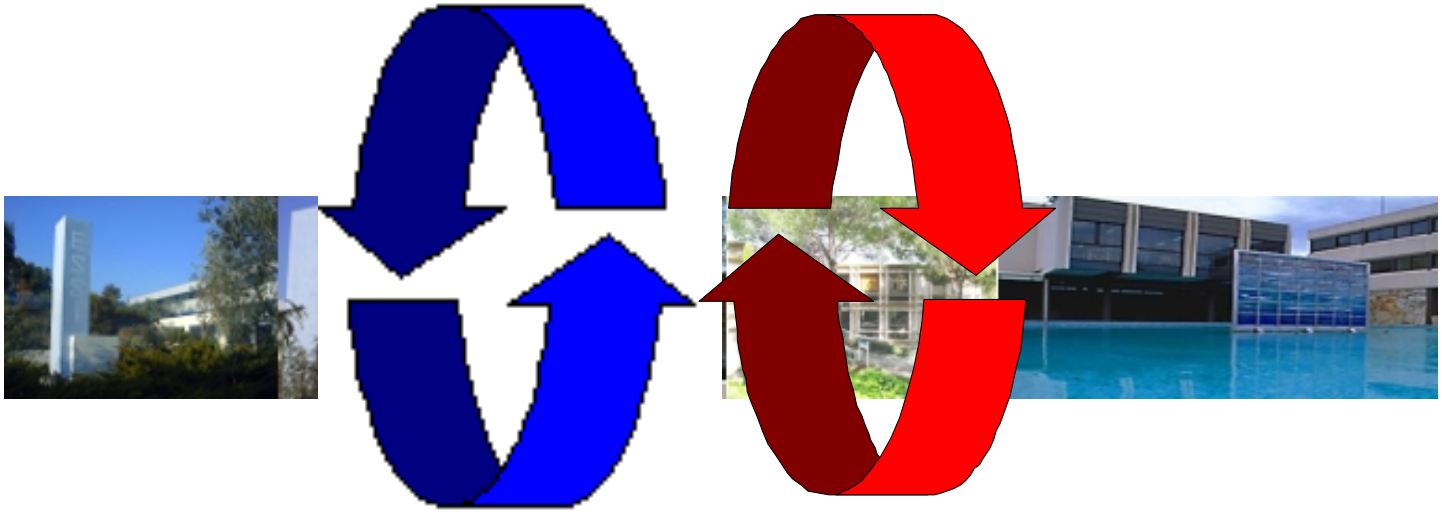
Modélisation des processus

riveill@unice.fr

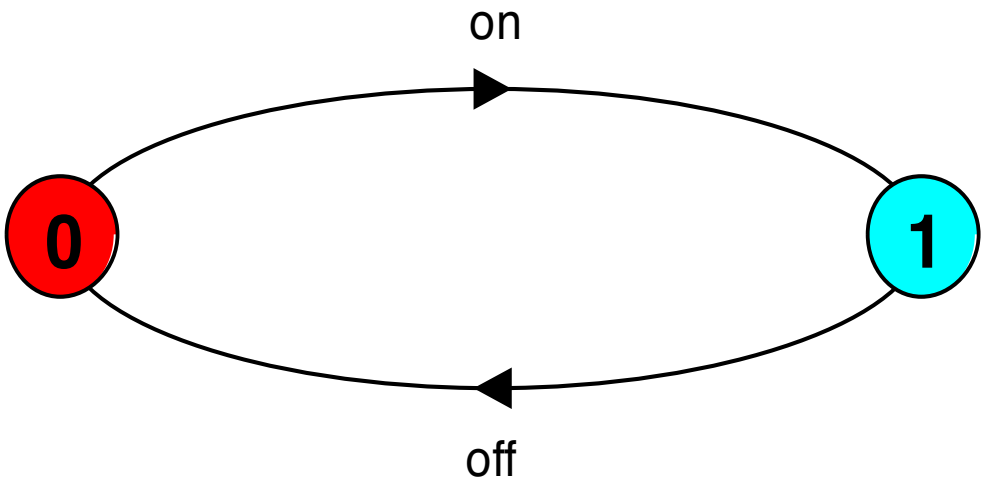
http://www.i3s.unice.fr/~riveill



Step 1 : modeling sequential processes

Modeling processes

- A process is the execution of a sequential program. It is modeled as a finite state machine which transits from state to state by executing a sequence of atomic actions.
- A light switch **LTS**

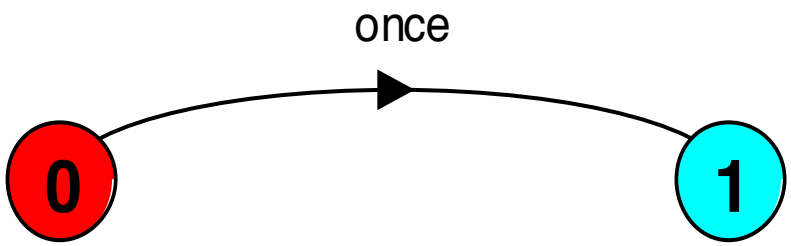


- A sequence of actions or ***trace***
  - `on→off→on→off→on→off→ .....`
- *Can finite state models produce infinite traces?*

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**.

- ONESHOT state machine (terminating process)
  - `ONESHOT = (once -> STOP).`

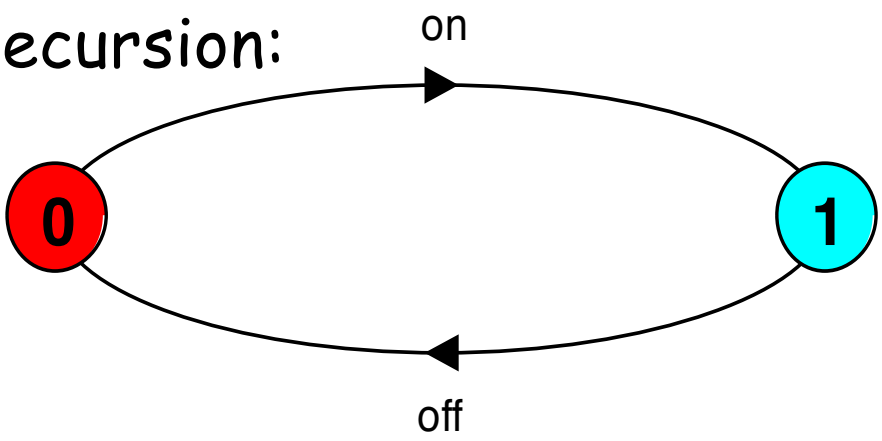


Convention:
- actions begin with lowercase letters
- PROCESSES begin with uppercase letters

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:

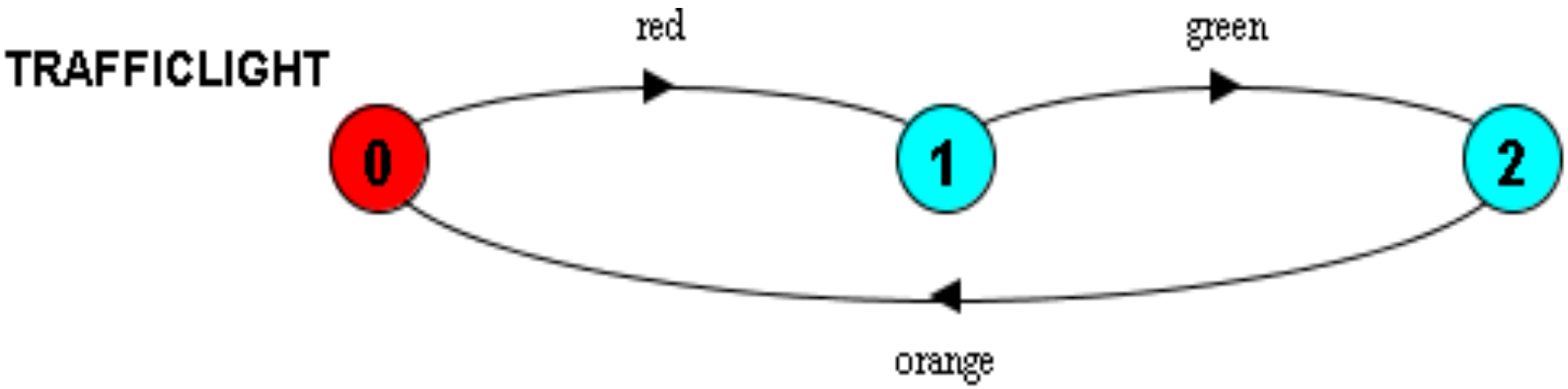
```
SWITCH = (on->off->SWITCH).
```

TEST

- Model in FSP a traffic light

```
TRAFFICLIGHT = (red->green->orange->TRAFFICLIGHT).
```

- Design in LTS a traffic light



- What is the trace ?
  - `red→green→orange→red→green` …
- What is the alphabet ?
  - `{red, green, orange}`

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**.

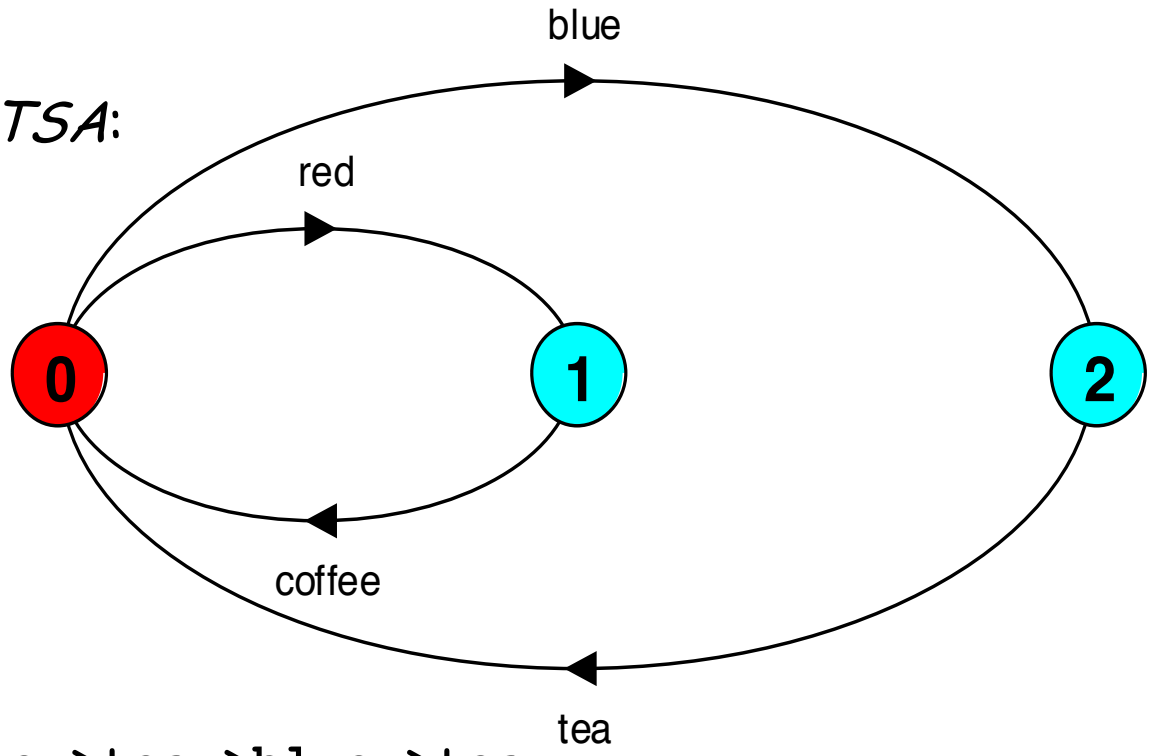
- *Who or what makes the choice?*
- *Is there a difference between input and output actions?*

FSP - choice

- FSP model of a drinks machine :

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

- LTS generated using *LTSA*:



- Possible traces?

