

# Validación y Verificación de Software

## **Sistemas Concurrentes y su Corrección**

# El problema de la corrección del software

Poder garantizar la corrección del software que construimos es una tarea deseable. En algunas aplicaciones, es, sin duda, crucial:

- Software para equipamiento médico

- Software para el control de vehículos

- Software para el control de procesos

- ...

Estos sistemas, cuyas fallas pueden ocasionar daños de gran importancia -- incluyendo la pérdidas de vidas humanas, catástrofes

# El problema de la corrección del software

Poder garantizar la corrección del software que construimos es una tarea deseable. Los sistemas críticos fundamentalmente responden a comportamientos reactivos, concurrentes y/o de tiempo real.

Software para equipamiento médico

Software para el control de vehículos

Software para el control de procesos

...

Estos sistemas, cuyas fallas pueden ocasionar daños de gran importancia -- incluyendo la pérdidas de vidas humanas, catástrofes

# El problema de la corrección del software

Poder garantizar la corrección del software que construimos es una tarea deseable. Los sistemas críticos fundamentalmente responden a comportamientos reactivos, concurrentes y/o de tiempo real.

Software para equipamiento médico

Software para el control de vehículos

Software para el control de procesos

...

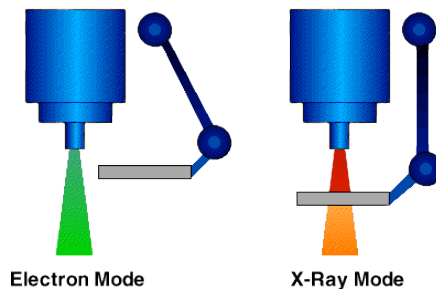
Estos sistemas, cuyas fallas pueden ocasionar daños de gran importancia -- incluyendo la pérdidas de vidas humanas, catástrofes

# Algunos Ejemplos

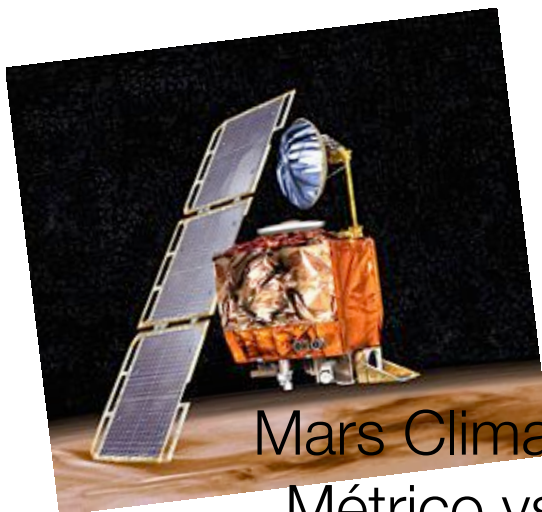


Pentium:  
FDIV

Ariane 5:  
64 bits fp  
vs 16 bits int



Therac-25



Mars Climate Orbiter:  
Métrico vs Imperial



Voto electrónico:  
Integridad/Confidencialidad

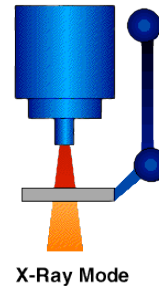
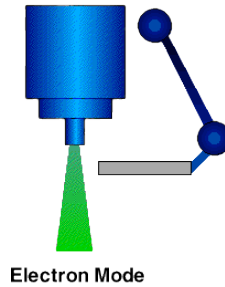
# Algunos Ejemplos



Pentium:  
FDIV

$$\frac{4195835 * 3145727}{3145727} = 4195579$$

Ariane 5:  
64 bits fp  
vs 16 bits int



Therac-25



Mars Climate Orbiter:  
Métrico vs Imperial



Voto electrónico:  
Integridad/Confidencialidad

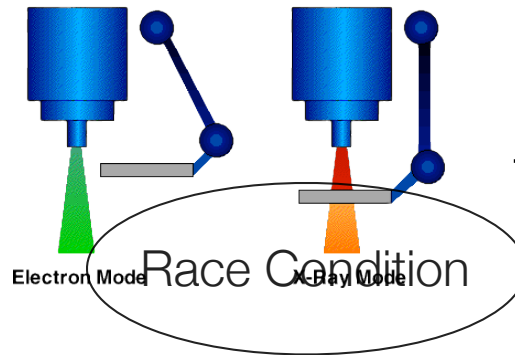
# Algunos Ejemplos



Pentium:  
FDIV

$$\frac{4195835 * 3145727}{3145727} = 4195579$$

Ariane 5:  
64 bits fp  
vs 16 bits int



Therac-25



Mars Climate Orbiter:  
Métrico vs Imperial



Voto electrónico:  
Integridad/Confidencialidad



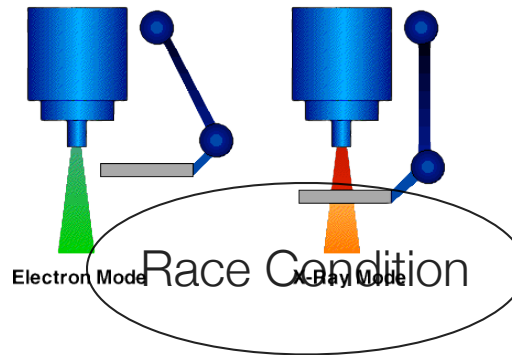
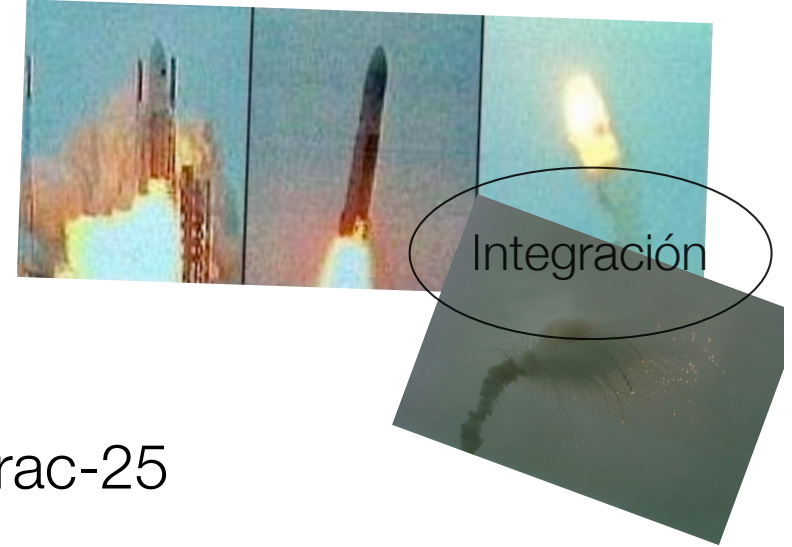
# Algunos Ejemplos



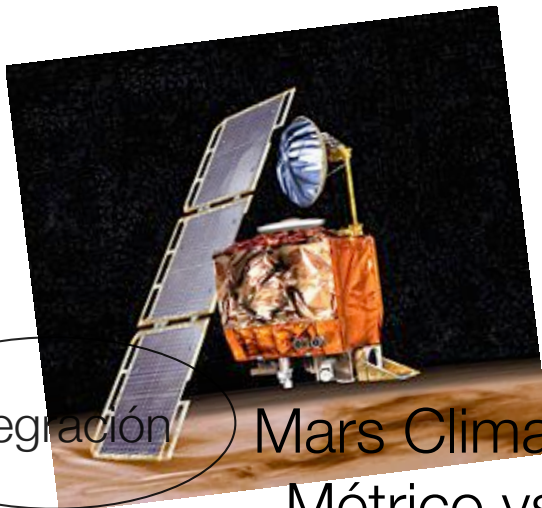
Pentium:  
FDIV

$$\frac{4195835 * 3145727}{3145727} = 4195579$$

Ariane 5:  
64 bits fp  
vs 16 bits int



Therac-25



Mars Climate Orbiter:  
Métrico vs Imperial



Voto electrónico:  
Integridad/Confidencialidad



# Algunos ejemplos

Ariane 5, 1996. Error de conversión de punto flotante de 64 bits a entero de 16 bits -- overflow aritmético (pérdidas estimadas en 500 millones de dólares)

Therac-25, 1985-1987. Software de control de equipamiento médico para radioterapia. Seis personas sobre-expuestas a radiación por error en el software.

Mars Climate Orbiter, 1999. Problemas de representación (un módulo usaba sistema imperial y otro sistema decimal).

Patriot Missile, 1991. 28 muertos y 100 heridos por un error de redondeo en el software de control de Patriot Missile.

# Algunos ejemplos

Ariane 5, 1996. Error de conversión de punto flotante de 64 bits a entero de 16 bits -- overflow aritmético (pérdidas estimadas en 500 millones de dólares)

• The 25,000 digit 1987 US software bug caused a radiation therapy error. Seis personas sobre-expuestas a radiación por error en el software.  
• <http://www.cs.tau.ac.il/~nachumd/verify/horror.html>

• Mars Climate Orbiter, 1999. Problemas de representación (un módulo usaba sistema imperial y otro sistema decimal)  
• [http://en.wikipedia.org/wiki/List\\_of\\_notable\\_software\\_bugs](http://en.wikipedia.org/wiki/List_of_notable_software_bugs)

Patriot Missile, 1991. 28 muertos y 100 heridos por un error de redondeo en el software de control de Patriot Missile.

# Limitaciones del testing y la simulación

Tanto el testing como la simulación involucran experimentos previos al lanzamiento o uso masivo del software. En general, ambos métodos proveen una serie de entradas al software, y estudian el comportamiento del mismo en esos casos.

El testing y la simulación raramente permiten garantizar la ausencia de errores. Una frase famosa al respecto: “El testing puede confirmar la presencia de errores pero nunca garantizar su ausencia”.

Afirmación capciosa: La verificación formal sólo puede hacerlo en teoría.

# Verificación (semi)automática de software

Existen serias limitaciones en lo que respecta a la verificación automática de software. Por ejemplo, el problema de decidir si un programa dado termina o no **no es computable**.

