# Finite State Process

Ali Safilian

Tutorial (Week #3)

CONCURRENT SYSTEM DESIGN (3BB4)

Fall 2015

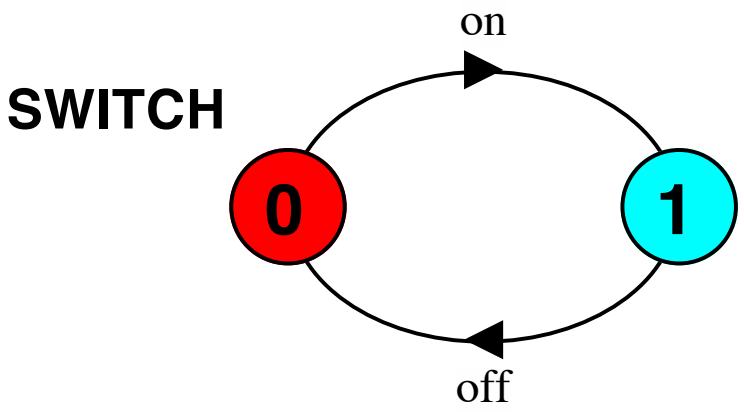
Instructor: Dr. Maibaum

# Finite State Process

- ✓ FSP is a *process algebra* for specifying a finite state process:

```
SWITCH = (on -> off-> SWITCH).
```

- ✓ The *semantics* is given via LTSs:




- ✓ What is the semantics of an LTS? *The set of valid traces*!

on→off → on → off → on → off → on → off → on ...

# FSP Summary

| | | | |
|---|---|---|---|
| ***Action prefix*** | (x->P) | ***Parallel composition*** | (P\|\|Q) |
| ***Choice*** | (x->P\|y->Q) | ***Replicator*** | forall [I:1..N] P(I) |
| ***Guarded Action*** | (when B x->P \| y->Q) | ***Process labeling*** | a: P |
| ***Alphabet extension*** | P + S | ***Process sharing*** | $\{a_1,...,a_n\}$::P |
| ***Conditional*** | If B then P else Q | ***Priority High*** | \|\|C=(P\|\|Q)<<$\{a_1,...,a_n\}$ |
| ***Relabelling*** | /{$new_1/old_1$,...} | ***Priority Low*** | \|\|C=(P\|\|Q)>>$\{a_1,...,a_n\}$ |
| ***Hiding*** | \{$a_1,...,a_n$} | ***Safety property*** | property P |
| ***Interface*** | @{$a_1,...,a_n$} | ***Progress property*** | progress P = $\{a_1,...,a_n\}$ |

# Primitive Processes

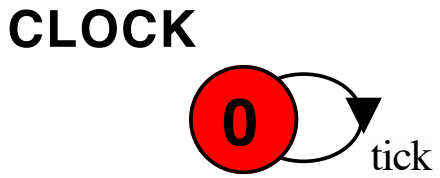
# Action Prefix

If x is an action and P a process then (x-> P) is a process that initially engages in the action x and then behaves like P.

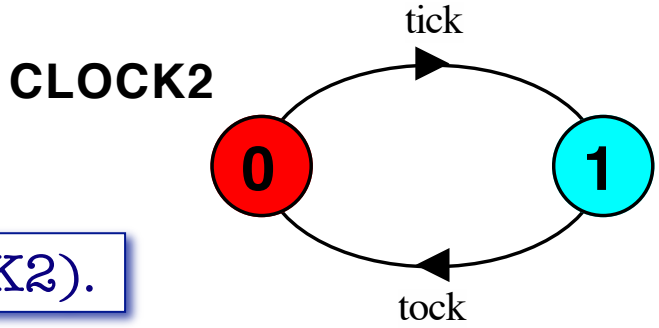
***Convention:***

Processes start with UPPERCASE, actions start with lowercase.

```
CLOCK = (tick -> CLOCK).
```




```
CLOCK2  = (tick -> TOCK),
TOCK = (tock -> CLOCK2).
```

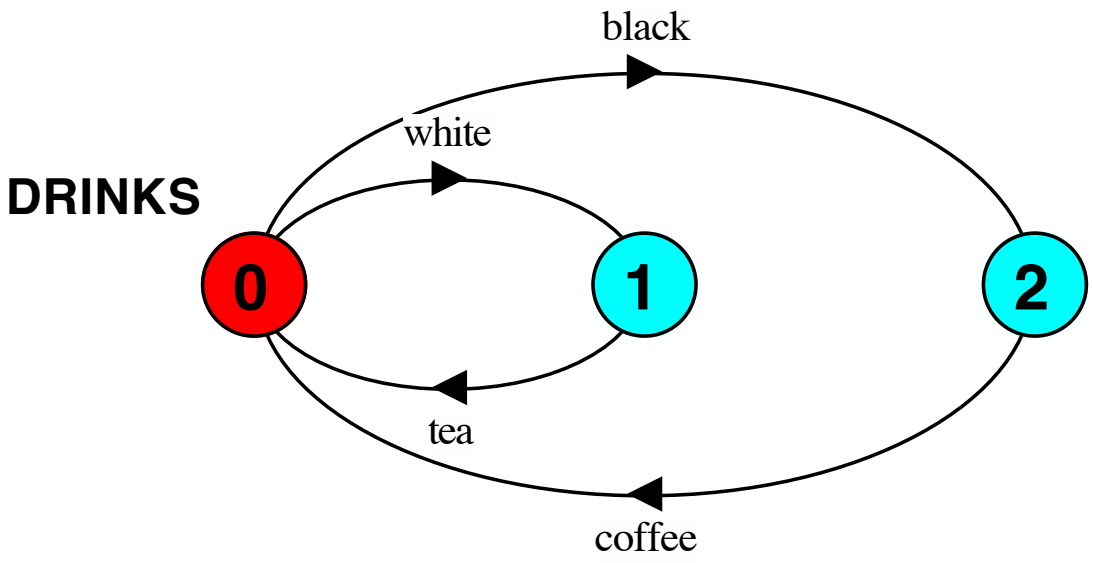



```
CLOCK2  = (tick -> tock -> CLOCK2).
```

# Choice

(x->P | y->Q) performs either x or y and then behaves as either P or Q, respectively.

