

Ingeniería del Software II

2 - Programación concurrente

Programación concurrente

• ¿Qué?

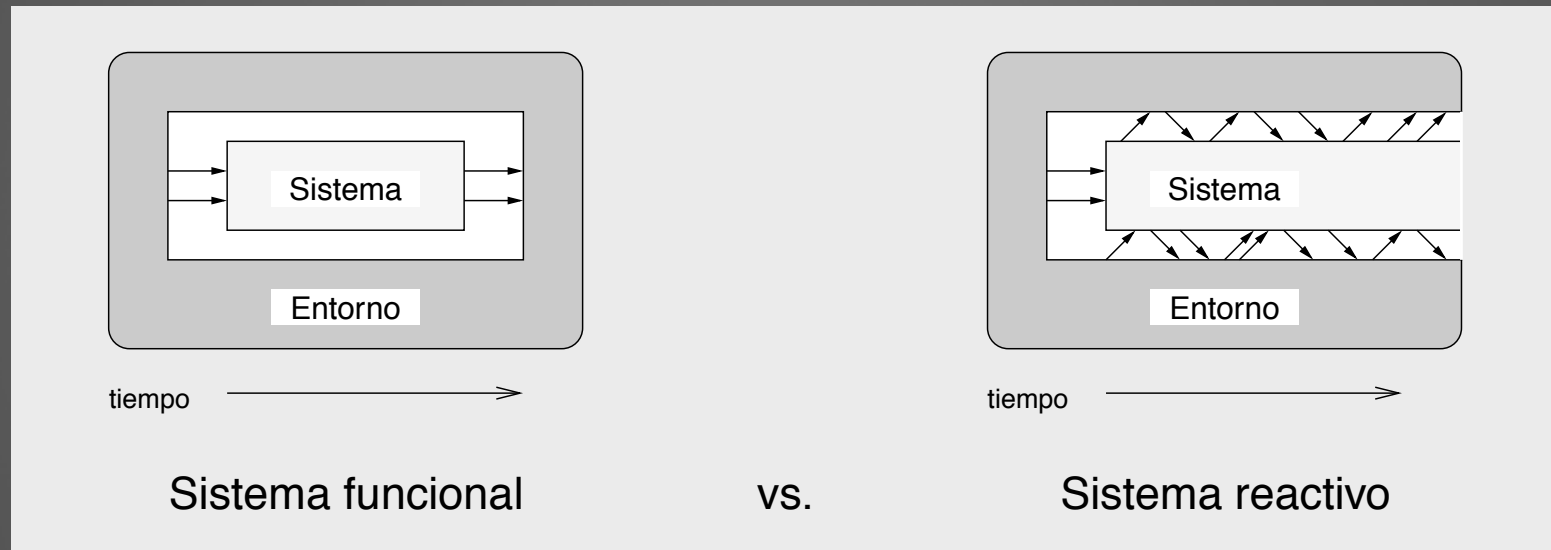
- Programación de sistemas compuestos de varios procesos que se ejecutan de manera superpuesta en un período de tiempo e interactúan entre sí.

• ¿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 (ej: múltiples procesos en un desktop)
- Mejorar velocidad de cálculo
- Mejorar uso de recursos
- Comunicación
- ...

Características reactivas de los programas concurrentes

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



Los sistemas reactivos tienen características diferentes a las de los programas convencionales. En muchos casos, éstos no computan resultados, y suele no requerirse que terminen.

Ejemplos: sistemas operativos, software de control, hardware, 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

- 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
- Livelock: Dos o más procesos no pueden avanzar en su ejecución porque continuamente responden a los cambios en el estado de otros procesos

Concurrencia: un ejemplo

```
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
}
```

¿Qué hace este programa?

Concurrencia: un ejemplo

Se bloquea
hasta que la aserción
se haga verdadera

```
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
}
```

¿Qué hace este programa?

¿Garantiza
exclusión mutua?

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.
- un nodo distinguido que reconoceremos como el **estado inicial**.

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:

(S, s_0, \rightarrow)

↓
Relación de
transición

↓
Estado inicial

↓
Conjunto de estados

Ejemplo

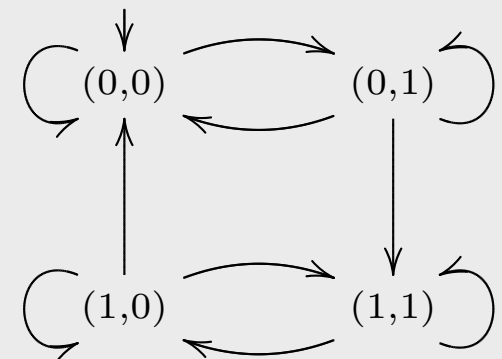
$$S = \{0, 1\} \times \{0, 1\}$$

$$s_0 = (0, 0)$$

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

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

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

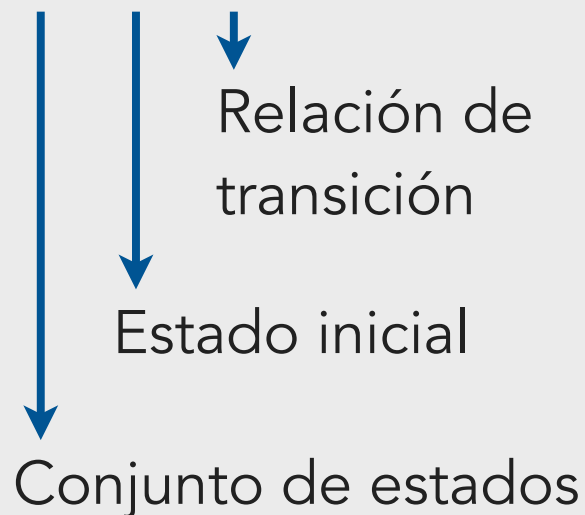


Semántica de programas concurrentes

Una semántica típica para programas concurrentes está basada en **sistema 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.

(S, s_0, \rightarrow)



Ejemplo

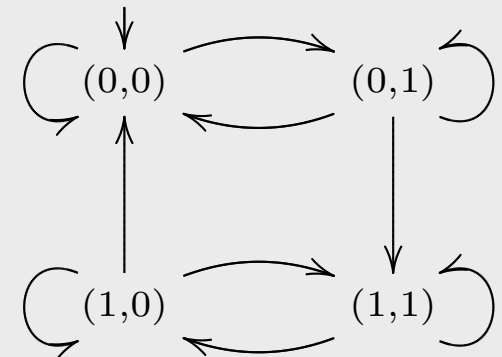
$$S = \{0, 1\} \times \{0, 1\}$$

$$s_0 = (0, 0)$$

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

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

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



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.



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



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

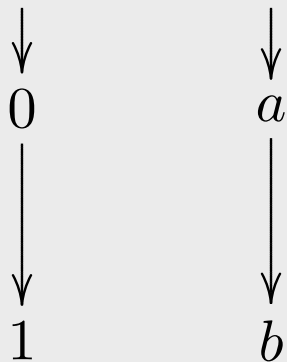


$(0, 0)(0, 1)(1, 0)(1, 1)(0, 0)(1, 1)(1, 0)(1, 0)(1, 1)(1, 0)(0, 0)(1, 1)(0, 1)(1, 1) \cdots$

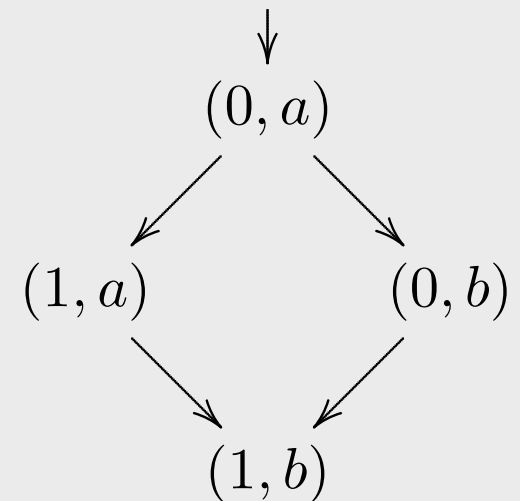
¿Cómo se ejecutan los procesos concurrentes?

