# Concurrency, Parallelism and Distribution (CPD)

## Concurrency: FSP, LTS and Shared Memory

Joaquim Gabarro, gabarro@cs.upc.edu

Computer Science
Universitat Politècnica de Catalunya

Basic Readings

Processes

Modeling Concurrency

Modeling Interaction

# Basic Readings

# Readings

Models

Jeff Magee and Jeff Cramer

Concurrency, State Models & Java Programs

John Wiley & and Sons, 2006.

`http://www.doc.ic.ac.uk/~jnm/book/`

# Processes

# Modeling Processes

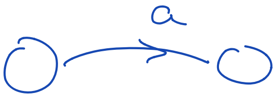
- A process is the execution of a sequential program.
- As a process executes, it transforms its states by executing statements.
- Each statement consists of a sequence of one or more atomic actions.

Install the Labelled Transition System Analyzer, LTS

`http://www.doc.ic.ac.uk/~jnm/book/`

↓

finite machine state



# FSP

- We introduce a simple algebraic notation called FSP (for Finite State Process)
- Every FSP description has a coresponding Labeled transition System.

syntax

# FSP-action prefix

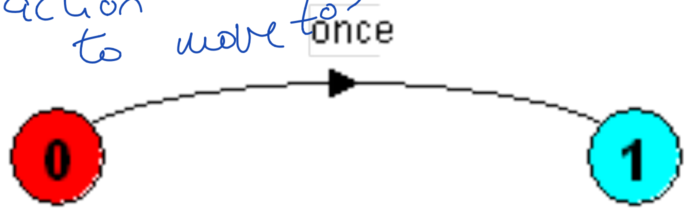
FSP-action prefix: If $x$ is an action and $P$ a process then $(x- > P)$ describes a process that initially engages in the action x and then behaves exactly as described by $P$.

action

P process

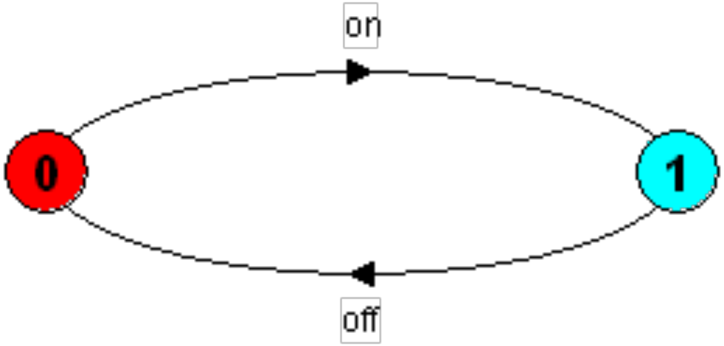
```
ONESHOT = (once -> STOP).
```

action to move to



# FSP - action prefix & recursion

Consider the following light switch



It is described in FSP as follows:

```
SWITCH = OFF,
OFF = (on -> ON),
ON = (off-> OFF).

SWITCH = OFF,
OFF = (on ->(off->OFF)).

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

# Trace

A trace corresponds to an execution of a process.

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

```
on->off->on->off->on->off->...
```

# FSP - choice

FSP - choice: If $x$ and $y$ are actions then $(x- > P|y- > Q)$ describes a process which initially engages in either of the actions $x$ or $y$. After the first action has occurred, the subsequent behavior is described by $P$ if the first action was $x$ and $Q$ if the first action was $y$.

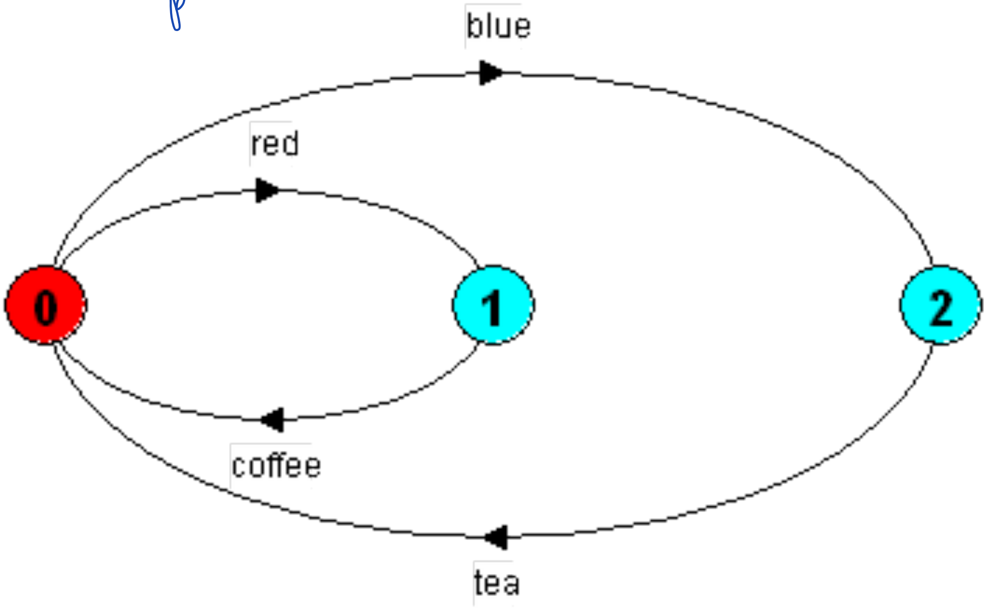
# Example: Drinking machine

push red → get coke