Sin embargo, si imponemos algunas restricciones sobre las propiedades que queremos verificar, y sobre qué sistemas, algunas tareas pueden realizarse mecánicamente. Model checking es un ejemplo de esto.

# Programación concurrente

# Programación concurrente

¿Qué?

Programación de sistemas compuestos de varios procesos/ programas que se ejecutan concurrentemente (o en paralelo, o en forma distribuida) e interactúan entre sí.



# Programación concurrente

¿Qué?

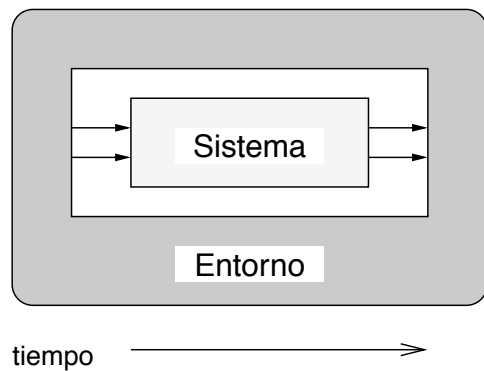
Programación de sistemas compuestos de varios procesos/ programas que se ejecutan concurrentemente (o en paralelo, o en forma distribuida) e interactúan entre sí.

¿Por qué?

Interacción con el entorno con el fin de observar/controlar dispositivos físicos (ej: dispositivos periféricos de una computadora)

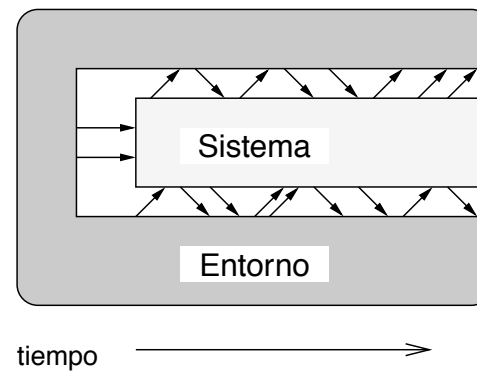
Mejorar el tiempo de respuesta en la interacción con el usuario

# Características reactivas de los programas



Sistema funcional

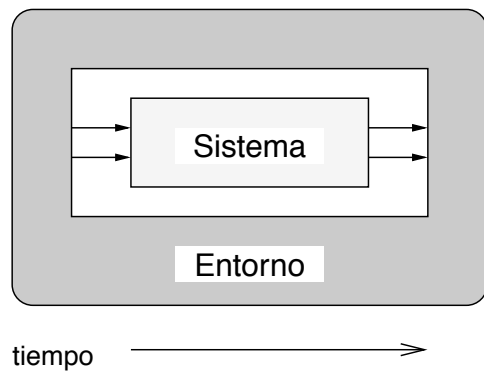
vs.



Sistema reactivo

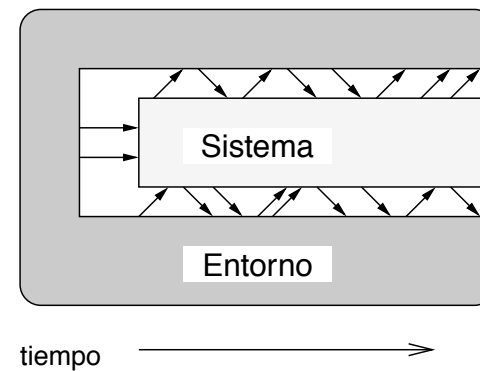
Muchos programas concurrentes suelen ser reactivos, es decir, su funcionalidad involucra la interacción permanente con el ambiente (y otros procesos).

# Características reactivas de los programas



Sistema funcional

vs.



Sistema reactivo

Muchos programas concurrentes suelen ser reactivos, es decir, su funcionalidad involucra la interacción permanente con el ambiente (y otros procesos).

Ejemplos: sistemas operativos, software de control, etc.

# Interacción de programas concurrentes

Los programas concurrentes están compuestos por procesos (o threads, o componentes) que necesitan interactuar. Existen varios mecanismos de interacción entre procesos. Entre éstos se encuentran la memoria compartida y el pasaje de mensajes.

Además, los programas concurrentes deben, en general, colaborar para llegar a un objetivo común, para lo cual la sincronización entre procesos es crucial.

# Algunos problemas comunes de los programas concurrentes

# Algunos problemas comunes de los programas concurrentes

Violación de propiedades universales (invariantes)

Starvation (inanición): Uno o más procesos quedan esperando indefinidamente un mensaje o la liberación de un recurso

Deadlock: dos o más procesos esperan mutuamente el avance del otro

Problemas de uso no exclusivo de recursos compartidos



# Semántica de programas concurrentes

Una semántica típica para programas concurrentes está basada en **sistemas de transición de estados**. Un sistema de transición de estados es un grafo dirigido en el cual:

- los **nodos** son los estados del sistema (posiblemente infinitos estados)

- las **aristas** son las transiciones atómicas de estados en estados, dadas por las sentencias del sistema.

# Semántica de programas concurrentes

Una semántica típica para programas concurrentes está basada en **sistemas de transición de estados**. Un sistema de transición de estados es un grafo dirigido en el cual:

(los **nodos** son los estados del sistema  
(posiblemente infinitos estados))

las **aristas** son las transiciones atómicas de estados en estados, dadas por las sentencias del sistema.

Estado inicial

Conjunto de estados

# Semántica de programas concurrentes

Una semántica típica para programas concurrentes está basada en **sistemas de transición de estados**. Un sistema de transición de estados es un grafo dirigido en el cual:

los **nodos** son los estados del sistema  
(posiblemente infinitos estados)

las **aristas** son las transiciones atómicas de estados en estados, dadas por las sentencias del sistema.

Estado inicial  
Conjunto de estados

Ejemplo

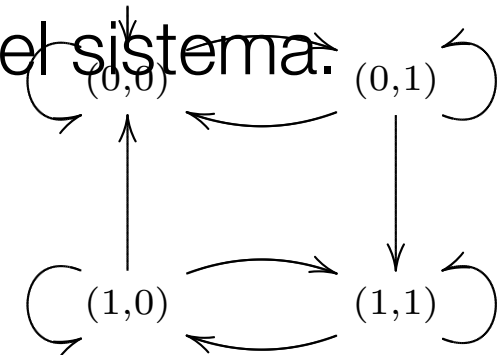
$$S = \{0, 1\} \times \{0, 1\}$$

$$s_0 = (0, 0)$$

$$(x, y) \rightarrow (x, 0)$$

$$(x, y) \rightarrow (x, 1)$$

$$(x, y) \rightarrow (y, y)$$



# Semántica de programas concurrentes

Una semántica típica para programas concurrentes está basada en **sistemas de transición de estados**. Un sistema de transición de estados es un grafo dirigido en el cual:

Muchas veces la transición aparece etiquetada con el evento que la origina.

los **nodos** son los estados del sistema  
(posiblemente infinitos estados)

las **aristas** son las transiciones atómicas de estados en estados, dadas por las sentencias del sistema.

Estado inicial

Conjunto de estados

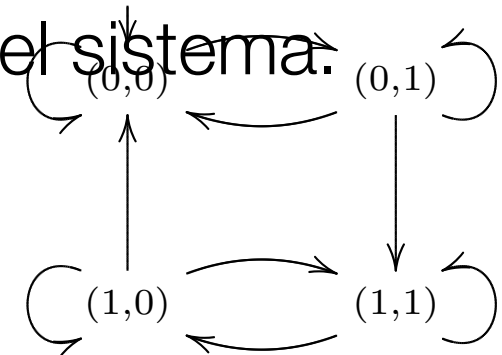
$$S = \{0, 1\} \times \{0, 1\}$$

$$s_0 = (0, 0)$$

$$(x, y) \rightarrow (x, 0)$$

$$(x, y) \rightarrow (x, 1)$$

$$(x, y) \rightarrow (y, y)$$



# Ejecuciones de un programa

Una **ejecución** es una secuencia de estados  $s_0s_1s_2\cdots$ , tal que:

$s_0$  es el estado inicial,

puede llegarse desde  $s_i$  a  $s_{i+1}$  por alguna sentencia atómica (i.e. transición) del sistema.

El conjunto de todas las ejecuciones determina el comportamiento de un programa concurrente modelado con un sistema de transición de estados.

# Ejecuciones de un programa

Una **ejecución** es una secuencia de estados  $s_0s_1s_2\cdots$ , tal que:

$s_0$  es el estado inicial,

puede llegarse desde  $s_i$  a  $s_{i+1}$  por alguna sentencia atómica (i.e. transición) del sistema.

El conjunto de todas las ejecuciones determina el comportamiento de un programa concurrente modelado con un sistema de transición de estados.

$(0, 0)(0, 0)(0, 1)(1, 1)(1, 0)(1, 1)(1, 0)(1, 0)(1, 1)(1, 0)(0, 0)(0, 1)(0, 1)(1, 1) \cdots$

$(0, 1)(0, 0)(0, 1)(1, 1)(1, 0)(1, 1)(1, 0)(1, 0)(1, 1)(1, 0)(0, 0)(0, 1)(0, 1)(1, 1) \cdots$

$(0, 0)(0, 1)(1, 0)(1, 1)(0, 0)(1, 1)(1, 0)(1, 0)(1, 1)(1, 0)(0, 0)(1, 1)(0, 1)(1, 1) \cdots$



# Ejecuciones de un programa

Una **ejecución** es una secuencia de estados  $s_0s_1s_2\cdots$ , tal que:

$s_0$  es el estado inicial,

puede llegarse desde  $s_i$  a  $s_{i+1}$  por alguna sentencia atómica (i.e. transición) del sistema.

El conjunto de todas las ejecuciones determina el comportamiento de un programa concurrente modelado con un sistema de transición de estados.

