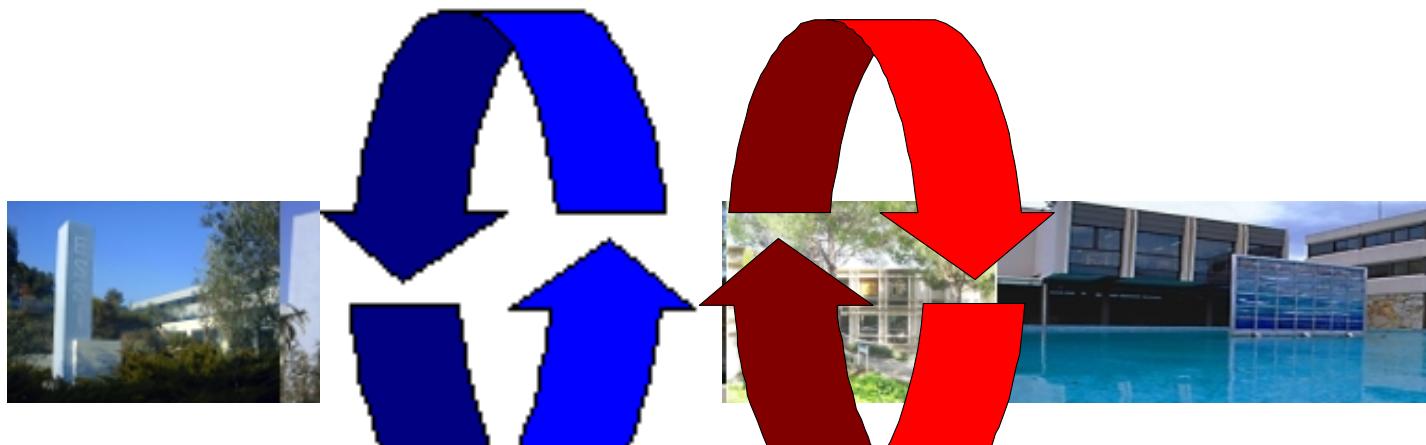


# Modélisation des processus

[riveill@unice.fr](mailto:riveill@unice.fr)

<http://www.i3s.unice.fr/~riveill>

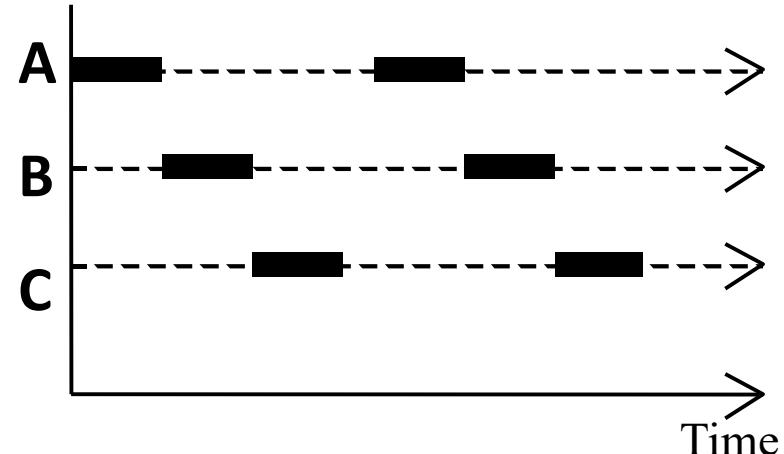


## Step 2 : modeling independent processus

# reminder

## ◆ *Concurrency*

- *Logical/simultaneous processing.*
- Does not imply multiple processing elements (PEs).
- Requires interleaved execution on a single PE.



## ◆ *Parallelism*

- *Physically simultaneous processing.*
- Involves multiple PEs and/or independent device operations.

Both concurrency and parallelism require controlled access to shared resources. We use the terms parallel and concurrent interchangeably and generally do not distinguish between real and pseudo-concurrent execution.

# How to model 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

- If  $P$  and  $Q$  are processes then  $(P \parallel Q)$  represents the concurrent execution of  $P$  and  $Q$ . The operator  $\parallel$  is the parallel composition operator.

ITCH = (scratch->STOP) .

CONVERSE = (think->talk->STOP) .

||CONVERSE\_ITCH = (ITCH || CONVERSE) .

Disjoint  
alphabets

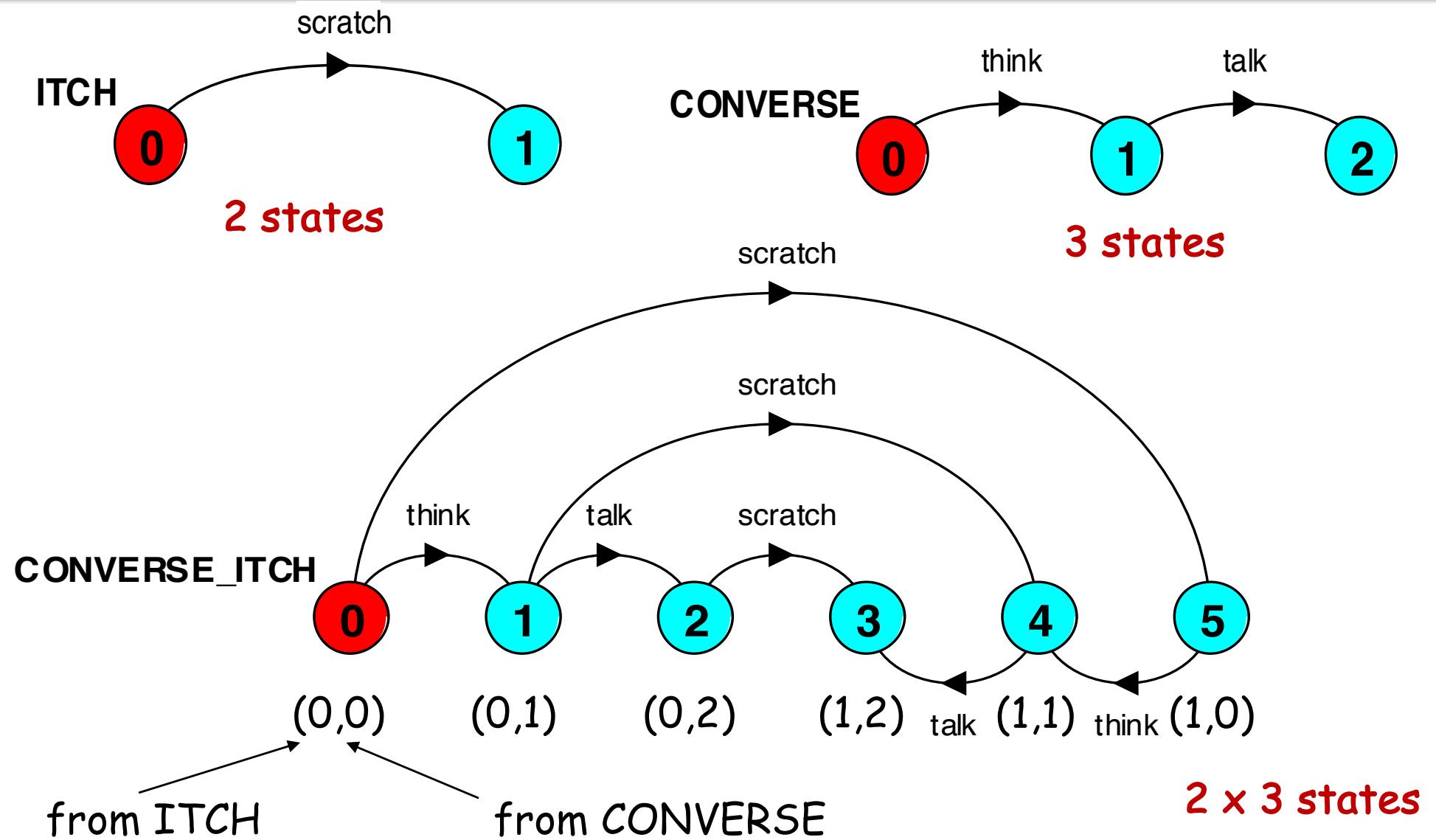
- Possible traces as a result of action interleaving.

think → talk → scratch

think → scratch → talk

scratch → think → talk

# parallel composition - action interleaving



# parallel composition - algebraic laws

- **Idempotente:**  $(P \parallel P) = P$
- **Commutative:**  $(P \parallel Q) = (Q \parallel P)$
- **Associative:**  $(P \parallel (Q \parallel R)) = ((P \parallel Q) \parallel R)$   
 $= (P \parallel Q \parallel R).$
- *Clock radio example:*
- $\text{CLOCK} = (\text{tick} \rightarrow \text{CLOCK}) .$
- $\text{RADIO} = (\text{on} \rightarrow \text{off} \rightarrow \text{RADIO}) .$
- $\text{CLOCK\_RADIO} = (\text{CLOCK} \parallel \text{RADIO}) .$
- *LTS? Traces? Number of states?*

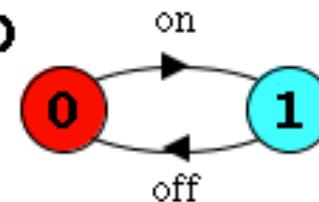
# parallel composition - algebraic laws

- *LTS :*

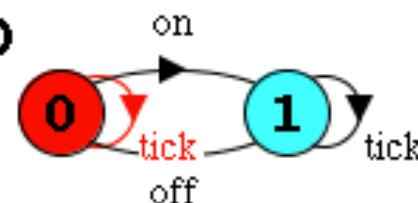
CLOCK



RADIO



CLOCK\_RADIO



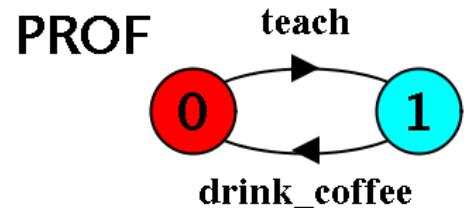
- *Traces = tick->tick->on->tick->off->...*

- *Number of states = 1 \* 2 = 2*

# TEST

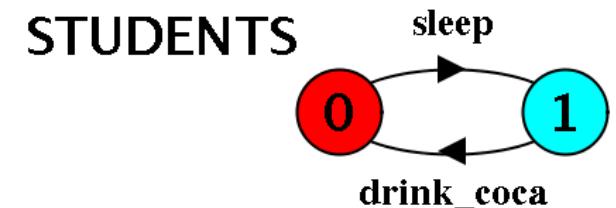
**PROF = (teach->drink\_coffee->PROF) .**

- *LTS ?*



**STUDENTS = (sleep->drink\_coca->STUDENTS) .**

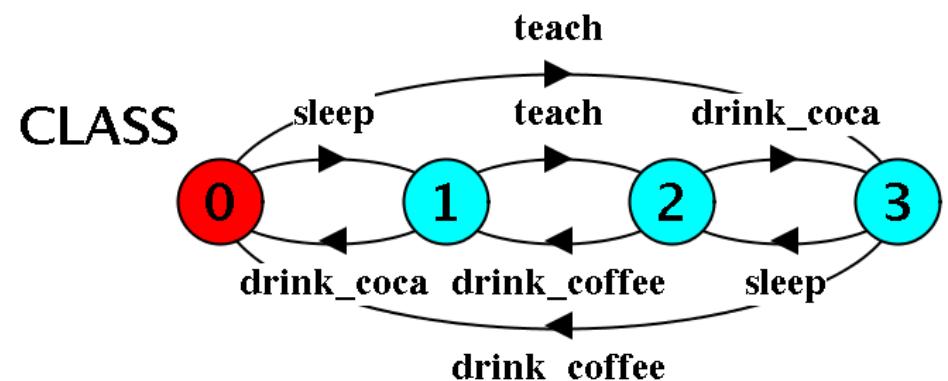
- *LTS ?*



**||CLASS = (PROF || STUDENTS) .**

- *How many states ?*
- *LTS of ||CLASS ?*

*2 x 2 states*



## Step 3 : modeling concurrent processus

# modeling interaction - shared actions

- If processes in a composition have actions in common, these actions are said to be *shared*. Shared actions are the way that process interaction is modeled. While unshared actions may be arbitrarily interleaved, a shared action must be executed at the same time by all processes that participate in the shared action.