```
DRINKS = (red->coffee->DRINKS
         |blue->tea->DRINKS
         ).
```

or

push blue get tea



# Example: Drinking machine, traces

A process my have many possible traces

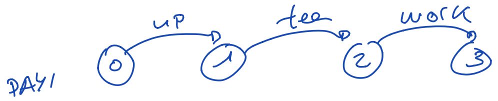
```
red->coffee-> blue->tea-> blue->tea->...

blue->tea-> red->coffee-> blue->tea->...
```

Look at the *Animator* part of the LST.

# Class Exercise: three DAYS

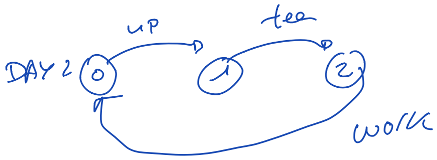
You should do this exercise by hand first and then check using the LTSA tool.

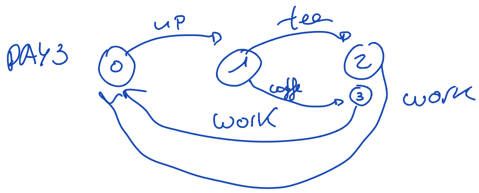
- Draw the three `DAY` LTSs, representing the actions of some-one getting up and going to work:
  - `DAY1`: get up (action `up`), then have tea (action `tea`), then go to work (action `work`), then stop
  - `DAY2`: do DAY1 repeatedly
  - `DAY3`: do DAY2, but choose between tea and coffee
- Write the FSP process definitions for the above. You can check these using the LTSA tool.
- Extend DAY3 to DAY4 to include the effects of an `alarm` with a `snooze` button, so prior to the up action, an alarm action is performed. However instead of then doing up you may do a snooze action and go back to the start.

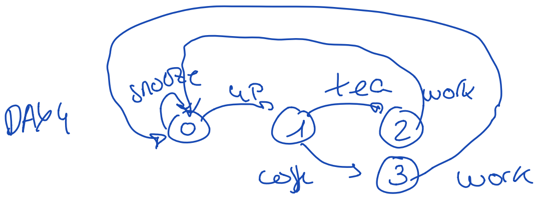


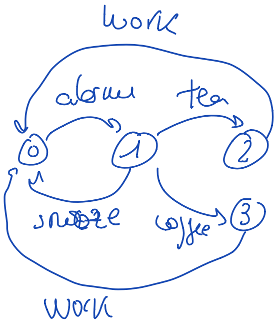
DAY 1 = (up -> tea -> work ->
STOP),

DAY 2 = (up -> tea -> work -> DAY2)

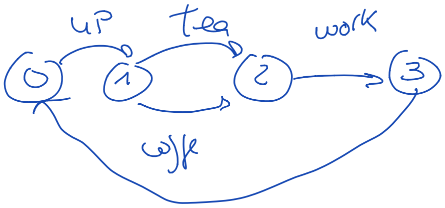




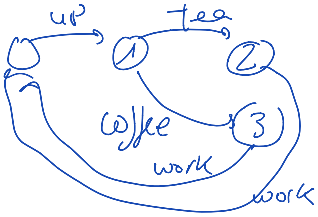




DAY 3 = (up -> (tea -> work |
coffee -> work), DAY3),



DAY 3 = (up -> (tea -> work -> DAY3 |
coffee -> work -> DAY3),



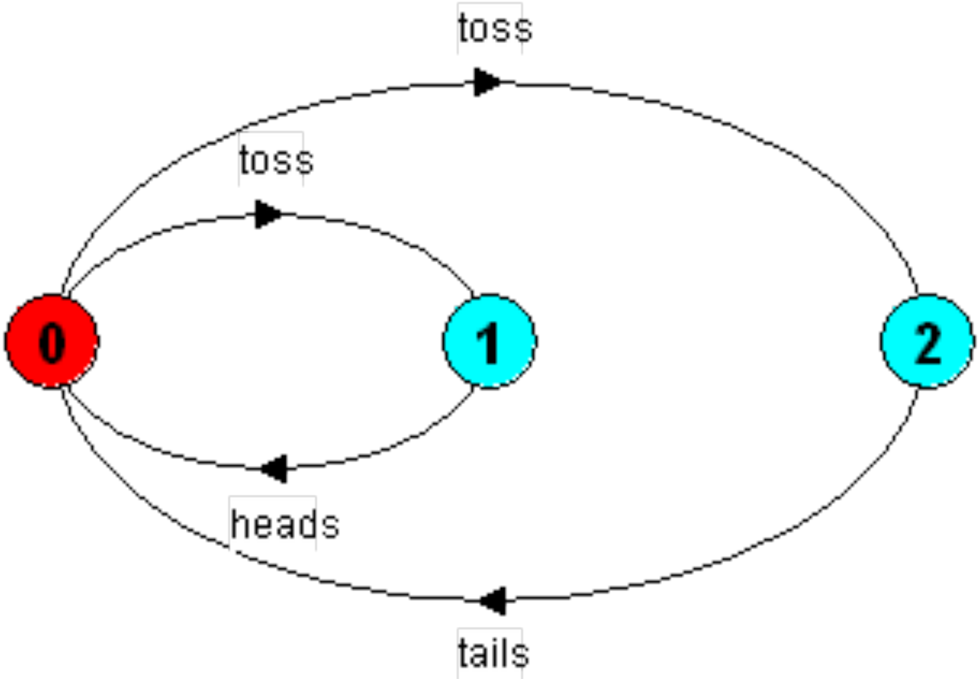
DAY4 = (alarm (snooze -> DAY4 | up -> (tea -> work | coffee -> WORK),

WORK = (work -> DAY4),

# Non-deterministic choice

Non-deterministic choice: Process $(x- > P|x- > Q)$ describes a process which engages in $x$ and then behaves as $P$ or $Q$.

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



# Modeling failure

How do we model an unreliable communication channel which accepts `in` actions and if a failure occurs produces no output, otherwise performs an `out` action?