✓  $(0, 0)(0, 0)(0, 1)(1, 1)(1, 0)(1, 1)(1, 0)(1, 0)(1, 1)(1, 0)(0, 0)(0, 1)(0, 1)(1, 1) \cdots$   
 $(0, 1)(0, 0)(0, 1)(1, 1)(1, 0)(1, 1)(1, 0)(1, 0)(1, 1)(1, 0)(0, 0)(0, 1)(0, 1)(1, 1) \cdots$   
 $(0, 0)(0, 1)(1, 0)(1, 1)(0, 0)(1, 1)(1, 0)(1, 0)(1, 1)(1, 0)(0, 0)(1, 1)(0, 1)(1, 1) \cdots$

# Ejecuciones de un programa

Una **ejecución** es una secuencia de estados  $s_0s_1s_2\cdots$ , tal que:

$s_0$  es el estado inicial,

puede llegarse desde  $s_i$  a  $s_{i+1}$  por alguna sentencia atómica (i.e. transición) del sistema.

El conjunto de todas las ejecuciones determina el comportamiento de un programa concurrente modelado con un sistema de transición de estados.

✓  $(0, 0)(0, 0)(0, 1)(1, 1)(1, 0)(1, 1)(1, 0)(1, 0)(1, 1)(1, 0)(0, 0)(0, 1)(0, 1)(1, 1) \cdots$

✗  $(0, 1)(0, 0)(0, 1)(1, 1)(1, 0)(1, 1)(1, 0)(1, 0)(1, 1)(1, 0)(0, 0)(0, 1)(0, 1)(1, 1) \cdots$

$(0, 0)(0, 1)(1, 0)(1, 1)(0, 0)(1, 1)(1, 0)(1, 0)(1, 1)(1, 0)(0, 0)(1, 1)(0, 1)(1, 1) \cdots$

# Ejecuciones de un programa

Una **ejecución** es una secuencia de estados  $s_0s_1s_2\cdots$ , tal que:

$s_0$  es el estado inicial,

puede llegarse desde  $s_i$  a  $s_{i+1}$  por alguna sentencia atómica (i.e. transición) del sistema.

El conjunto de todas las ejecuciones determina el comportamiento de un programa concurrente modelado con un sistema de transición de estados.

✓  $(0, 0)(0, 0)(0, 1)(1, 1)(1, 0)(1, 1)(1, 0)(1, 0)(1, 1)(1, 0)(0, 0)(0, 1)(0, 1)(1, 1) \cdots$

✗  $(\boxed{0, 1})(0, 0)(0, 1)(1, 1)(1, 0)(1, 1)(1, 0)(1, 0)(1, 1)(1, 0)(0, 0)(0, 1)(0, 1)(1, 1) \cdots$

$(0, 0)(0, 1)(1, 0)(1, 1)(0, 0)(1, 1)(1, 0)(1, 0)(1, 1)(1, 0)(0, 0)(1, 1)(0, 1)(1, 1) \cdots$

# Ejecuciones de un programa

Una **ejecución** es una secuencia de estados  $s_0 s_1 s_2 \dots$ , tal que:

$s_0$  es el estado inicial,

puede llegarse desde  $s_i$  a  $s_{i+1}$  por alguna sentencia atómica (i.e. transición) del sistema.

El conjunto de todas las ejecuciones determina el comportamiento de un programa concurrente modelado con un sistema de transición de estados.

✓  $(0, 0)(0, 0)(0, 1)(1, 1)(1, 0)(1, 1)(1, 0)(1, 0)(1, 1)(1, 0)(0, 0)(0, 1)(0, 1)(1, 1) \dots$

✗  $(\boxed{0, 1})(0, 0)(0, 1)(1, 1)(1, 0)(1, 1)(1, 0)(1, 0)(1, 1)(1, 0)(0, 0)(0, 1)(0, 1)(1, 1) \dots$

✗  $(0, 0)(0, 1)(1, 0)(1, 1)(0, 0)(1, 1)(1, 0)(1, 0)(1, 1)(1, 0)(0, 0)(1, 1)(0, 1)(1, 1) \dots$

# Ejecuciones de un programa

Una **ejecución** es una secuencia de estados  $s_0s_1s_2\cdots$ , tal que:

$s_0$  es el estado inicial,

puede llegarse desde  $s_i$  a  $s_{i+1}$  por alguna sentencia atómica (i.e. transición) del sistema.

El conjunto de todas las ejecuciones determina el comportamiento de un programa concurrente modelado con un sistema de transición de estados.

✓  $(0, 0)(0, 0)(0, 1)(1, 1)(1, 0)(1, 1)(1, 0)(1, 0)(1, 1)(1, 0)(0, 0)(0, 1)(0, 1)(1, 1) \cdots$

✗  $(\boxed{0, 1})(0, 0)(0, 1)(1, 1)(1, 0)(1, 1)(1, 0)(1, 0)(1, 1)(1, 0)(0, 0)(0, 1)(0, 1)(1, 1) \cdots$

✗  $(0, 0)(\boxed{0, 1}(1, 0))(1, 1)(0, 0)(1, 1)(1, 0)(1, 0)(1, 1)(1, 0)(0, 0)(1, 1)(0, 1)(1, 1) \cdots$

# Cómo se ejecutan los procesos concurrentes?

De acuerdo al modelo computacional descripto, los procesos concurrentes se ejecutan **intercalando** las acciones atómicas que los componen. Llamamos a esto, **interleaving**.

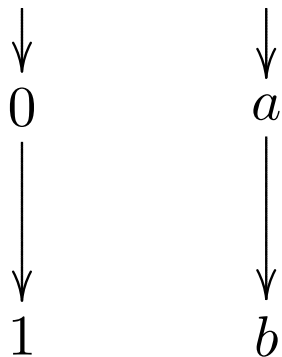
El orden en que se ejecutan las acciones atómicas no puede decidirse en general, y un mismo par de procesos puede tener diferentes ejecuciones debido al **no determinismo** en la elección de las acciones atómicas a ejecutar.



# Cómo se ejecutan los procesos concurrentes?

De acuerdo al modelo computacional descripto, los procesos concurrentes se ejecutan **intercalando** las acciones atómicas que los componen. Llamamos a esto, **interleaving**.

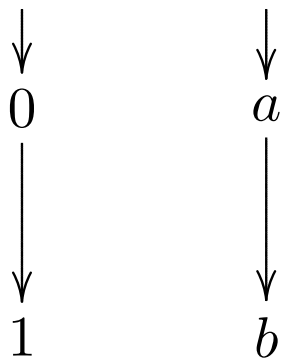
El orden en que se ejecutan las acciones atómicas no puede decidirse en general, y un mismo par de procesos puede tener diferentes ejecuciones debido al **no determinismo** en la elección de las acciones atómicas a ejecutar.



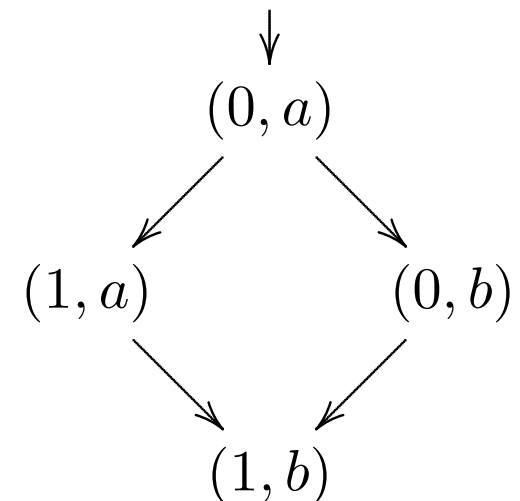
# Cómo se ejecutan los procesos concurrentes?

De acuerdo al modelo computacional descripto, los procesos concurrentes se ejecutan **intercalando** las acciones atómicas que los componen. Llamamos a esto, **interleaving**.

El orden en que se ejecutan las acciones atómicas no puede decidirse en general, y un mismo par de procesos puede tener diferentes ejecuciones debido al **no determinismo** en la elección de las acciones atómicas a ejecutar.



La composición paralela de los STE de la izquierda, da como resultado el STE de la derecha.



# Concurrencia: un ejemplo

¿Qué hace este programa?

```
int y1 = 0;
int y2 = 0;
short in_critical = 0;
active proctype process_1() {
    do
        :: true ->
            y1 = y2+1;
            ((y2==0) || (y1<=y2));
            in_critical++;
            in_critical--;
            y1 = 0;
    od
}
```

```
active proctype process_2() {
    do
        :: true ->
            y2 = y1+1;
            ((y1==0) || (y2<y1));
            in_critical++;
            in_critical--;
            y2 = 0;
    od
}
```

# Concurrencia: un ejemplo

En este ejemplo, el conjunto de estados está dado por la combinación de todos los valores posibles de las variables globales `y1`, `y2` e `in_critical`, además de dos variables implícitas: los program counters (pc) de los dos procesos.

El estado inicial es aquel en el cual las tres variables valen 0 y cada pc se sitúa al inicio de cada proceso.

