2SD3 Final v0.2 – The Triumph of Bureaucracy and Vocal Minority: You Shall (Not) Pass Edition

If you're reading this, please contribute!

REMINDER! This is a template! The cheat sheet maintainer (.json) *intentionally* leaves extra space for you to add your own notes! If something's missing, add it yourself! (and if it's important enough please contribute!)

FSP

Syntax

```
// instance prefixing - a:P
SWITCH = (on -> off -> SWITCH).
||TWO_SWITCH = (a:SWITCH || b:SWITCH).
// relabeling - /{new1/old1, new2/old2, ...}.
CLIENT = (call -> wait -> continue -> CLIENT).
SERVER = (request -> service -> reply -> SERVER).
||CLIENT_SERVER = (CLIENT||SERVER)/{call/request,
        ↪ reply/wait}.
// process prefixing (mutex) - {a1, ..., ax} :: P
RESOURCE = (acquire -> release -> RESOURCE).
USER = (acquire -> user -> release -> USER).
||RESOURCE_SHARE = (a:USER || b:USER || {a, b}::
        ↪ RESOURCE).
// RESOURCE is a single shared instance between
        ↪ the two USERs
// for loop syntax
|| SWITCHES (N = 3) = (forall [i:1..N] s[i]:SWITCH
        ↪ )
// or alternatively
range Seats = 1..3
|| SEATS=(seat[i:1..3]:SEAT).
// hiding operator - \{a1, ..., ax}
// interface operator - @{a1, ..., ax}
// these two do the same thing
USER = (acquire -> use -> release -> USER)\{use}.
USER = (acquire -> use -> release -> USER)@{
        ↪ acquire, release}.
// syntax for progress
progress P = {a1, ..., an}
// syntax for high priority
||C = (P||Q)<<{a1, ..., an}.
// syntax for low priority
||C = (P||Q)>>{a1, ..., an}.
// simulating a boolean
const False = 0
const True  = 1
range Bool  = 0..1
```

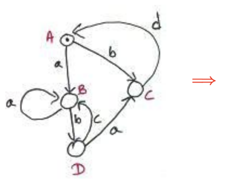
Maker-user example

```
MAKER = (make -> ready -> MAKER).
USER  = (ready -> user -> USER).
||MAKER_USER = (MAKER || USER).
```

Garden example (maybe move this to a diff section later)

```
const N = 4
range T = 0..N
set VarAlpha = {value.{read[T],write[T]}}
VAR      = VAR[0],
VAR[u:T] = (read[u]   ->VAR[u]
           |write[v:T]->VAR[v]).
TURNSTILE = (go     -> RUN),
RUN       = (arrive-> INCREMENT
            |end    -> TURNSTILE),
INCREMENT = (value.read[x:T]
            ->value.write[x+1]->RUN)
            +VarAlpha.
DISPLAY =(value.read[T]->DISPLAY)+{value.write[T
        ↪ ]}.
||GARDEN = (east:TURNSTILE || west:TURNSTILE ||
        ↪ display:DISPLAY
           || {east,west,display}::value:VAR)
            /{go /{east,west}.go,
              end/{east,west}.end}.
```

LTS

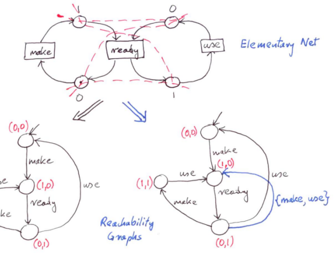


$\Rightarrow$

$A = (a \to B \mid b \to C)$
$B = (a \to B \mid b \to D)$
$C = (d \to A)$
$D = (a \to C \mid c \to B)$

Petri Nets

Reachability Graphs



Hiding/Labeling

relabeling: (PROCESS)/$\{newlabel1/oldlabel1, ..., newlabeln/oldlabeln\}$
interface: (PROCESS)@$\{a1...ax\}$ hides all actions except a1 . . . ax
hiding: (PROCESS) \$\{a1...an\}$ hides actions a1...an
$\{a1, ..., ax\}$ :: P replaces every action label n in the alphabet of P with the labels a1.n, ... , ax.n. Thus, every transition (n → X) in the definition of P is replaced with the transitions $(\{a1.n, ..., ax.n\} \to X)$

Bisimulation

State Bisimilarity – $p \approx q$ iff whatever action executed at p can also be executed at q, and vice versa.
LTS Bisimilarity – $P \approx Q$ iff each state p_t, reachable from the initial state by a trace t in P is bisimilar to an appropriate state q_t that is reachable from the initial state by the same trace t in Q.