```
CHAN = (in->CHAN
       |in->out->CHAN).
```

# FSP - indexed processes and actions

Single slot buffer that inputs a value in the range 0 to 3 and then outputs that value.

```
BUFF = (in[i:0..3]->out[i]-> BUFF).
```

equivalent to

```
BUFF = (in[0]->out[0]->BUFF
         |in[1]->out[1]->BUFF
         |in[2]->out[2]->BUFF
         |in[3]->out[3]->BUFF
         ).
```

Indexed actions generate labels of the form action.index

# FSP - indexed processes and actions

```
const N = 1
range T = 0..N
range R = 0..2*N

SUM = (in[a:T][b:T]->TOTAL[a+b]),
       TOTAL[s:R] = (out[s]->SUM).
```

# Process Parameters

Using a process parameter with default value:

```
BUFF(N=3) = (in[i:0..N]->out[i]-> BUFF).
```

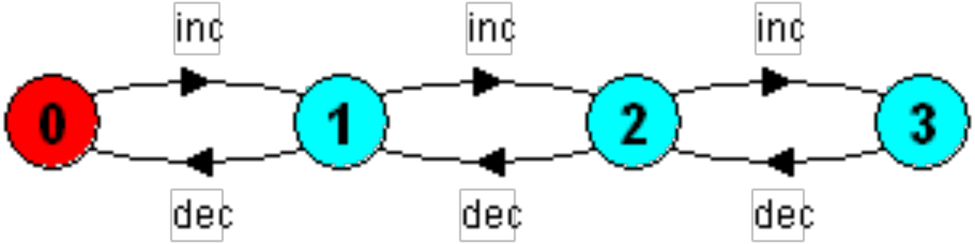
buffer parametrized by N

to have a picture, give N a value

# FSP - guarded actions

if

FSP - guarded actions: The choice (**when** $Bx- > P|y- > Q$) means that when the guard $B$ is true then the actions $x$ and $y$ are both eligible to be chosen, otherwise if $B$ is false then the action $x$ cannot be chosen.
Example:

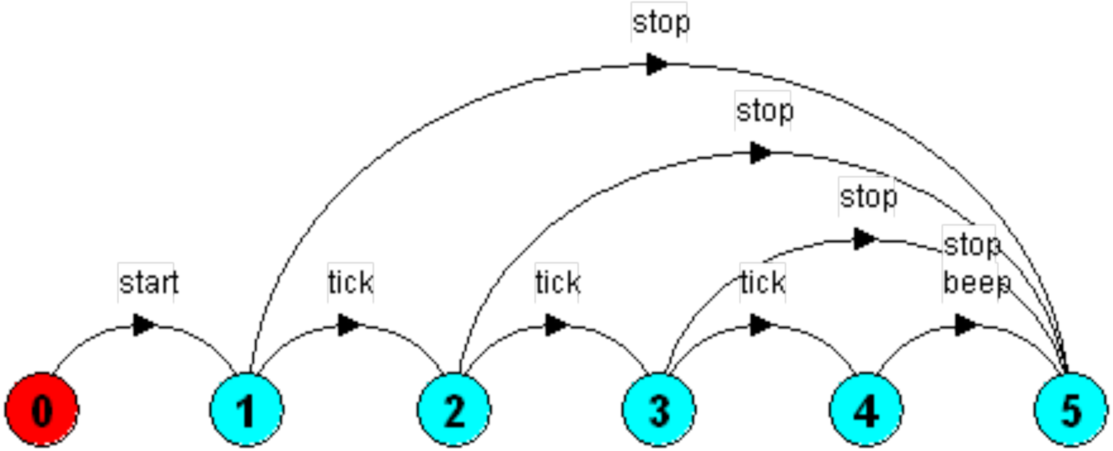