Por cada sentencia atómica tenemos una transición. Por ejemplo, la sentencia

```
in_critical++
```

define un conjunto de transiciones donde cada una relaciona todos los estados con aquellos en los cuales `y1` e `y2` no cambian su valor, e `in_critical` se incrementa en uno (y el pc del proceso correspondiente también se incrementa).

```

int y1 = 0;
int y2 = 0;
short in_critical = 0;

active proctype process_1() {
    do
        :: true ->
0:      y1 = y2+1;
1:      ((y2==0) || (y1<=y2));
        in_critical++;
2:      in_critical--;
3:      y1 = 0;
    od
}

```

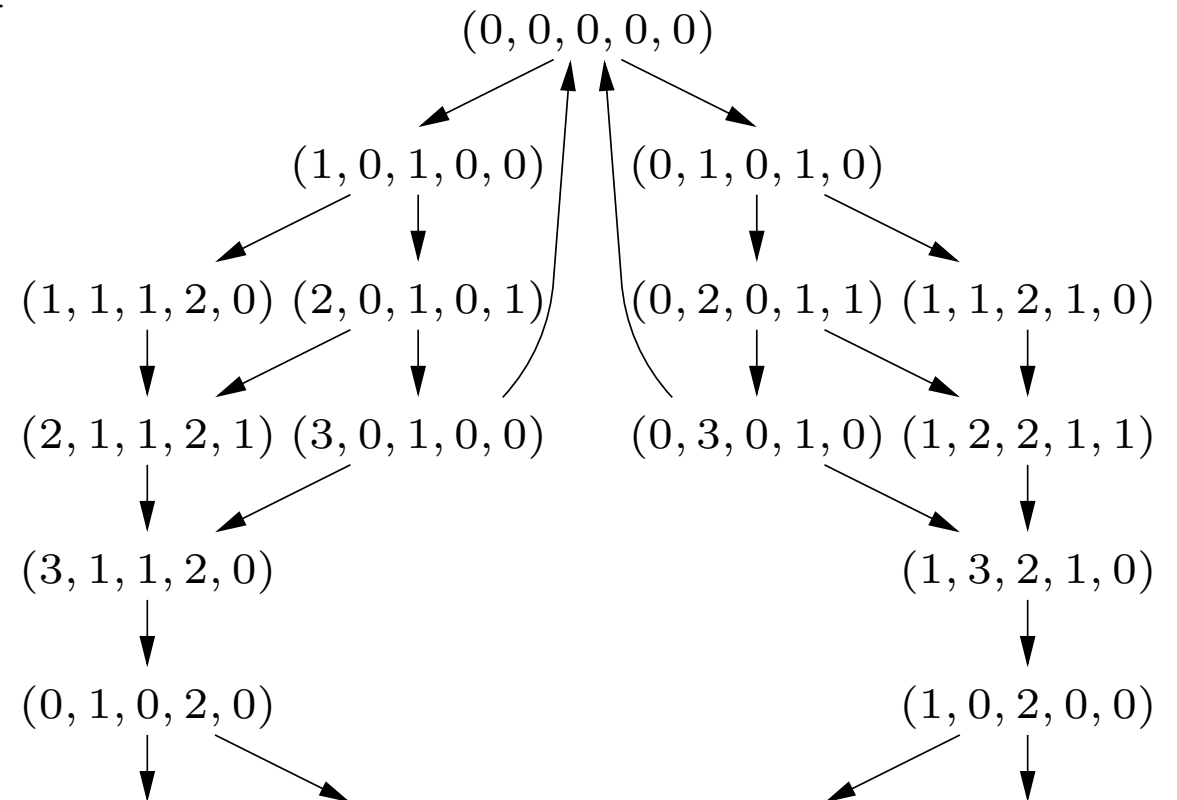
```

active proctype process_2() {
    do
        :: true ->
0:      y2 = y1+1;
1:      ((y1==0) || (y2<y1));
        in_critical++;
2:      in_critical--;
3:      y2 = 0;
    od
}

```

Estructura del estado:

$(pc_1, pc_2, y1, y2, in\_critical)$



# Razonamiento sobre programas concurrentes

En general, es muy difícil razonar sobre programas concurrentes. Luego, garantizar que un programa concurrente es correcto es también muy difícil.

Una de las razones tiene que ver con que diferentes interleavings de acciones atómicas pueden llevar a diferentes resultados o comportamientos de los sistemas concurrentes.

El número de interleavings posibles, por su parte, es en general muy grande, lo que hace que el testing difícilmente pueda brindarnos confianza de que nuestros programas concurrentes funcionan correctamente.

# Razonamiento sobre programas concurrentes

En general, es muy difícil razonar sobre programas concurrentes. Luego, garantizar que un programa concurrente es correcto es también muy difícil.

Una de las razones tiene que ver con que diferentes interleavings de acciones atómicas pueden llevar a diferentes resultados o comportamientos de los sistemas concurrentes.

El número de interleavings posibles, por su parte, es en general muy grande, lo que hace que el testing difícilmente pueda

brindarnos confianza de que nuestros programas concurrentes funcionan correctamente. En el ejemplo anterior es prácticamente imposible realizar el test que lleve al overflow.

# Razonamiento sobre programas concurrentes

En general, es muy difícil razonar sobre programas concurrentes. Luego, garantizar que un programa concurrente es correcto es también muy difícil.

Una de las razones tiene que ver con que diferentes interleavings de acciones atómicas pueden llevar a diferentes resultados o comportamientos de los sistemas concurrentes. El espacio de estados crece exponencialmente con el número de componentes.

El número de interleavings posibles, por su parte, es en general muy grande, lo que hace que el testing difícilmente pueda

brindarnos confianza de que nuestros programas concurrentes funcionan correctamente. En el ejemplo anterior es prácticamente imposible realizar el test que lleve al overflow.



# Abstracción: Modelos de programas concurrentes

Una forma de aliviar, en parte, el problema de razonar sobre programas concurrentes es considerar **representaciones abstractas** de éstos. Estas representaciones abstractas, llamadas **modelos**, nos permiten concentrarnos en las características particulares que queremos analizar.

Las álgebras de procesos (CSP, CCS, ACP, LOTOS, etc.) permiten construir estos modelos, concentrándose en las propiedades funcionales de sistemas concurrentes. Para esto, es importante considerar los **eventos** en los cuales cada proceso puede estar involucrado, y los **patrones de ocurrencia** que éstos siguen.

# El lenguaje FSP

El lenguaje que utilizaremos es FSP (Finite State Processes). Incluye, entre otras cosas:

Prefijos de acciones:

$x \rightarrow P$

# El lenguaje FSP

El lenguaje que utilizaremos es FSP (Finite State Processes). Incluye, entre otras cosas:

Evento

Prefijos de acciones:

$x \rightarrow P$

Proceso

# El lenguaje FSP

El lenguaje que utilizaremos es FSP (Finite State Processes). Incluye, entre otras cosas:

Construyen un

Prefijos de un proceso

$x \rightarrow P$

# El lenguaje FSP

El lenguaje que utilizaremos es FSP (Finite State Processes). FSP es una variante simple de CSP, que incluye, entre otras cosas:

Prefijos de acciones:

$$x \rightarrow P$$

Definiciones recursivas:

$$\text{OFF} = (\text{on} \rightarrow (\text{off} \rightarrow \text{OFF}))$$

Elección:

$$(x \rightarrow P \mid y \rightarrow Q)$$

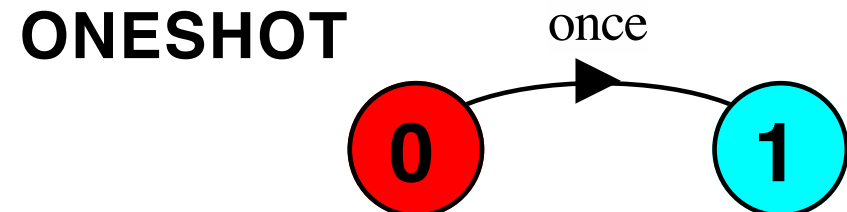
# Semántica de procesos

La semántica de los procesos FSP está dada en términos de sistemas de transición de estados y trazas. En particular, los procesos pueden verse gráficamente como sistemas de transición de estados.

Por ejemplo, el proceso

ONESHOT = once -> STOP.

puede visualizarse como:



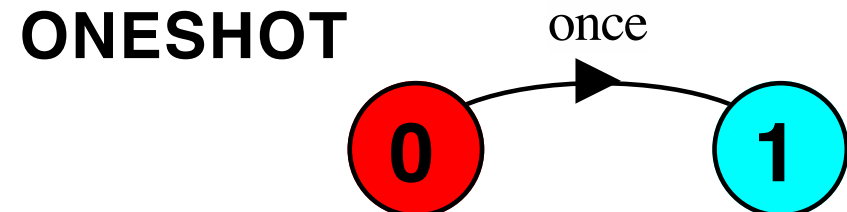
# Semántica de procesos

La semántica de los procesos FSP está dada en términos de sistemas de transición de estados y trazas. En particular, los procesos pueden verse gráficamente como sistemas de transición de estados.

Por ejemplo, el proceso

ONESHOT = once -> STOP.

puede visualizarse como:



Los gráficos están hechos con LTSA que es la herramienta que soporta a FSP

# FSP: Prefijos de acciones

Uno puede definir un proceso que, luego de realizar un evento o acción atómica  $x$ , se comporta exactamente como cierto proceso  $P$  usando prefijos:

$$(x \rightarrow P)$$

Esto es utilizado, por ejemplo, en el proceso ONESHOT visto anteriormente.

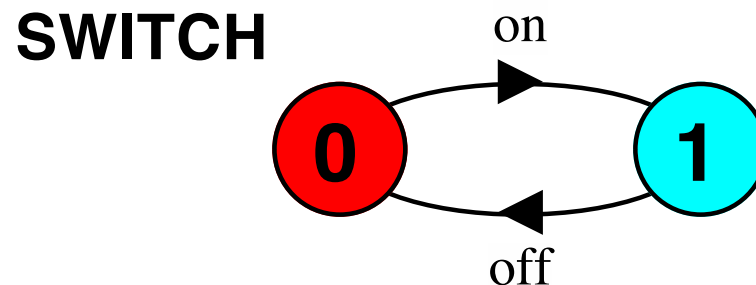


# FSP: Recursión

El comportamiento de un proceso también puede definirse usando recursión, por ejemplo:

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

Por supuesto, estos procesos pueden verse gráficamente como sistemas de transición de estados (y la herramienta LTSA lo hace por nosotros!).

