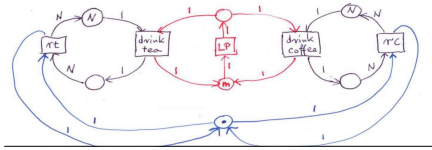
The saving account problem. A saving account is shared by several people. Each person may deposit or withdraw funds from the account subject to the constraint that the balance of the account must never become negative.

```
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]
).
```

BINARY SEMAPHORE

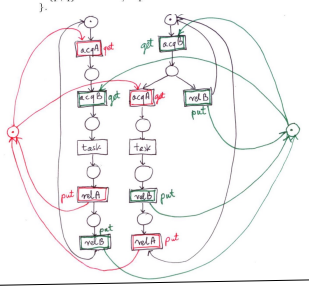
```
BSEMA = (up -> down -> BSEMA).
PROCESS = (console.up -> console.down -> PROCESS).
set Processes = {user[1..2],system[1..2]}
OS = (Processes:PROCESS || Processes::console:BSEMA)>>{user}.
```

```
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}
```

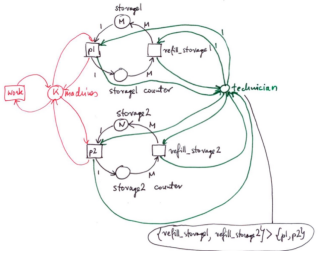


A possible model for the above scenario can be defined as

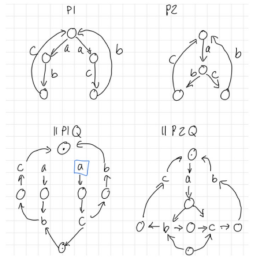
```
RESOURCE = ( get -> put -> RESOURCE ).

/* 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
  }.
```




```
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}
```



EQUIVALENCE/TRACE Q



Both P1 and P2 follow the same path with the same transitions, we can see this by looking at the trace.
Traces(P1) = Traces(P2) = Prefix((a(bc U cb))*)
Although, there is also a deadlock in ||P1Q, while ||P2Q does not have a deadlock, therefore they are not equivalent.

```
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).
```

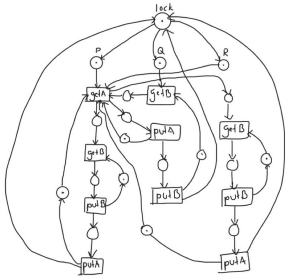
```
const False = 0
const True = 1
range Bool = False..True

ROOM = ROOM[False],
ROOM[reserved:Bool] = (when(!reserved) reserve -> ROOM[True]
      | query[reserved] -> ROOM[reserved]
      | when (reserved) reserve -> STOP).

range R = 1..4
||ROOMS = (room[R]:ROOM).

CUSTOMER = (choose[r:R] -> room[r].query[reserved:Bool]
  -> if (!reserved) then
     (room[r].reserve -> CUSTOMER | cancel[r] -> room[r].cancel -> CUSTOMER)
     else
     (room[r].wait -> CUSTOMER | cancel[r] -> room[r].cancel -> CUSTOMER)).

||HOTEL = (ROOMS || CUSTOMER).
```

[a]:
RESOURCE = (resourceTaken -> resourceAvailable -> RESOURCE).

LOCK = (lockAvailable -> releaseLock -> LOCK).

P = (getLock -> getA -> getB -> putB -> putA -> releaseLock -> P).
Q = (getLock -> getB -> getA -> putA -> putB -> releaseLock -> Q).
R = (getLock -> getA -> getB -> putB -> putA -> releaseLock -> R).

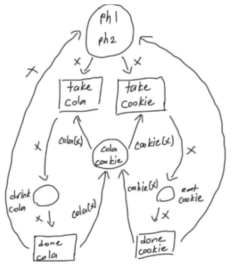
||PQR = ({a,b}:RESOURCE || {lock}:LOCK || p:P || q:Q || r:R)
  /{{p,q,r}.getLock/lock.lockAvailable, {p,q,r}.releaseLock/lock.releaseLock,
   {p,q,r}.releaseLock/lock.resourceAvailable,
   {p,q,r}.getA/{a,b}.resourceTaken,
   {p,q,r}.putA/{a,b}.resourceAvailable}.



Cookie

Consider two philosophers system. Each philosopher either think, or eats cookies or drinks cola. However there is only one cookie dispenser and only one cola distributor (they are separate machines), so only one philosopher can get cookies or cola at a time, the other must wait.

```
PHILOSOPHER = (think >
        ↪ check_cookie_available >
        ↪ get_cookie -> eat_cookie
        ↪ -> put_cookie
-> check_cola_available -> get_cola
        ↪  -> drink_cola ->
        ↪ put_cola -> PHILOSOPHER),
COOKIE_DISPENSER = (
        ↪ cookie_available -> get
        ↪ _cookie -> cookieEaten ->
        ↪  put_cookie ->
COOKIE_DISPENSER),
COLA_DISPENSER = (cola_available ->
        ↪  get_cola -> colaDrank ->
        ↪  put_cola ->
COLA_DISPENSER),
||  Parallel composition with
        ↪ synchronization on
        ↪ resource access
||SYSTEM = (PHILOSOPHER1:
        ↪ PHILOSOPHER ||
        ↪ PHILOSOPHER2:PHILOSOPHER
        ↪ ||
COOKIE_DISPENSER || COLA_DISPENSER)
        ↪ .
```




```
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}
```