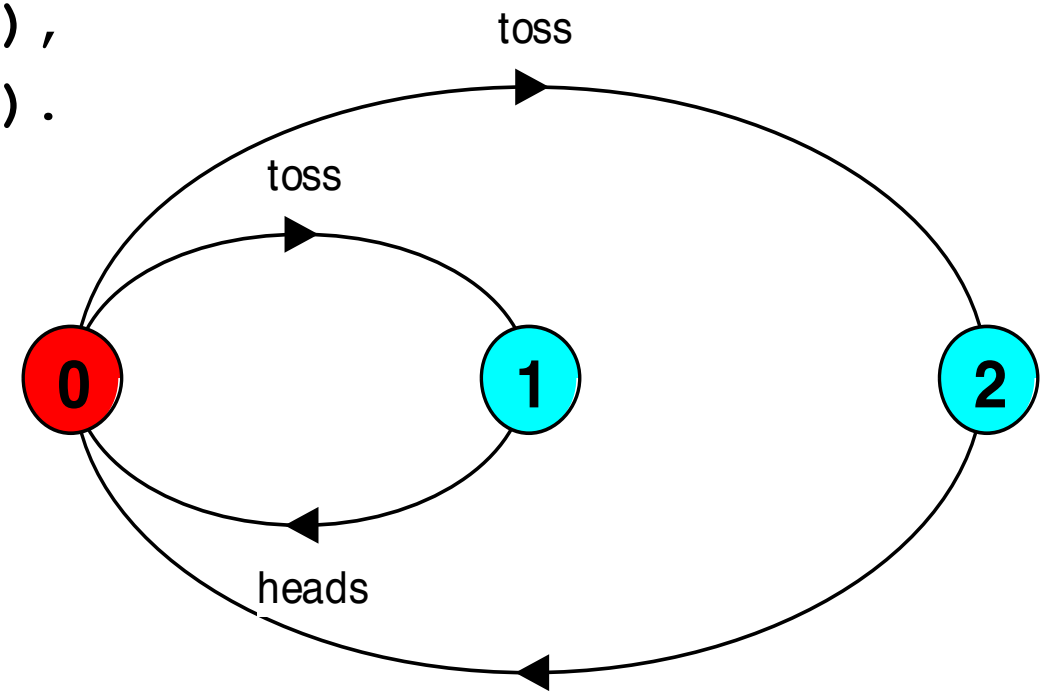
```
red->coffee->blue->tea->blue->tea->
```

Non-deterministic choice

- Process (x-> P | x -> 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.**



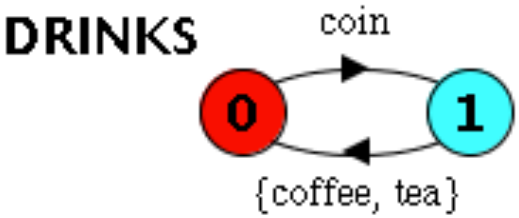
- Possible traces?
  - `toss->heads->toss->heads->toss->tails`

TEST

- Model in FSP a random drinks machine

```
DRINKS = (coin->{coffee,tea}->DRINKS).
```

- Design in LTS a drinks machine



- What is the trace ?
  - `coin->coffee->coin->coffee->coin->tea->...`
- What is the alphabet ?
  - `{coin,coffee,tea}`

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*

- or using a process parameter with default value:

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

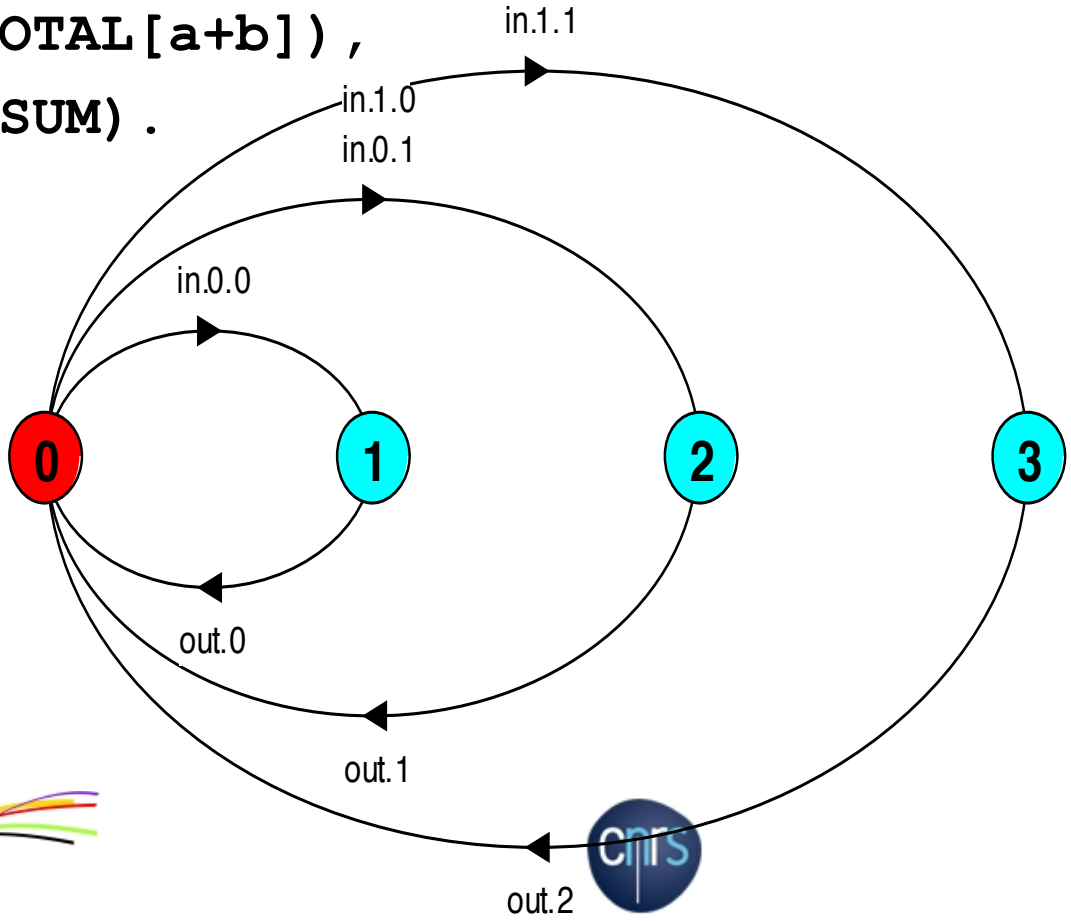
- Alphabet = {in.0, in.1, in.2, in.3,
  out.0, out.1, out.2, out.3}

FSP - indexed processes and actions

- index expressions to model calculation:

```
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).
```

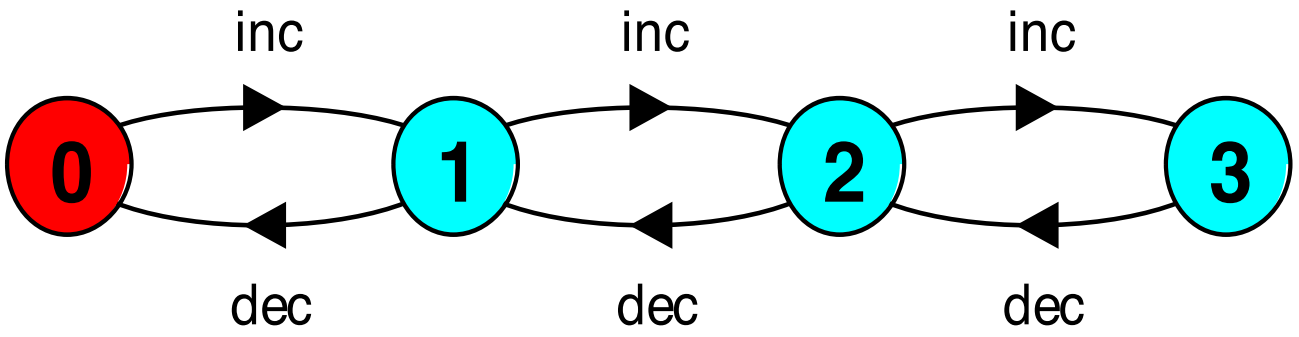
Local indexed process definitions are equivalent to process definitions for each index value



FSP - guarded actions

- The choice **(when B x -> 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.

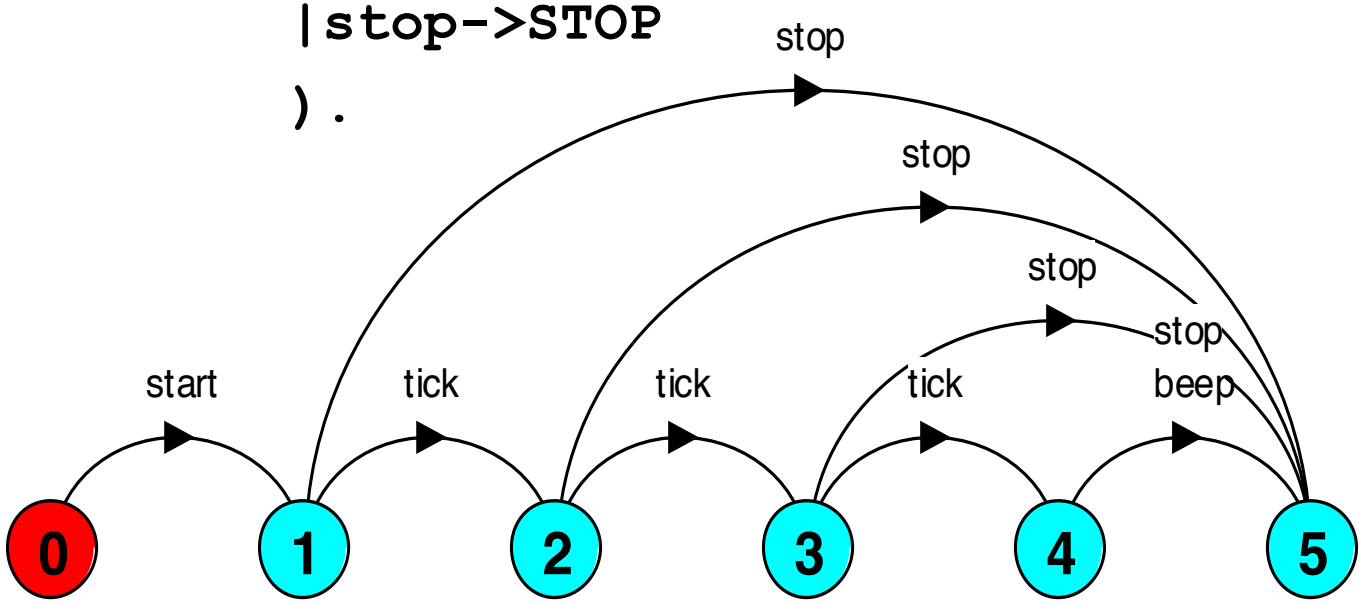
```
COUNT  (N=3)     = COUNT[0],
COUNT[i:0..N] = (when(i<N)  inc->COUNT[i+1]
                |when(i>0)  dec->COUNT[i-1]
                ).
```



FSP - guarded actions

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

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



FSP - process alphabets

- The alphabet of a process is the set of actions in which it can engage.
- Process alphabets are implicitly defined by the actions in the process definition.
- The alphabet of a process can be displayed using the LTSA alphabet window.

```
Process:
     COUNTDOWN
Alphabet:
     { beep,
       start,
       stop,
       tick
     }
```

FSP - process alphabet extension

- Alphabet extension can be used to extend the implicit alphabet of a process:

  - ***implicit***

    ```
    WRITER = (write[1]->write[3]->WRITER).
    ```

  - Alphabet of `WRITER` is the set `{write.1, write.3}`

  - ***explicit***

    ```
    WRITER = (write[1]->write[3]->WRITER)
                +{write[0..3]}.
    ```

  - Alphabet of `WRITER` is the set `{write[0..3]}`

FSP - process alphabet adjustment

- The implicit alphabet of a process can be extended and/or reduced, by two kinds of suffixes to a process description P:
  - extension `P + {…}`
  - hiding `P\{…}`
- Examples:

```
MEALS = (breakfast->lunch->dinner->MEALS)\{lunch}.
```

  - Now “lunch” becomes an internal action, *tau*, not visible nor shared.
  - You want to eat alone.

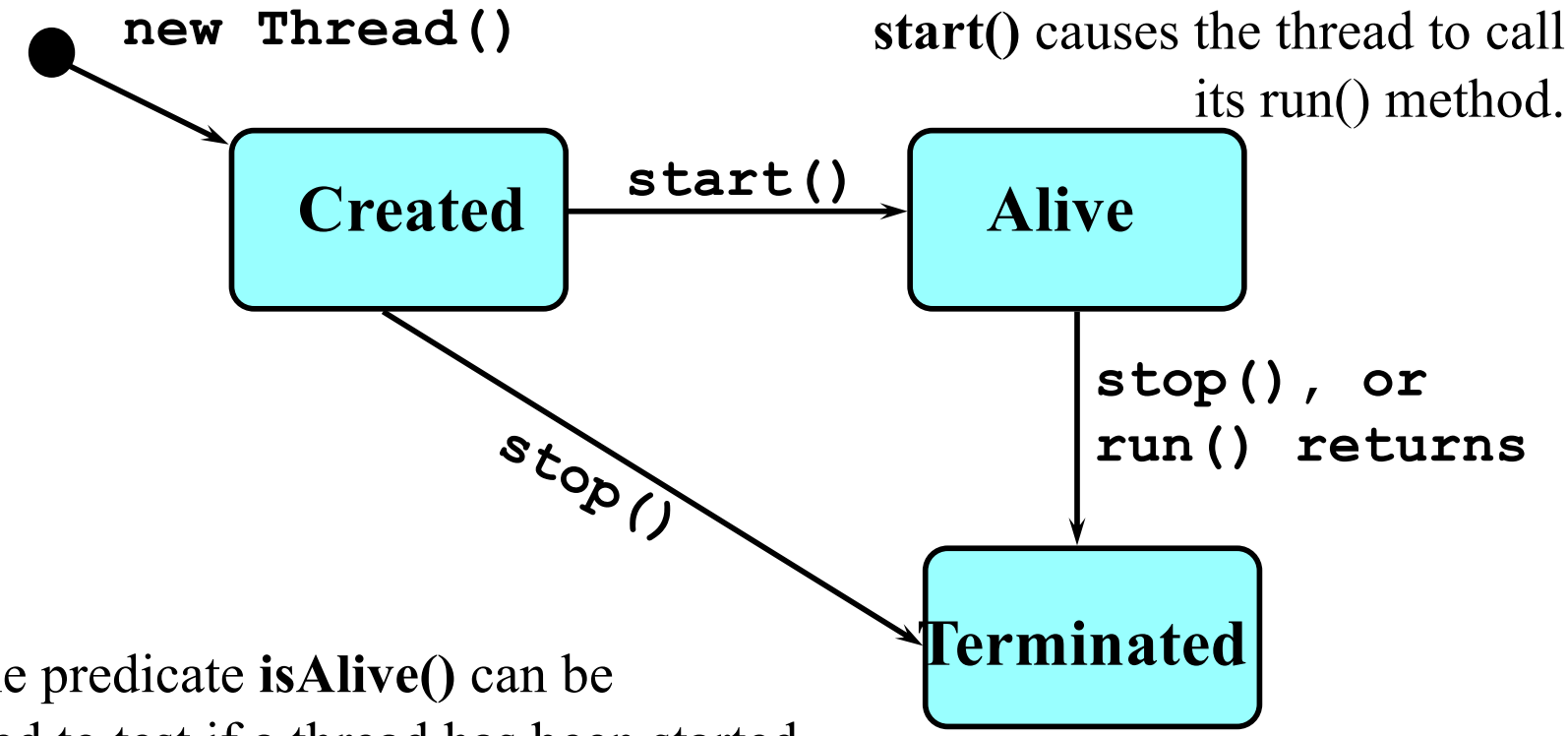
```
LISTEN = ({latin, jazz, pop}->LISTEN)+{hiphop}.
```

  - You do not want to listen to hiphop, and block on hiphop actions.