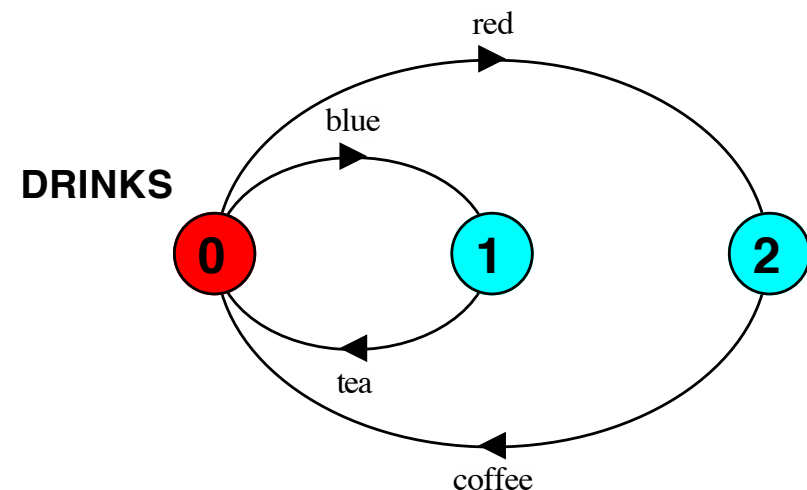


# FSP: Elección

La ramificación en el flujo de ejecución de un proceso se describe mediante la elección.

Ejemplo:  $\text{DRINKS} = (\text{red} \rightarrow \text{coffee} \rightarrow \text{DRINKS} \mid \text{blue} \rightarrow \text{tea} \rightarrow \text{DRINKS})$ .

Y, nuevamente, estos procesos pueden verse gráficamente como sistemas de transición de estados.

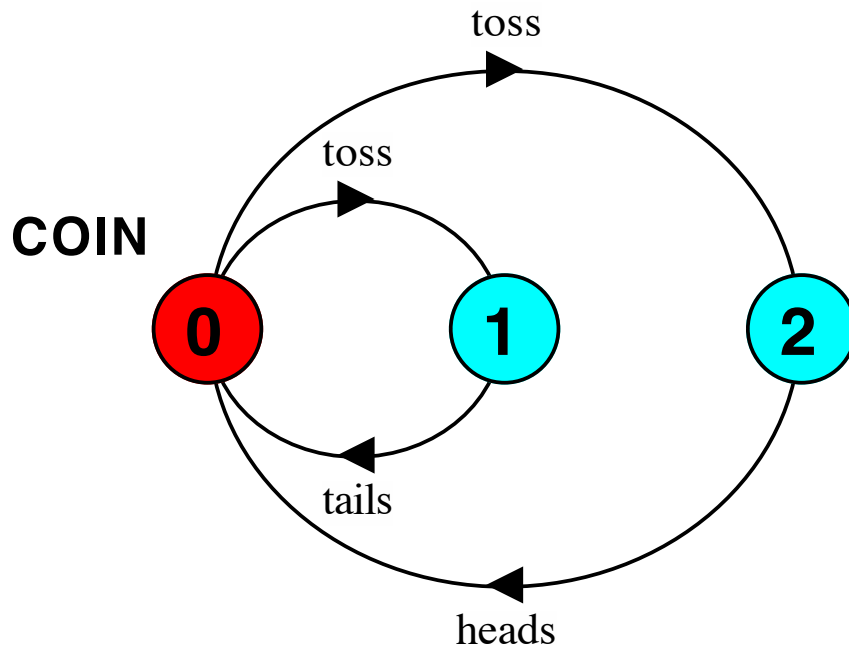


# FSP: Elección no determinista

Es simplemente un caso particular de elección.

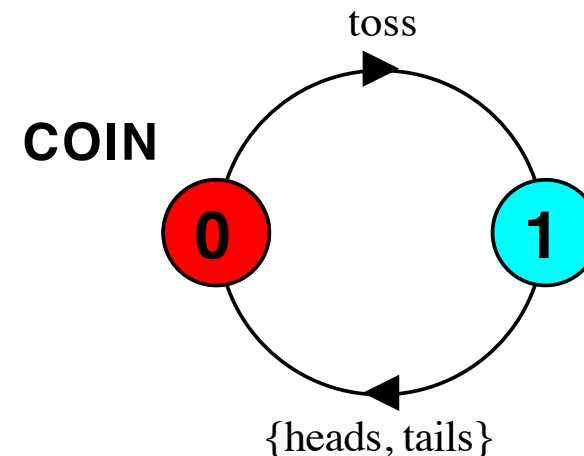
Ejemplo:

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



Comparar:

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

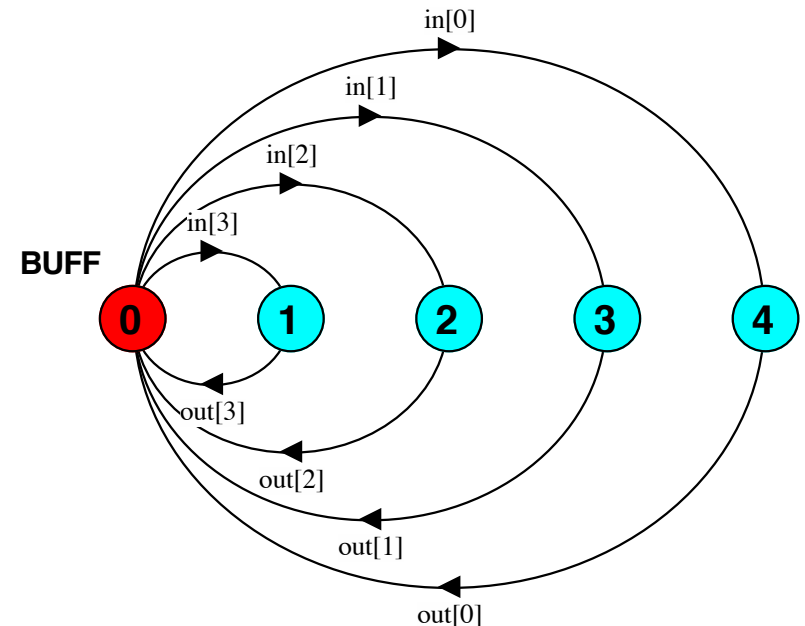


# FSP: Procesos y acciones indexados

Cuando se necesita modelar procesos que tomen un número grande de posibles valores, pueden utilizarse acciones (y procesos) indexadas, donde el rango del índice debe ser finito.

Esta facilidad tiene múltiples usos. En particular, puede emplearse para modelar estado explícito.


$\text{BUFF} = (\text{in}[i:0..3] \rightarrow \text{out}[i] \rightarrow \text{BUFF}).$

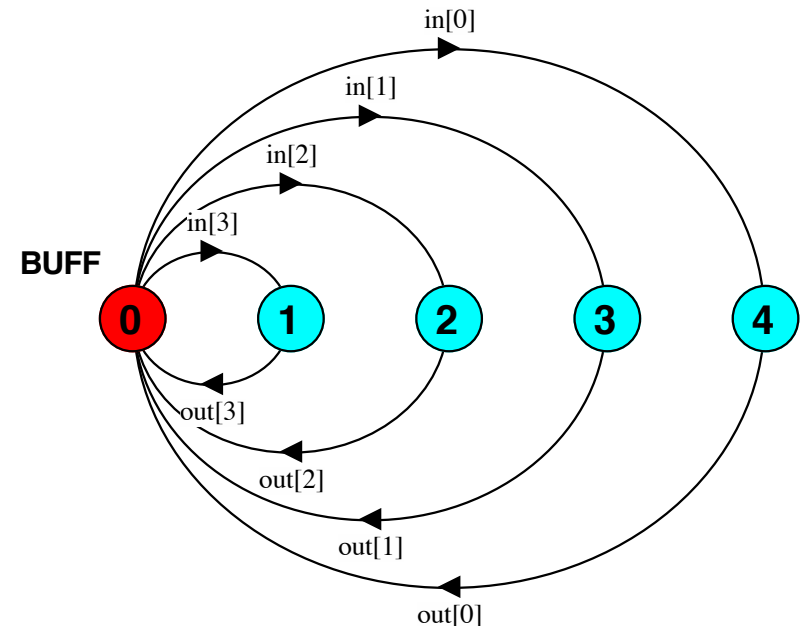


# FSP: Procesos y acciones indexados

Cuando se necesita modelar procesos que tomen un número grande de posibles valores, pueden utilizarse acciones (y procesos) indexadas, donde el rango del índice debe ser finito.

Esta facilidad tiene múltiples usos. En particular, puede emplearse para modelar estado explícito.


  
 $\text{BUFF} = (\text{in}[i:0..3] \rightarrow \text{out}[i] \rightarrow \text{BUFF}).$



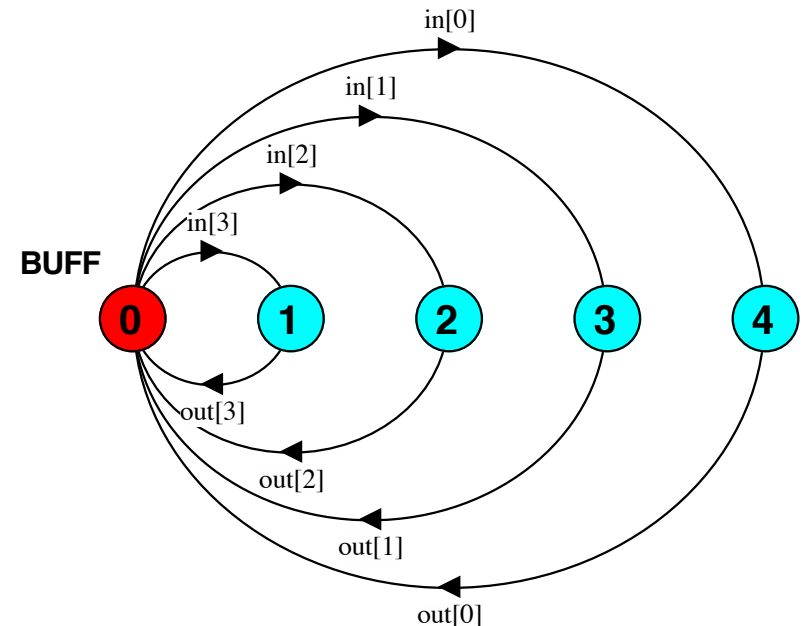
# FSP: Procesos y acciones indexados

Cuando se necesita modelar procesos que tomen un número grande de posibles valores, pueden utilizarse acciones (y procesos) indexadas, donde el rango del índice debe ser finito.