```
COUNT (N=3) = COUNT[0],
COUNT[i:0..N] = (when(i<N) inc->COUNT[i+1]
                 |when(i>0) dec->COUNT[i-1]
                ).
```

# Example

A countdown timer which beeps after N ticks, or can be stopped.

`http://www.doc.ic.ac.uk/∼jnm/`
`    book/book_applets/CountDown.html`



```
COUNTDOWN (N=3) = (start->COUNTDOWN[N]),
COUNTDOWN[i:0..N] =
          (when(i>0) tick->COUNTDOWN[i-1]
          |when(i==0)beep->STOP
          |stop->STOP).
```

# Class Exercise: SENSOR

You should do this exercise by hand first and then check using the LTSA tool.

A sensor measures the water level of a tank. The level (initially 5) is measured in units 0::9. The sensor outputs a `low` signal if the level is less than 2, a `high` signal if the level is greater than 8 and otherwise it outputs `normal`. Model the sensor as an FSP process, `SENSOR`.

Hint: The alphabet of SENSOR is

$$\{\texttt{level}[0 :: 9]; \texttt{high}; \texttt{low}; \texttt{normal}\}$$

When the sensor receives a new level it should output low, normal or high as required. This can be done either via a choice, or by specifying that each level input is followed by the appropriate output.

# Modeling Concurrency

# Modeling Concurrency

- How should we model process execution speed?
  - Arbitrary speed.
    We abstract away time.
- How do we model concurrency?
  - Arbitrary relative order of actions from different processes.
    Interleaving but preservation of each process order.
- What is the result?
  - Provides a general model independent of scheduling.
    Asynchronous model of execution.

# Parallel composition - action interleaving

Parallel Composition: If *P* and *Q* are processes then $(P||Q)$ represents the concurrent execution of *P* and *Q*. The operator $||$ is the parallel composition operator.

```
ITCH = (scratch->STOP).
CONVERSE = (think->talk->STOP).
||CONVERSE_ITCH = (ITCH || CONVERSE).
```

Possible traces as a result of action interleaving.