- We make use of alphabet extensions in later lectures

thread life-cycle in Java

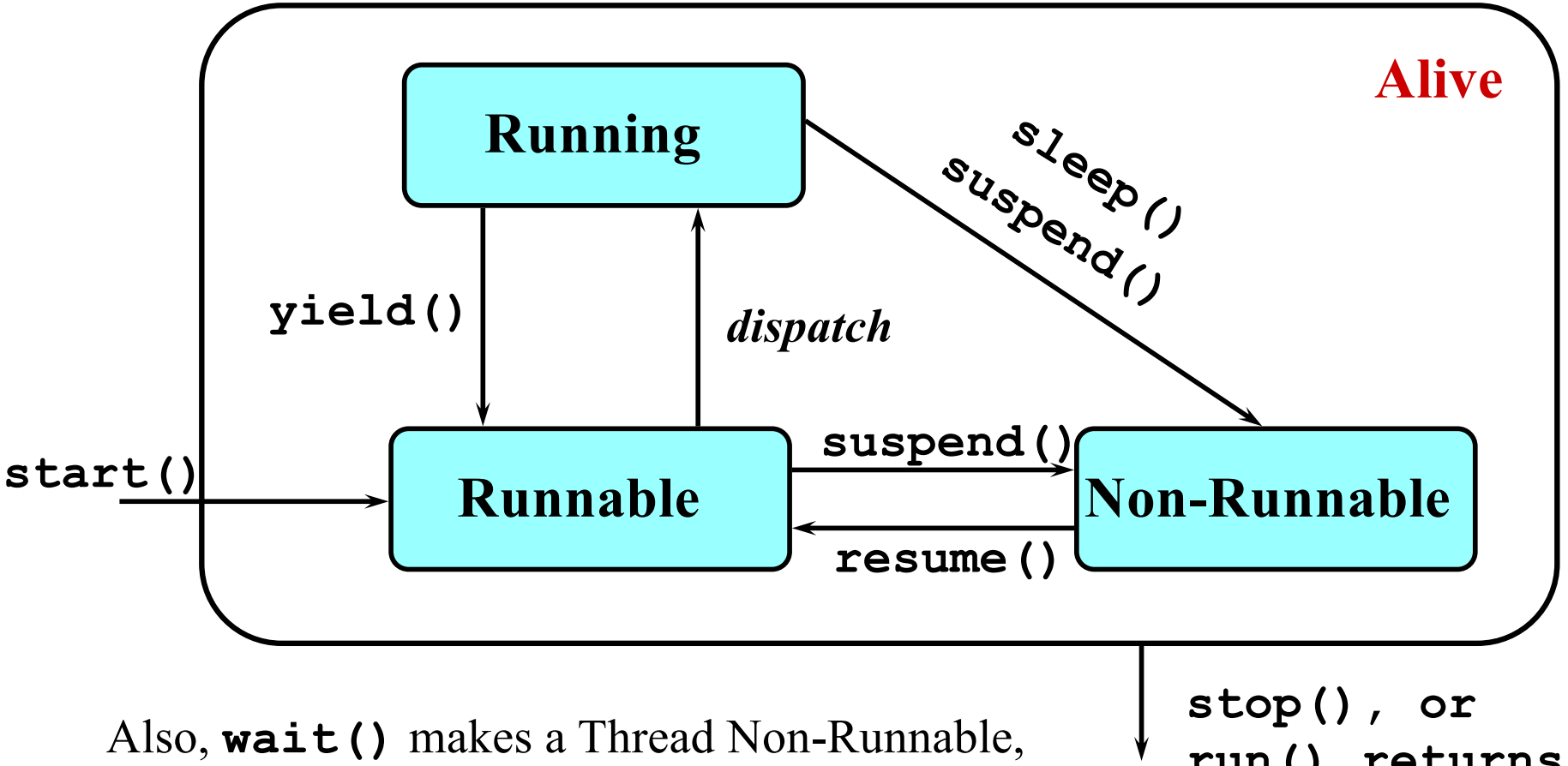
An overview of the life-cycle of a thread as state transitions:



The predicate **isAlive()** can be used to test if a thread has been started but not terminated. Once terminated, it cannot be restarted (cf. mortals).

thread alive states in Java

Once started, an **alive** thread has a number of substates :

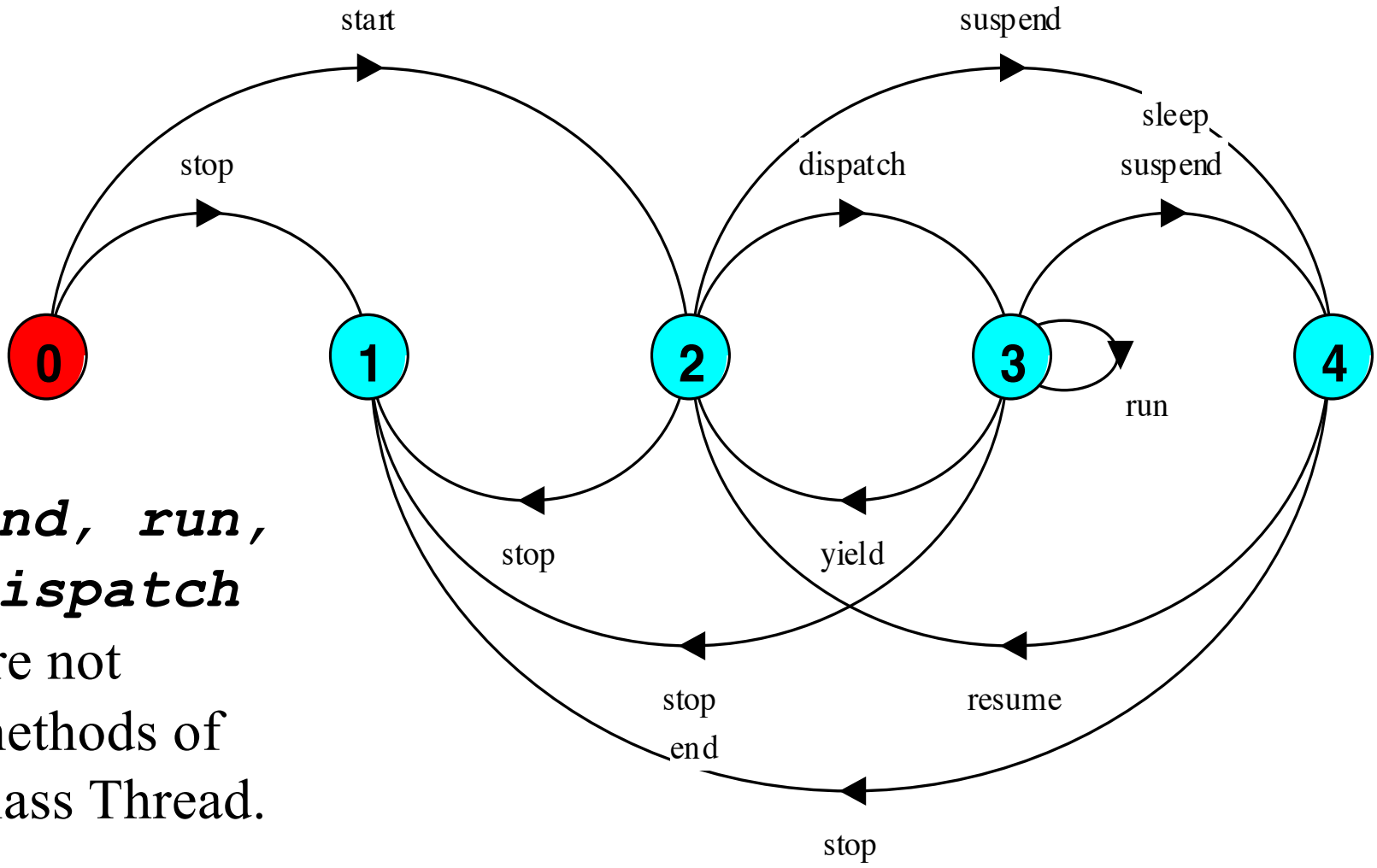


Also, **`wait()`** makes a Thread Non-Runnable,
and **`notify()`** makes it Runnable
(used in later chapters).

Java thread lifecycle - an FSP specification

```
THREAD      = CREATED,
CREATED     = (start              ->RUNNABLE
               |stop               ->TERMINATED),
RUNNING     = ({suspend,sleep}    ->NON_RUNNABLE
               |yield              ->RUNNABLE
               |{stop,end}         ->TERMINATED
               |run                ->RUNNING),
RUNNABLE    = (suspend            ->NON_RUNNABLE
               |dispatch           ->RUNNING
               |stop               ->TERMINATED),
NON_RUNNABLE = (resume            ->RUNNABLE
               |stop               ->TERMINATED),
TERMINATED  = STOP.
```

Java thread lifecycle - an FSP specification



end, run, dispatch are not methods of class Thread.

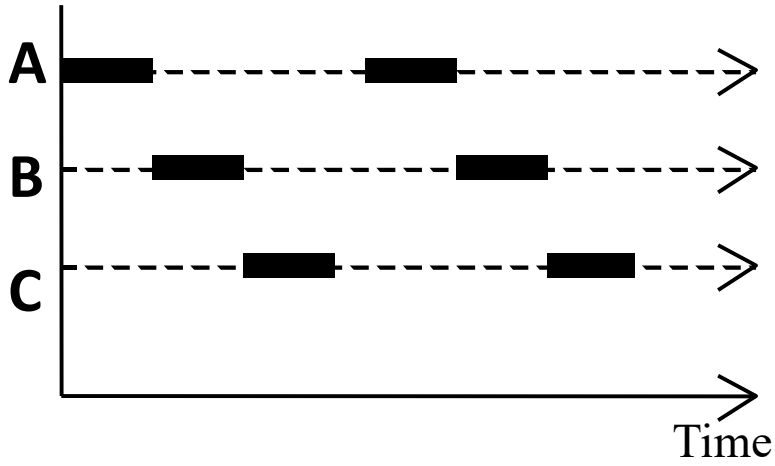
Université Nice Sophia Antipolis

States 0 to 4 correspond to **`CREATED`**, **`TERMINATED`**, **`RUNNABLE`**, **`RUNNING`**, and **`NON-RUNNABLE`** respectively.

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||Q) represents the concurrent execution of P and Q. The operator || is the parallel composition operator.

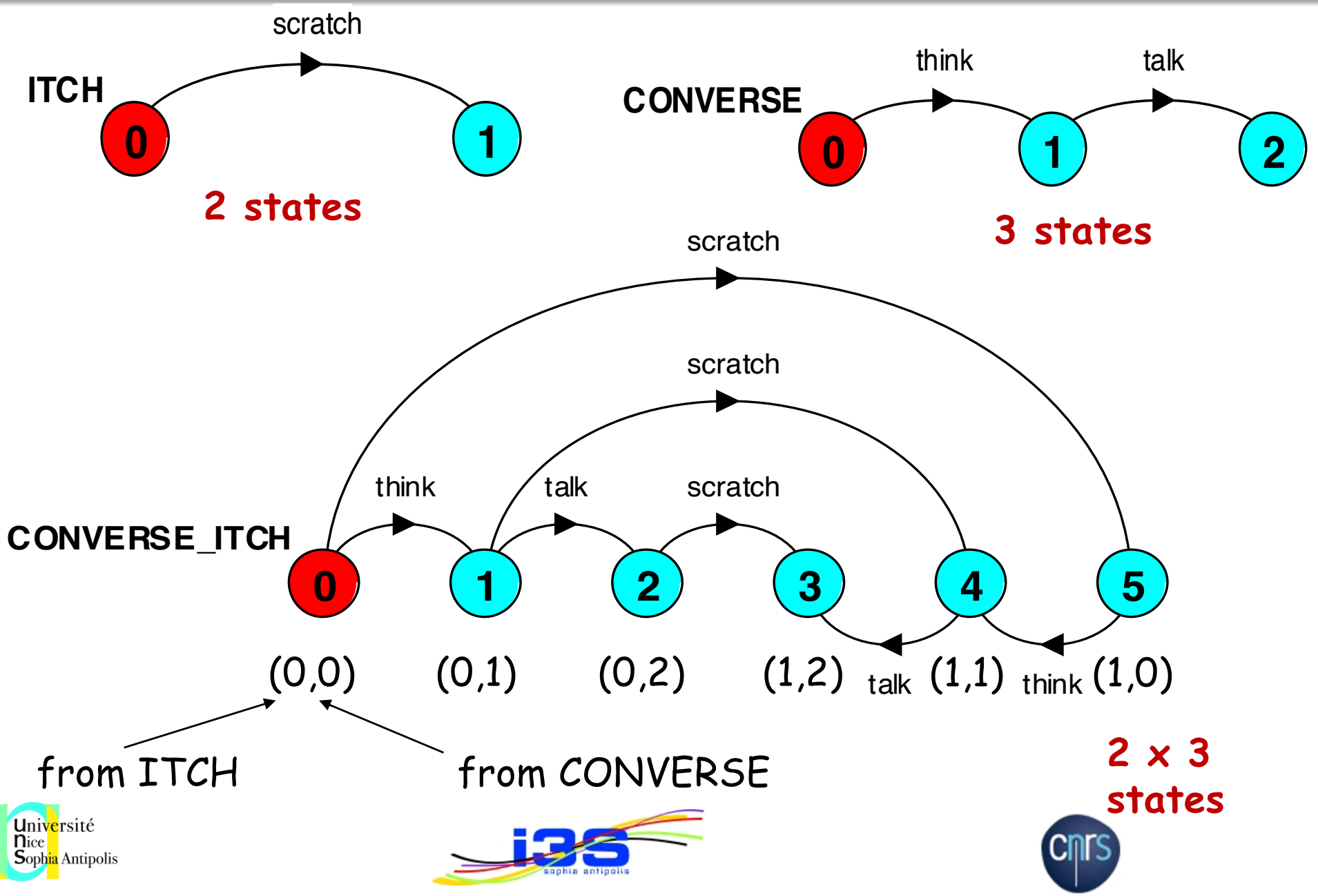
```
ITCH     = (scratch->STOP).
CONVERSE = (think->talk->STOP).
||CONVERSE_ITCH = (ITCH || CONVERSE).
```