De acuerdo al modelo computacional descrito, 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.



Volviendo al ejemplo...

Volviendo a nuestro ejemplo anterior, 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 decrementa 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
}

```

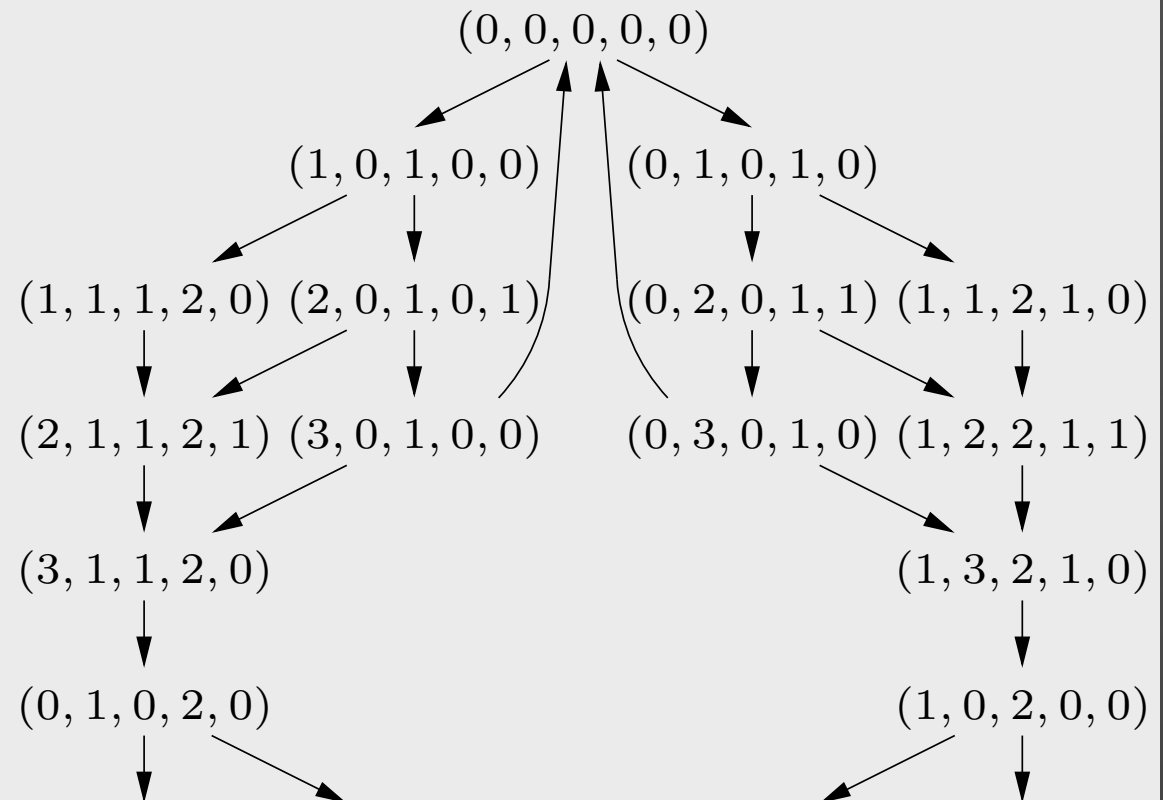
Estructura del estado:

$(pc_1, pc_2, y1, y2, in_critical)$

```

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

```



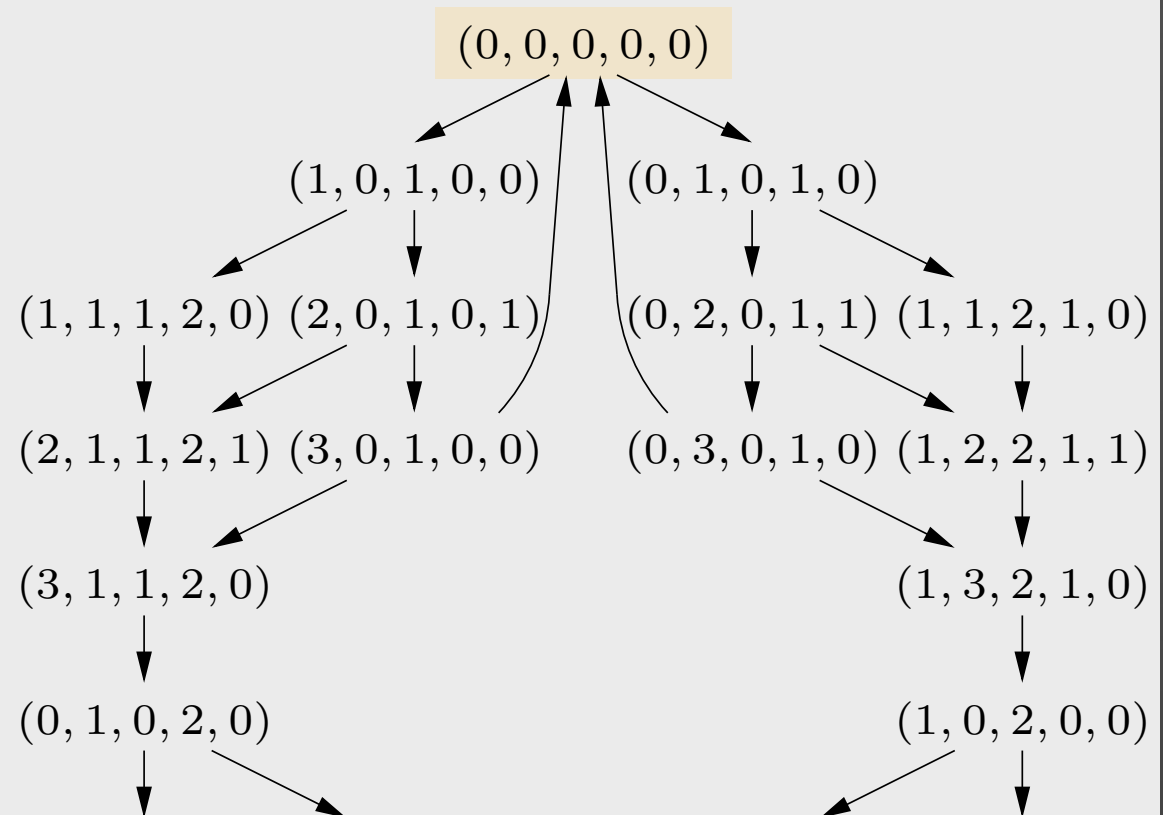
```
int y1 = 0;
int y2 = 0;
short in_critical = 0;

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

Estructura del estado:

$$(pc_1, pc_2, y1, y2, \text{in_critical})$$

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



```

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
}