```
think->talk->scratch
think->scratch->talk
scratch->think->talk
```

# Modeling Interaction

# Modeling Interaction

Shared actions: If processes in a composition have actions in common, these actions are said to be shared.

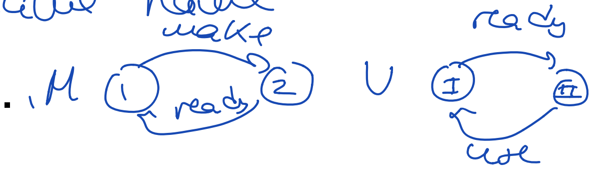
- Unshared actions may be arbitrarily interleaved.
- shared action must be executed at the same time by all processes that participate in the shared action.

# Maker user example

A MAKER manufacturates and item (action make) and signals to the process USER that the item is ready (by a shared action ready). The USER process can only use the item (action use) after the signal.

share = same name

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



LTS Analyzer → Edit (write the program) → C (Compile) → || (compose) → Draw → click ||MAKER_USER

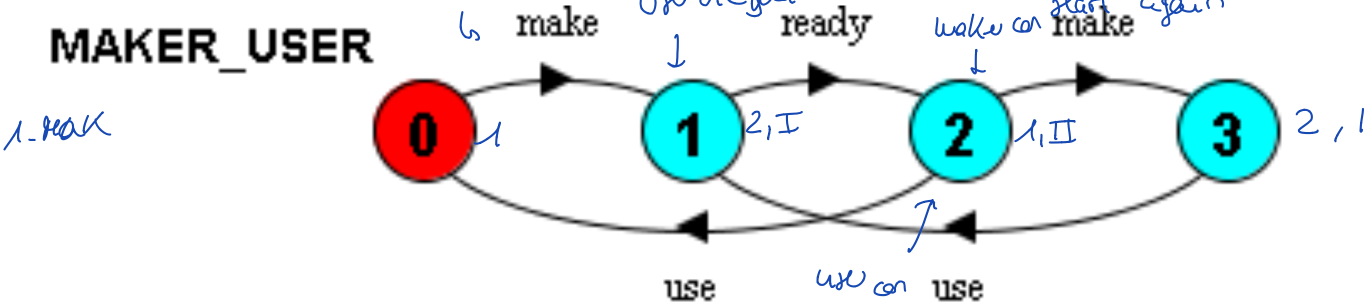
it's the only thing I can do to start, make does it

use and make can execute ready

make can start again

use can use

1_MAK



# Manual construction of the LTS (1)

1) Unfolding the processes in the initial state 0 we get

(MAKER || USER)

= (make->ready->MAKER ||ready->use->USER)

2) As ready needs to be executed by both processes ”at the same time”, the only possible transition from the initial state is to the state 1

$$(\text{MAKER} \,||\, \text{USER}) \xrightarrow{\text{make}} (\text{ready->MAKER}||\text{ready->use->USER})$$

3) Both processes execute (at the same time) ready and go to state 2:

$$(\text{ready->MAKER}||\text{ready->use->USER}) \xrightarrow{\text{ready}} (\text{MAKER}||\ \text{use->USER})$$

# Manual construction of the LTS (2)

4) Unfolding state 2 we get

(MAKER|| use->USER) = (make->ready->MAKER|| use->USER)

Two transitions are available going to states 3 and 0:

(make->ready->MAKER|| use->USER)
$\xrightarrow{\text{make}}$ (ready->MAKER|| use->USER)

(make->ready->MAKER|| use->USER)
$\xrightarrow{\text{use}}$ (ready->MAKER||USER)

5) From state 3 the following trantion moves to state 1:

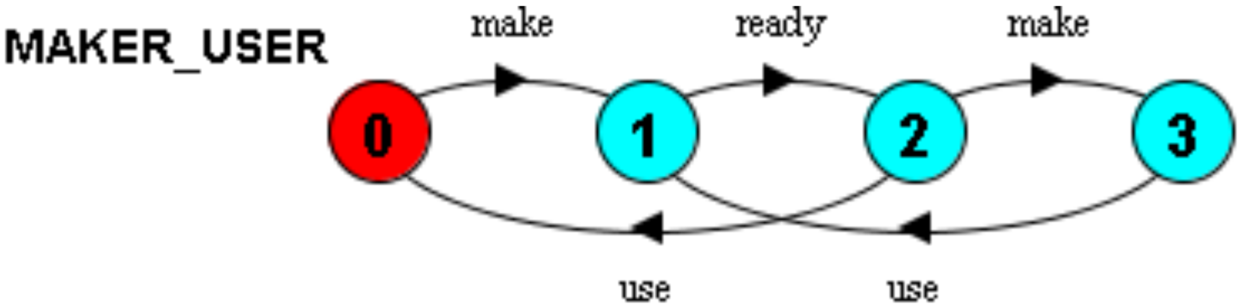
(ready->MAKER|| use->USER) $\xrightarrow{\text{use}}$ (ready->MAKER||USER)

# Manual construction of the LTS (3)

Give a fine grained description of the states using unfolding when needed.

<table>
<tr><th>state</th><th>description</th></tr>
<tr><td>0</td><td>(MAKER || USER) =<br>(make->ready->MAKER || USER)=<br>(make->ready->MAKER || ready->use->USER)</td></tr>
<tr><td>1</td><td>(ready->MAKER || USER)=<br>(ready->MAKER || ready->use->USER)</td></tr>
<tr><td>2</td><td>(MAKER || use->USER)=<br>(make->ready->MAKER || use->USER)</td></tr>
<tr><td>3</td><td>(ready->MAKER || use->USER)</td></tr>
</table>

# Traces

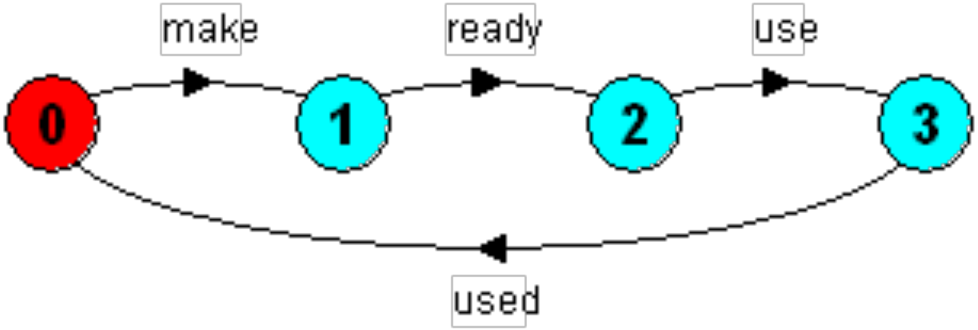


As expected
make->ready->use->make->ready->use-> · · ·

Also
make->ready->make->use->ready->use->make-> · · ·

# Handshake