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

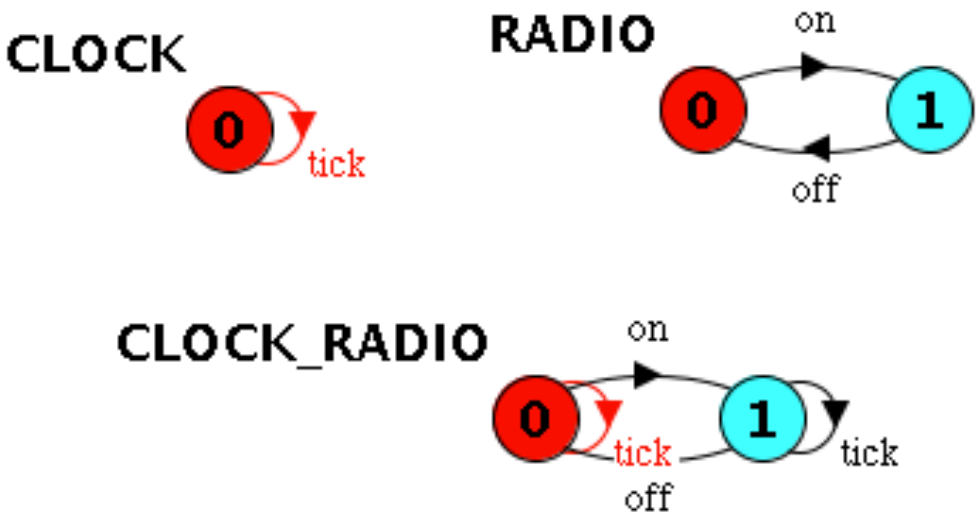
- **Idempotente:** `(P||P) = P`
- **Commutative:** `(P||Q) = (Q||P)`
- **Associative:** `(P||(Q||R)) = ((P||Q)||R)`
  `= (P||Q||R).`

- Clock radio example:
- `CLOCK = (tick->CLOCK).`
- `RADIO = (on->off->RADIO).`
- `||CLOCK_RADIO = (CLOCK || RADIO).`

- *LTS? Traces? Number of states?*

parallel composition - algebraic laws

- *LTS :*



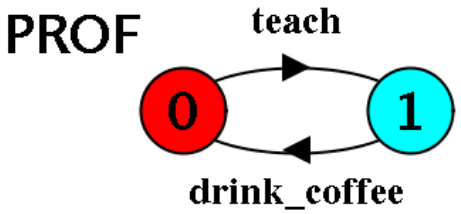
- *Traces =* `tick->tick->on->tick->off->`…

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

TEST

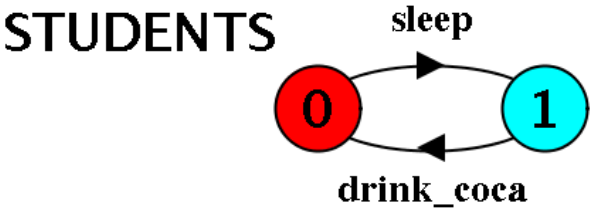
```
PROF = (teach->drink_coffee->PROF).
```

- *LTS ?*




```
STUDENTS  = (sleep->drink_coca->STUDENTS).
```

- *LTS ?*

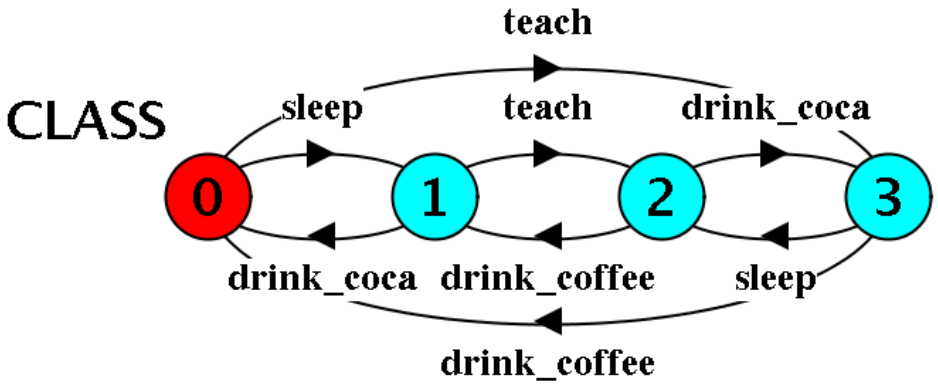



```
||CLASS = (PROF || STUDENTS).
```

- *How many state ?*

2 x 2 states

- *LTS of ||CLASS ?*



Université Nice Sophia Antipolis

i3S sophia antipolis

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`.

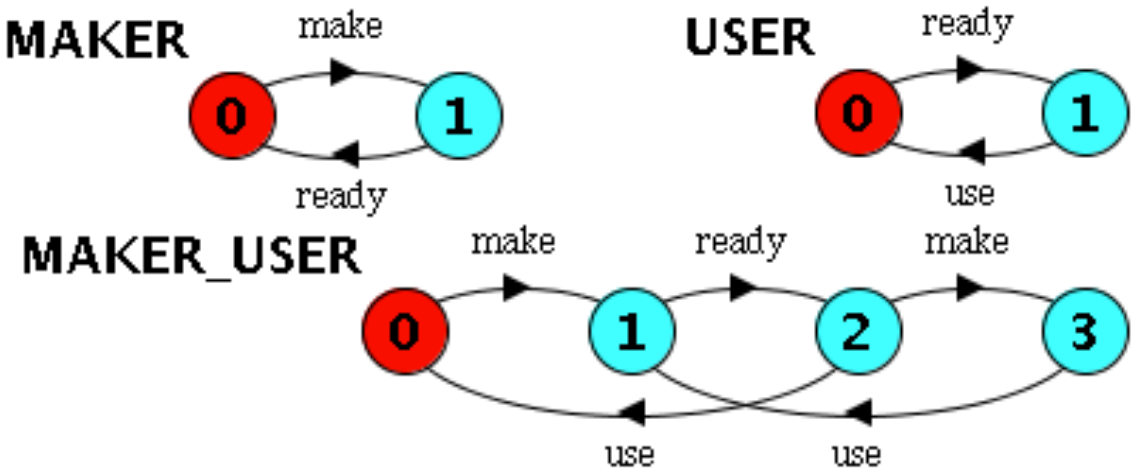
- ***LTS? Traces? Number of states?***

modeling interaction - shared actions

- *FSP =*

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

- *LTS =*

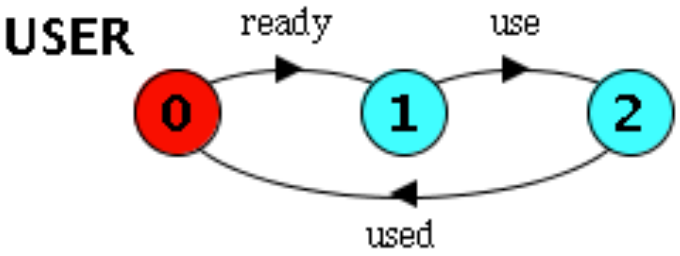


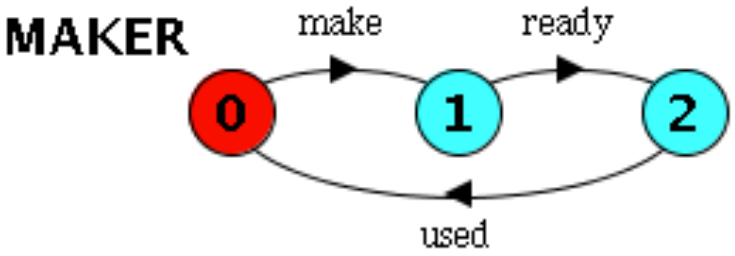
- *Traces*
  - `make->ready->use->`...
- *Number of states? 4*

TEST

- A handshake is an action acknowledged by another:

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



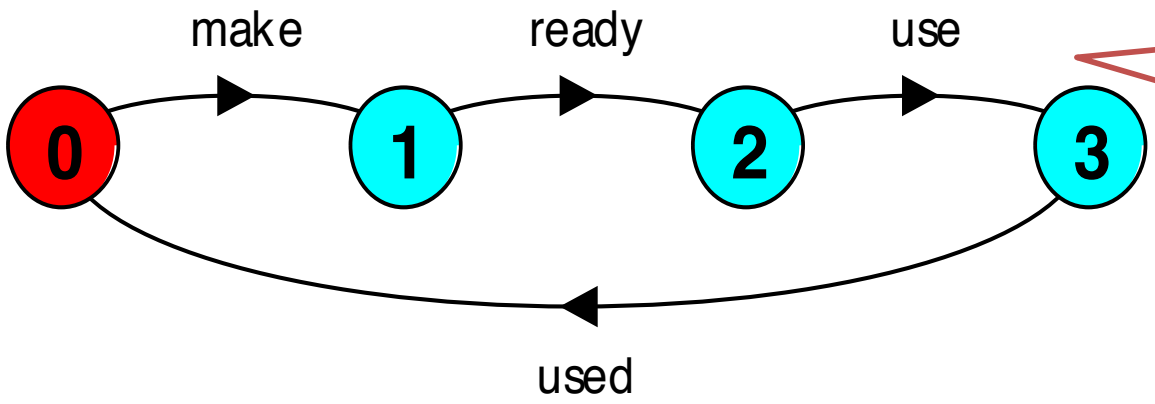


- *LTS ?*

3 x 3 states?

No : only 4 states (non disjoint alphabet)

→ ready actions occur simultaneously



Interaction constrains the overall behaviour.

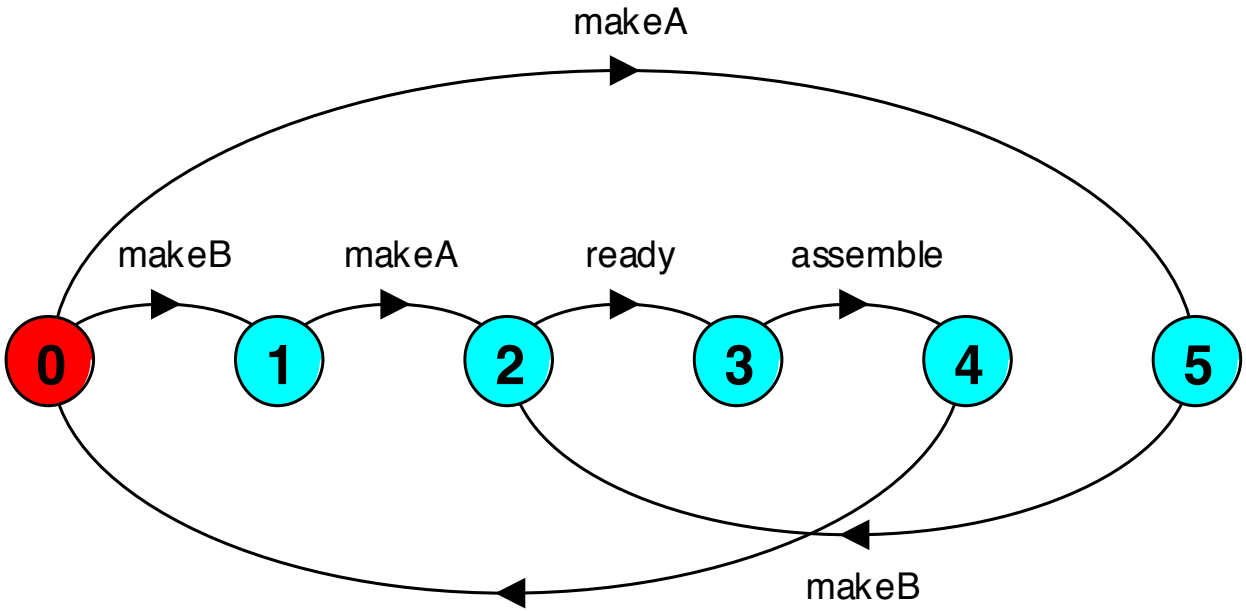
Université Nice Sophia Antipolis

cnrs

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.