```

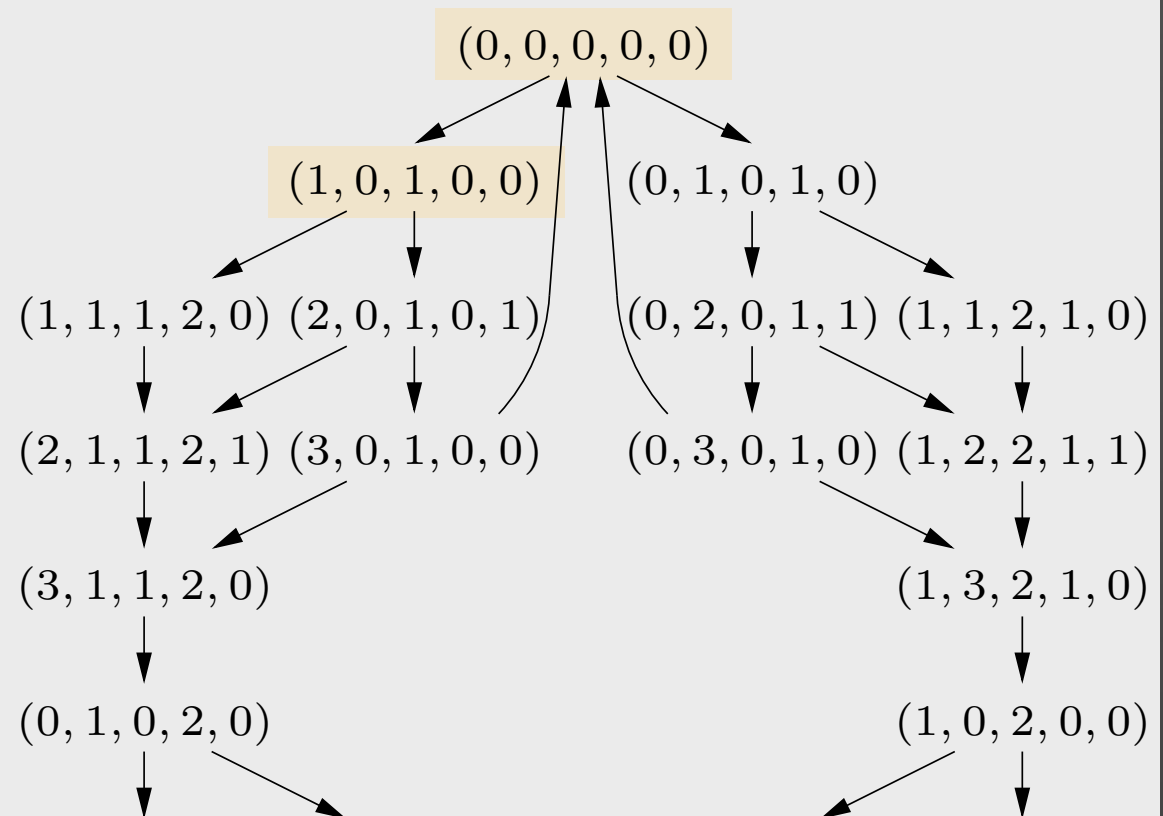
Estructura del estado:

$(pc_1, pc_2, y1, y2, in_critical)$

```

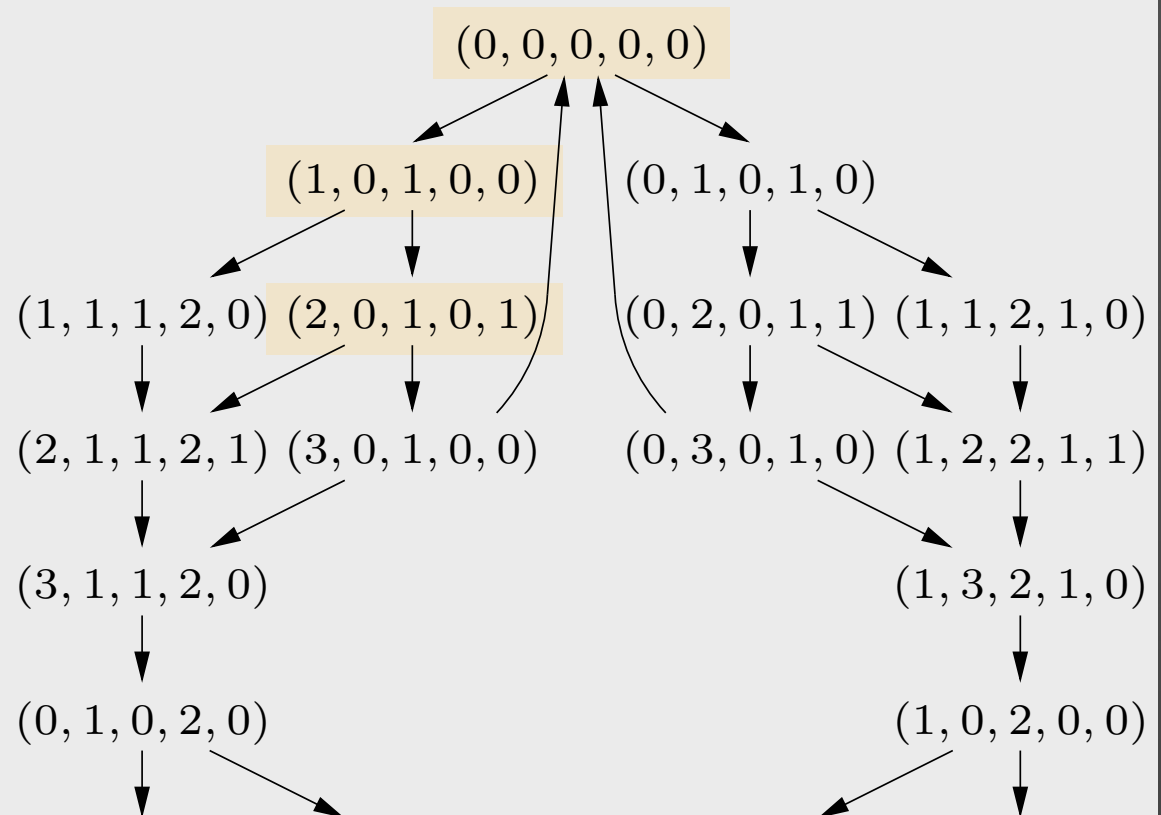
active proctype process_2() {
    do
        :: true ->
0:      y2 = y1+1;
1:      ((y1==0) || (y2<y1));
        in_critical++;
2:      in_critical--;
3:      y2 = 0;
    od
}

```




```
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 ->
            :      y2 = y1+1;
            :      ((y1==0) || (y2<y1));
            :      in_critical++;
            :      in_critical--;
            :      y2 = 0;
    od
}
```

$$(pc_1, pc_2, y1, y2, in_critical)$$


```

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
}

```

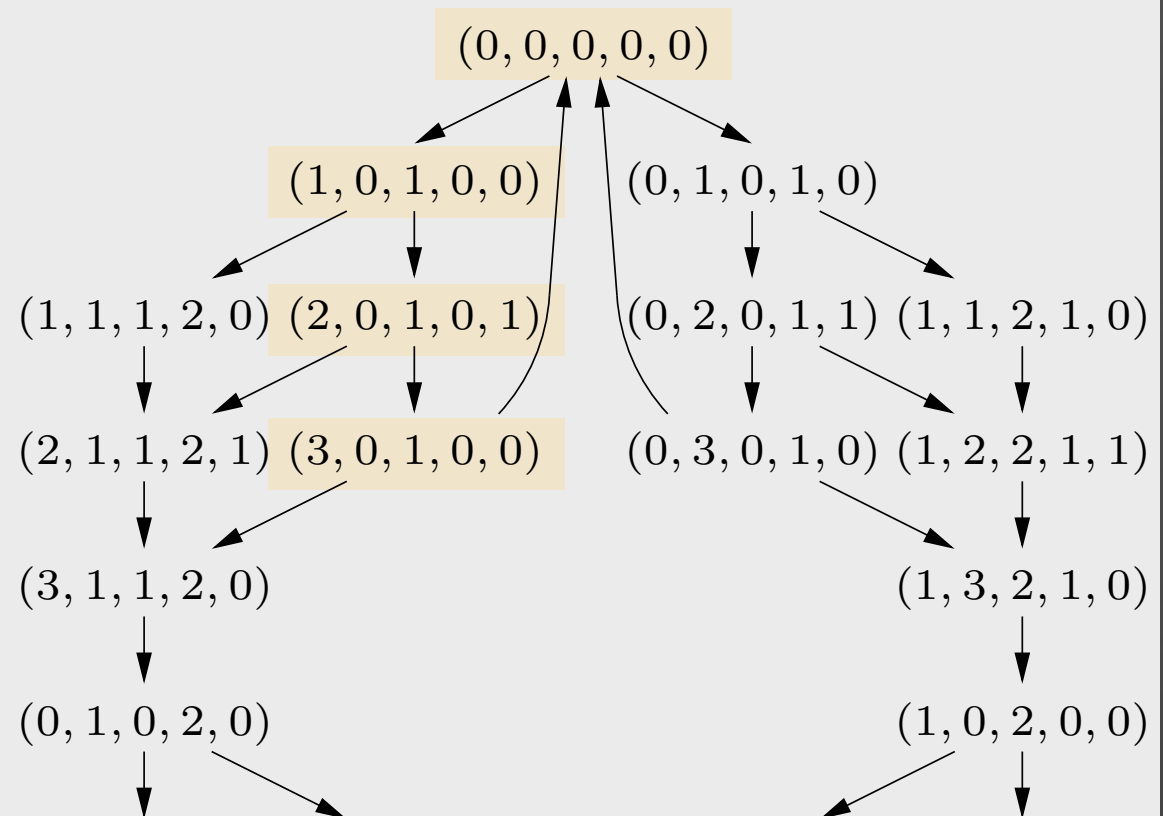
Estructura del estado:

$(pc_1, pc_2, y1, y2, in_critical)$

```

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

```



```

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
}

```

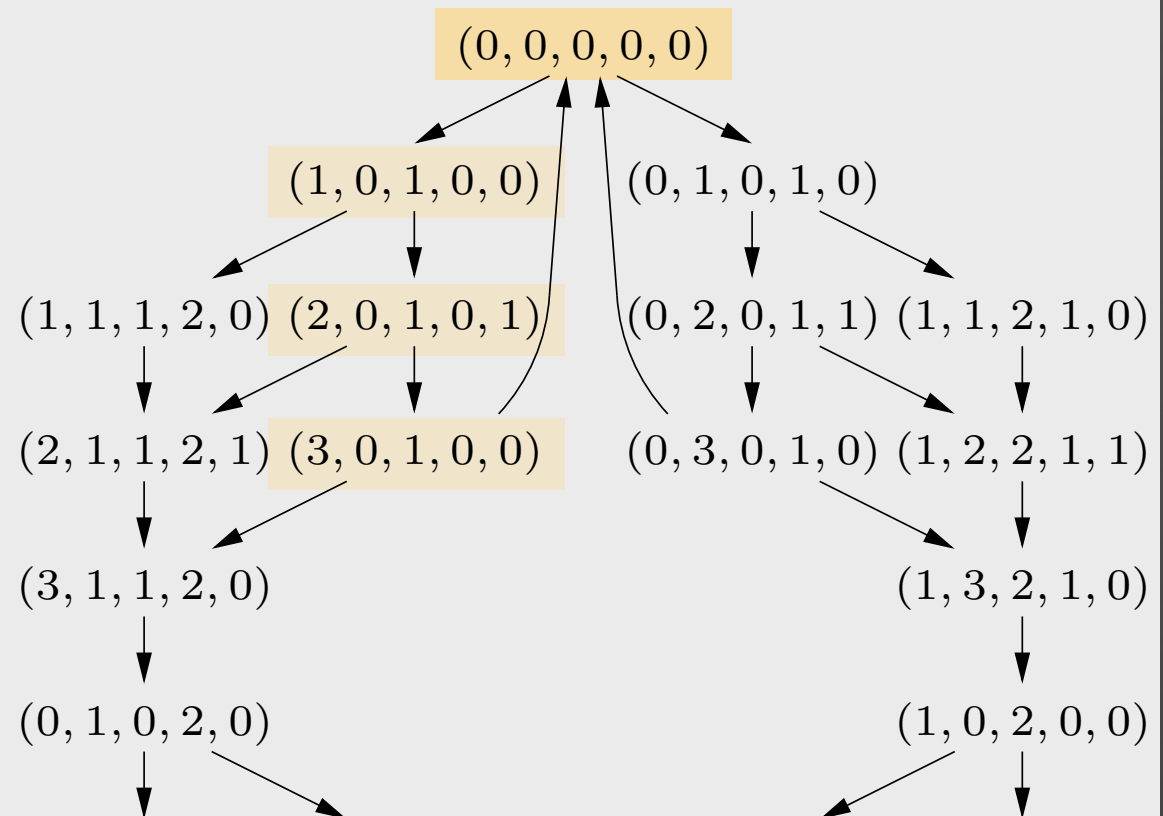
Estructura del estado:

$(pc_1, pc_2, y1, y2, in_critical)$

```

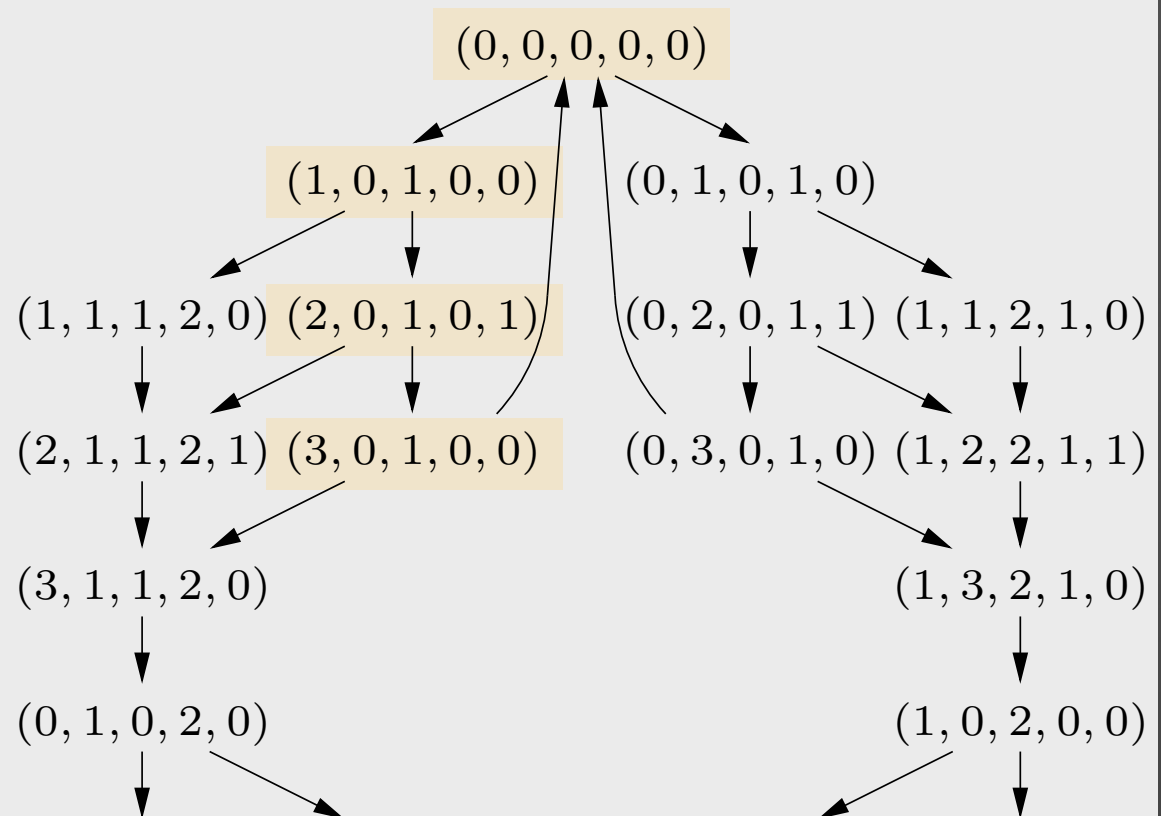
active proctype process_2() {
    do
        :: true ->
0:      y2 = y1+1;
1:      ((y1==0) || (y2<y1));
        in_critical++;
2:      in_critical--;
3:      y2 = 0;
    od
}

```



```
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 ->
:       y2 = y1+1;
:       ((y1==0) || (y2<y1));
        in_critical++;
:       in_critical--;
:       y2 = 0;
    od
}
```

$$(pc_1, pc_2, y1, y2, \text{in_critical})$$


```

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
}