Esta facilidad tiene múltiples usos. En particular, puede emplearse para modelar estado explícito.

  
 $\text{BUFF} = (\text{in}[i:0..3] \rightarrow \text{out}[i] \rightarrow \text{BUFF}).$

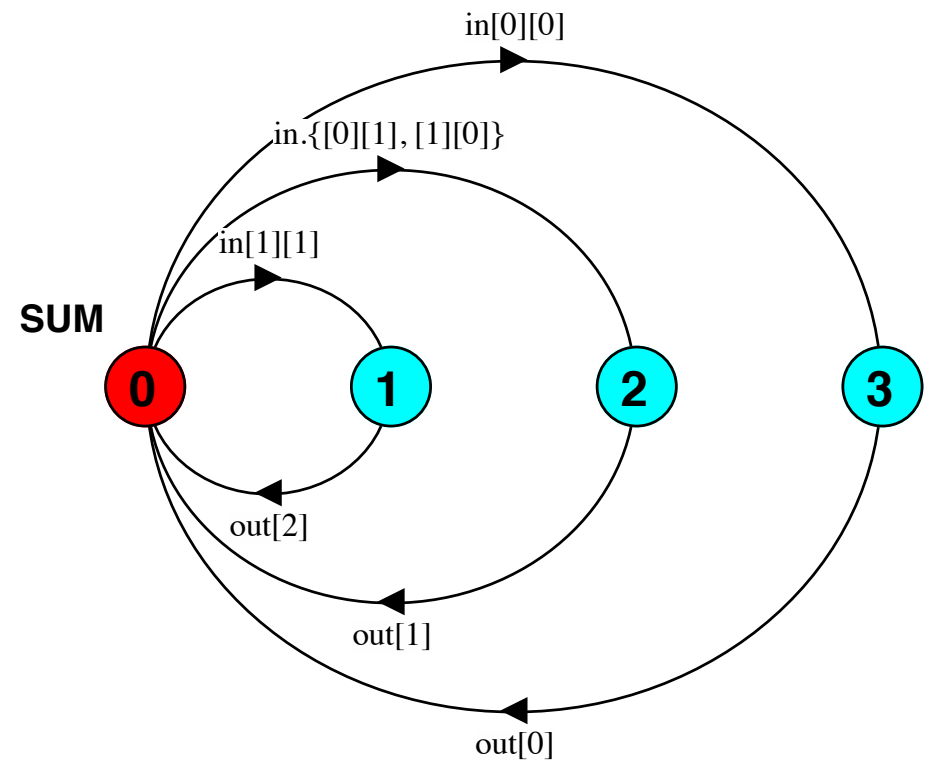
Es equivalente a escribir:

$$\begin{aligned} \text{BUFF} = & (\text{in}[0] \rightarrow \text{out}[0] \rightarrow \text{BUFF} \\ & | \text{in}[1] \rightarrow \text{out}[1] \rightarrow \text{BUFF} \\ & | \text{in}[2] \rightarrow \text{out}[2] \rightarrow \text{BUFF} \\ & | \text{in}[3] \rightarrow \text{out}[3] \rightarrow \text{BUFF}). \end{aligned}$$


# FSP: Procesos y acciones indexados

Ejemplo:

const  $N = 1$   
range  $T = 0..N$   
range  $R = 0..2*N$   
 $SUM = (in[a:T][b:T] \rightarrow TOTAL[a+b]),$   
 $TOTAL[s:R] = (out[s] \rightarrow SUM).$

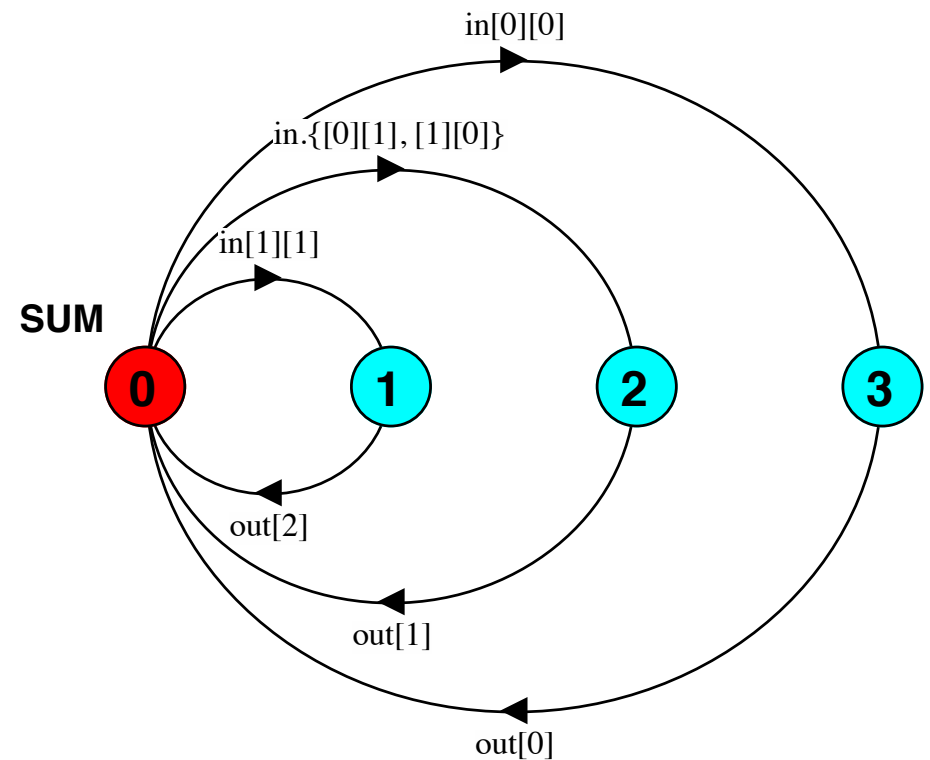


# FSP: Procesos y acciones indexados

Ejemplo:

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).

Intenten con otros  
valores de N

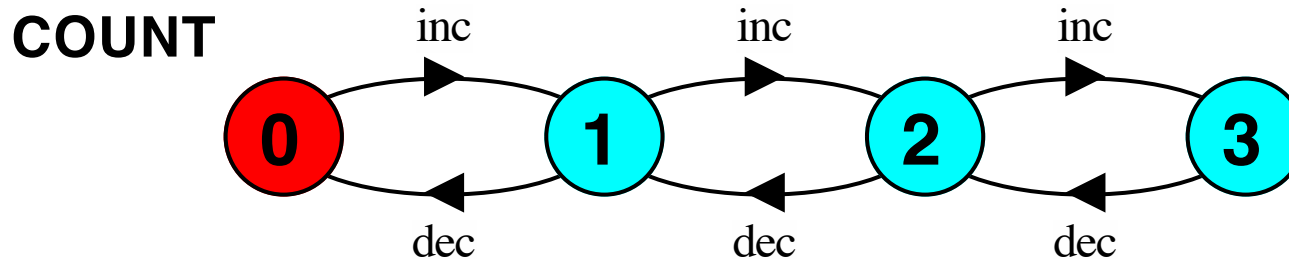




# FSP: Acciones con guardas

Es en general útil contar con acciones que se ejecuten condicionalmente, con respecto al estado de la máquina o sistema modelado. Esto puede expresarse usando la notación “when” en FSP.

Ejemplo:

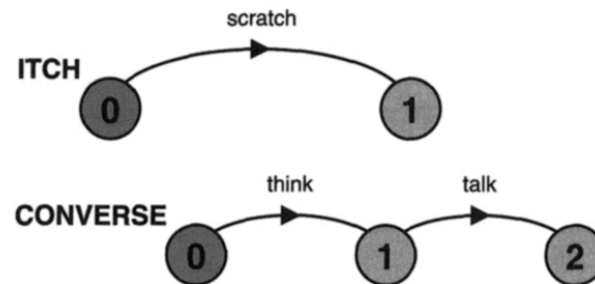
$$\begin{aligned} \text{COUNT } (N=3) &= \text{COUNT}[0], \\ \text{COUNT}[i:0..N] &= ( \text{when}(i < N) \text{ inc} \rightarrow \text{COUNT}[i+1] \\ &\quad | \text{when}(i > 0) \text{ dec} \rightarrow \text{COUNT}[i-1] \\ &\quad ). \end{aligned}$$


# FSP: Composición paralela

Hasta el momento, ninguna de las construcciones vistas nos permite modelar concurrencia. La construcción que nos permite hacer esto, y las más compleja de comprender, es la composición paralela de procesos.

Dados dos procesos P y Q,  $P||Q$  denota la composición paralela de estos procesos.

Ejemplo:



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

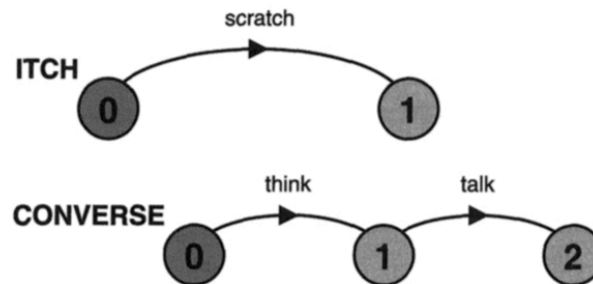
**ITCH = (scratch-> STOP).**

# FSP: Composición paralela

Hasta el momento, ninguna de las construcciones vistas nos permite modelar concurrencia. La construcción que nos permite hacer esto, y las más compleja de comprender, es la composición paralela de procesos.