Mutual Exclusion

Arbitrary interleaving of read and write actions leads to **interference**. Interference bugs are difficult to locate. We use **mutual exclusion** to only give one process access to the shared resource at a time.

```
LOCK = (acquire->release->LOCK).
U1 = (acquire -> use -> release -> U1).
U2 = (acquire -> use -> release -> U2).
||SYSTEM = (u1:U1||u2:U2||{u1,u2}::LOCK).
```

Above allows for lock→use→release for either user but only one of them at a time.

Monitors and Semaphores

Monitor – A threadsafe class where each function is wrapped by a mutex. Essentially, only one process may access the class at a time. Entirely syntactic sugar. **Semaphore** – Essentially a mutex with a queue of processes

```
down(s):     if s > 0 then
                 decrement s
             else
                 block execution of calling process
up(s):       if processes blocked on s then
                 awaken one of them
             else
                 increment s
```

```
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)).
```

Bounded Buffer

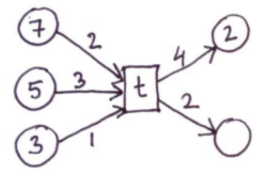
A buffer with a fixed number of slots

```
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).
```

Nested Monitor Problem

P/T nets

Each place in a P/T net can hold multiple tokens. Each transition has a weight, w, associated with it. If it is an input transition, firing takes w tokens from the input place. If its an output transition, firing adds w tokens to the output place. An action can only be fired if enough input tokens are present in all input



places.

Deadlocks

Dining Philosophers Problem

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
```

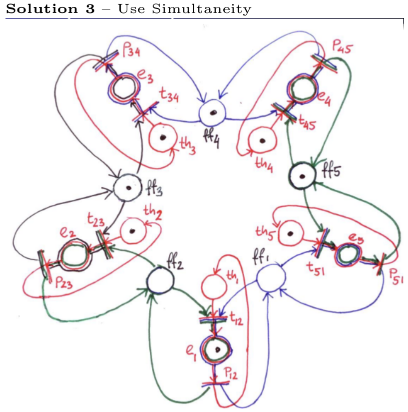
Solution 1 – Add asymmetry into the composition, where 1, 3, 5 always perform 'left.get -> right.get', while 2, 4 always perform 'right.get -> left.get'.

```
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) ...
||{phil[i:1..N]}::BUTLER(K=4)).
```

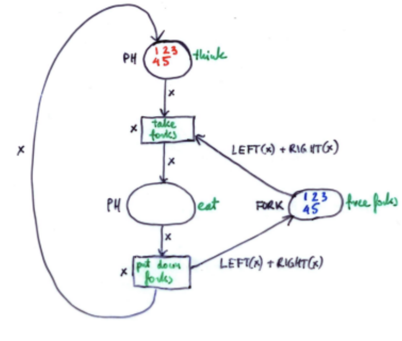
Solution 3 – Use Simultaneity



Only fire a transition if both forks are available

Coloured Petri Nets

"Colours" are simply types of tokens that are passed around the petri net Paths to transitions are either labeled with variables or functions that transform one of the input variables into the object to remove from a state.