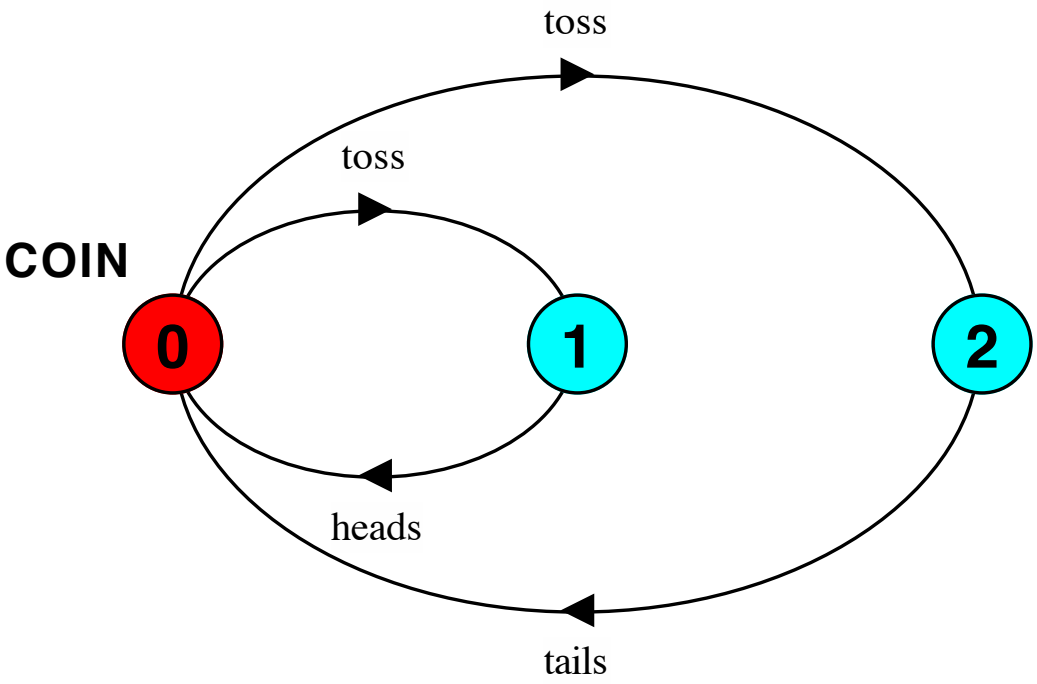
```
DRINKS = (black -> coffee -> DRINKS | white -> tea -> DRINKS).
```



# Non-deterministic Choice

(x->P | x->Q) performs x and then behaves as either P or Q.

```
COIN = (toss -> heads -> COIN |toss -> tails -> COIN).
```




FSP follows CCS rather than CSP in not distinguishing between internal and external choice

# Action Sets

If a number of different actions are followed by the same behavior, you may want to use an action set rather than explicit choice.

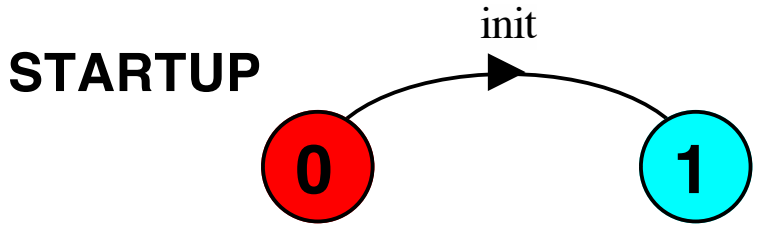
```
DOOR = (open -> DOOR | close -> DOOR).
```

```
DOOR = ({open, close} -> DOOR).
```

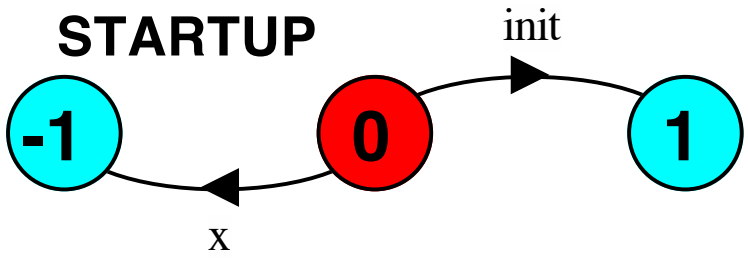
# STOP and ERROR

Two predefined processes STOP and ERROR.

```
STARTUP = (init -> STOP).
```




```
STARTUP = (init -> STOP | x -> ERROR).
```



# Alphabet Extension

What are the alphabets of the following process? Let LTSA do it!

```
DRINKS = (black -> coffee -> DRINKS | white -> tea -> DRINKS).
```