**MAKER** = (make->**ready**->MAKER) .

**USER** = (**ready**->use->USER) .

| | **MAKER\_USER** = (MAKER || USER) .

Non-disjoint alphabets

- **MAKER** synchronizes with **USER** when **ready**.
- *LTS? Traces? Number of states?*

# modeling interaction - shared actions

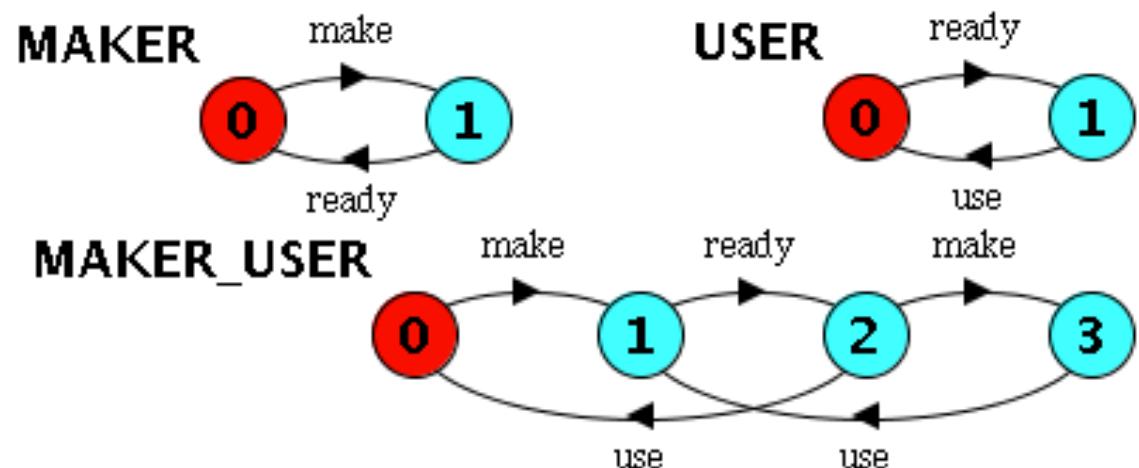
- *FSP =*

MAKER = (make->ready->MAKER) .

USER = (ready->use->USER) .

| | MAKER\_USER = (MAKER || USER) .

- *LTS =*



- *Traces*

- make->ready->use->...

- *Number of states? 4*

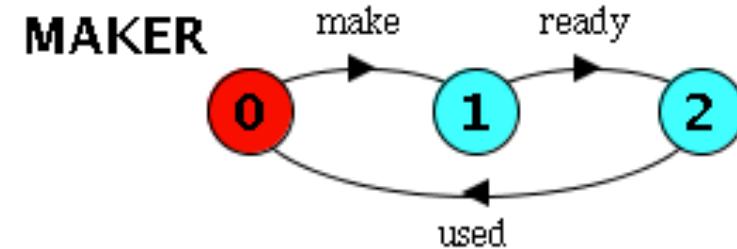
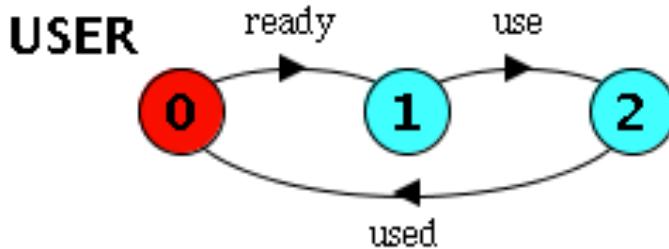
# TEST

- A handshake is an action acknowledged by another:

MAKER = (make->ready->used->MAKER) .

USER = (ready->use->used ->USER) .

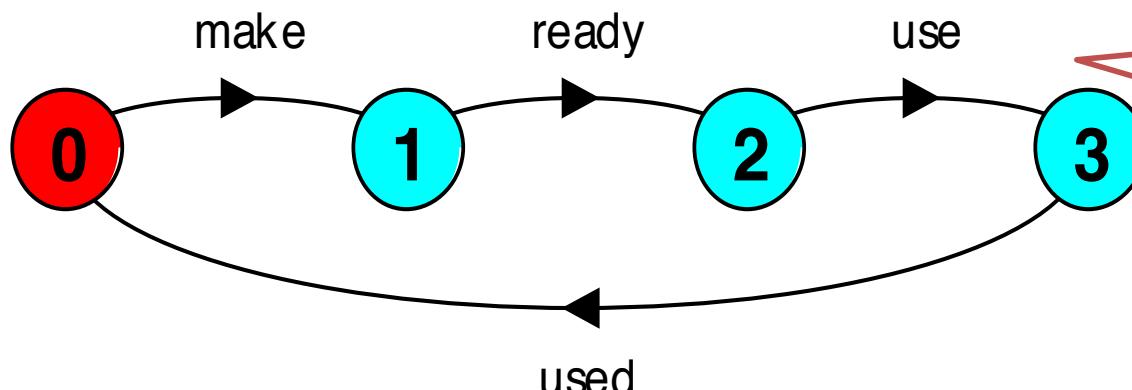
| | MAKER\_USER = (MAKER || USER) .



- LTS?

3 x 3  
states?

No : only 4 states (non disjoint alphabet)  
→ ready actions occur simultaneously



Interaction  
constrains the  
overall  
behaviour.

# modeling interaction - multiple processes

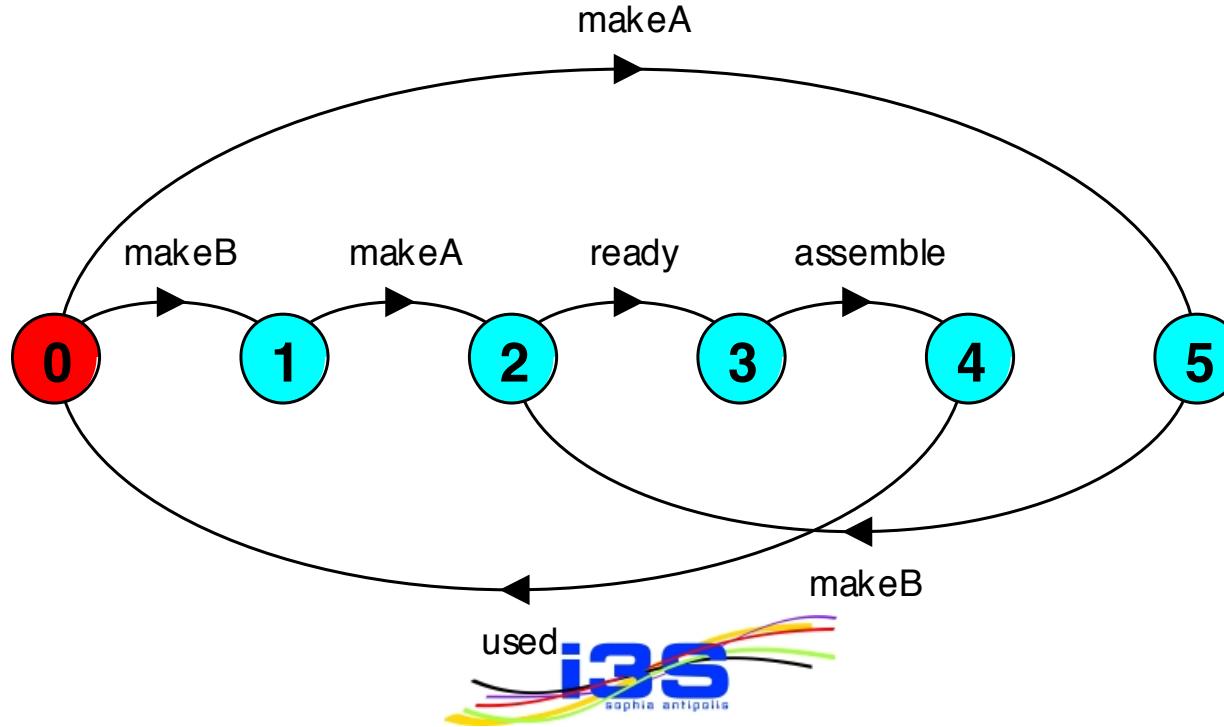
- Multi-party synchronization:

**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**) .



# composite processes

- A composite process is a parallel composition of primitive processes. These composite processes can be used in the definition of further compositions.

`||MAKERS = (MAKE_A || MAKE_B) .`

`||FACTORY = (MAKERS || ASSEMBLE) .`

- Substituting the definition for `MAKERS` in `FACTORY` and applying the commutative and associative laws for parallel composition results in the original definition for `FACTORY` in terms of primitive processes.