```
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
```

Semaphores and Extensions

Dijkstra's Semaphore Operations

C(s) – initial value of a semaphore variable s
ndown(s) – number of times down(s) was executed **nup(s)** – number of times up(s) was executed **npdown(s)** – number of times down(s) was passed
Then we define down and up:
down(s): ndown(s) = ndown(s)+1: if ndown(s) <= nup(s) + C(s) then npdown(s) = npdown(s) + 1;
up(s): if ndown(s) > nup(s) + C(s) then npdown(s) = npdown(s) + 1; nup(s) = nup(s)+1;

Theorem 1. *npdown(s) = min(ndown(s), C(s) + nup(s))*

Multidimensional Semaphores of Agerwala

edown($s1, \ldots, s_n, s_{n+1}, \ldots, s_{n+m}$: if for all $i, 1 \leq i \leq n, s_i > 0$ and for all $j, 1 \leq j \leq m, S_{n+j} = 0$ then for all $i, 1 \leq i \leq n, s_i = s_i - 1$ else block execution of calling processes
eup($s_1, s_2, \ldots, s_n$: if processes blocked on $(s_1, \ldots, s_n)$ then awaken al of them else for all $i, i \leq i \leq n, s_i = s_i + 1$

Inhibitor Nets

Add a circle to the transition side of an arc to make it an inhibitor arc Now the transition can only be fired if the places connected by inhibitor arcs are empty.

Smokers' Problem

3 Smokers each have an unlimited type of either tobacco, cigarette paper, matches. 2 ingredients are placed on the table, the smoker with the third ingredient needed should pick up the ingredients, make a cigarette, and smoke it. Next set of ingredients won't be placed until smoking is completed.

Simple-minded 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_M).
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_cigarette,
 s_m.smoking_completed/s_m.smoke_cigarette,
 s_p.smoking_completed/s_p.smoke_cigarette}.
```

Property (safety)

```
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).
```

Ask first, do later

```
SMOKER_T=( no_tobacco -> get_paper -> get_match->
        ↪ roll_cigarette ->
smoke_cigarette -> SMOKER_T)
SMOKER_P=( no_paper -> get_tobacco -> get_match->
        ↪ roll_cigarette ->
smoke_cigarette -> SMOKER_P)
SMOKER_M=( no_match -> get_tobacco -> get_paper->
        ↪ roll_cigarette ->
smoke_cigarette -> 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_cigarette,
 s_m.smoking_completed/s_m.smoke_cigarette,
 s_p.smoking_completed/s_p.smoke_cigarette}.
```

Safety and Liveness

Safety – asserts that nothing bad happens

Safety Property P – defines a process that asserts any trace including the actions in the alphabet of P is accepted by P, otherwise they are transitions to the ERROR state, safety checks are compositional hence they should be composed with the appropriate (sub)system

Liveness – asserts that something good eventually happens

Progress Property – asserts that it is always the case that a particular action is eventually executed, opposite of starvation, progress checks are not compositional hence they should be conducted after safety checks

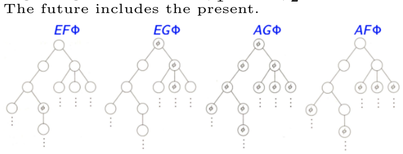
Starvation – situation in which an action is never executed

Terminal Set of States – set of states in which every state is reachable from every other state in the set and there is no transition from within to set to any state outside the set

Priority – specifies actions that have a higher/lower priority than any other action in the alphabet of some state

CTL and LTL
CTL
A: along all paths
E: exists a path
X: next state
F: some future state
G: all future states
U: until ("killer" event must happen)
W: weak until ("killer" event may never happen).
In CTL, must start with path operator (A/E). W in CTL: $A[p\,\mathrm{W}\,q] \equiv A[p\,\mathrm{U}\,q] \vee \mathrm{AG}\,p$ (same with E).

Common Pattenrs $AX\phi$ - in every next state.
$EX\phi$ - in some next state.
$AG\phi$ - All computation paths beginning with s the property ϕ holds Globally.
$EG\phi$ - There Exists a path beginning in s such that ϕ holds Globally along the path.
$AF\phi$ - For All computation paths beginning with s there will be some Future state where ϕ holds.
$EF\phi$ - There Exists a computation bath beginning in s such that ϕ holds in some Future states.
$A[\phi_1 U \phi_2]$ - All computation paths beginning in s satisfy that ϕ_1 Until ϕ_2 holds.
$E[\phi_1 U \phi_2]$ - There Exists a computation path beginning in s such that ϕ_1 Until ϕ_2 holds on it.
The future includes the present.



Equivalences $\neg AF\phi \equiv EG\neg\phi$
$\neg EF\phi \equiv AG\neg\phi$
$\neg AX\phi \equiv EX\neg\phi$
$AF\phi \equiv A[\top U \phi]$
$EF\phi \equiv E[\top I \phi]$

Elevator Example 1

"An upwards travelling elevator at the second floor does not change direction when it has passengers wishing to go to the fifth floor"
CTL: $AG(floor = 2 \wedge direction = up \wedge ButtonPressed5 \Rightarrow A[direction = up\ U\ floor = 5])$.