Alphabet:
{black, coffee, tea, white}

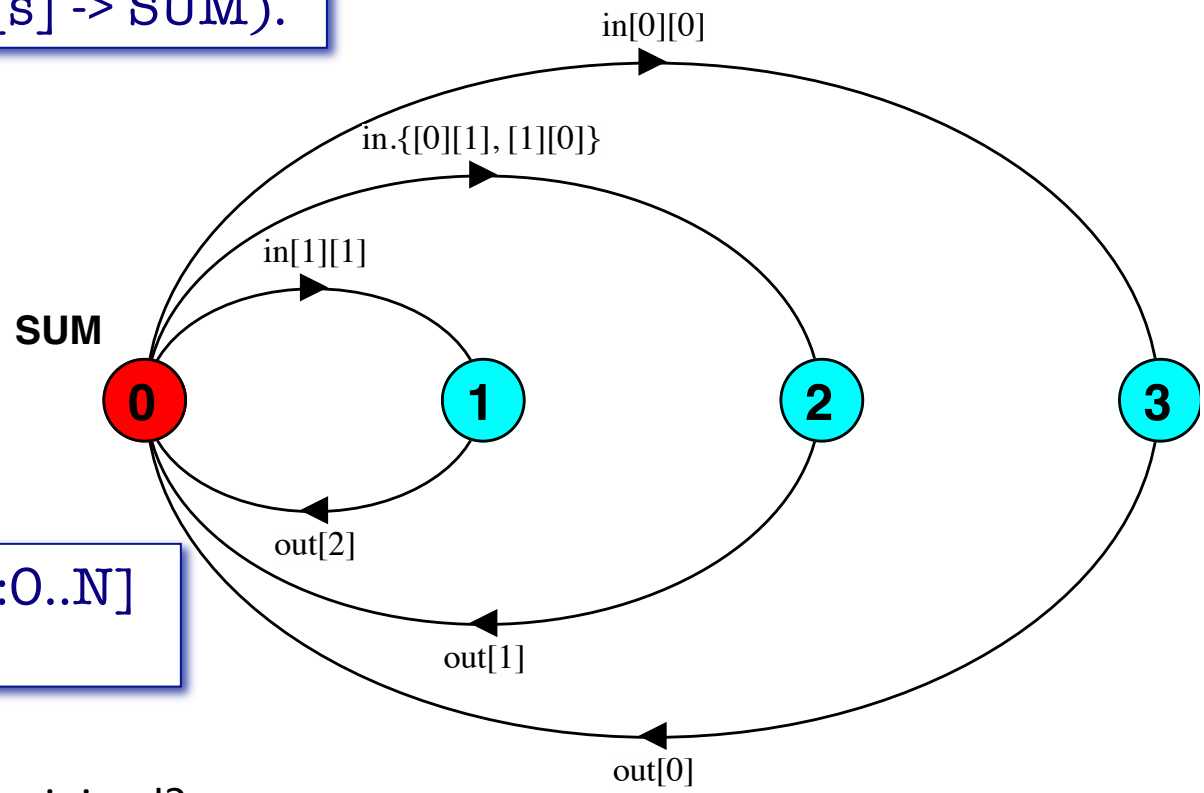
It is sometimes useful to extend the alphabet of a process with actions that it does not engage in.

```
NODRINKS = STOP + {coffee, tea}.
```

Alphabet:
{coffee, tea}

# Indexing

```
const N = 1
range I = 0 .. N
SUM = (in[i: I][j: I] -> out[s] -> SUM).
```




Pls run the following process:

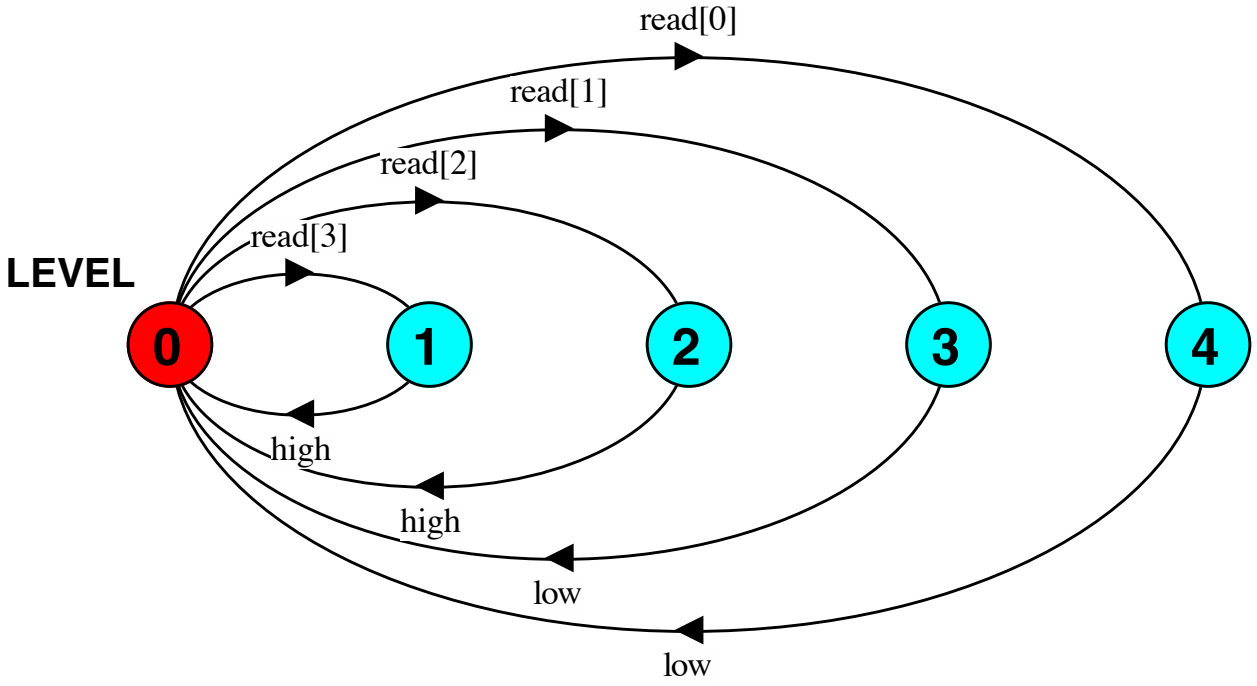
```
SUM(N=1) = (in[a:0..N][b:0..N]
-> out[a+b] -> SUM).
```

Is the LTSA generated by tool minimal?