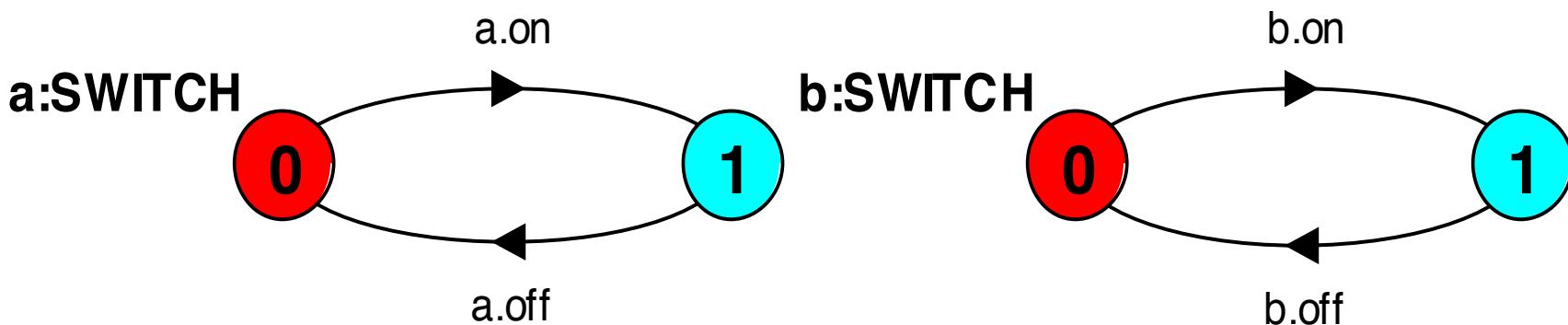
`||FACTORY = (MAKE_A || MAKE_B || ASSEMBLE) .`

# process instances and labeling

- $a:P$ 
  - prefixes each action label in the alphabet of  $P$  with  $a$ .
- Two **instances** of a switch process:

$\text{SWITCH} = (\text{on} \rightarrow \text{off} \rightarrow \text{SWITCH}) .$

$\parallel \text{TWO\_SWITCH} = (a:\text{SWITCH} \parallel b:\text{SWITCH}) .$



$\parallel \text{TWO\_SWITCH} = (\{a, b\}:\text{SWITCH}) .$

- An array of **instances** of the switch process:

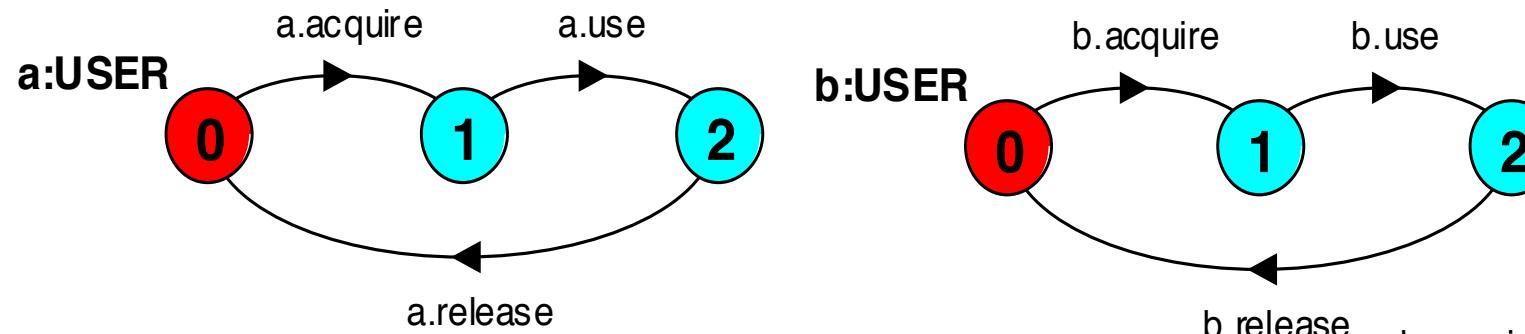
$\parallel \text{SWITCHES (N=3)} = (\text{forall } [i:1..N] \ s[i]:\text{SWITCH}) .$

$\parallel \text{SWITCHES (N=3)} = (\{s[i]:1..N\}:\text{SWITCH}) .$

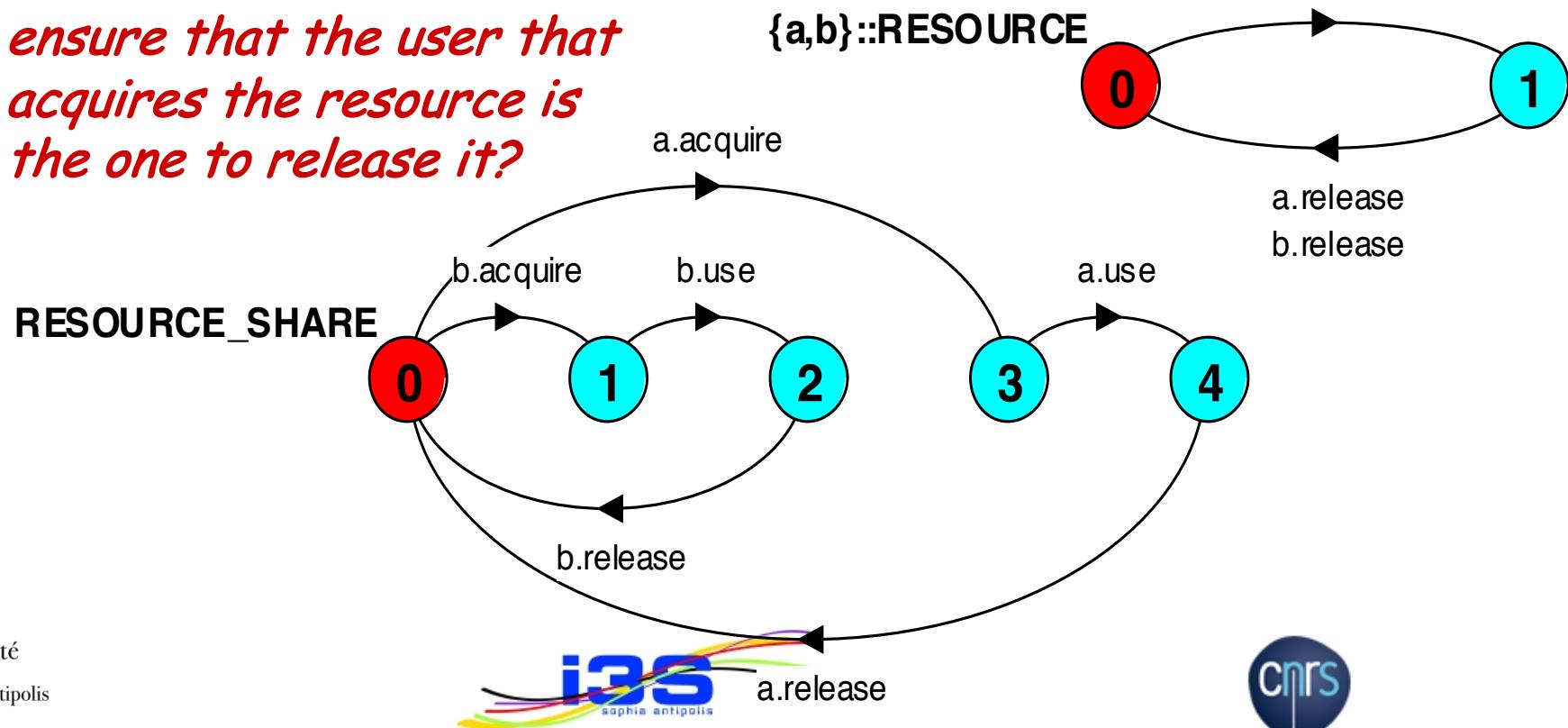
# process labeling by a set of prefix labels

- $\{a_1, \dots, a_x\} :: P$ 
  - replaces every action label  $n$  in the alphabet of  $P$  with the labels  $a_1.n, \dots, a_x.n$ . Further, every transition  $(n \rightarrow X)$  in the definition of  $P$  is replaced with the transitions  $(\{a_1.n, \dots, a_x.n\} \rightarrow X)$ .
- Process prefixing is useful for modeling **shared** resources:  
 $\text{RESOURCE} = (\text{acquire} \rightarrow \text{release} \rightarrow \text{RESOURCE}) .$   
 $\text{USER} = (\text{acquire} \rightarrow \text{use} \rightarrow \text{release} \rightarrow \text{USER}) .$   
 $\text{|| RESOURCE\_SHARE} = (\text{a:USER} \parallel \text{b:USER}$   
 $\qquad \qquad \qquad \parallel \{a,b\} :: \text{RESOURCE}) .$

# process prefix labels for shared resources



*How does the model ensure that the user that acquires the resource is the one to release it?*



# action relabeling

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

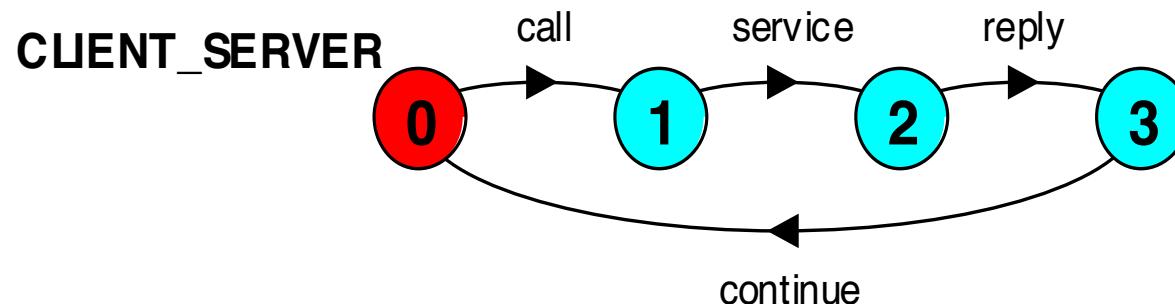
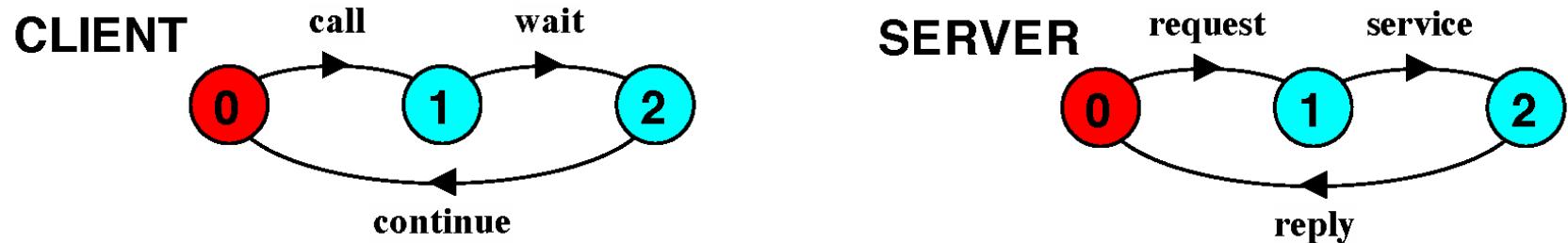
*/ {newlabel\_1/oldlabel\_1, ... newlabel\_n/oldlabel\_n}.*

- Relabeling to ensure that composed processes synchronize on particular actions.

CLIENT = (**call**->**wait**->**continue**->CLIENT) .  
          ↑  
          ↔  
          ↓  
SERVER = (**request**->**service**->**reply**->SERVER) .