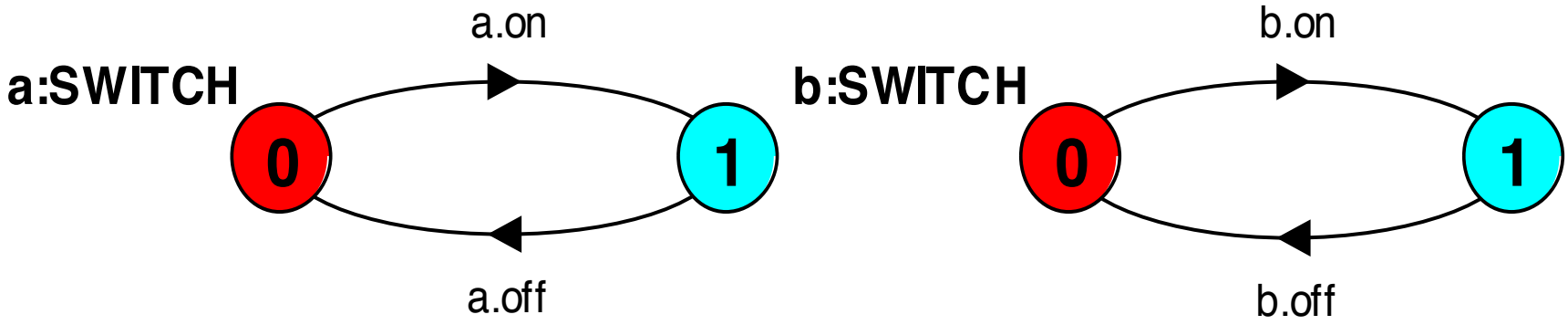
```
||FACTORY = (MAKE_A || MAKE_B || ASSEMBLE).
```

process instances and labeling

- **a:P**
  - prefixes each action label in the alphabet of P with a.

- Two **instances** of a switch process:

```
SWITCH =  (on->off->SWITCH).
||TWO_SWITCH = (a:SWITCH || b:SWITCH).
```




```
||TWO_SWITCH = ({a, b}:SWITCH || :SWITCH).
```

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

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

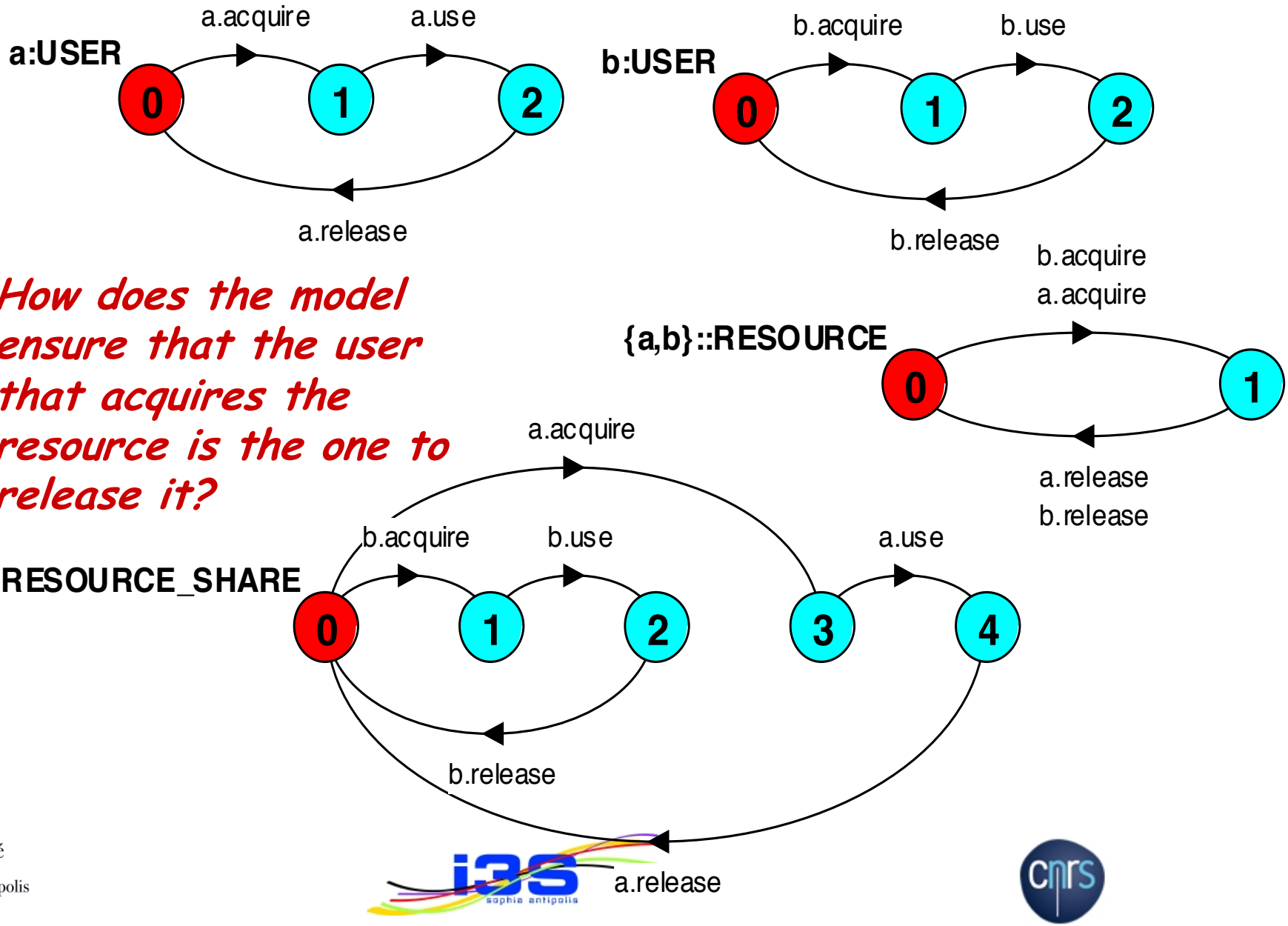
process labeling by a set of prefix labels

- **{a1,..,ax}::P**
  - replaces every action label n in the alphabet of P 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).

- Process prefixing is useful for modeling **shared** resources:

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

process prefix labels for shared resources



Université Nice Sophia Antipolis

i3S sophia antipolis

cnrs

action relabeling

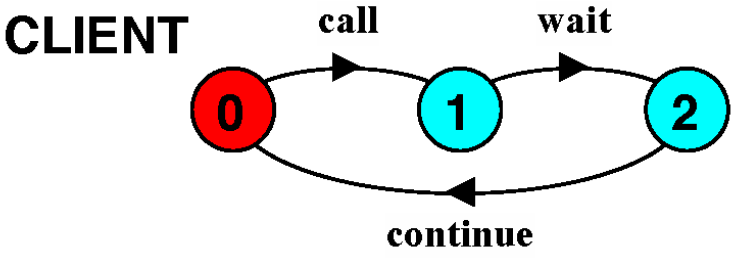
- Relabeling functions are applied to processes to change the names of action labels. The general form of the relabeling function is:
  /{newlabel_1/ oldlabel_1,… newlabel_n/ oldlabel_n}.

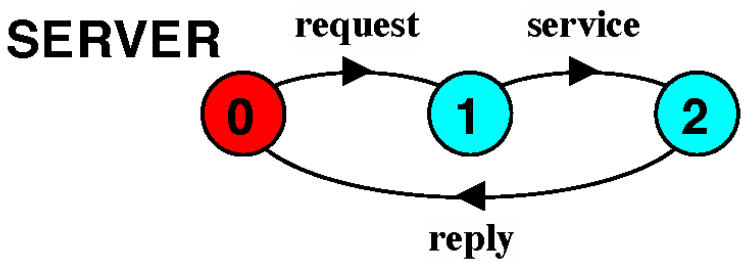
- Relabeling to ensure that composed processes synchronize on particular actions.

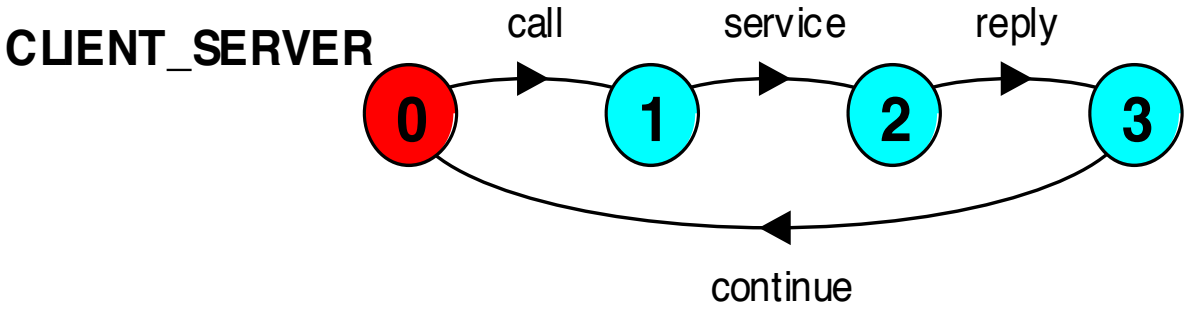
```
CLIENT = (call->wait->continue->CLIENT).

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

action relabeling

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







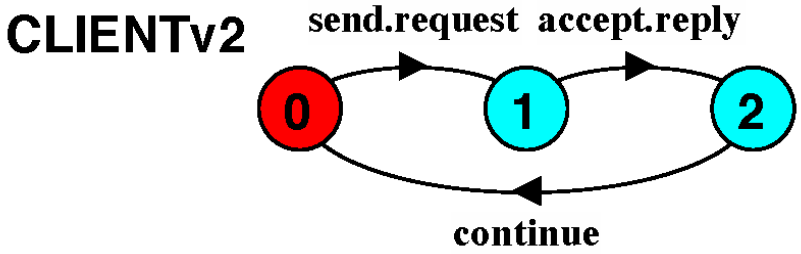
action relabeling - prefix labels

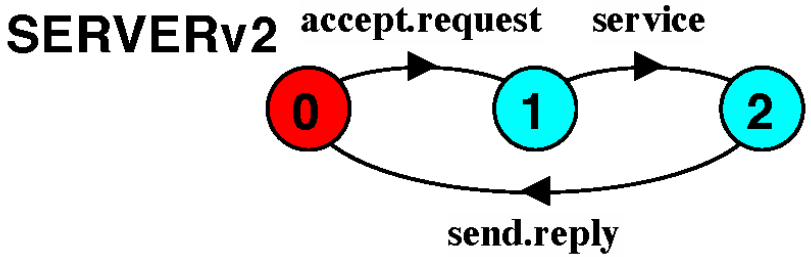
- An alternative formulation of the client server system is described below using qualified or prefixed labels:

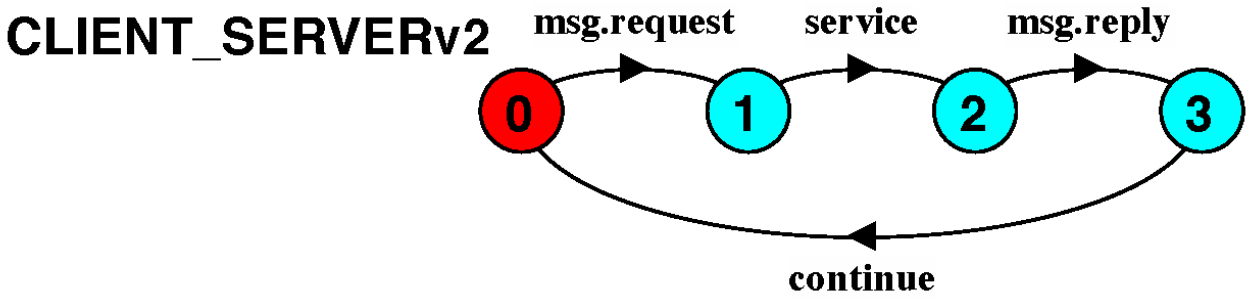
```
SERVERv2 = (accept.request
            ->service->accept.reply->SERVERv2).
CLIENT = (call.request
            ->call.reply->continue->CLIENT).
||CLIENT_SERVERv2 = (CLIENT || SERVERv2)
                    /{msg/accept, msg/call}.
```

action relabeling - prefix labels

```
SERVERv2 = (accept.request
            ->service->send.reply->SERVERv2).
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 \{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.

- Sometimes it is more convenient to specify the set of labels to be exposed....