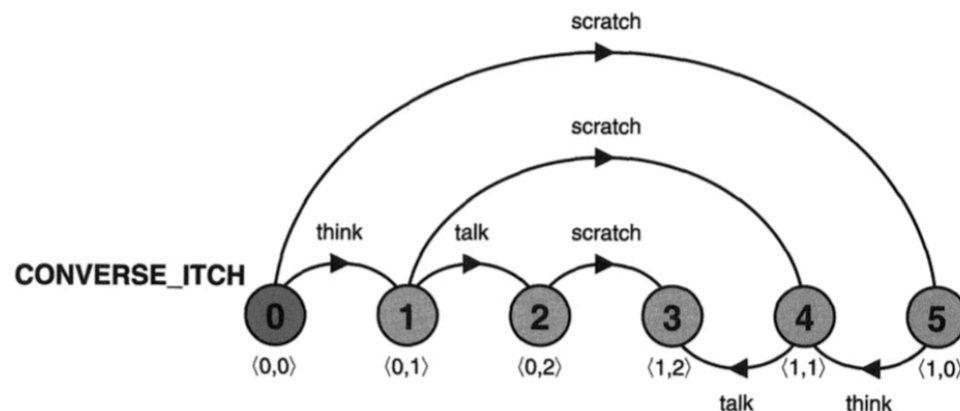
Dados dos procesos P y Q,  $P||Q$  denota la composición paralela de estos procesos.

Ejemplo:



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

**ITCH = (scratch-> STOP).**



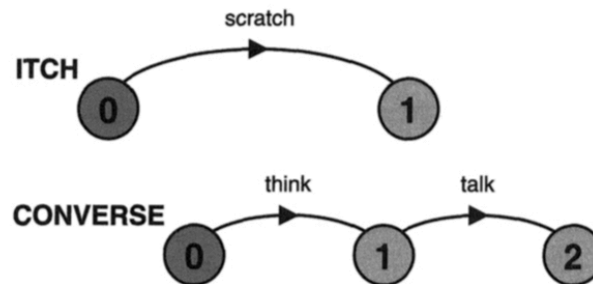
**$||\text{CONVERSE\_ITCH} = (\text{ITCH} || \text{CONVERSE}).$**

# FSP: Composición paralela

Hasta el momento, ninguna de las construcciones vistas nos permite modelar concurrencia. La construcción que nos permite hacer esto, y las más compleja de comprender, es la composición paralela de procesos.

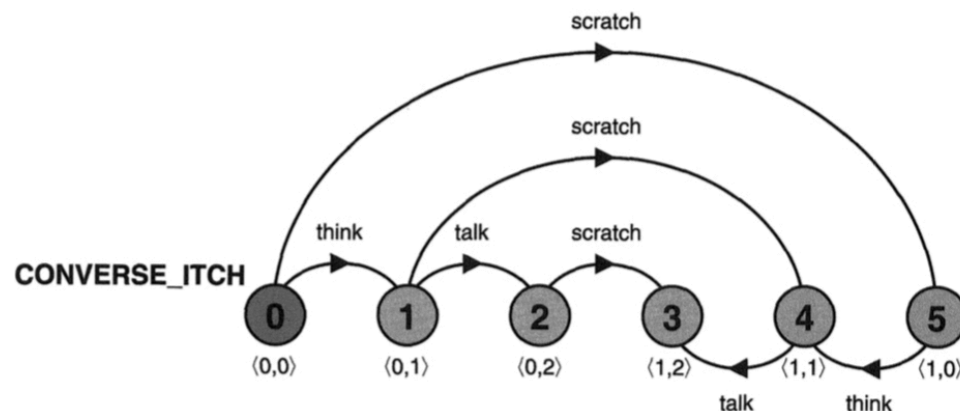
Dados dos procesos P y Q,  $P||Q$  denota la composición paralela de estos procesos.

Ejemplo:



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

**ITCH = (scratch-> STOP).**



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

# FSP: Composición paralela

Hay varios puntos importantes con respecto a la composición paralela en FSP:

La sincronización se realiza en las acciones comunes (y puede involucrar a más de dos procesos).

No se puede identificar explícitamente al proceso “activo” y al proceso “pasivo” en la sincronización de acciones comunes. Esta interpretación corre por cuenta del diseñador

El modelo de concurrencia es interleaving, donde las acciones atómicas independientes de diferentes procesos pueden ejecutarse en interleavings arbitrarios.

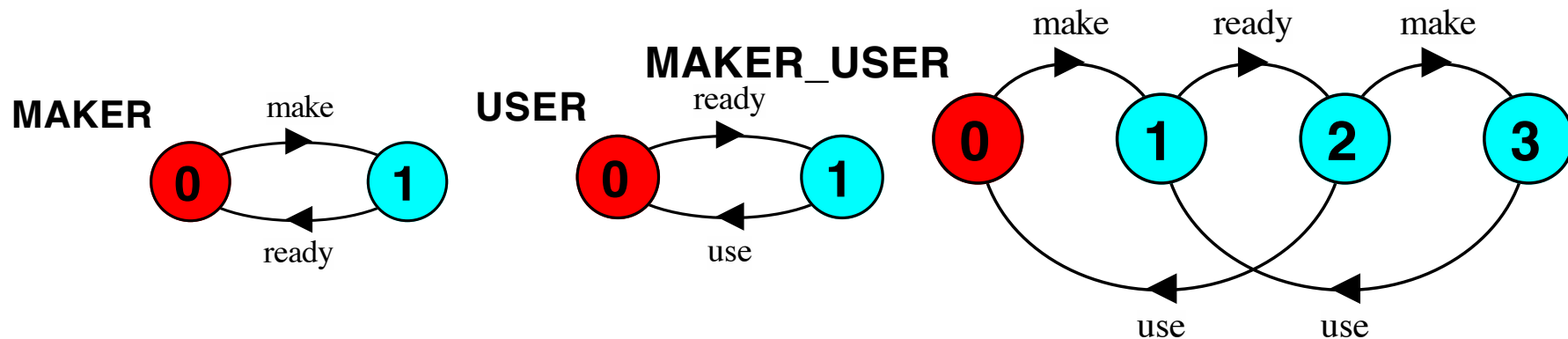
# FSP: Composición paralela

Ejemplo:

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

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

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



# FSP: Composición paralela

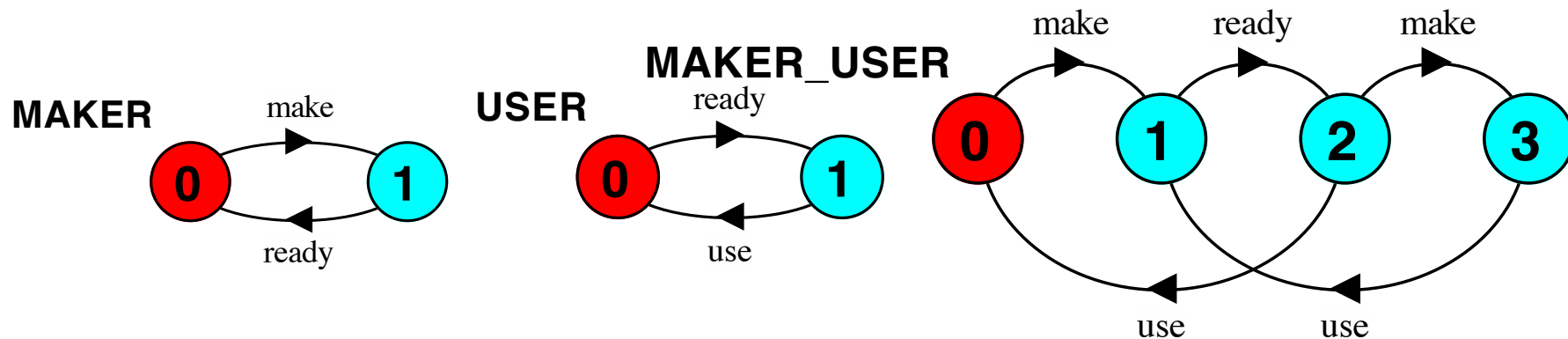
Ejemplo:

**ready** debe ser ejecutada al mismo tiempo por los dos procesos

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

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

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



# FSP: Otros conceptos y operaciones

**Alfabeto de un proceso:** Conjunto de acciones al cual este proceso puede involucrar. Es importante para saber como se sincronizan los procesos.

**Etiquetado de procesos:** Utilizado para generar diferentes copias de un mismo proceso.



# Etiquetado de Procesos

Dada la definición de un proceso muchas veces queremos más de una copia de un proceso en un programa o un modelo de un sistema:

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

# Etiquetado de Procesos

Dada la definición de un proceso muchas veces queremos más de una copia de un proceso en un programa o un modelo de un sistema:

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

# Etiquetado de Procesos

Dada la definición de un proceso muchas veces queremos más de una copia de un proceso en un programa o un modelo de un sistema:

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

$(\text{SWITCH} \parallel \text{SWITCH}).$

Procesos sincronizan en las acciones compartidas.  
Indistinguible de SWITCH

# Etiquetado de Procesos

Dada la definición de un proceso muchas veces queremos más de una copia de un proceso en un programa o un modelo de un sistema:

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

~~(SWITCH || SWITCH).~~

Procesos sincronizan en las acciones compartidas.  
Indistinguible de SWITCH

# Etiquetado de Procesos

Dada la definición de un proceso muchas veces queremos más de una copia de un proceso en un programa o un modelo de un sistema:

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

~~$(\text{SWITCH} \parallel \text{SWITCH}).$~~

Procesos sincronizan en las acciones compartidas.  
Indistinguible de SWITCH

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

alfabeto = {a.on, a.off}

alfabeto = {b.on, b.off}

# Etiquetado de Procesos

Dada la definición de un proceso muchas veces queremos más de una copia de un proceso en un programa o un modelo de un sistema:

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

~~$(\text{SWITCH} \parallel \text{SWITCH}).$~~

Procesos sincronizan en las acciones compartidas.  
Indistinguible de SWITCH

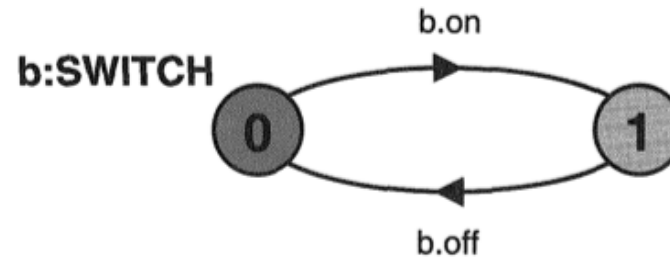
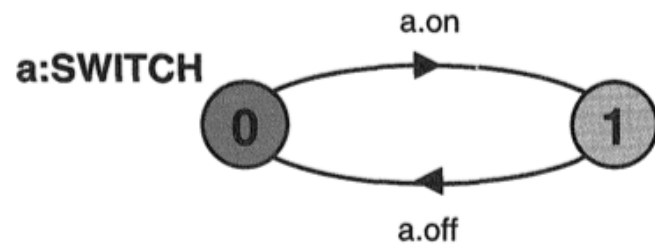
$\parallel\text{TWO\_SWITCH} = (a:\text{SWITCH} \parallel b:\text{SWITCH}).$

Alfabetos  
disjuntos!