```

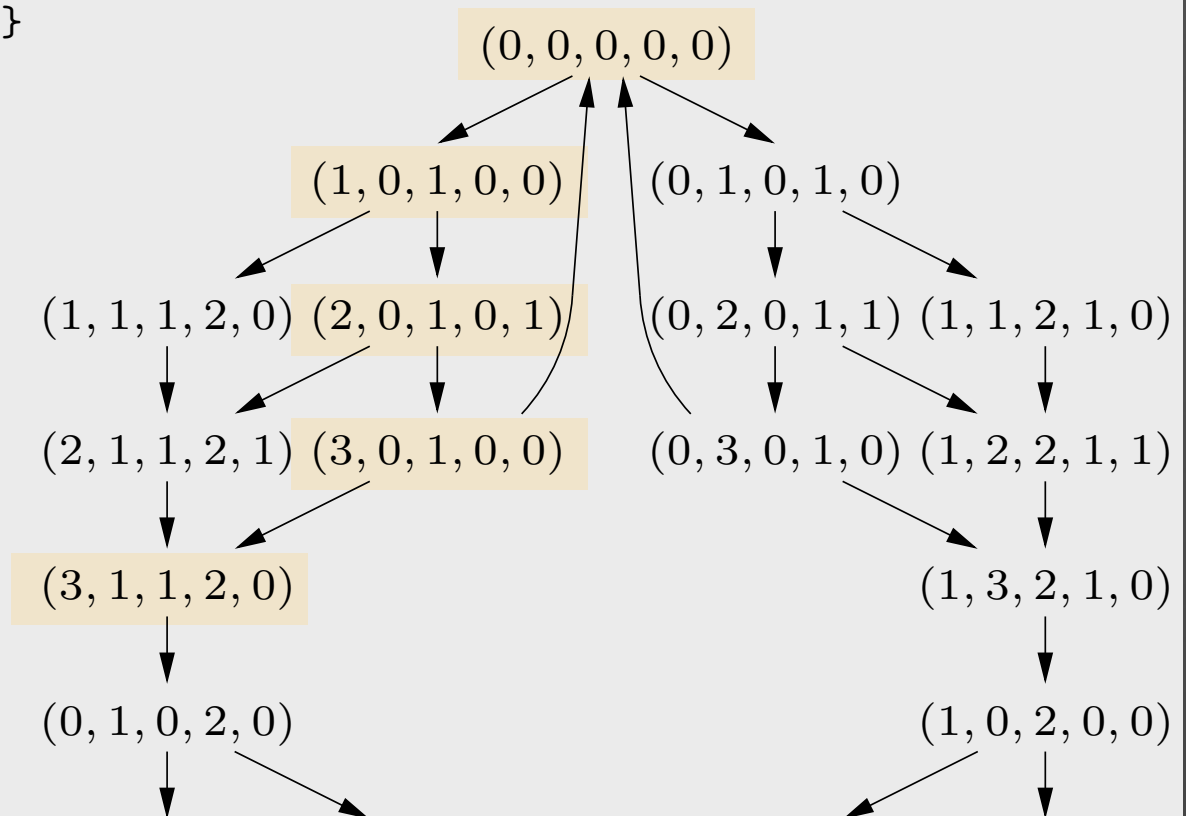
```

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:

$(pc_1, pc_2, y1, y2, in_critical)$




```

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
}

```

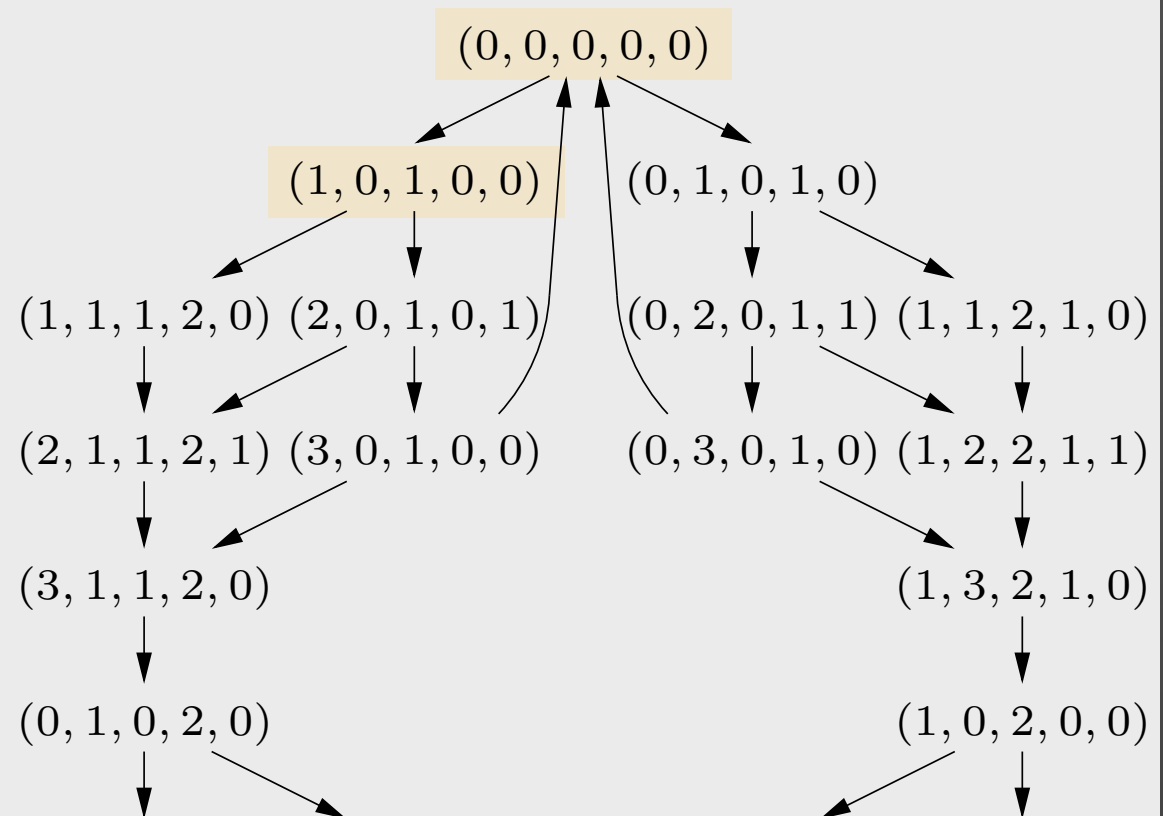
Estructura del estado:

$(pc_1, pc_2, y1, y2, in_critical)$

```

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

```



```

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
}

```

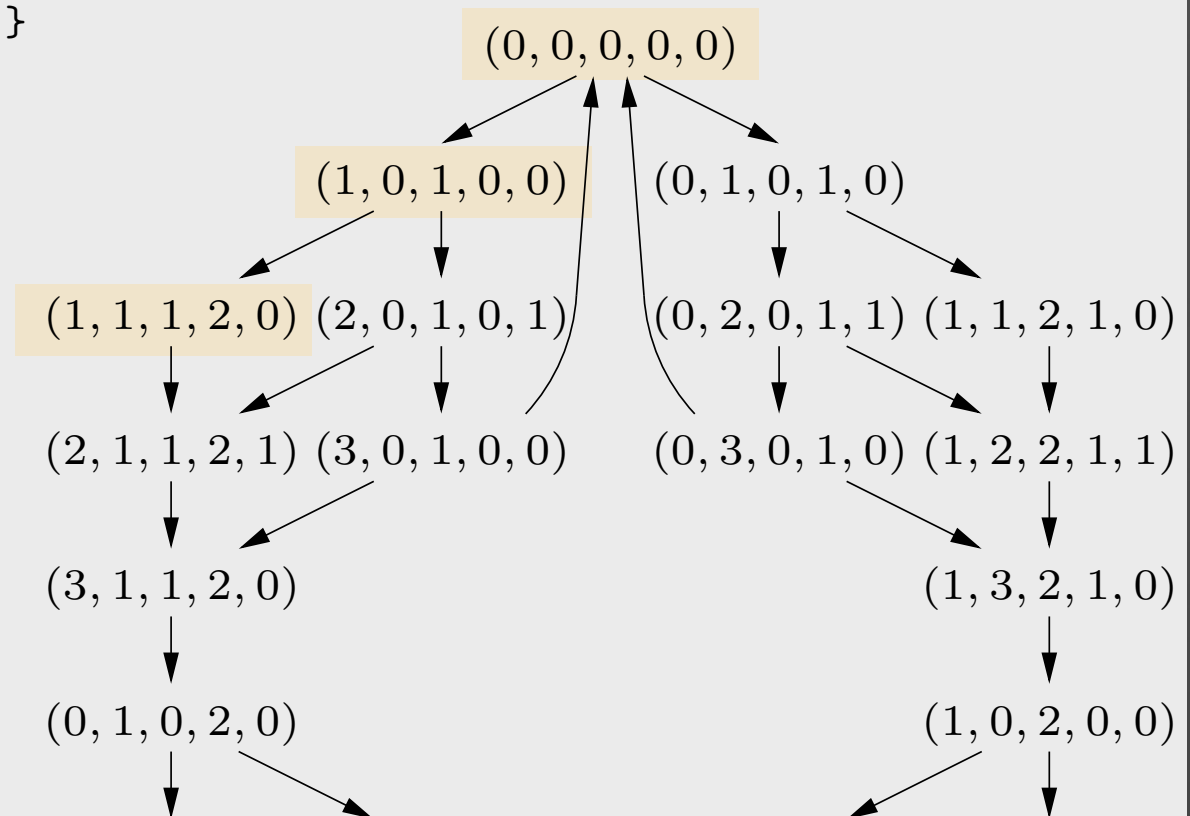
Estructura del estado:

$(pc_1, pc_2, y1, y2, in_critical)$