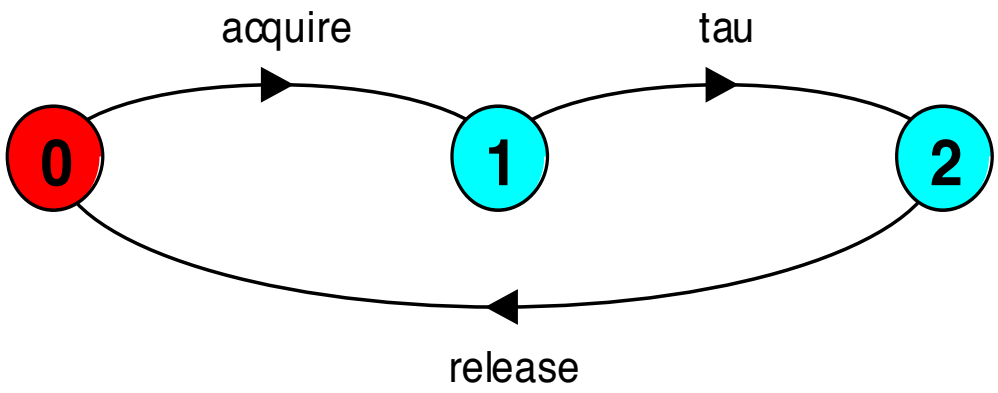
- When applied to a process P, the interface operator @{a1..ax} hides all actions in the alphabet of P not labeled in the set a1..ax.

action hiding

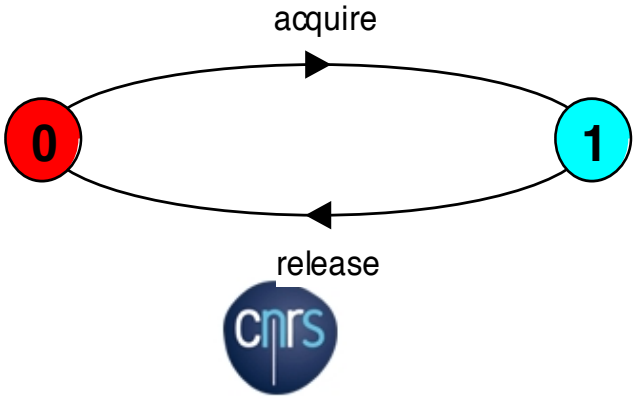
- The following definitions are equivalent:

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

```
USER = (acquire->use->release->USER)
       @{acquire,release}.
```



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

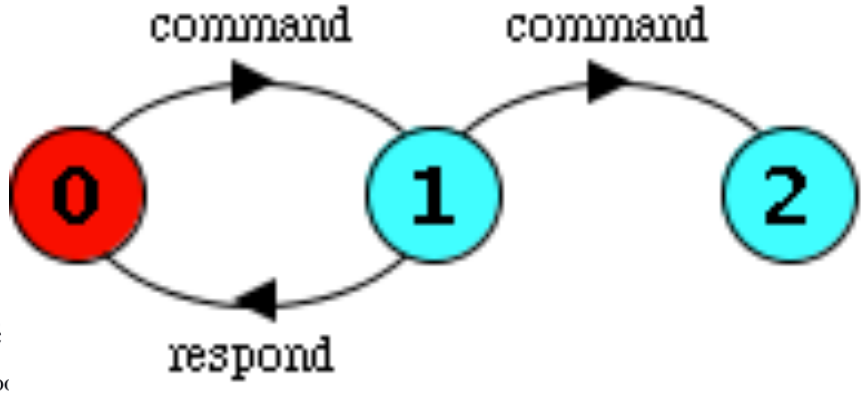


Step 3 : Preuve

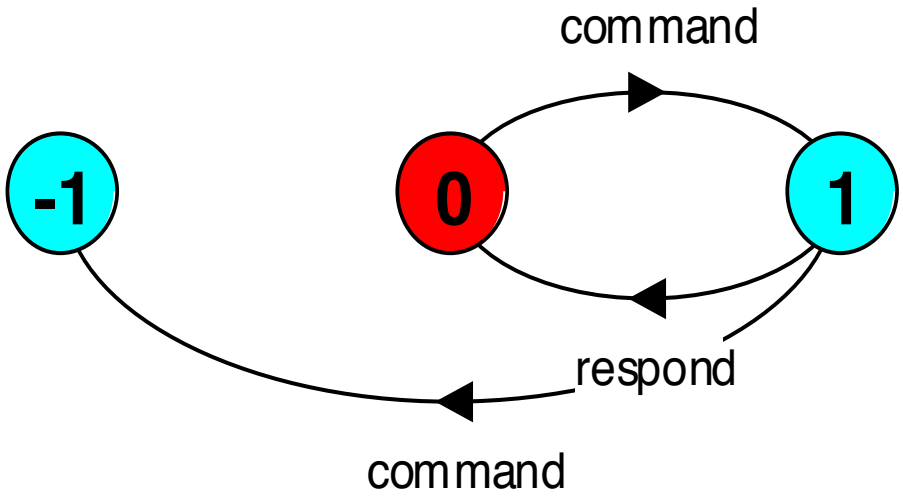
Safety

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

```
ACTUATOR
    =(command->ACTION),
ACTION
    =(respond->ACTUATOR
      |command->STOP).
```

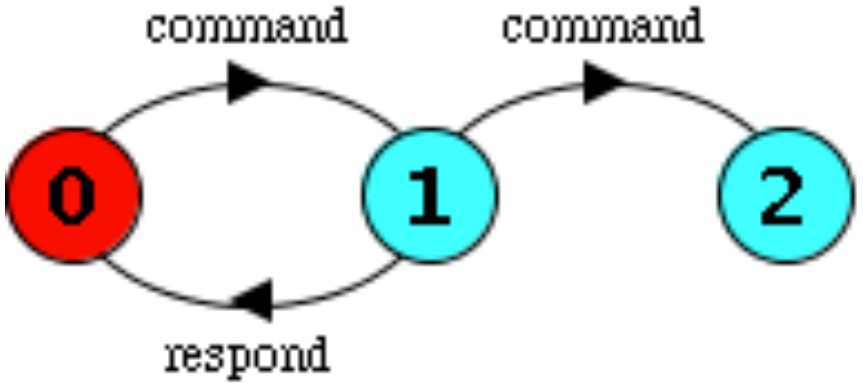



```
ACTUATOR
    =(command->ACTION),
ACTION
    =(respond->ACTUATOR
      |command->ERROR).
```



STOP

```
ACTUATOR
    =(command->ACTION),
ACTION
    =(respond->ACTUATOR
      |command->STOP).
```



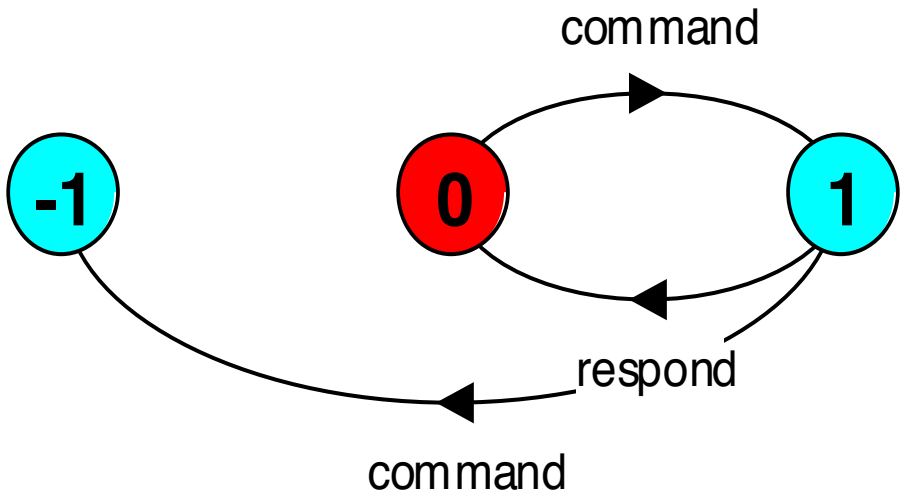
- analysis using LTSA:
  - Check -> Safety

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**

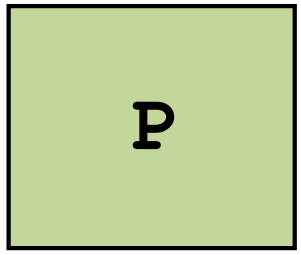
- Le processus à prouver
  - **P = (on -> Q), R = (off -> Q), Q = (off -> on -> P).**
- La preuve
  - PREUVE = (on -> OFF | off -> PREUVE),
    OFF = (off -> PREUVE | on -> ERROR).
- On compose
  - ||S = (P || PREUVE).

- On demande à LTS de vérifier
  - Compilation
    - property PREUVE violation.
  - Check -> Safety
    - Trace to property violation in PREUVE:
    - on
    - off
    - on
    - on

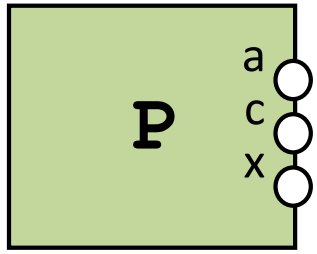
Schéma d'architecture
(ADL : architectural description language)