# action relabeling

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



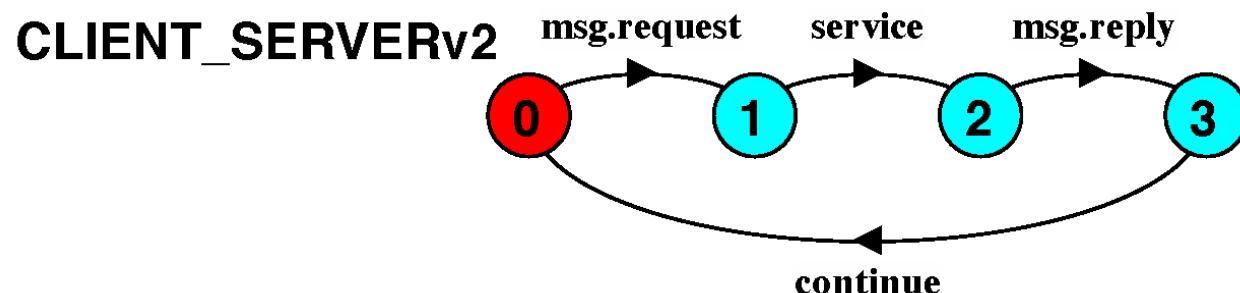
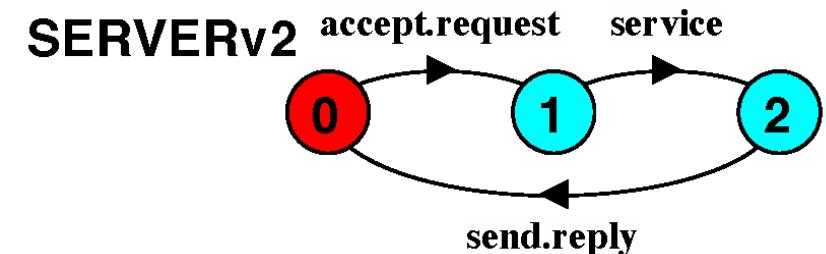
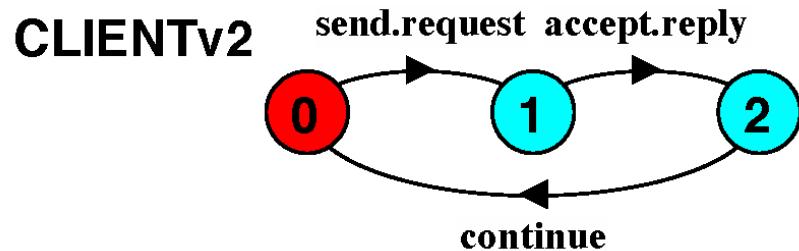
# action relabeling - prefix labels

- An alternative formulation of the client server system is described below using qualified or prefixed labels:

```
SERVERv2 = (accept.request  
             ->service->accept.reply->SERVERv2) .  
  
CLIENT = (call.request  
             ->call.reply->continue->CLIENT) .  
  
||CLIENT_SERVERv2 = (CLIENT || SERVERv2)  
                     /{msg/accept, msg/call} .
```

# action relabeling - prefix labels

```
SERVERV2 = (accept.request  
           ->service->send.reply->SERVRV2) .  
  
CLIENTv2 = (send.request  
           ->accept.reply->continue->CLIENTv2) .  
  
||CLIENT_SERVERv2 = (CLIENTv2 || SERVERv2)  
                     / {msg/accept, msg/send} .
```



# action **hiding** - abstraction to reduce complexity

- When applied to a process P, the hiding operator  $\backslash\{a_1..a_x\}$  removes the action names  $a_1..a_x$  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.
- Sometimes it is more convenient to specify the set of labels to be **exposed**....
- When applied to a process P, the interface operator  $@\{a_1..a_x\}$  hides all actions in the alphabet of P not labeled in the set  $a_1..a_x$ .

# action hiding

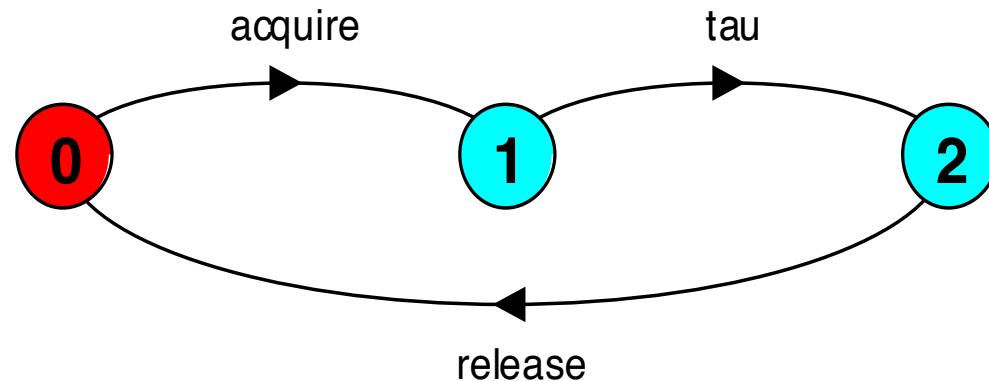
- The following definitions are equivalent:

$\text{USER} = (\text{acquire} \rightarrow \text{use} \rightarrow \text{release} \rightarrow \text{USER})$

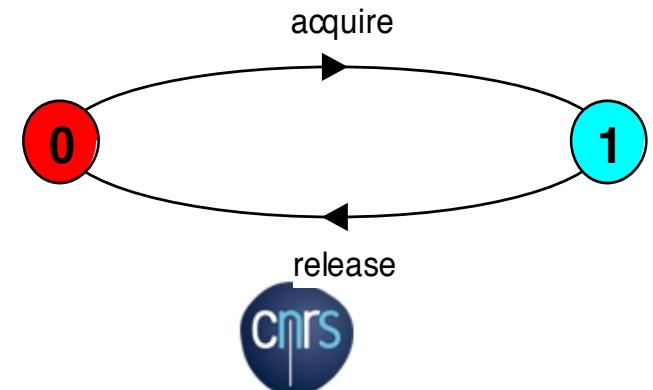
$\backslash \{\text{use}\}$ .

$\text{USER} = (\text{acquire} \rightarrow \text{use} \rightarrow \text{release} \rightarrow \text{USER})$

$\text{@}\{\text{acquire, release}\}$ .



Minimization removes hidden tau actions to produce an LTS with equivalent observable behavior.



## Step 3 : Preuve

# Safety

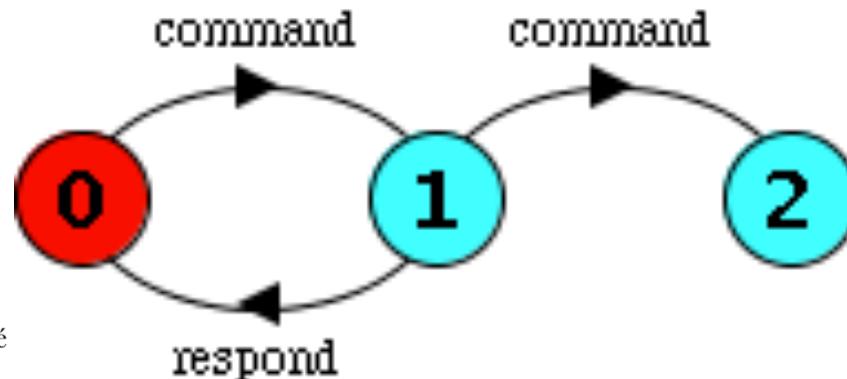
- A **safety** property asserts that nothing bad happens.
  - ◆ **STOP** or deadlocked state (no outgoing transitions)
  - ◆ **ERROR** process (-1) to detect erroneous behaviour

**ACTUATOR**

$= (\text{command} \rightarrow \text{ACTION}) ,$

**ACTION**

$= (\text{respond} \rightarrow \text{ACTUATOR} \mid \text{command} \rightarrow \text{STOP}) .$

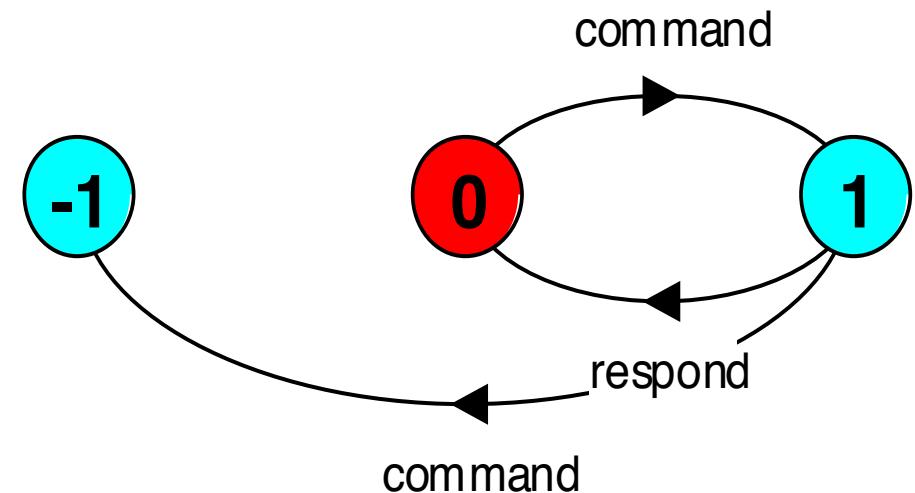


**ACTUATOR**

$= (\text{command} \rightarrow \text{ACTION}) ,$

**ACTION**

$= (\text{respond} \rightarrow \text{ACTUATOR} \mid \text{command} \rightarrow \text{ERROR}) .$



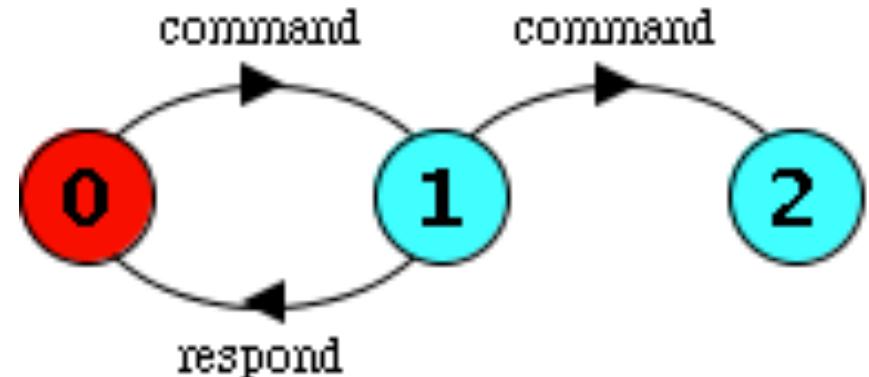
# STOP

**ACTUATOR**

= (command->ACTION) ,

**ACTION**

= (respond->ACTUATOR  
| command->**STOP**) .



- ◆ analysis using LTSA:
  - ◆ Check -> Safety
  - Trace to DEADLOCK:**  
**command**  
**command**

Give the shortest path

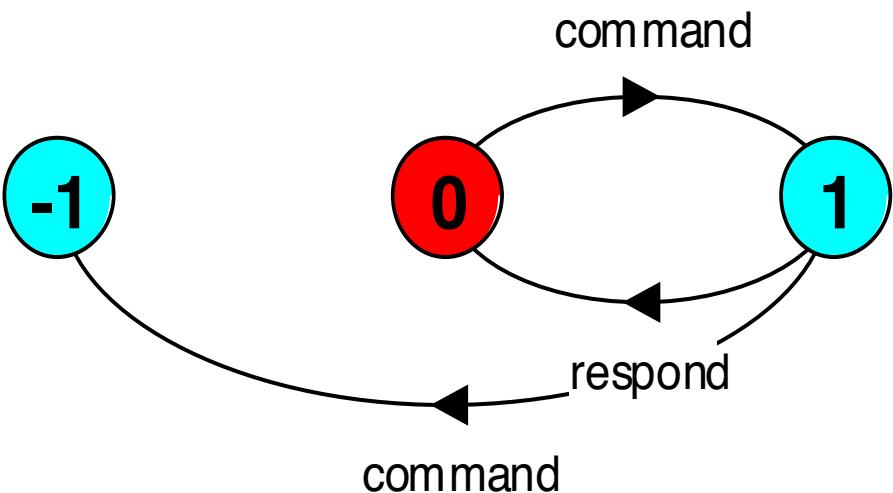
# ERROR

**ACTUATOR**

= (command->ACTION) ,

**ACTION**

= (respond->ACTUATOR  
| command->**ERROR**) .