```

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

```



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.

Además, los tests
no pueden controlar el
no determinismo

Una de las razones es que diferentes interleavings de acciones atómicas producen diferentes resultados o comportamientos de los sistemas concurrentes.

El número de interleavings posible es muy grande, lo que hace que el testing pierda confianza de que nuestros programas funcionan correctamente.

El espacio de estados
crece exponencialmente con
el número de componentes y
variables.

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 en la primera parte de la asignatura es FSP (Finite State Processes). FSP es una variante simple de CSP, que incluye, entre otras cosas:

- Prefijos de acciones:

$x \rightarrow P$

Evento

Proceso

El lenguaje FSP

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

- Prefijos de acciones:

$x \rightarrow P$

Construyen un
nuevo proceso

El lenguaje FSP

El lenguaje que utilizaremos en la primera parte de la asignatura 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:

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

- Elección no determinista:

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

¿Cuál es la diferencia?

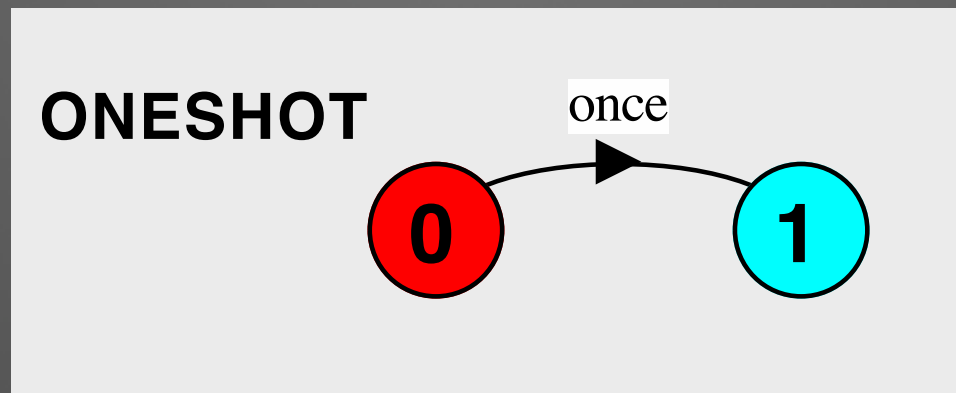
Semántica de procesos

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` \rightarrow **STOP**.

puede visualizarse como:



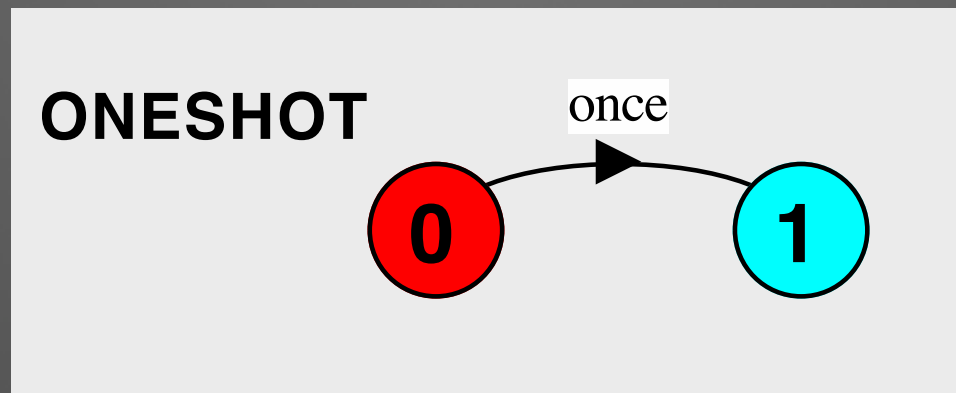
Semántica de procesos

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*