Structure diagrams

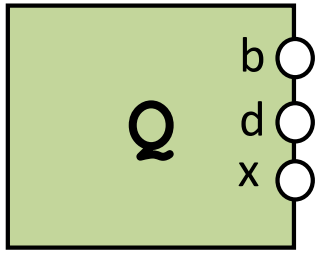
- Process P



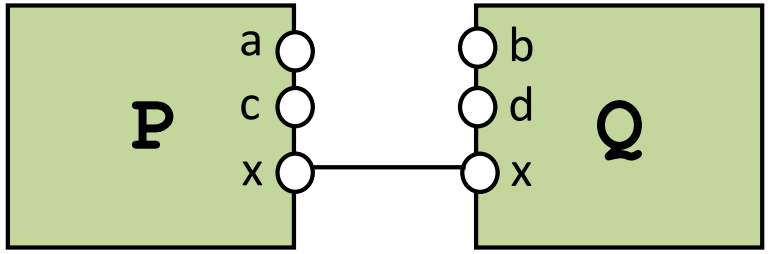
- Process P with alphabet {a, c, x}



- Process Q with alphabet {b, d, x}

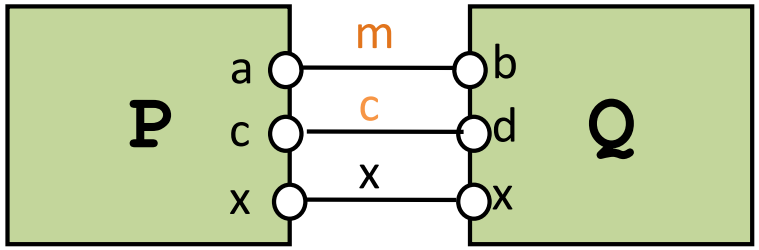


- Parallel composition P||Q

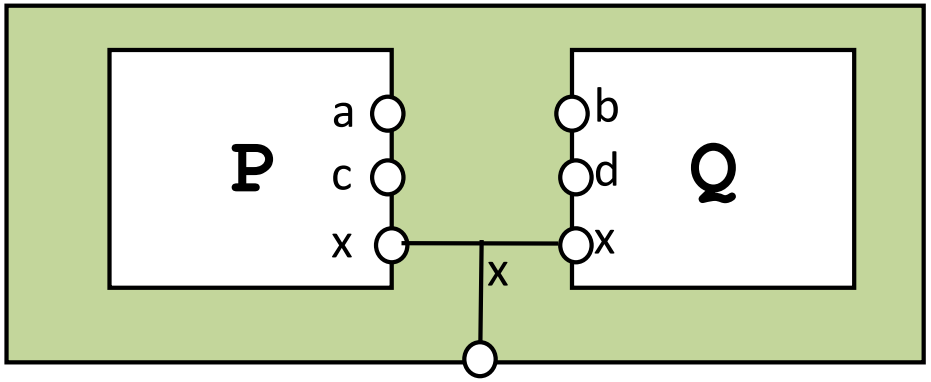


Structure diagrams

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



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

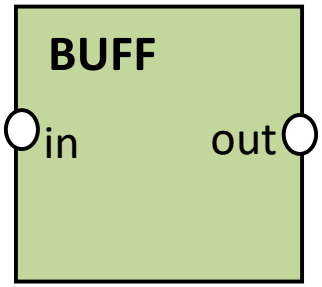


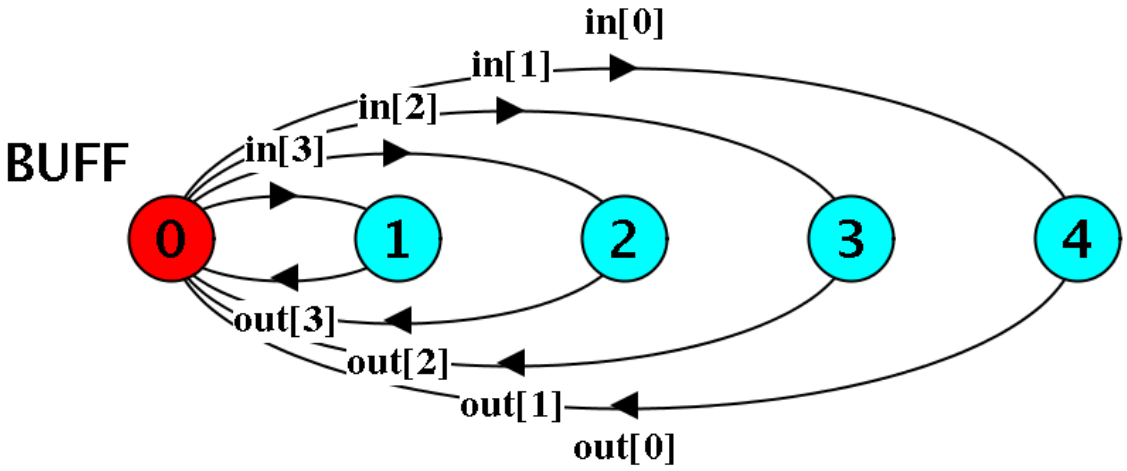
structure diagrams

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).
```





Université Nice Sophia Antipolis

i3S sophia antipolis

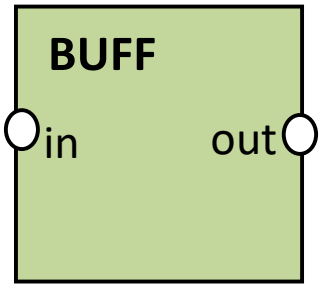
cnrs

structure diagrams

We use structure diagrams to capture the structure of a model expressed by the static combinators:

parallel composition, *relabeling* and *hiding*.

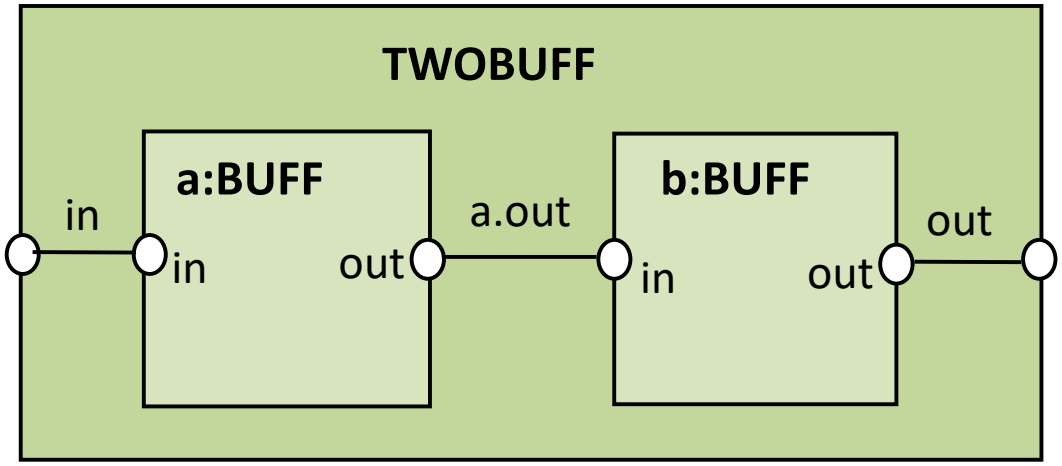
```
range T = 0..3
BUFF = (in[i:T]->out[i]->BUFF).
```



||TWOBUF = ?

Witch ||TWOBUF ??

This one →

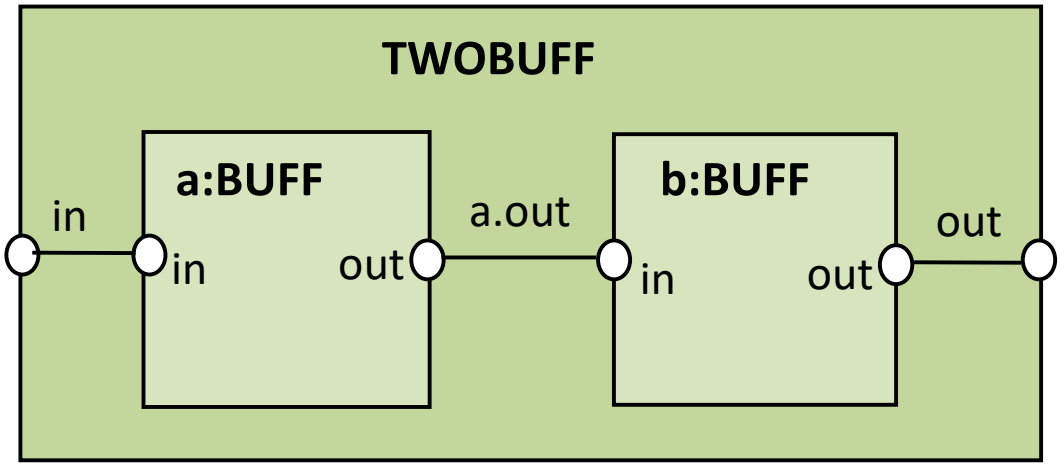


Université Nice Sophia Antipolis

i3S sophia antipolis

cnrs

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]}.
```

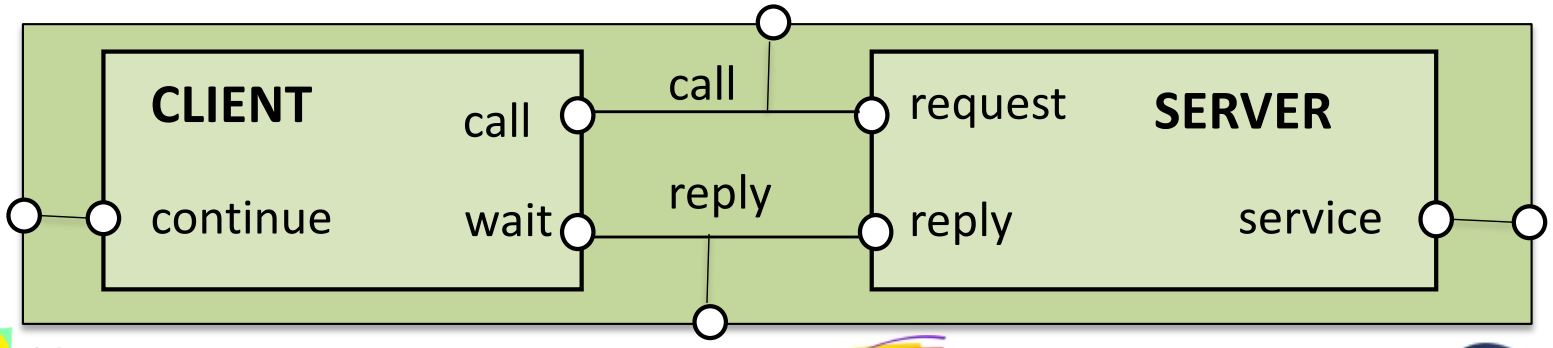
Etes vous d'accord ?

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}.
```

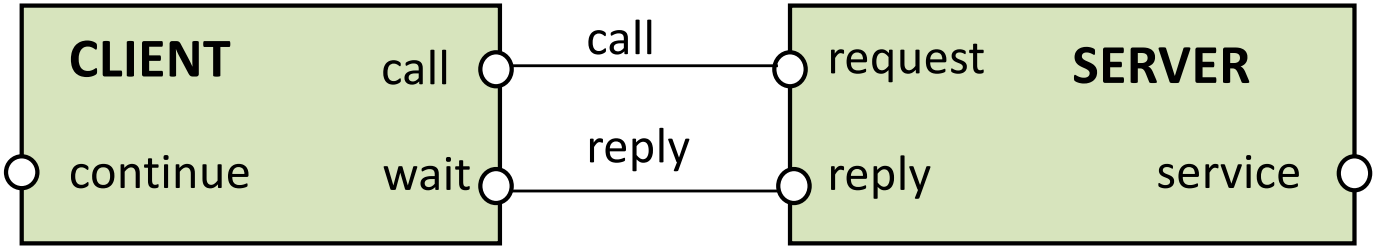


Test: Structure diagram for CLIENT_SERVER ?

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

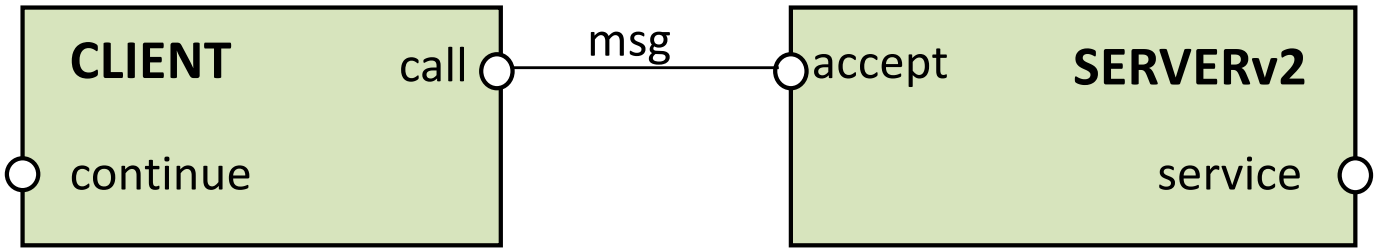
```
CLIENT = (call->wait->continue->CLIENT).
SERVER = (request->service->reply->SERVER).
||CLIENT_SERVER = (CLIENT || SERVER)
    /{call/request, reply/wait}.
```

Par :



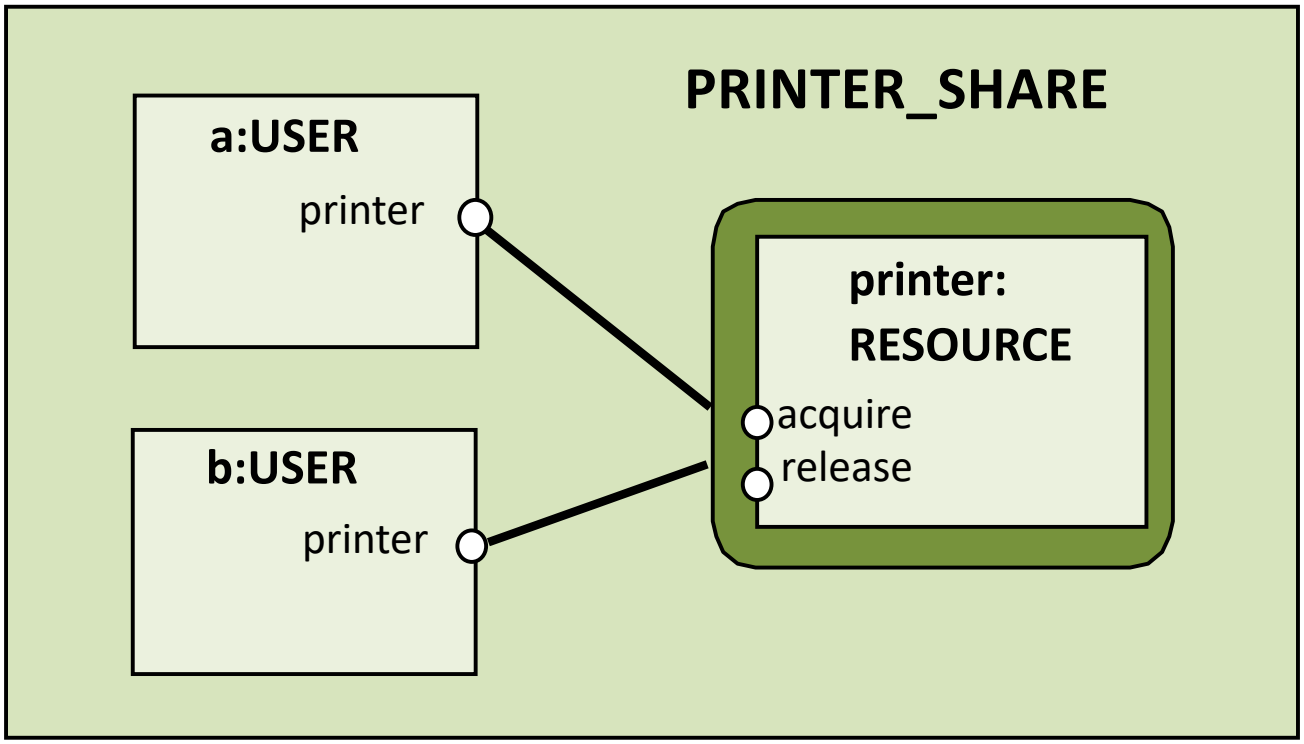
Test: Structure diagram for the second **CLIENT_SERVER** ?

```
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 ?