- ◆ Analysis using LTSA:
    - ◆ Check -> Safety
- Trace to ERROR:**
- command
- command

Give the shortest path

# Comment prouver

Programme à prouver  
(programme FSP)



Propriété à vérifier

(programme FSP qui termine en ERROR pour les cas incorrects)

Programme FSP

Si existe état -1

Si existe état puit

Sinon

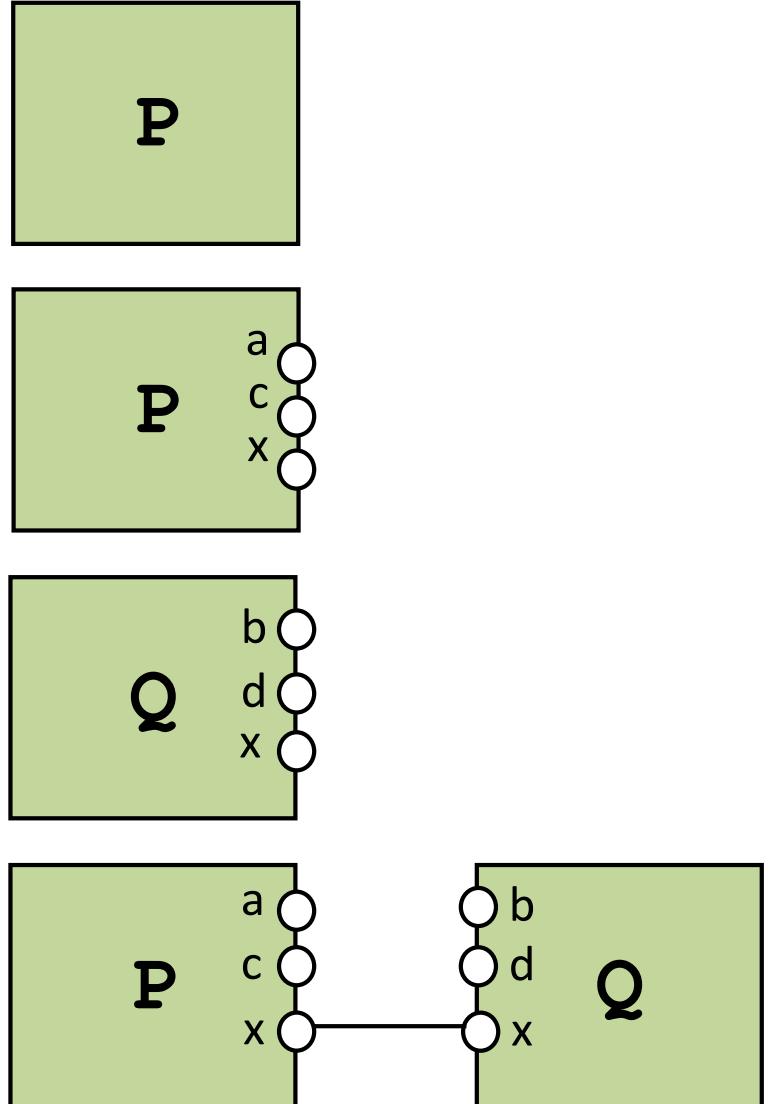
# Exemple

- Chaque action **on** est suivie de **off**
- Le processus à prouver
  - $P = (\text{on} \rightarrow Q), R = (\text{off} \rightarrow Q), Q = (\text{off} \rightarrow \text{on} \rightarrow P)$ .
- La preuve
  - PREUVE =  $(\text{on} \rightarrow \text{OFF} \mid \text{off} \rightarrow \text{PREUVE})$ ,  
 $\text{OFF} = (\text{off} \rightarrow \text{PREUVE} \mid \text{on} \rightarrow \text{ERROR})$ .
- On compose
  - $\parallel S = (P \parallel \text{PREUVE})$ .
- On demande à LTS de vérifier
  - Compilation
    - property PREUVE violation.
  - Check -> Safety
    - Trace to property violation in PREUVE:
      - on
      - off
      - on
      - on

# Schéma d'architecture (ADL : architectural description language)

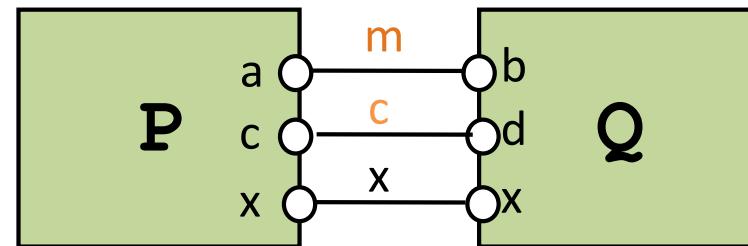
# Structure diagrams

- Process P
- Process P with alphabet  $\{a, c, x\}$
- Process Q with alphabet  $\{b, d, x\}$
- Parallel composition  $P \parallel Q$

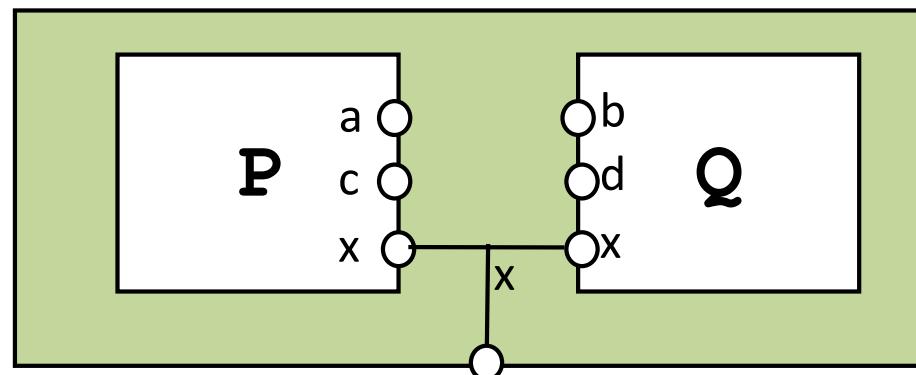


# Structure diagrams

- Parallel Composition  $(P \parallel Q) / \{m/a, m/b, c/d\}$



- Composite process  $||S = (P \parallel Q) @ \{x\}$

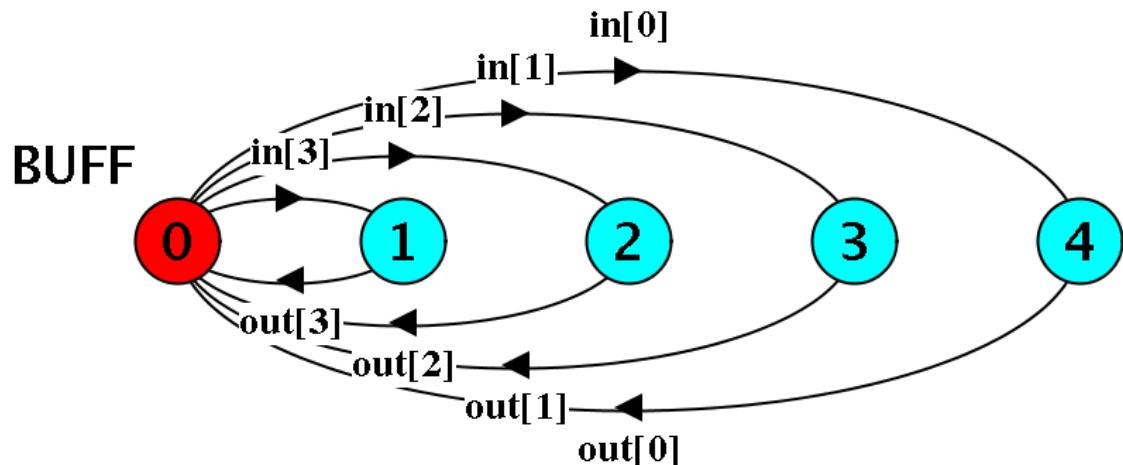
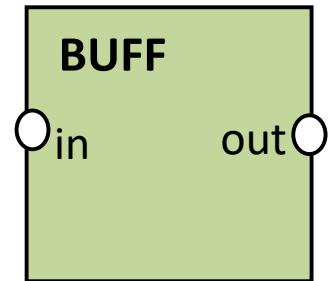


# structure diagrams

We use structure diagrams to capture the structure of a model expressed by the static combinators:  
*parallel composition, relabeling* and *hiding*.

```
range T = 0..3
```

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

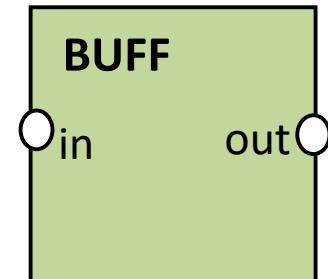


# structure diagrams

We use structure diagrams to capture the structure of a model expressed by the static combinators:  
*parallel composition, relabeling* and *hiding*.

**range T = 0..3**

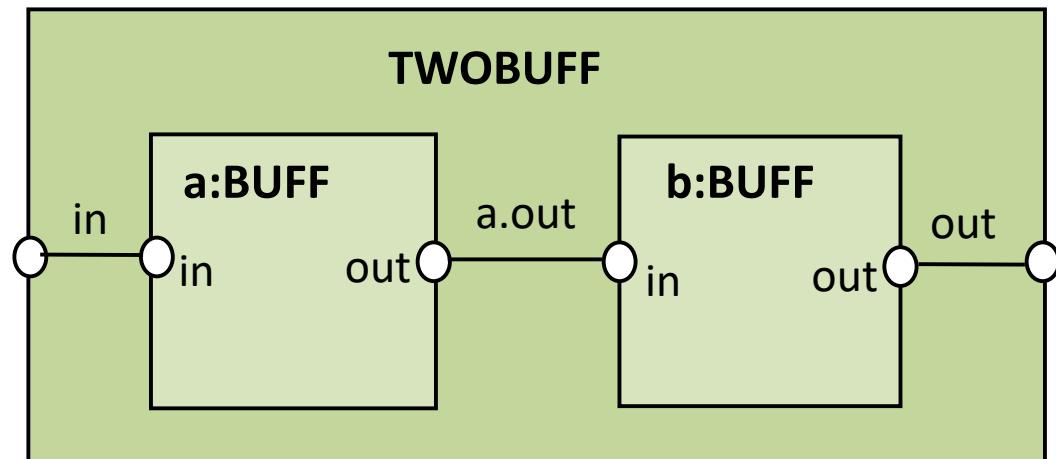
**BUFF = (in[i:T] -> out[i] -> BUFF) .**