# Conditional

**if** *expr* **then** *local_process* **else** *local_process*

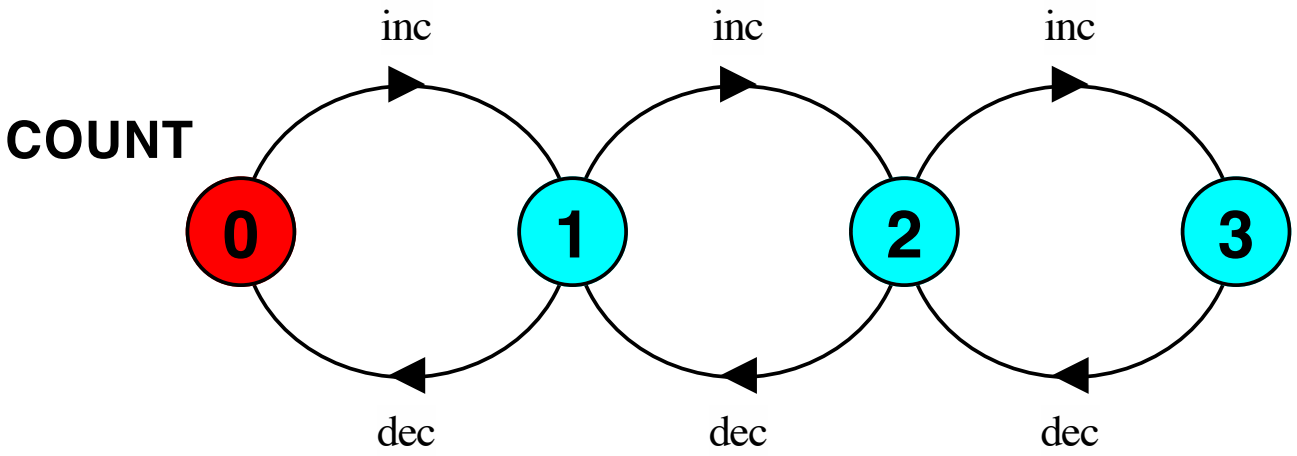
```
LEVEL = (read[x:0..3] ->
            if x>=2 then  (high -> LEVEL)
            else (low -> LEVEL)).
```



# Guards

(**when** B x->P) means that the action x is eligible when the guard B is true, otherwise x cannot be chosen for execution.

```
Proc (N=3)      =     COUNT[0],
COUNT[i:0..N]   =     ( when(i<N) inc -> COUNT[i+1]
                      | when(i>0) dec -> COUNT[i-1]
                      ).
```

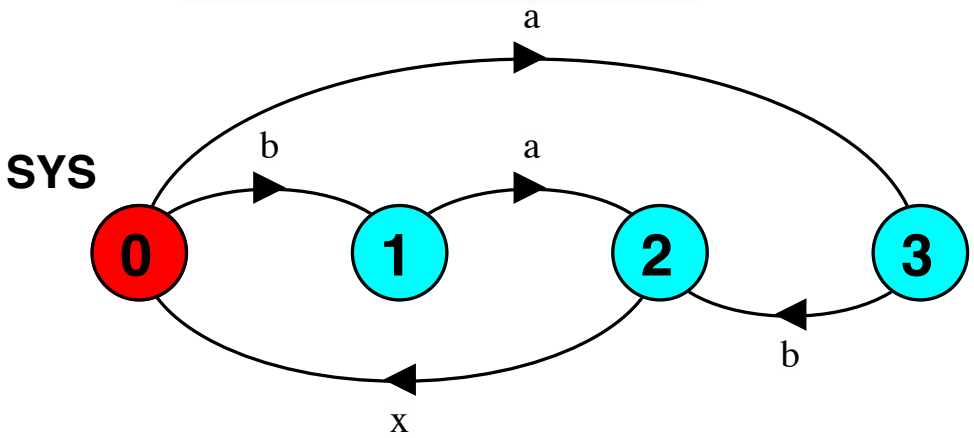


# Composite Processes

# Parallel Composition

**(P || Q)** expresses the parallel composition of the processes P and Q. It constructs an LTS which allows all the possible interleavings of the actions of the two processes.

```
A = (a -> x -> A).
B = (b -> x -> B).
||SYS = ( A || B).
```