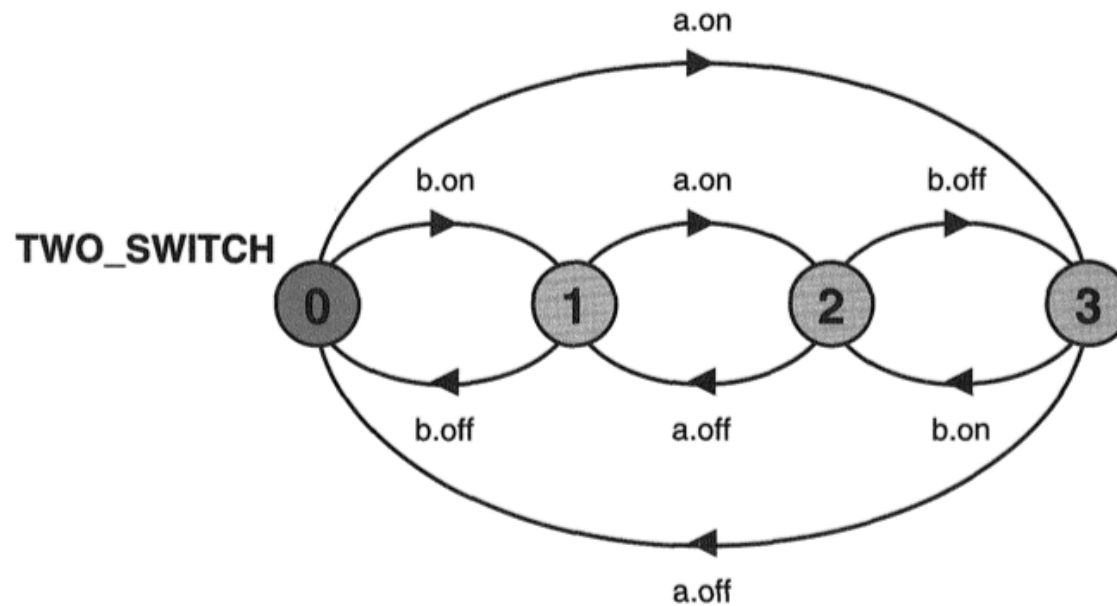
alfabeto = {a.on, a.off}

alfabeto = {b.on, b.off}

# Etiquetado de Procesos



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



# FSP: Otros conceptos y operaciones

**Alfabeto de un proceso:** Conjunto de acciones al cual este proceso puede involucrar. Es importante para saber como se sincronizan los procesos.

**Etiquetado de procesos:** Utilizado para generar diferentes copias de un mismo proceso.

**Reetiquetado:** Utilizado usualmente para asegurar que procesos compuestos sincronicen en las acciones deseadas.



# Reetiquetados de Procesos

Utilizado usualmente para asegurar que procesos compuestos sincronicen en las acciones deseadas.

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

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

Podemos asociar la acción call del CLIENT con la acción request del SERVER

$\parallel$ CLIENT\_SERVER= (CLIENT  $\parallel$  SERVER)/**{call/request, reply/wait}**.

# Reetiquetados de Procesos

Utilizado usualmente para asegurar que procesos compuestos sincronicen en las acciones deseadas.

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

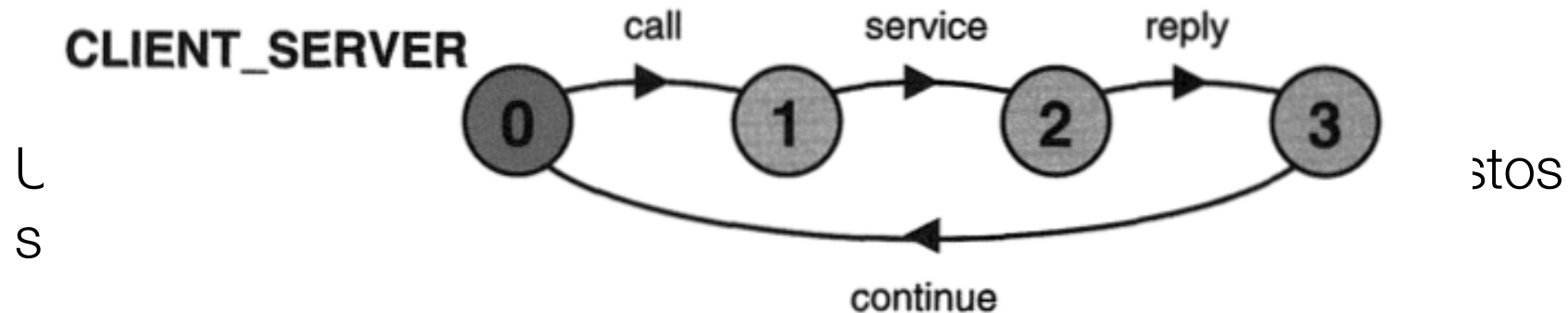
SERVER=(request->service->reply->SERVER).

Podemos asociar la acción call del CLIENT con la acción request del SERVER

||CLIENT\_SERVER= (CLIENT || SERVER)/**{call/request, reply/wait}**.

la etiqueta **call** reemplaza la etiqueta **request** y  
**reply** reemplaza a **wait**

# Reetiquetados de Procesos



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

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

Podemos asociar la acción call del CLIENT con la acción request del SERVER

$\parallel$ CLIENT\_SERVER= (CLIENT  $\parallel$  SERVER)/{**call/request**, **reply/wait**}.

la etiqueta **call** reemplaza la etiqueta **request** y  
**reply** reemplaza a **wait**

# FSP: Otros conceptos y operaciones

**Alfabeto de un proceso:** Conjunto de acciones al cual este proceso puede involucrar. Es importante para saber como se sincronizan los procesos.

**Etiquetado de procesos:** Utilizado para generar diferentes copias de un mismo proceso.

**Reetiquetado:** Utilizado usualmente para asegurar que procesos compuestos sincronicen en las acciones deseadas.

**Ocultamiento:** Las acciones ocultas no pueden compartirse con otros procesos (desaparecen del alfabeto del proceso). Utiliza una acción distinguida **tau** (denominada acción sigilosa o invisible)

# Ocultamiento de Acciones

Las acciones ocultas no pueden compartirse con otros procesos (desaparecen del alfabeto del proceso). Utiliza una acción distinguida **tau** (denominada acción sigilosa o invisible) que no tiene permitido sincronizar.

Esencial para reducir la complejidad de sistemas grandes durante el análisis.

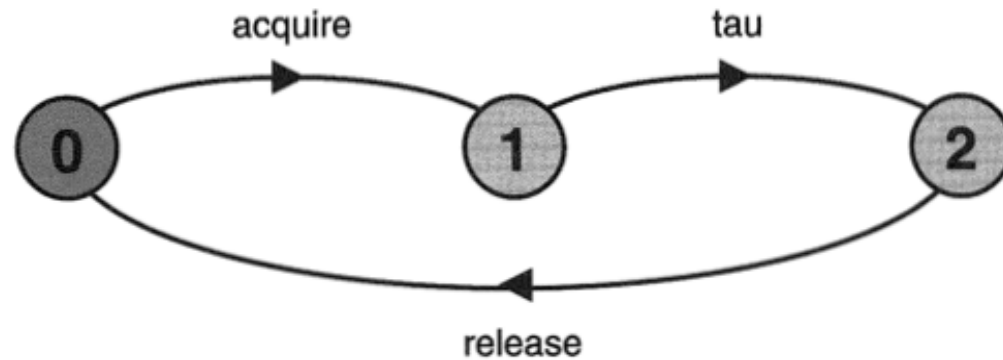
$$\text{USER} = (\text{acquire} \rightarrow \text{use} \rightarrow \text{release} \rightarrow \text{USER}) \backslash \{\mathbf{use}\}.$$

Oculto la acción **use**

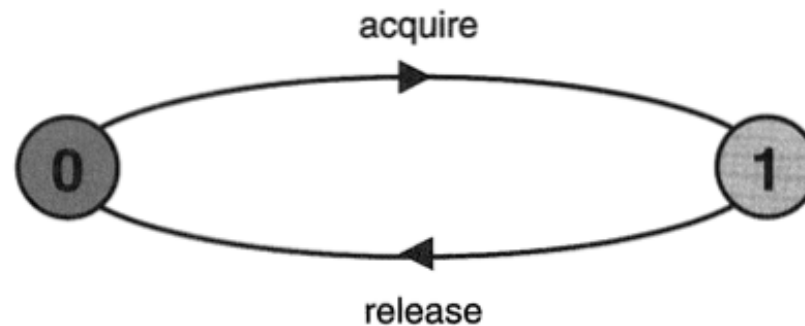
$$\text{USER} = (\text{acquire} \rightarrow \text{use} \rightarrow \text{release} \rightarrow \text{USER}) @ \{\mathbf{acquire, release}\}.$$

Oculto las acción que no están en este conjunto

# Ocultamiento de Acciones



minimizado:



# Bibliografia

capítulos 1, 2 y 3 de Concurrency, State Models and  
Java Programs, Magee and Kramer 1999

Download LTSA: [http://countingfluents.weebly.com/  
download.html](http://countingfluents.weebly.com/download.html)