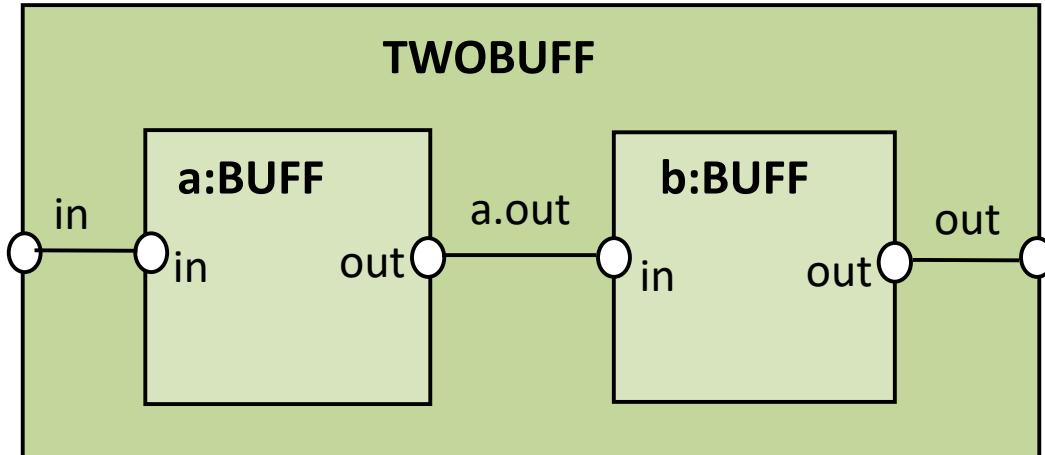
**|| TWOBUF = ?**

**Witch || TWOBUF ??**

**This one →**



# structure diagrams



**range**  $T = 0..3$

**BUFF** = (**in**[ $i:T$ ] -> **out**[ $i$ ] -> **BUFF**) .

**| | TWO\_BUFF** = (**a:BUFF** || **b:BUFF**)  
/ {**a.out**[ $i:T$ ] / **b.in**[ $i$ ],  
**in**[ $i:T$ ] / **a.in**[ $i$ ],  
**out**[ $i:T$ ] / **b.out**[ $i$ ] }  
@{**in**[ $T$ ] , **out**[ $T$ ] } .

Etes vous  
d'accord ?

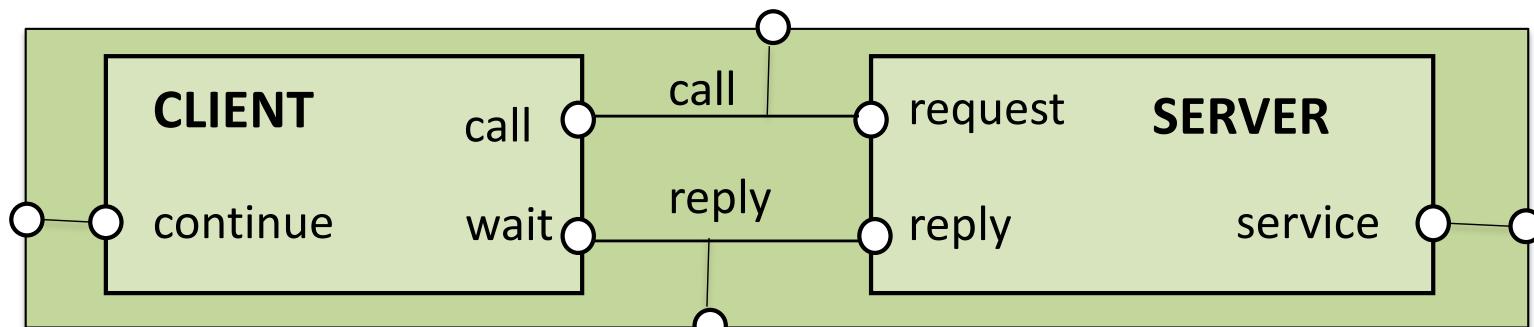


# Test: Structure diagram for CLIENT\_SERVER ?

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

If you want to hide the ports **call** and **reply** ?

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

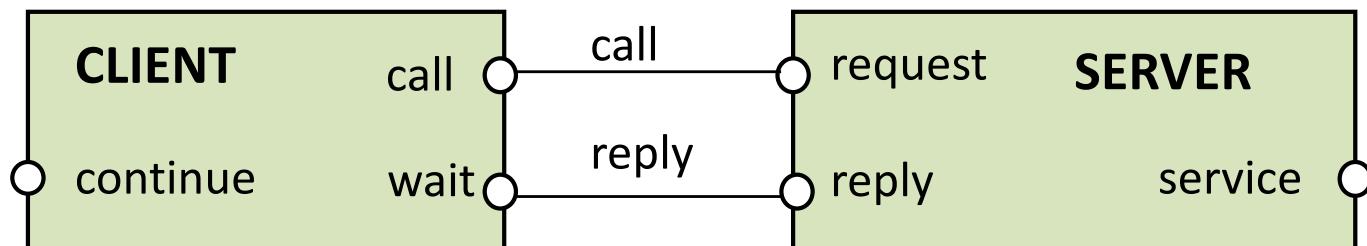


# Test: Structure diagram for CLIENT\_SERVER ?

*Sauf besoin spécifique (réutilisation) et pour des raisons de "simplification", nous représenterons :*

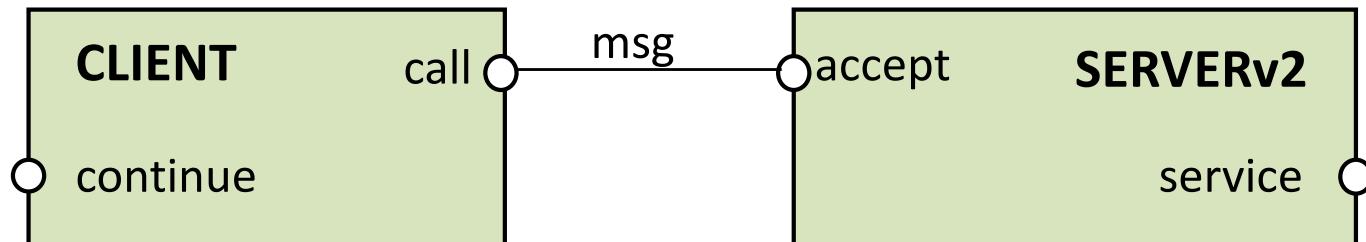
```
CLIENT = (call->wait->continue->CLIENT) .  
SERVER = (request->service->reply->SERVER) .  
||CLIENT_SERVER = (CLIENT || SERVER)  
/{call/request, reply/wait} .
```

Par :



# Test: Structure diagram for the second CLIENT\_SERVER ?

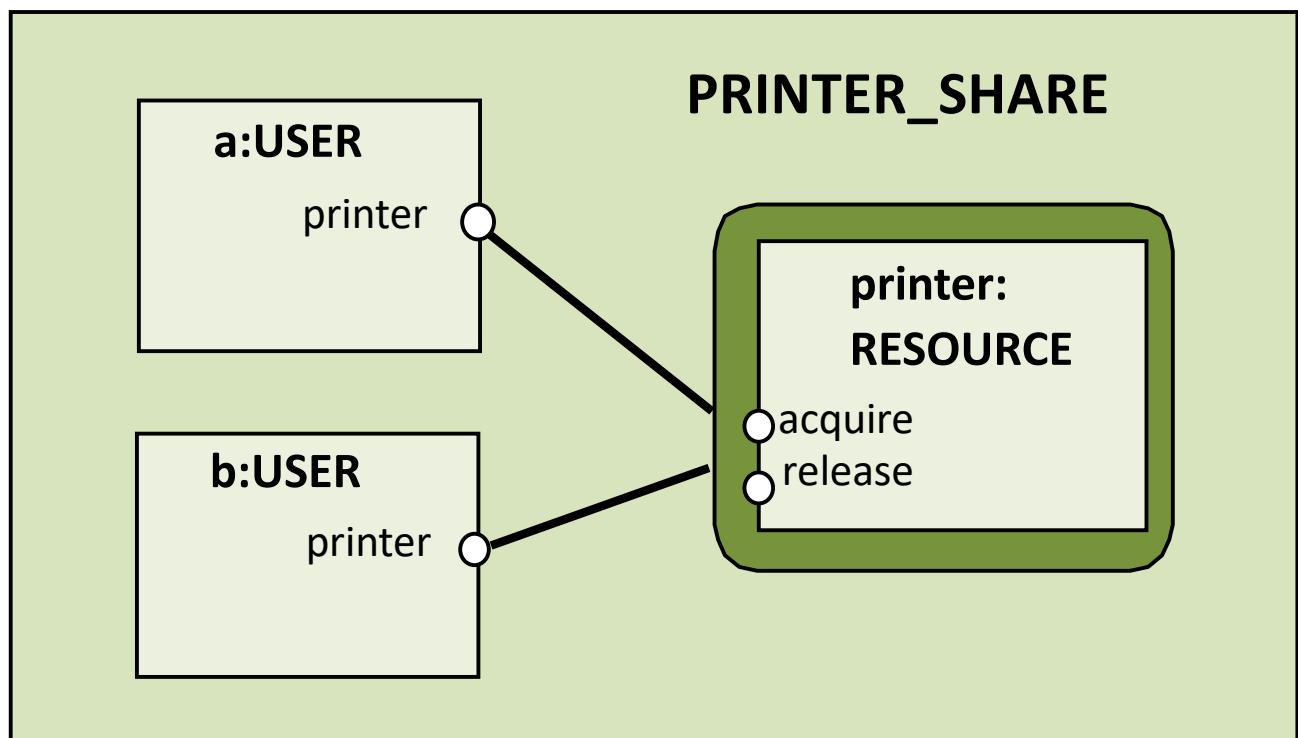
```
SERVERv2 = (accept.request  
            ->service->accept.reply->SERVERv2) .  
  
CLIENT = (call.request  
           ->call.reply->continue->CLIENT) .  
  
||CLIENT_SERVERv2 = (CLIENT || SERVERv2)  
                     /{msg/accept, msg/call} .
```



# Test: Structure diagram for RESSOURCE\_SHARE ?

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

```
| | PRINTER_SHARE  
= ({a,b}:USER| | {a,b}::printer:RESOURCE) .
```

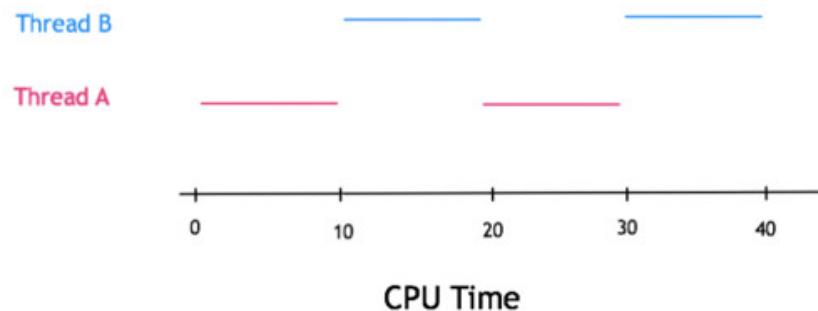


Rappel de tout ce que l'on vient de voir  
(mes fiches de révisions)