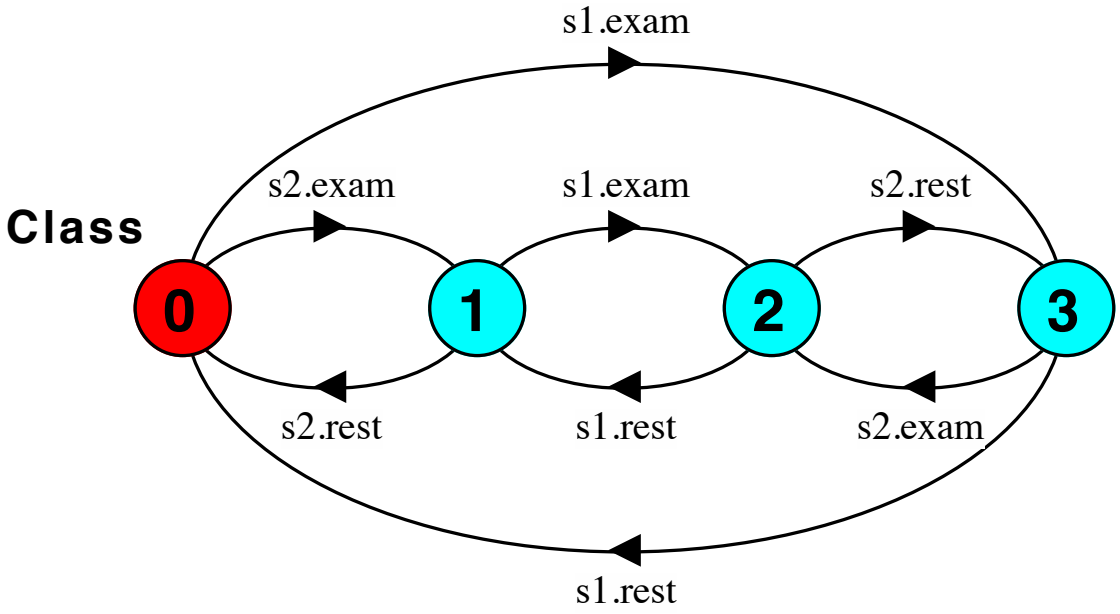



Composite process declarations are distinguished from primitive process declarations by prefixing with the symbol ||.

# Process Labeling

The process **a: P** has an alphabet in which every action label in the alphabet of P is prefixed with label a.

```
Student = (exam -> rest -> Student).

||Class  = (s1:Student || s2: Student).
```



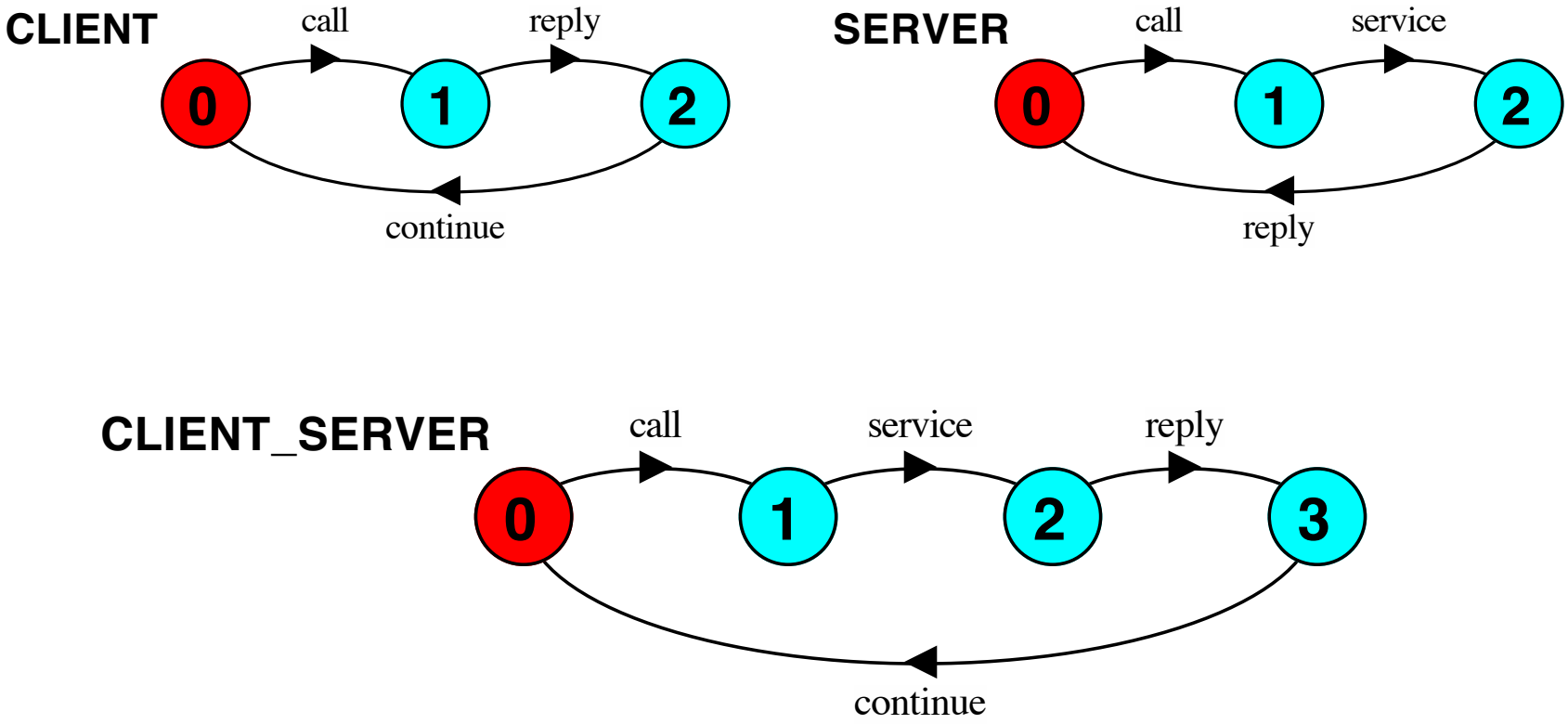
# Action reLabeling

*Relabeling* functions are applied to processes and change the names of action labels. This is usually done to ensure that composed processes *synchronize* on the correct actions.