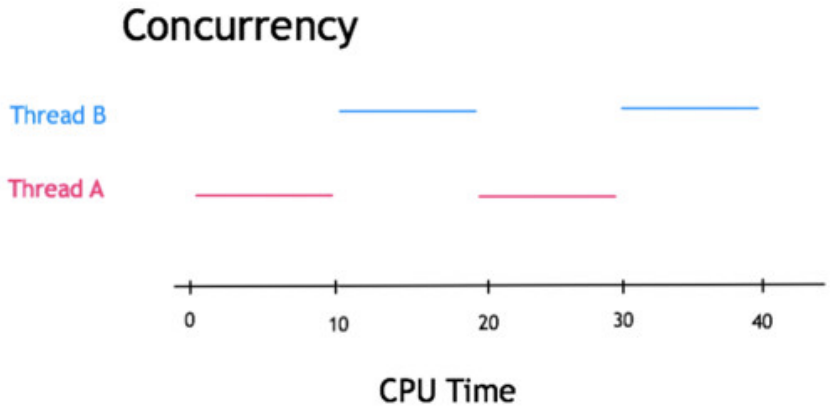
```
RESOURCE = (acquire->release->RESOURCE).
USER =     (printer.acquire->use
            ->printer.release->USER)\{use}.

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

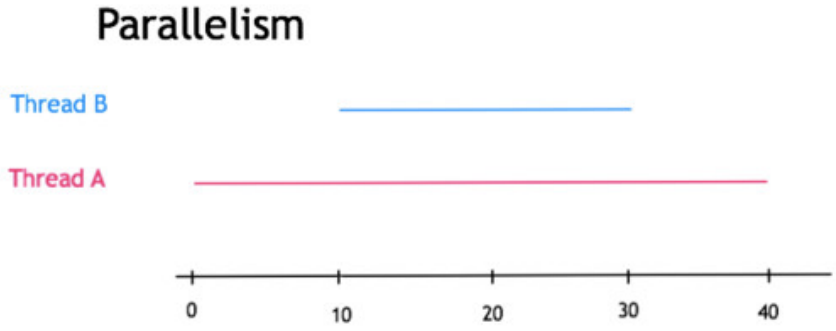


Rappel de tout ce que l'on vient de voir
(mes fiches de révisions)

Concurrency and parallelism

- Concurrency is not (only) parallelism
- Interleaved concurrency
  - Logically simultaneous processing
  - Interleaved execution on a single processor
- Parallelism
  - Physically simulteous processing
  - Require a multiprocessors, a multicore system or a distributed system





- **New processor** → multi-core
  - Concurrency and 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**

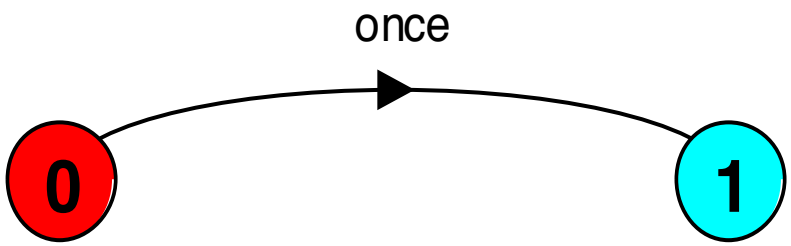
- **2 parts**
  - **finite state processes (FSP)** - algebraic form
    - to model processes as sequences of actions.
  - **labelled transition systems (LTS / LTSA)**
    - to analyse, display and animate behavior.

  - **FSP** - algebraic form
  - **LTS** - graphical form
  - **LTSA** – analysing tools

FSP - action prefix

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

- ONESHOT state machine (terminating process)
  - `ONESHOT = (once -> STOP).`

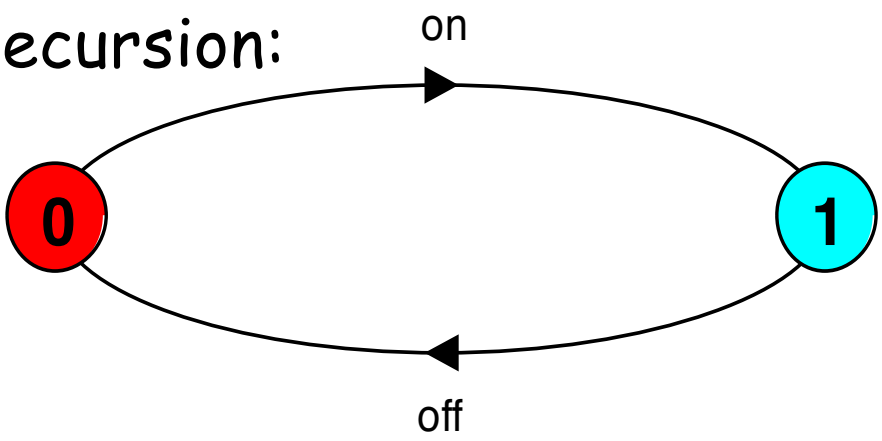


Convention:
- actions begin with lowercase letters
- PROCESSES begin with uppercase letters

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-> 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**.

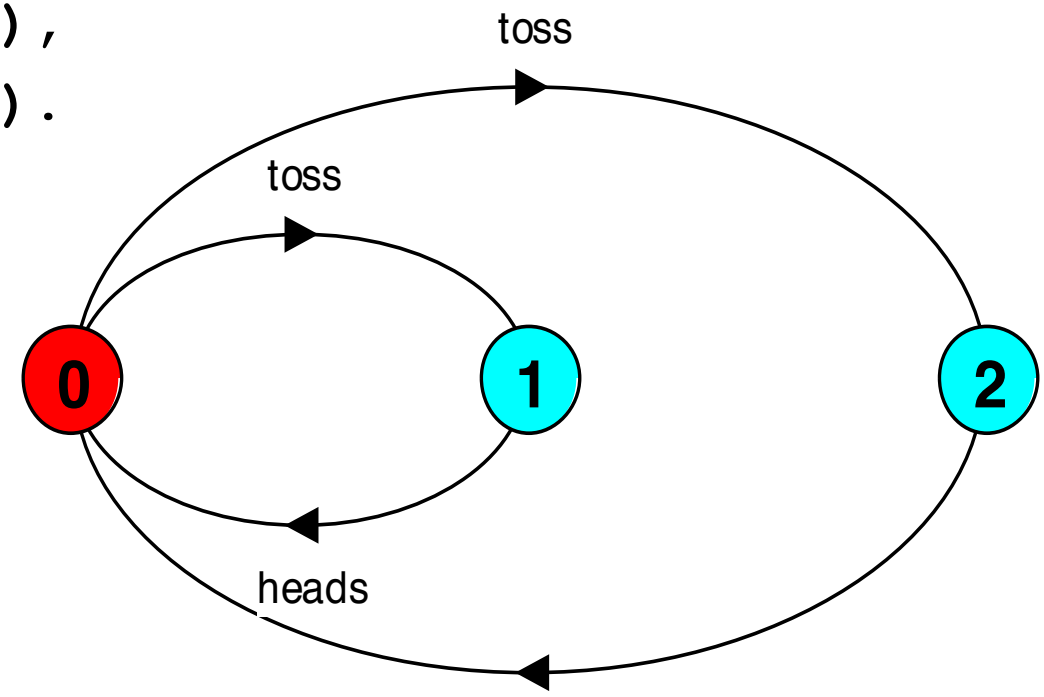
- *Who or what makes the choice?*
- *Is there a difference between input and output actions?*

Non-deterministic choice

- Process (x-> P | x -> 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.**

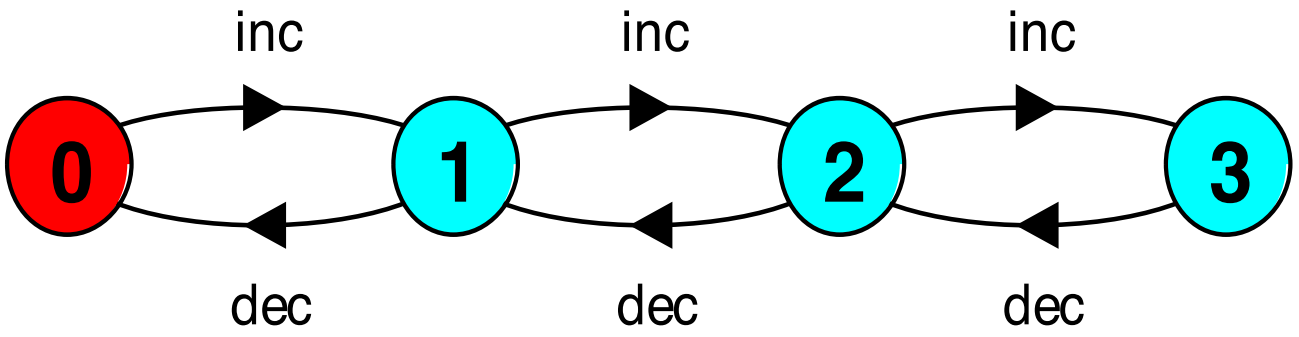


- Possible traces?
  - `toss->heads->toss->heads->toss->tails`

FSP - guarded actions

- The choice **(when B x -> 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.

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



parallel composition - action interleaving

- 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).
```

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`.

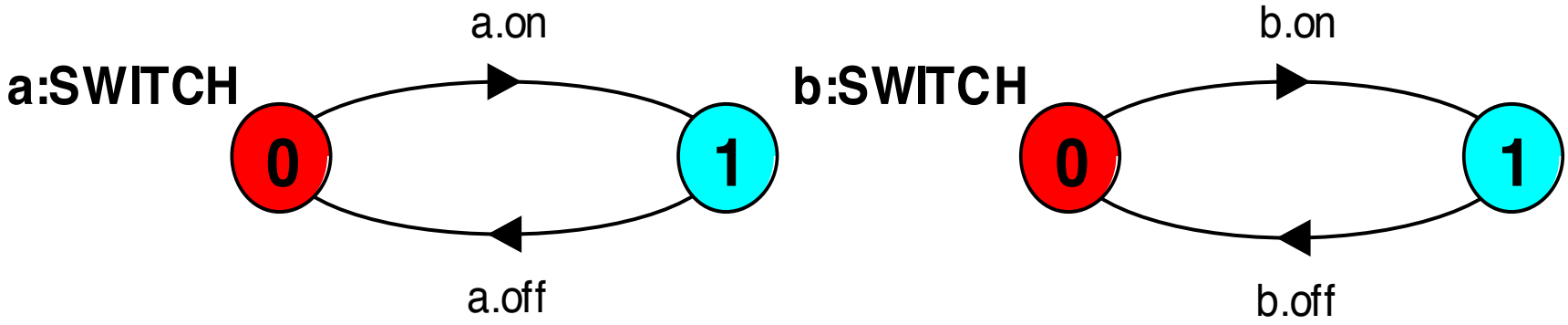
- ***LTS? Traces? Number of states?***

process instances and labeling

- **a:P**
  - prefixes each action label in the alphabet of P with a.

- Two **instances** of a switch process:

```
SWITCH =  (on->off->SWITCH).
||TWO_SWITCH = (a:SWITCH || b:SWITCH).
```

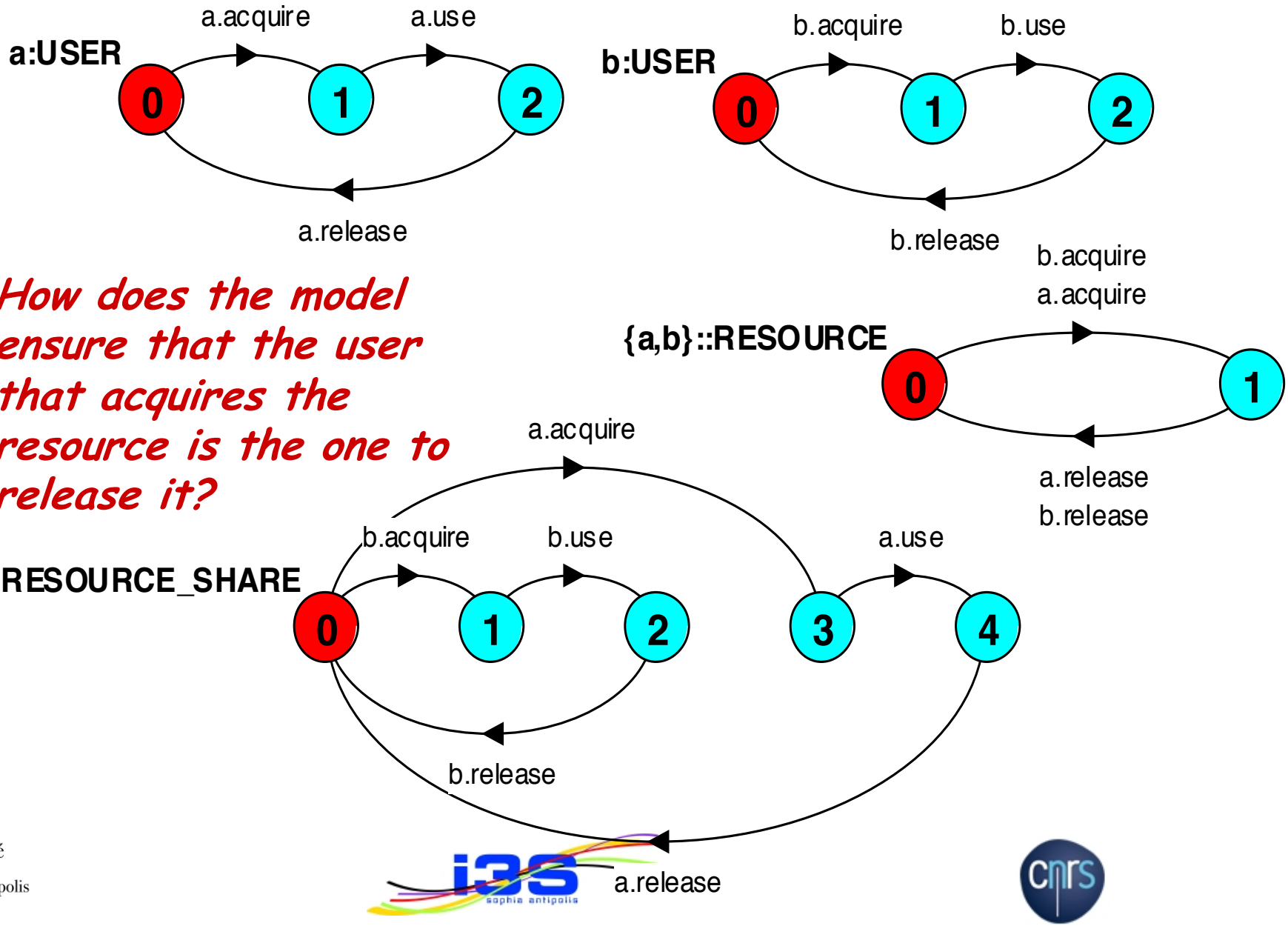



```
||TWO_SWITCH = ({a, b}:SWITCH ||  :SWITCH).
```

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

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

process prefix labels for shared resources



Université Nice Sophia Antipolis

i3S sophia antipolis

cnrs

process labeling by a set of prefix labels

- **{a1,..,ax}::P**
  - replaces every action label n in the alphabet of P 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).

- Process prefixing is useful for modeling **shared** resources:

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

action hiding - abstraction to reduce complexity

- When applied to a process P, the hiding operator \{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.

- Sometimes it is more convenient to specify the set of labels to be exposed....

- When applied to a process P, the interface operator @{a1..ax} hides all actions in the alphabet of P not labeled in the set a1..ax.

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