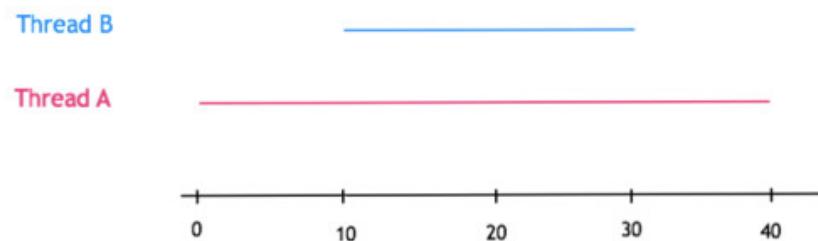
# Concurrency and parallelism

- Concurrency is not (only) parallelism
- **Interleaved concurrency**
  - Logically simultaneous processing
  - Interleaved execution on a single processor
- **Parallelism**
  - Physically simultaneous processing
  - Require a multiprocessors, a multicore system or a distributed system
- **New processor → multi-core**
  - Concurrency and parallelism

Concurrency



Parallelism



# Synchronization

- All the interleavings of the threads are NOT acceptable correct programs
- All **languages/systems** provide synchronization mechanism to restrict the interleavings → Java or Linux
- Synchronization serves two purposes:
  - Ensure safety for shared updates
    - Avoid race conditions
  - Coordinate actions of threads
    - Parallel computation
    - Event notification

# Safety

- Multiple threads access shared resources simultaneously
- Safe only if:
  - All accesses have no effect on resource,
    - e.g. reading a variable
  - or
  - All accesses idempotent
    - e.g.  $y = \sinus(a)$  or  $a = 125$
  - or
  - Only one access at a time
    - **mutual exclusion**
- **SAFETY/sûreté : propriété la plus importante**
  - **Quelque chose de mauvais ne peut jamais arriver**

# A garder en mémoire... pour toute la durée du semestre et si possible après

- The main challenge in designing concurrent programs is ensuring the **correct sequencing** of the interactions or communications between different computational executions, and **coordinating** access to resources that are shared among executions.
- Potential problems include :
  - Race conditions
  - Deadlocks
  - Resource starvation

Les définitions wikipedia des **mots en rouge** sont excellentes.

Vous pouvez aussi regarder sur wikipedia les définitions de :

- [Concurrent\\_computing](#) : présente globalement le problème avec
- [Concurrency\\_control](#) : présente des moyens pour résoudre les problèmes mais principalement dans le cadre des bases de données

# Races

- Race conditions - insidious bugs
  - Non-deterministic, timing dependent
  - Cause data corruption, crashes
  - Difficult to detect, reproduce, eliminate
- Many programs contain races
  - Inadvertent programming errors
  - Failure to observe locking discipline

# Data races

- Problem with data races: non-determinism
  - Depends on interleaving of threads
- Usual question
  - ◆ *Is the system safe?*
  - ◆ *Would testing be sufficient to discover all errors*
- In “sequential programming”
  - Safe programming is easy, we use
    - Pre and post condition
    - Invariant
- With concurrent programming
  - All interleaving execution could be safe
  - we need a new approach to explore all the solution
  - We **need a model** in order to evaluate all possible execution

# Models

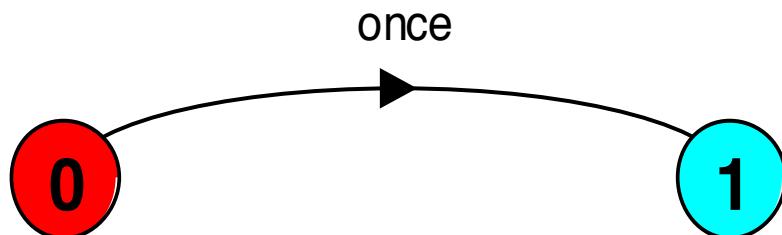
- A model is a simplified representation of the real world.
- Engineers use models to gain confidence in the adequacy and validity of a proposed design.
  - ◆ focus on an aspect of interest - concurrency
  - ◆ model animation to visualise a behaviour
  - ◆ mechanical verification of properties (safety & progress)
- Models are described using state machines, known as Labelled Transition Systems **LTS**. These are described textually as finite state processes (**FSP**) and displayed and analysed by **LTSA** (Labelled Transition Systems Analysis tool).

# Our modelling tool

- Based on model-checking or temporal logic results
- 2 parts
  - finite state processes (FSP) - algebraic form
    - to model processes as sequences of actions.
  - labelled transition systems (LTS / LTSA)
    - to analyse, display and animate behavior.
- FSP - algebraic form
- LTS - graphical form
- LTSA - analysing tools

# FSP - action prefix

- If  $x$  is an action and  $P$  a process then  $(x \rightarrow P)$  describes a process that initially engages in the action  $x$  and then behaves exactly as described by  $P$ .
- ONESHOT state machine (terminating process)
  - $\text{ONESHOT} = (\text{once } \rightarrow \text{STOP})$ .



## Convention:

- actions begin with lowercase letters
- PROCESSES begin with uppercase letters

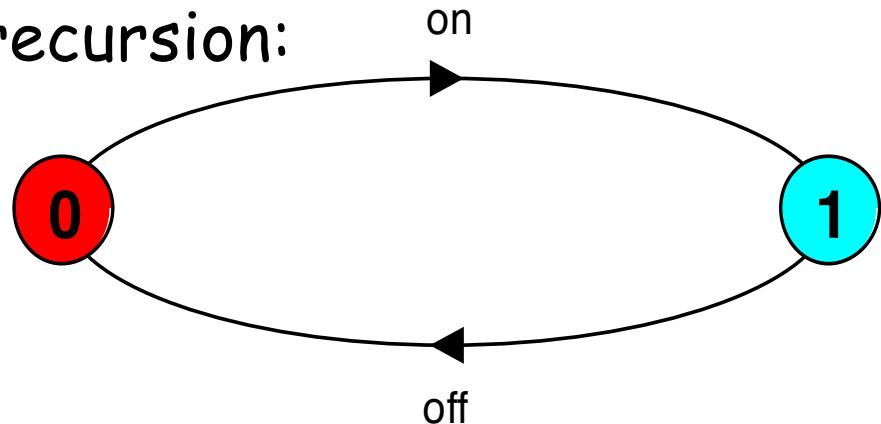
# FSP - action prefix & recursion (*infinite traces*)

- Repetitive behaviour uses recursion:

SWITCH = OFF ,

OFF = (on -> ON) ,

ON = (off-> OFF) .



- Substituting to get a more succinct definition:

SWITCH = OFF ,

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

- And again:

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

# FSP - choice

- If  $x$  and  $y$  are actions then  $(x \rightarrow P \mid y \rightarrow Q)$  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  $y$ .
- *Who or what makes the choice?*
- *Is there a difference between input and output actions?*

# Non-deterministic choice

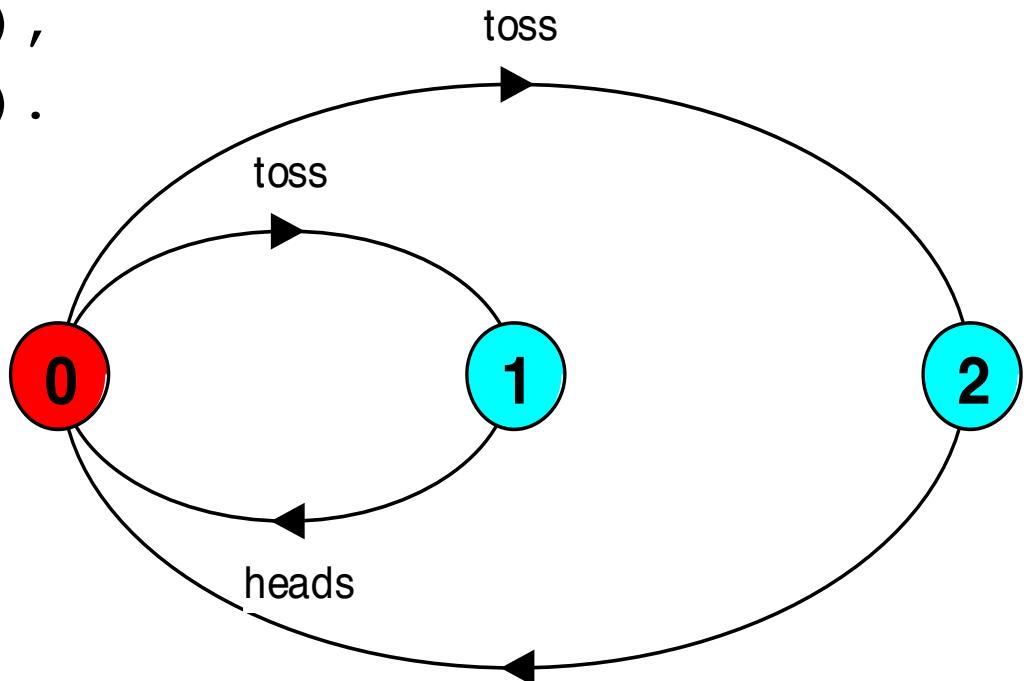
- Process  $(x \rightarrow P \mid x \rightarrow Q)$  describes a process which engages in  $x$  and then behaves as either  $P$  or  $Q$ .

COIN = (toss->HEADS | toss->TAILS) ,

HEADS= (heads->COIN) ,

TAILS= (tails->COIN) .

- Tossing a coin.



- Possible traces?

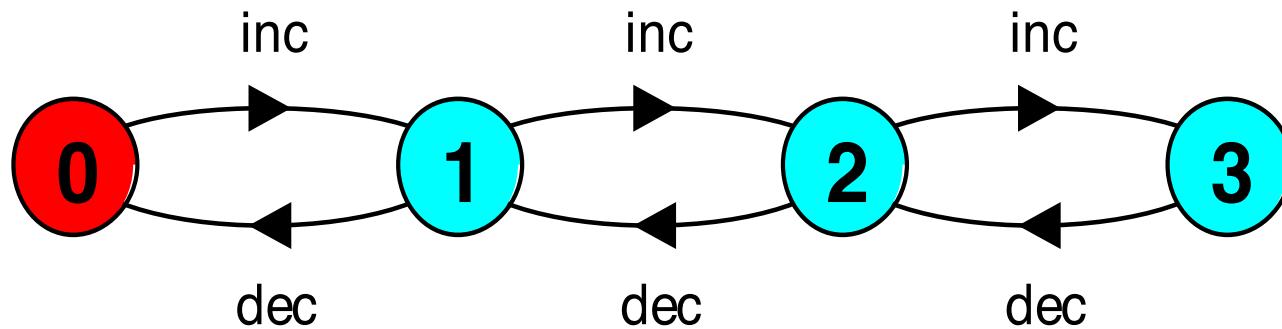
- toss->heads->toss->heads->toss->tails      tails

# FSP - guarded actions

- The choice ( $\text{when } B \ x \rightarrow P \mid y \rightarrow Q$ ) 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.