**/{newlabel_1/oldlabel_1,.... , newlabel_n/oldlabel_n}**

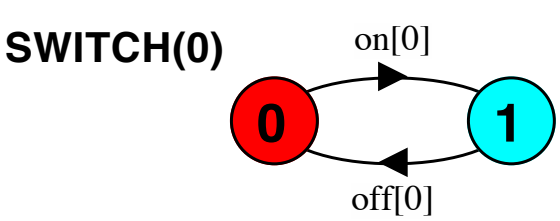
```
CLIENT = (call -> wait -> continue -> CLIENT).

SERVER = (request -> service -> reply -> SERVER).

||CLIENT_SERVER = (CLIENT || SERVER)
          /{call/request, reply/wait}.
```

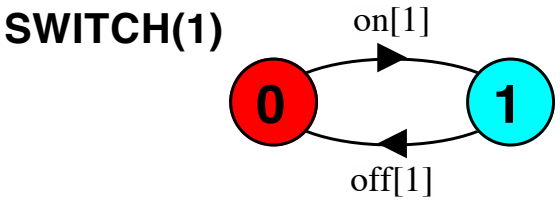
# Action reLabeling – Cont.

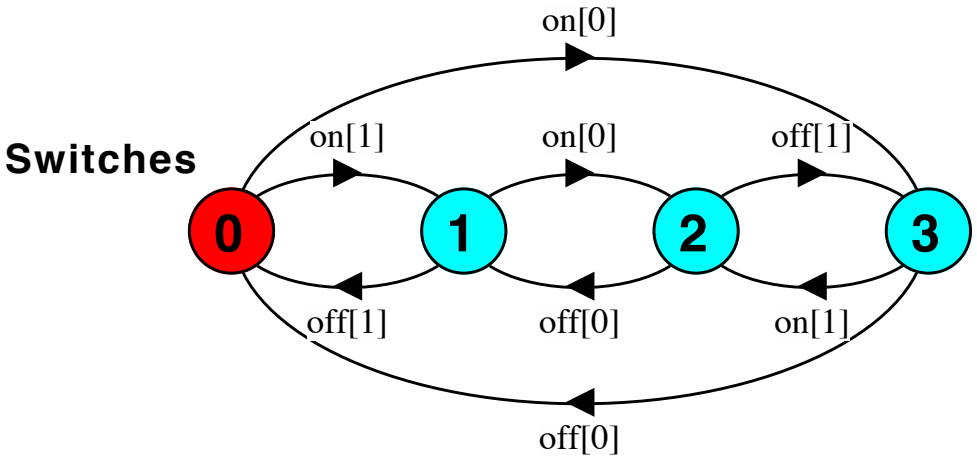


# For all – process replication

```
SWITCH(K=0) =  (on[K]->off[K]->SWITCH).

||Switches = (forall[i:0..1] SWITCH(i)).
```



# Quiz (Monday, Sep 28th)

A weekly SCHEDULE is as follows:

working and resting in weekdays and weekends, respectively.
Alphabet = {in[1..7], work, rest}, Main process: SCHEDULE

in[i], denoting the i'th day of a week, is considered as an input.

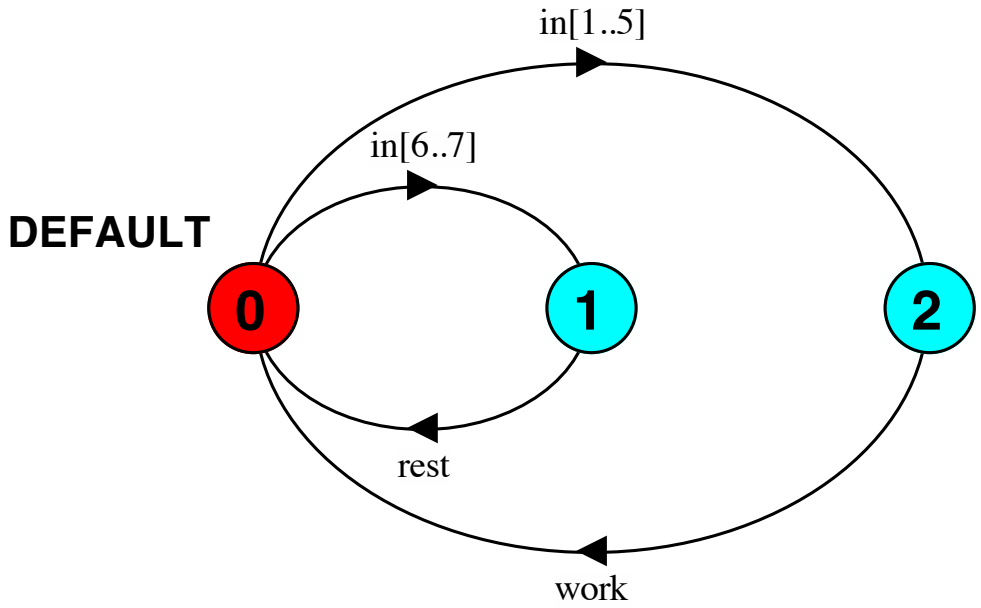
1. Specify an FSP expression for the above process.
   (Submit your FSP as "student#_Schedule.lts".)

2. Compose two instances "a" and "b" of the SCHEDULE process as || SCHEDULES (minimize it).
   (Submit the LTS and transition diagram as "student#_Schedule_Compose".lts and "student#_Schedule_Compose".pct, resp.)