```
MAKERv2 = (make->ready->used->MAKERv2).
USERv2 = (ready->use->used ->USERv2).
||MAKER_USERv2 = (MAKERv2 || USERv2).
```



make->ready->use->used->make->ready->use->use-> · · ·

The model does not distinguish wich process instigates a shared action even though it is natural to think of the MAKER instigating the ready and the USER instigating the used action.

# Multi-party synchronization

```
MAKE_A =(makeA->ready->used->MAKE_A).
MAKE_B = (makeB->ready->used->MAKE_B).
ASSEMBLE = (ready->assemble->used->ASSEMBLE).

||FACTORY = (MAKE_A || MAKE_B || ASSEMBLE).
```

# Class Exercise: `||MICROWAVE`

(1) Draw (at hand and check with the program) the LTS:

```
MICROWAVE = (put_food_in -> SETTINGS),
SETTINGS = (set_heat_level -> set_time -> COOK
            |set_time -> set_heat_level -> COOK),
COOK = (cook -> take_food_out -> MICROWAVE).
```

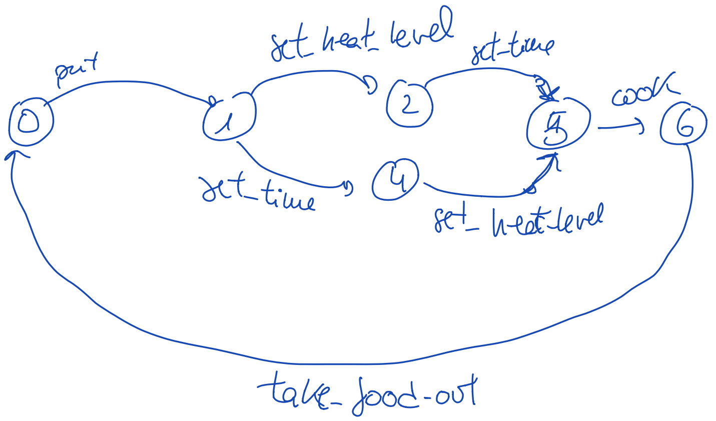
(2) Model again the `MICROWAVE` using parallel composition. *Hint*: You will need to use handshaking with shared actions, so that it is not possible to produce silly action traces. eg to cook after take food out.

```
COOK = ( put_food_in -> .... ->  take_food_out ->COOK).
SET_HEAT = ( put_food_in -> ... -> cook -> SET_HEAT).
SET_TIME = ...
```

such that

```
||MICROWAVE = ( COOK||SET_HEAT||SET_TIME).
```

MICROWAVE



COOK = (put_food-in, food_introduced,

SET_HEAT = (put-food_in, food-introduced, set_heat

SET_TIME = (put-food_in, food_introduced, set_time

COOK = (put_food-in → cook → take_food-out → COOK)

SET_HEAT = (put_food-in, set_heat_level, → cook →

SET_HEAT

sync

SET_TIME = (put_food_in → set_time → cook → SET_TIME)

S = (on → off → S)

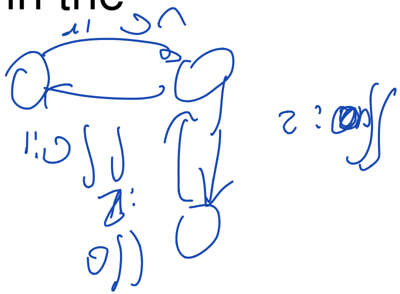
S2 = (on → off → S2)

(S || S2) = (on → off) → (S || S2))

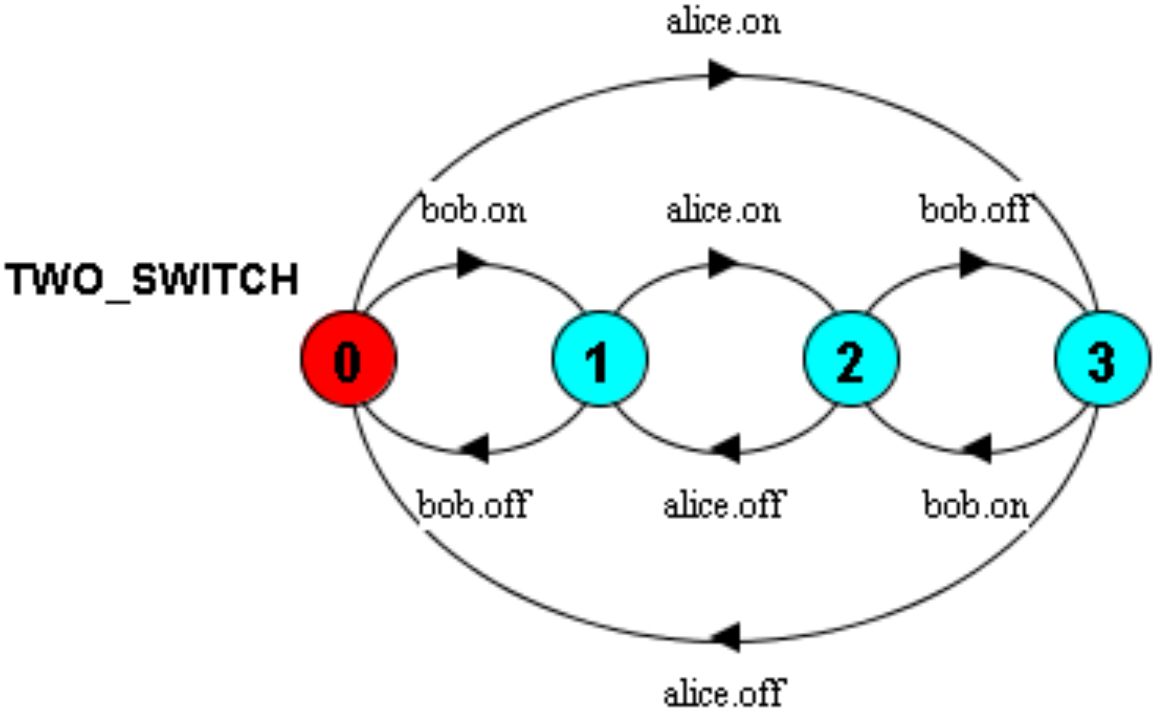
we need to give names

# Process relabeling

Process naming: $a : P$ prefixes each action label in the alphabet of $P$ with $a$.



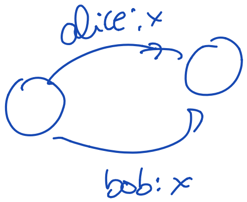
SWITCH = (on->off->SWITCH).
||TWO_SWITCH =(alice:SWITCH||bob:SWITCH).



# An array of instances of processes