$\text{COUNT } (N=3) = \text{COUNT}[0],$

$\text{COUNT}[i:0..N] = (\text{when } (i < N) \text{ inc} \rightarrow \text{COUNT}[i+1]$   
 $\quad \mid \text{when } (i > 0) \text{ dec} \rightarrow \text{COUNT}[i-1]$   
     $).$



# parallel composition - action interleaving

- If  $P$  and  $Q$  are processes then  $(P \parallel Q)$  represents the concurrent execution of  $P$  and  $Q$ . The operator  $\parallel$  is the parallel composition operator.

ITCH = (scratch->STOP) .

CONVERSE = (think->talk->STOP) .

||CONVERSE\_ITCH = (ITCH || CONVERSE) .

Disjoint  
alphabets

- Possible traces as a result of action interleaving.

think → talk → scratch

think → scratch → talk

scratch → think → talk

# modeling interaction - shared actions

- If processes in a composition have actions in common, these actions are said to be *shared*. Shared actions are the way that process interaction is modeled. While unshared actions may be arbitrarily interleaved, a shared action must be executed at the same time by all processes that participate in the shared action.

**MAKER** = (make->**ready**->MAKER) .

**USER** = (**ready**->use->USER) .

| | **MAKER\_USER** = (MAKER || USER) .

Non-disjoint alphabets

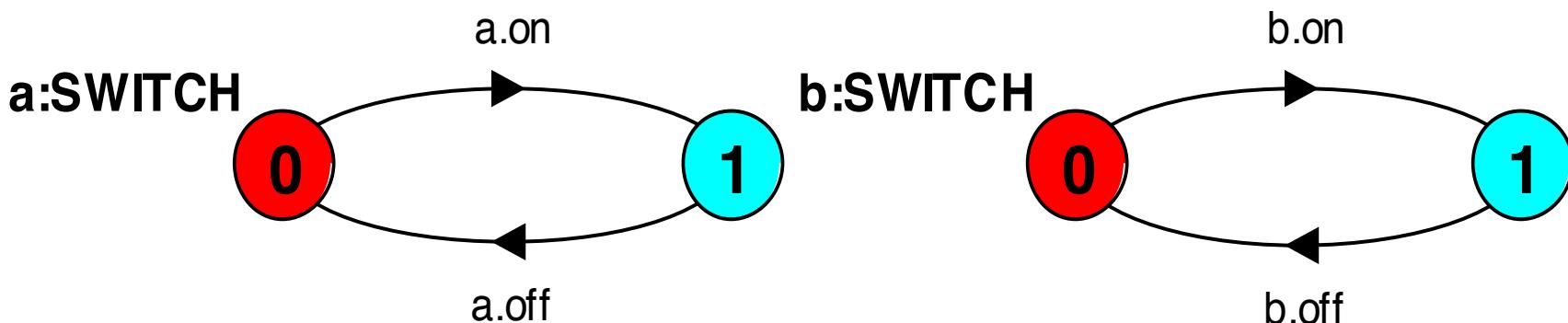
- **MAKER** synchronizes with **USER** when **ready**.
- *LTS? Traces? Number of states?*

# process instances and labeling

- $a:P$ 
  - prefixes each action label in the alphabet of  $P$  with  $a$ .
- Two **instances** of a switch process:

$\text{SWITCH} = (\text{on} \rightarrow \text{off} \rightarrow \text{SWITCH}) .$

$\parallel \text{TWO\_SWITCH} = (a:\text{SWITCH} \parallel b:\text{SWITCH}) .$



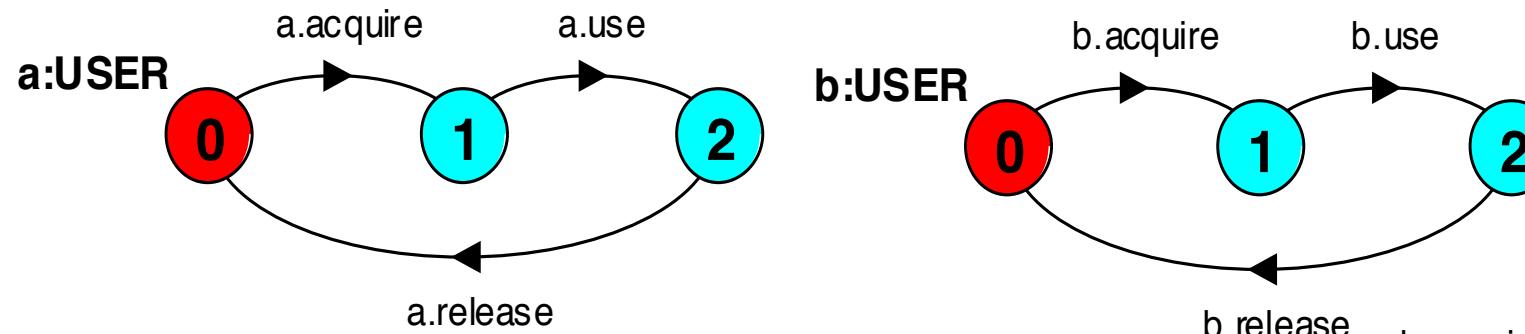
$\parallel \text{TWO\_SWITCH} = (\{a, b\}:\text{SWITCH} \parallel : \text{SWITCH}) .$

- An array of **instances** of the switch process:

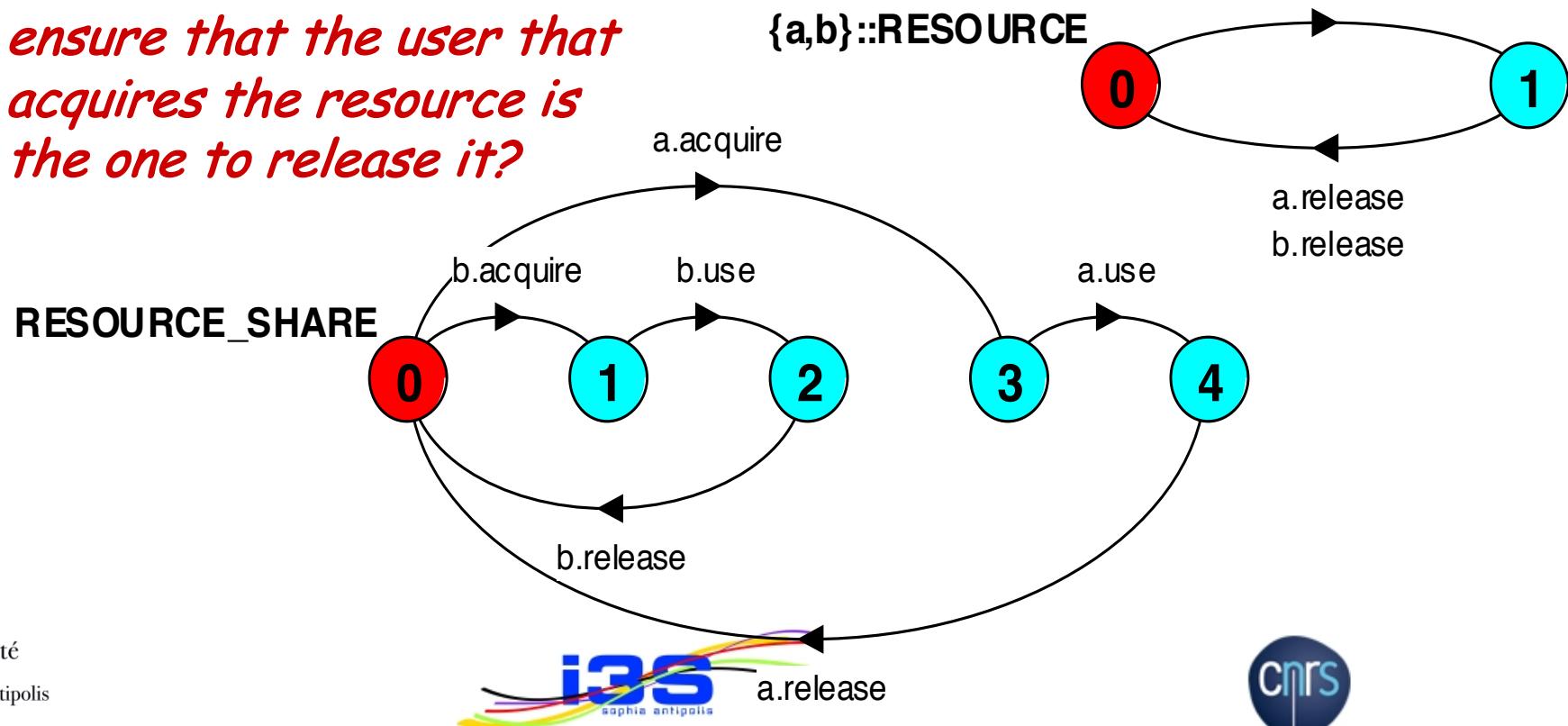
$\parallel \text{SWITCHES (N=3)} = (\text{forall}[i:1..N] s[i]:\text{SWITCH}) .$

$\parallel \text{SWITCHES (N=3)} = (s[i:1..N]:\text{SWITCH}) .$

# process prefix labels for shared resources



*How does the model ensure that the user that acquires the resource is the one to release it?*



# process labeling by a set of prefix labels

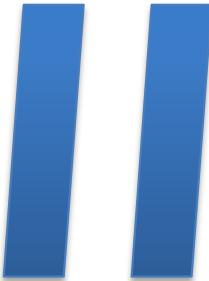
- $\{a_1, \dots, a_x\} :: P$ 
  - replaces every action label  $n$  in the alphabet of  $P$  with the labels  $a_1.n, \dots, a_x.n$ . Further, every transition  $(n \rightarrow X)$  in the definition of  $P$  is replaced with the transitions  $(\{a_1.n, \dots, a_x.n\} \rightarrow X)$ .
- Process prefixing is useful for modeling **shared** resources:  
 $\text{RESOURCE} = (\text{acquire} \rightarrow \text{release} \rightarrow \text{RESOURCE}) .$   
 $\text{USER} = (\text{acquire} \rightarrow \text{use} \rightarrow \text{release} \rightarrow \text{USER}) .$   
 $\text{|| RESOURCE\_SHARE} = (\text{a:USER} \parallel \text{b:USER}$   
 $\parallel \{a,b\} :: \text{RESOURCE}) .$

# action **hiding** - abstraction to reduce complexity

- When applied to a process P, the hiding operator  $\backslash\{a_1..a_x\}$  removes the action names  $a_1..a_x$  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.
- Sometimes it is more convenient to specify the set of labels to be **exposed**....
- When applied to a process P, the interface operator  $@\{a_1..a_x\}$  hides all actions in the alphabet of P not labeled in the set  $a_1..a_x$ .

# Comment prouver

Programme à prouver  
(programme FSP)



Propriété à vérifier

(programme FSP qui termine en ERROR pour les cas incorrects)

Programme FSP

Si existe état -1

Si existe état puit

Sinon

Tout ce que l'on vient de voir...  
doit être connu la semaine prochaine

# Q&A

<http://www.i3s.unice.fr/~riveill>