# Solution (Monday, Sep 28th)

```
Schedule  = ( in[i:1..7] ->
                    if (i < 6) then (work -> Schedule)
                    else (rest -> Schedule)
                ).
```

If you run the above FSP in LTSA and then minimize its corresponding LTS, then you get the following minimal LTS:

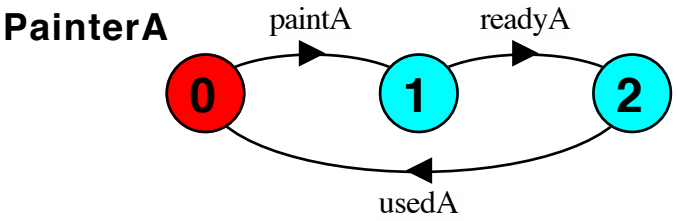


# Solution (Monday, Sep 28th) – Cont.

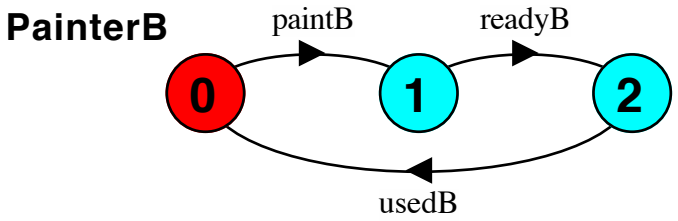
```
Schedule  = ( in[i:1..7] ->
                    if (i < 6) then (work -> Schedule)
                    else (rest -> Schedule)
                ).
||Schedules = (a: Schedule || b:Schedule).
```

# Quiz (Wed, Sep 30th)

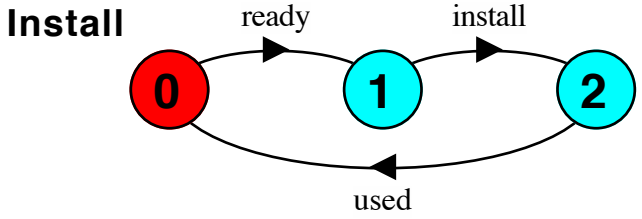
1. Give an FSP specification for each of the LTSs PainterA, PainterB, and Install.




2. Compose the three processes synchronized on {ready, readyA, readyB} and {used, usedA, usedB}. Name the composed process ||Factory.




Submit your solutions all in one file as "st#_Tut_3.lts."

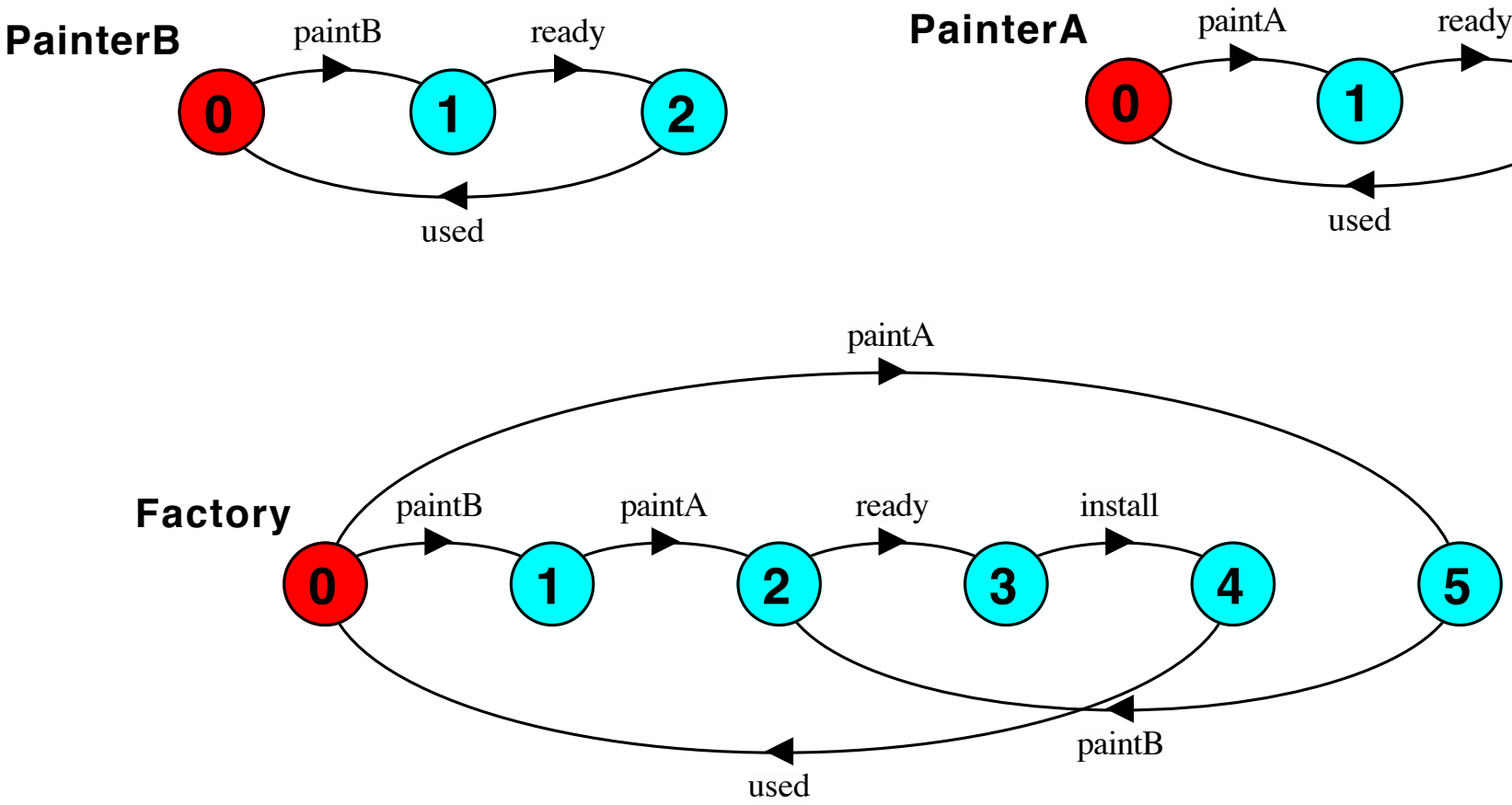


# Solution (Wed, Sep 30th)

```
PainterA = (paintA -> readyA -> usedA -> PainterA).

PainterB = (paintB -> readyB -> usedB -> PainterB).

Install = (ready -> install -> used -> Install).

||Factory = (PainterA || PainterB || Install)
                / {ready/readyA, ready/readyB, used/usedA, used/usedB}.
```