```
||SWITCHES(N=3) = (forall[i:1..N] s[i]:SWITCH).

||SWITCHES(N=3) = (s[i:1..N]:SWITCH).
```

# Process labeling by a set of prefix labels



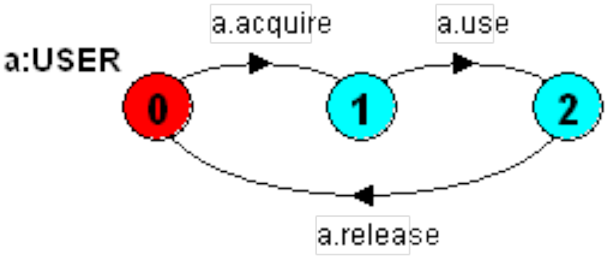
Process labeling by a set of prefix labels: Given P,

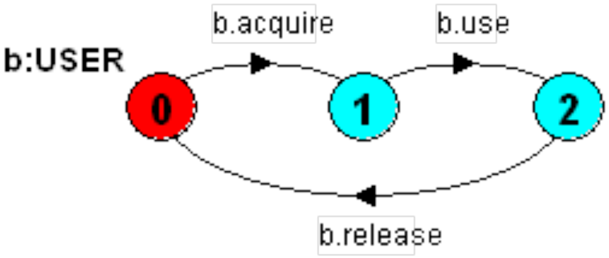
- {a1,..,ax}::P replaces every action label n with the labels a1.n,. . . ,ax.n. ;)
- Further, every transition (n->X) in the definition of P is replaced with the transitions ({a1.n,. . . ,ax.n} ->X).

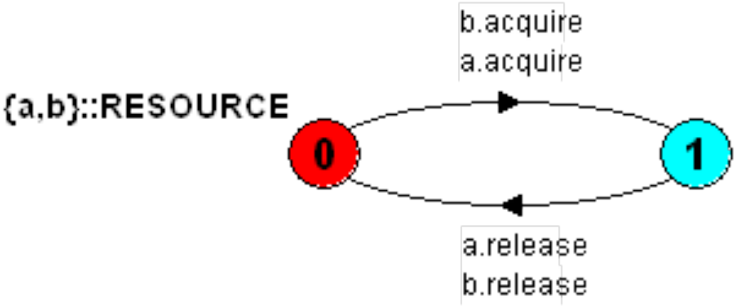
Action can be executed by Alice or Bob

# Class Exercise: RESOURCE_SHARE