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.

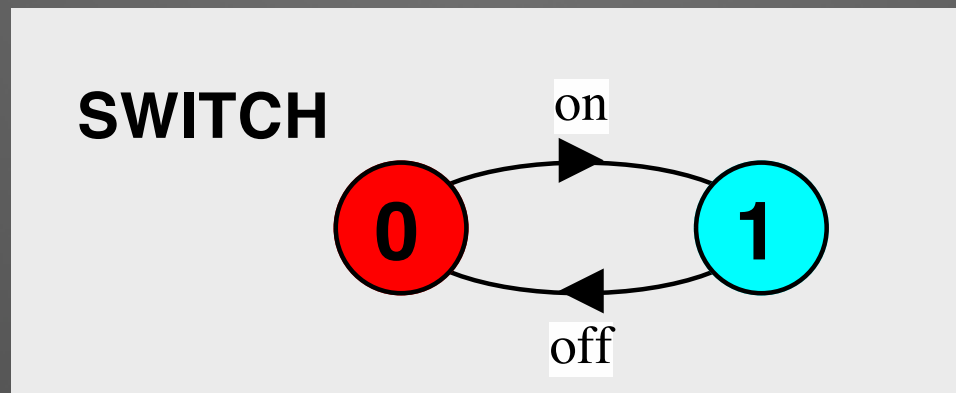
FSP: Recursión

El comportamiento de un proceso también puede definirse usando recursión, donde tenemos algunas restricciones sintácticas:

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!).

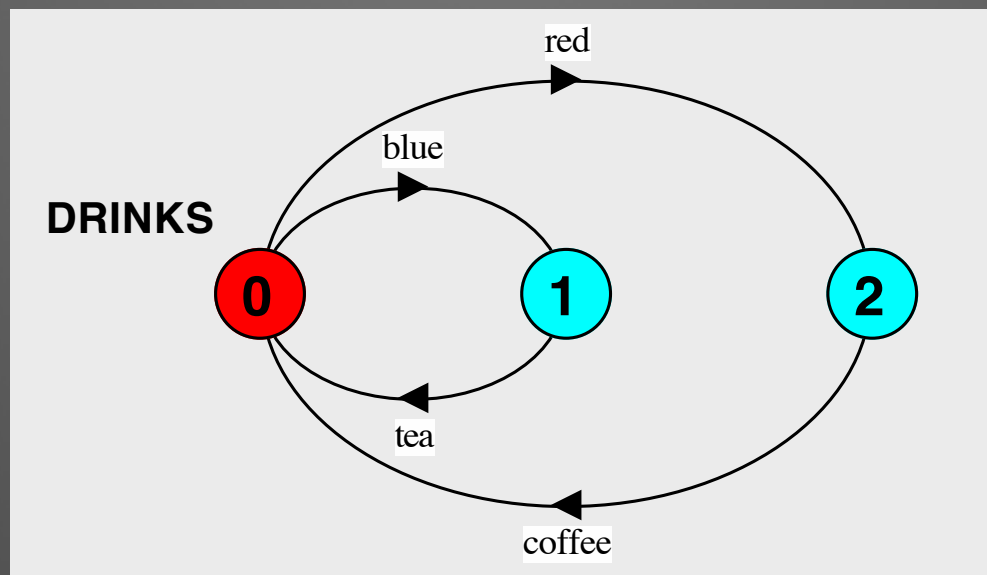


FSP: Elección

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

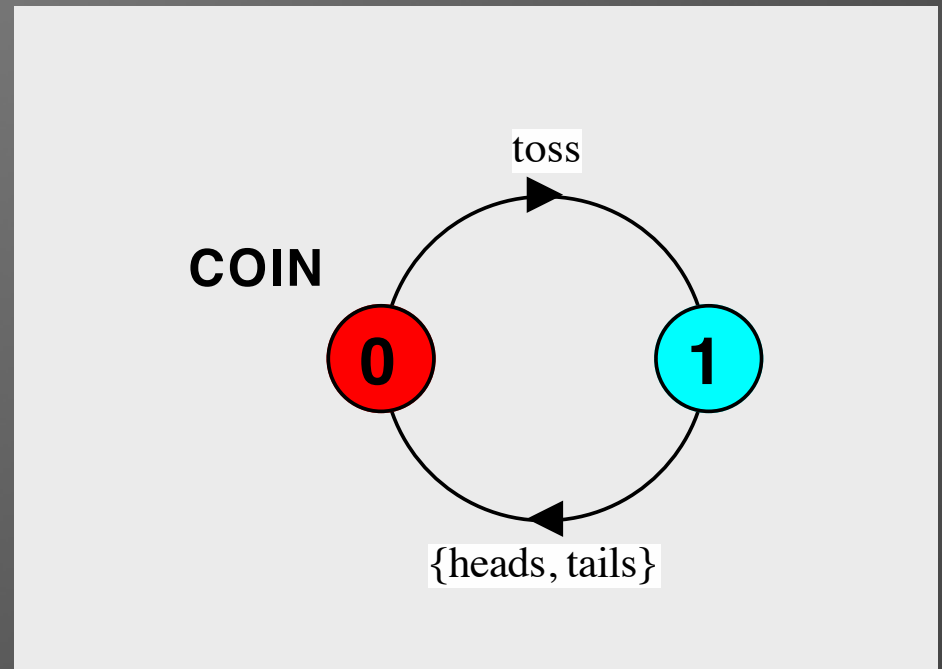
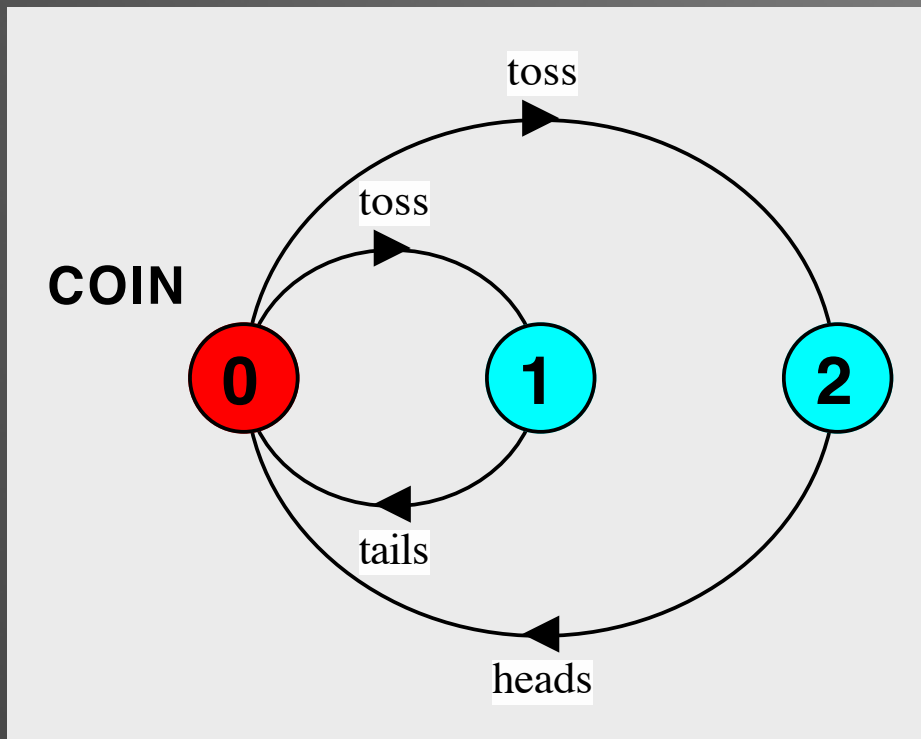
Es simplemente un caso particular de elección.

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.

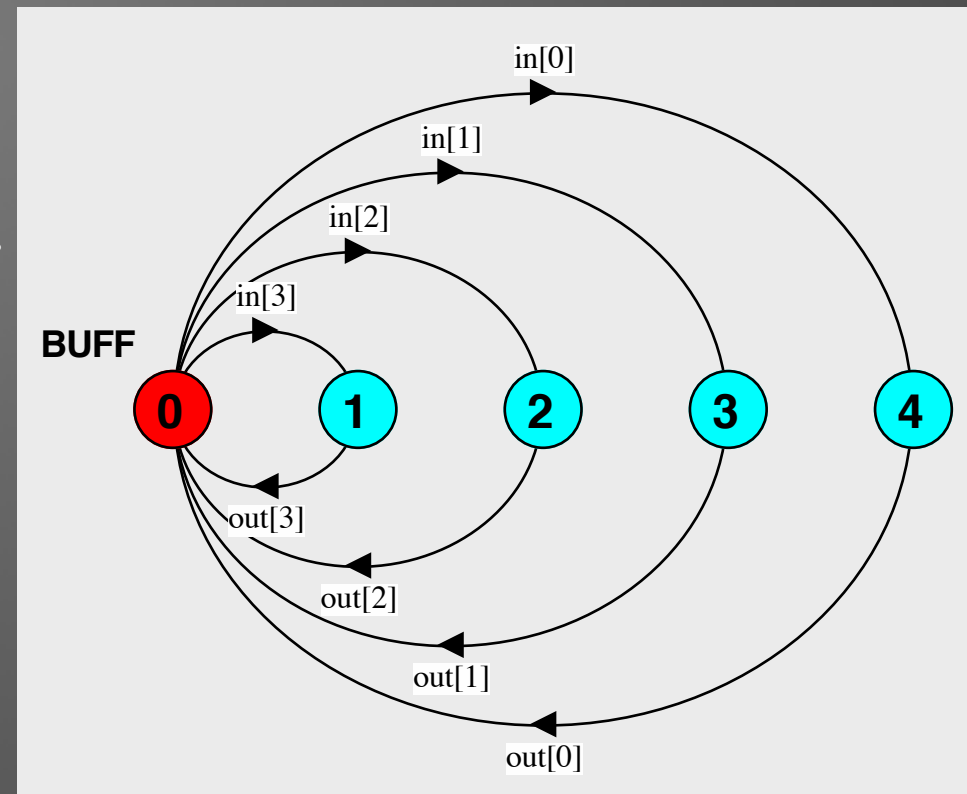
Esta facilidad tiene múltiples usos. En particular, puede emplearse para modelar estado explícito.

Ejemplo:

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

Es equivalente a escribir:

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



FSP: Procesos y acciones indexados

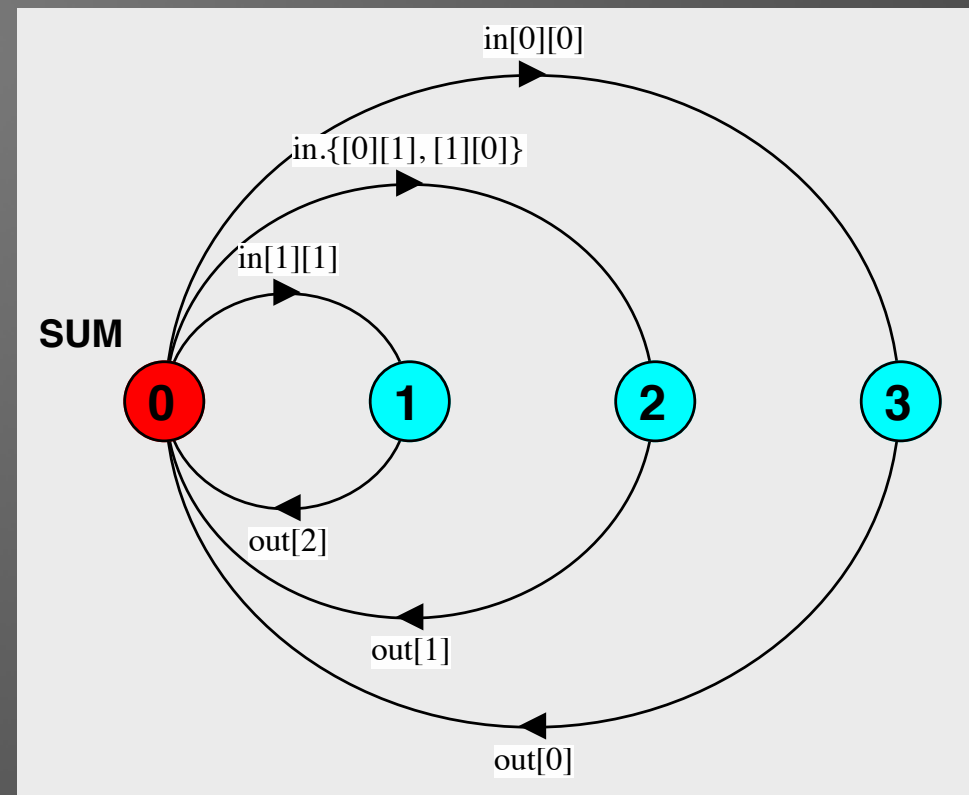
Cuando se necesita modelar procesos que tomen un número grande de posibles valores, pueden utilizarse acciones (y procesos) indexadas, donde el rango del índice debe ser finito.

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

Ejemplo:

Intenten con otros valores de N

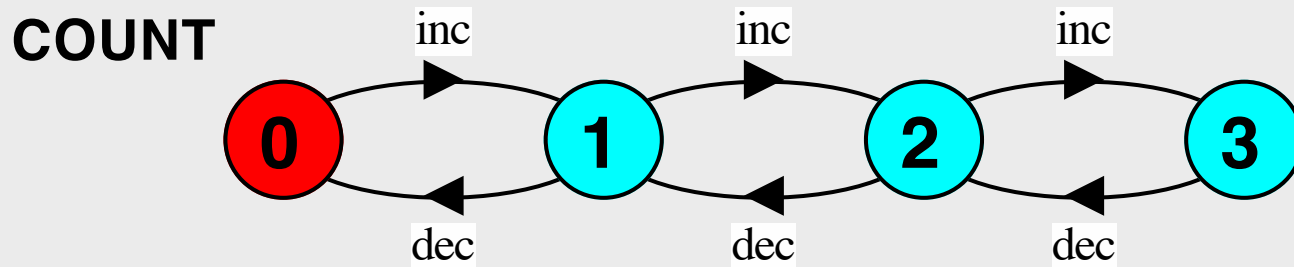
```
const N = 1
range T = 0..N
range R = 0..2*N
SUM = (in[a:T][b:T] -> TOTAL[a+b]),
TOTAL[s:R] = (out[s] -> SUM).
```



FSP: Acciones con guardas

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

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