# Solution (Wed, Sep 30th) – Cont.

Running the FSP (previous slide)in LTSA, you would get the following LTSs:



# Quiz (Fri, Oct 2th)

The behavioral of the speed of a car is as follows:

In a normal case, the speed must be 4

In an emergency case, the speed is either 5 or 6.

If the speed of the car is greater than 4, a ticket is issued as follows:

$500 if the speed is 5

$600 if the speed is 6.

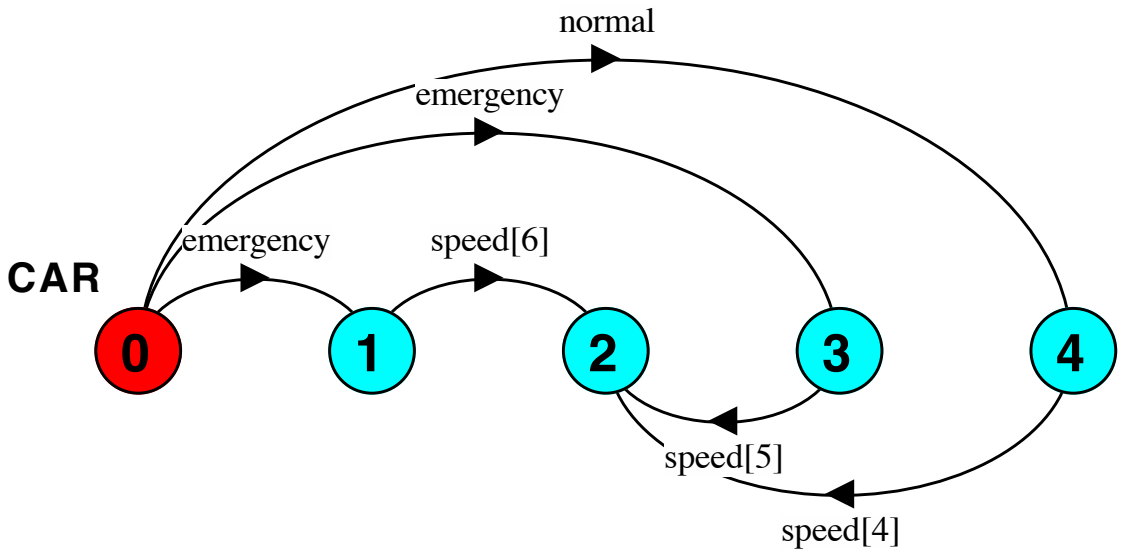
1. Write a process called CAR modeling the behavior of the speed of this car (terminating process). Hint: Alphabet = {normal, emergency, speed[4..6]}
2. Write a process called FINE modeling the behavior of fine issuing (terminating). . Hint: Alphabet = {speed[5..6], fine[500], fine[600]}
3. Compose the two process FINE and CAR (name it ||CAR_FINE).

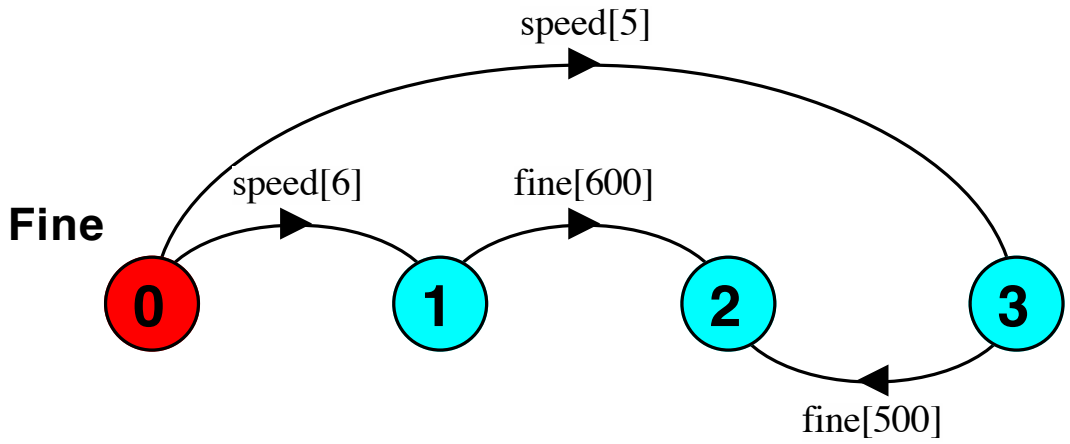
Submit your solutions all in one file as "st#_Tut_3_Fri.lts."

# Solution (Fri, Oct 2th)

```
CAR =   ( normal -> speed[4] -> STOP
        | emergency -> speed[5] -> STOP
        | emergency -> speed[6] -> STOP).

Fine = ( speed[i: 5..6] ->
          if (i == 5) then (fine[500] -> STOP)
          else (fine[600] -> STOP) ).

||CAR_FINE = (Fine || CAR).
```

# Solution (Fri, Oct 2th) – Cont.



# Solution (Fri, Oct 2th) – Cont.