Elevator Example 2

"The elevator can remain idle on the third floor with its doors closed"
CTL: $AG((floor = 3 \wedge idle \wedge door = closed) \Rightarrow EG(floor = 3 \wedge idle \wedge door = closed))$

LTL
$\perp$ - false, $\top$ - true Other symbols mean the same as above. A set of paths satisfies ϕ if every path in the set satisfies ϕ
Equivalences $\neg G\phi \equiv F\neg\phi$
$\neg F\phi \equiv G\neg\phi$
$\neg X\phi \equiv X\neg\phi$
$F(\phi \vee \psi) \equiv F\phi \vee F\psi$
$G(\phi \wedge \psi) \equiv G\phi \wedge G\psi$
$F\phi \equiv \top U \phi$
$G\phi \equiv \perp R \phi$
$\phi U \psi \equiv \phi W \psi \wedge F\psi$
$\phi W \psi \equiv \phi U \psi \vee G\phi$
$\phi W \psi \equiv \psi R(\phi \vee \psi)$
$\phi R \psi \equiv \psi W(\phi \wedge \psi)$

Dynamic Systems
Golf Club Program
Players at a golf club borrow and then return golf balls. Different players need different numbers of balls. How do we model the infinite stream of players? We can only model infinite behaviours.

Adverse Scheduling
Intentionally schedule priorities to try to break things
Eg for golf club scheduling:

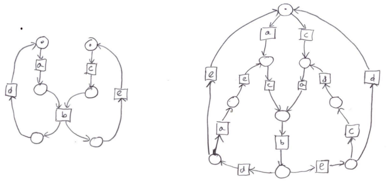
```
progress NOVICE = {NOVICES.get[R]}
progress EXPERT = {EXPERTS.get[R]}
||ProgressCheck = GOLFCLUB >> {Players.put[R]}.
```

Q7 - A1

```
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).
```

For the above FSPs, they both share the same LTS diagram (LTS version of the right petri net), however, since ||S1 has simultaneous actions, its petri net will be show simultaneity whereas S2 will not.



Q5 - Midterm
A central computer is connected to remote terminals, with seats for concert hall. Clients choose a free seat and clerk enters the seat into the system and gives a ticket. We need to prevent double booking while letting clients choose any available seat.

```
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).
```

Q7 - Midterm

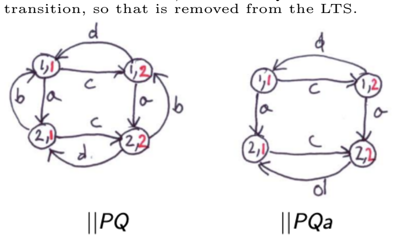
```
P1 = (a -> b -> c -> P1 | a -> c -> b -> P1).
P2 = (a -> (b -> c -> P2 | c -> b -> P2)).
Q = (b -> c -> Q).
||P1Q = (P1 || Q).
||P2Q = (P2 || Q).
```

For P1 there are two possible paths which have an a transition. For P2, it starts with one a transition, so both the starting states are equivalent in this case. In both branches of P1, you either do $b \to c \to P1$ or $c \to b \to P1$. For P2, it splits into two possible paths, $b \to c \to P2$ or $c \to b \to P2$, which is the same as the traces for P1. Therefore P1 and P2 are equivalent.
For ||P1Q and ||P2Q,both P1 and Q, and P2 and Q, share $b \to c$. In P1 and Q, this becomes a shared action and both P1 and Q will want to use the same transitions at the same time (deadlock). For ||P2Q, the FSP can choose to go in the alternate direction instead of waiting for Q to finish, so this one would not have a deadlock whereas ||P1Q will have one.

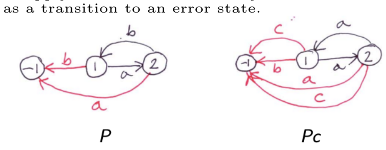
Alphabet Extension
Processes

```
P = (a -> b -> P).
Q = (c -> d -> Q).
Qa = (c -> d -> Q) + {b}.
```

When taking the composition of ||PQ and ||PQa, we take the union of both processes and remove duplicate transitions. In the case of PQ, there are no similar transitions, however PQa shares the b transition, so that is removed from the LTS.