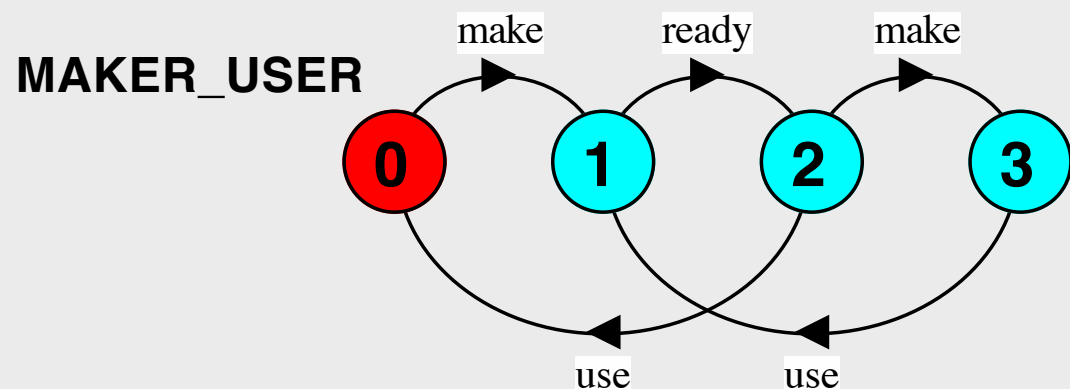
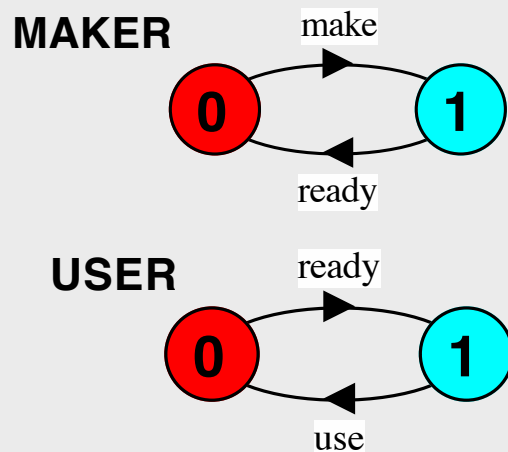


FSP: Composición paralela

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

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

Ejemplo: $\text{MAKER} = (\text{make} \rightarrow \text{ready} \rightarrow \text{MAKER}).$
 $\text{USER} = (\text{ready} \rightarrow \text{use} \rightarrow \text{USER}).$
 $\parallel \text{MAKER_USER} = (\text{MAKER} \parallel \text{USER}).$

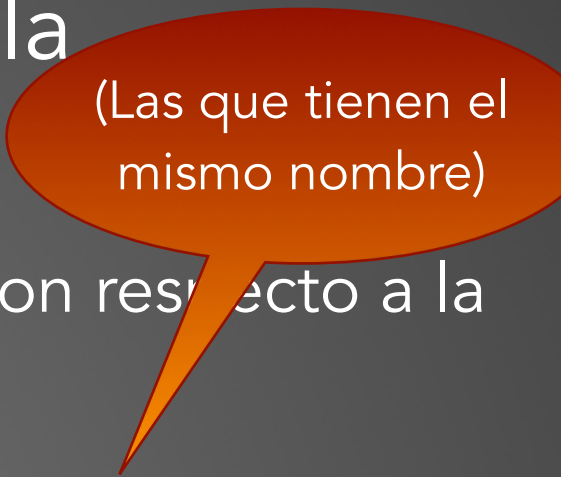


FSP: Composición paralela

Hay varios puntos importantes por recordar 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 (i.e. ustedes).
- El modelo de concurrencia es interleaving, donde las acciones atómicas independientes de diferentes procesos pueden ejecutarse en interleavings arbitrarios.

FSP: Composición paralela



(Las que tienen el mismo nombre)

Hay varios puntos importantes por recordar 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 (i.e. ustedes).
- El modelo de concurrencia es interleaving, donde las acciones atómicas independientes de diferentes procesos pueden ejecutarse en interleavings arbitrarios.

FSP: Otros conceptos y operaciones

- 👁 **Alfabeto de un proceso:** Conjunto de acciones al cual este proceso debe responder. 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) que no tiene permitido sincronizar.

FSP: Eliminación de operadores

Todo proceso (finito) puede escribirse usando sólo:

- prefijo
- elección, y
- recursión

Estas son las **operaciones básicas**