```
RESOURCE=(acquire->release->RESOURCE).
USER=(acquire->use->release->USER).
||RESOURCE_SHARE=(a:USER||b:USER||{a,b}::RESOURCE).
```







Give a picture of the LTS corresponding to RESOURCE_SHARE

# Action relabeling

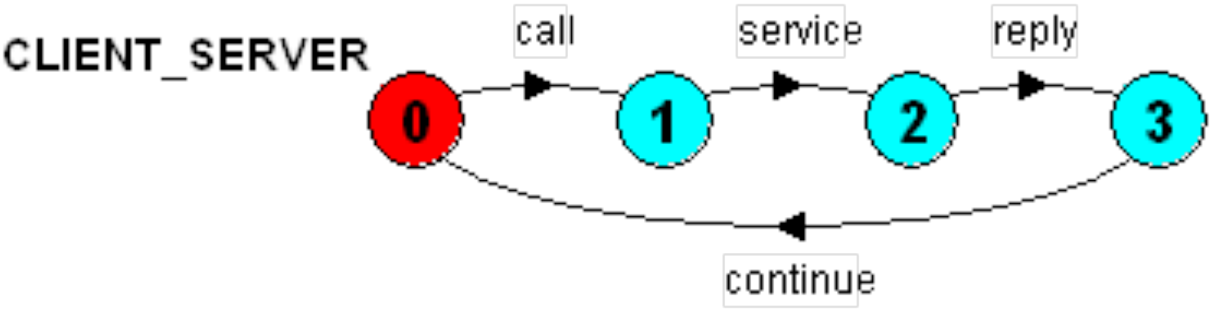
cambio de formal parametros a ~~para~~ valores

Action relabeling: Relabeling functions are applied to processes to change the names of action labels. The general form of the relabeling function is:

$$/\{newlabel\_1/oldlabel\_1, \ldots newlabel\_n/oldlabel\_n\}$$

Example: A SERVER process that provides some service and a CLIENT process that invoques the service.

CLIENT = (call->wait->continue->CLIENT).
SERVER = (request->service->reply->SERVER).
||CLIENT_SERVER = (CLIENT || SERVER)/{call/request, reply/wait}.



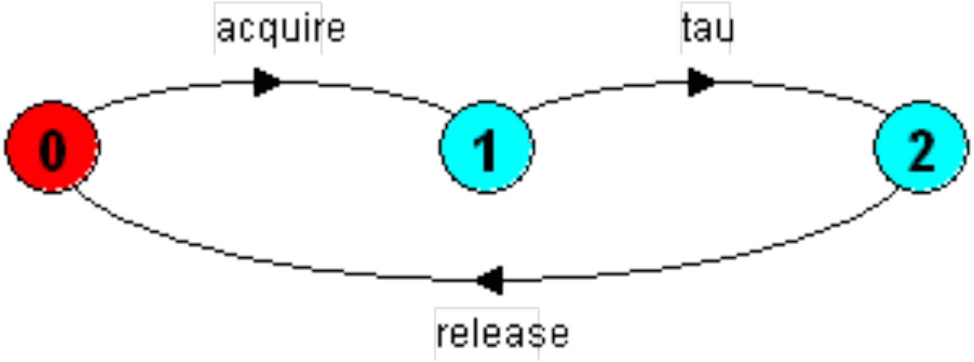
# Action hiding

Action hiding: When applied to a process $P$, the hiding operator $\backslash\{a1, .., ax\}$ removes the action names $a1, .., ax$ from the alphabet of P and makes these concealed actions silent.

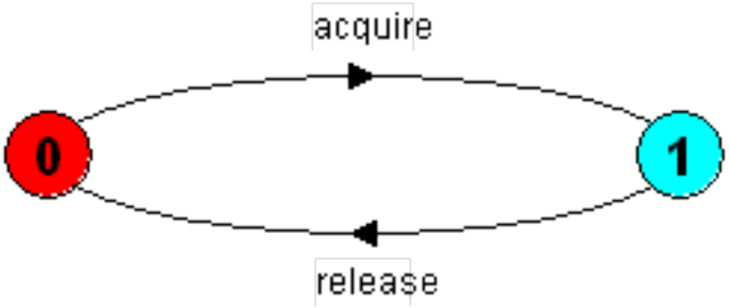
- These silent actions are labeled tau.
- Silent actions in different processes are not shared.

# Hiding example

USER = (acquire->use->release->USER) \{use}.



Minimizing



# Class Exercise: ||FACTORY

Consider the following FACTORY assembling three parts make_A, make_B and make_C into a final output:

```
MAKER_A=(make_A->ready->restart->MAKER_A).
MAKER_B=(make_B->ready->restart->MAKER_B).
ASSEMBLER_A_B =
   (ready->assemble_A_B->ready_two->ASSEMBLER_A_B).
ASSEMBLER=
   (ready_two->make_C->assemble_A_B_C
    ->output->restart->ASSEMBLER).

||FACTORY= (MAKER_A||MAKER_B||ASSEMBLER_A_B
              ||ASSEMBLER)\{ready,ready_two}.
```

Give a picture the LST corresponding to FACTORY. Comment briefly the result.
Give a picture of the preceding LST after minimising (pressing the button $\mathcal{M}$). Explain intuitively the result.