Properties
With properties, whenever you add more transitions through alphabet extension, you need to apply this transition to every state and use it as a transition to an error state.



Tut 6 Mutex Example
Workers in an office share a printer. The printer is able to print any number of jobs before it runs out of toner. This is replaced by a technician when necessary.

```
const J=3
range Jobs = 0..J
// j Represents toner
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}.
```

Binary Semaphore Question
In an operating system, a binary semaphore is used to control access to the console. The console is used by user and system processes. Write down a model with FSP for this system. Discuss when user processes may be denied access to the console.

```
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}.
```

Dining Savages
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 refills the pot with M servings. The behavior of the savages and the cook is described by

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

Model the behaviour of the pot and of the system as FSP processes.

```
const M = 5
SAVAGE = (getserving -> SAVAGE).
COOK = (fillpot -> COOK).
POT = SERVINGS[0],
SERVINGS[i:0..M] = (when (i==0) fillpot ->
      ↪ SERVINGS[M] | when (i>0) getserving ->
      ↪ SERVINGS[i-1]).
||SAVAGES = (SAVAGE || COOK || POT).
```

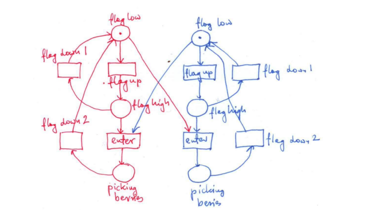
Simplified Multidimensional Semaphores
The extended primitives edown and eup are atomic (indivisible) and each operates on a set of semaphore variables which must be initialized with non-negative integer value. edown(S1,...,Sn): if for all i, $1 \leq i \leq n$, Si>0 then for all i, $1 \leq i \leq n$,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

```
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}.
```

Two Warring Neighbours
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.

```
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}
progress ENTER2 = {n2.enter}
||GREEDY = FIELD<<{{n1,n2}.{flag1,flag2}.setTrue}.
```

(b)[10]



CTL and LTL Examples
If the process is enabled infinitely often, then it runs infinitely often.

$$\mathrm{G}\,\mathrm{F}\,enabled \Rightarrow \mathrm{G}\,\mathrm{F}\,running$$

$$\mathrm{AG}(\mathrm{AF}\,enabled) \Rightarrow \mathrm{AG}(\mathrm{AF}\,running)$$

A passenger entering the elevator on 5th floor and pushing 2nd-floor button will never reach 6th floor, unless 6th-floor button is already lit or somebody will push it, no matter if she/he entered an upwards or upward travelling elevator.

$$\mathrm{AG}(floor = 5 \wedge ButtonPressed2 \Rightarrow$$

$$\mathrm{A}[\neg floor = 6\,\mathrm{U}\,ButtonPressed6])$$

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

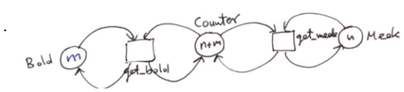
```
FORK = ( reserve_right -> take_right -> put_right
      ↪ -> FORK
| reserve_left -> take_left -> put_left -> FORK ).
PHIL = (think -> reserve_forks -> USE_FORKS).
USE_FORKS = ( take_right -> take_left -> eat ->
      ↪ PUT_FORKS
| take_left -> take_right -> eat -> PUT_FORKS ),
PUT_FORKS = ( put_left -> 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
}.
```

A3Q2 - Gas Question

```
const N = 3 //number of customers
const M = 2 //number of pumps
range C = 1..N
range P = 1..M
range A = 1..2 //Amount of money or Gas
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)
      ↪ .
// part b
range T = 1..2
property FIFO = (customer[i:T].prepay[A] -> PAID[i
      ↪ ]),
PAID[i:T] = (customer[i].gas[A] -> FIFO
|customer[j:T].prepay[A] -> PAID[i][j]),
PAID[i:T][j:T]= (customer[i].gas[A] -> PAID[j]).
||CHECK_FIFO = (GASSTATION || FIFO).
```

A3Q3 - Cheese Question

```
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).
```



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

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