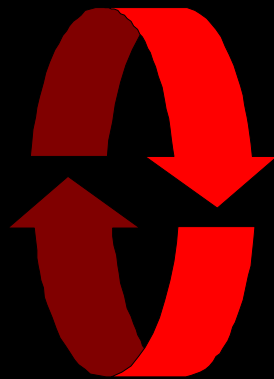


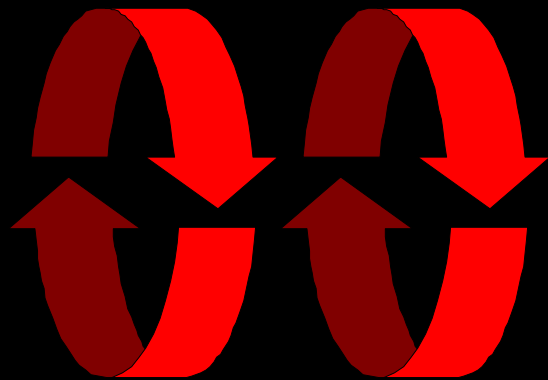
Programas Concurrentes

Ingeniería del Software 2

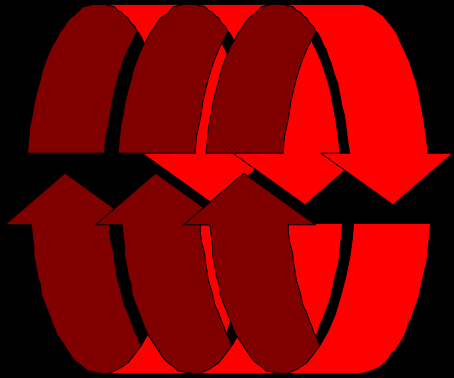
¿Qué es un programa concurrente?



- Un programa **secuencial** tiene un único thread de ejecución.



- Un programa **concurrente** tiene múltiples threads de ejecución, permitiéndole realizar múltiples cálculos en paralelo y controlar múltiples actividades externas que ocurren al mismo tiempo.

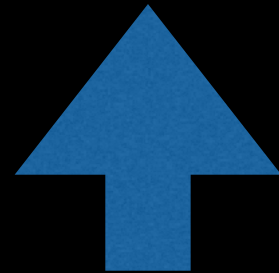


¿Para qué programas concurrentes?

- Mejora de performance en hardware paralelo.
- Mejora de throughput (ej. E/S no bloquean otros cálculos).
- Mejora de tiempo de respuesta de la aplicación (ej. alta prioridad para atender a input del usuario).
- Estructura mas apropiada para programas que interactúan con el ambiente, controlan múltiples actividades y reaccionan a múltiples eventos (sistemas reactivos)

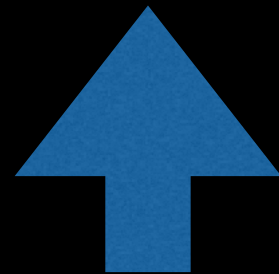
Concurrencia:

Procesamiento lógicamente simultáneo.
No implica múltiples unidades de procesamiento.
Requiere ejecución “interleaved” en un UP.



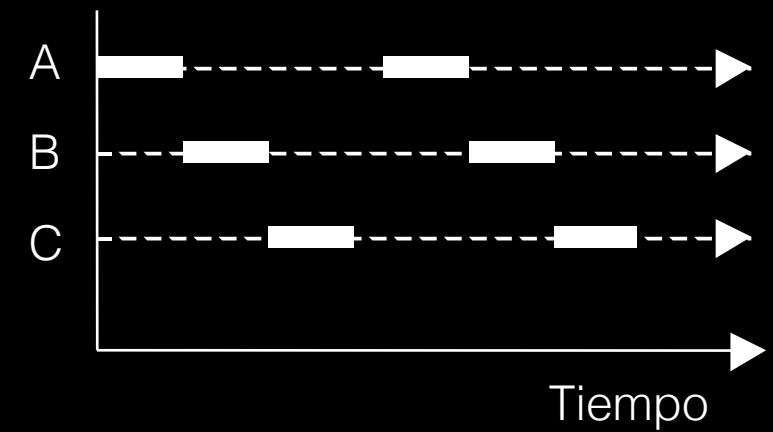
Paralelismo

Procesamiento físicamente simultáneo.
Involucra múltiples UPs.



Distribución

Procesamiento paralelo en UPs
distribuidas físicamente conectadas por una red



Preguntas de Interés

(Para Programas Concurrentes)

Compartidas con programas secuenciales

- Post-condiciones
- Invariante de ciclo
- Aserciones en código
- Terminación
- ...

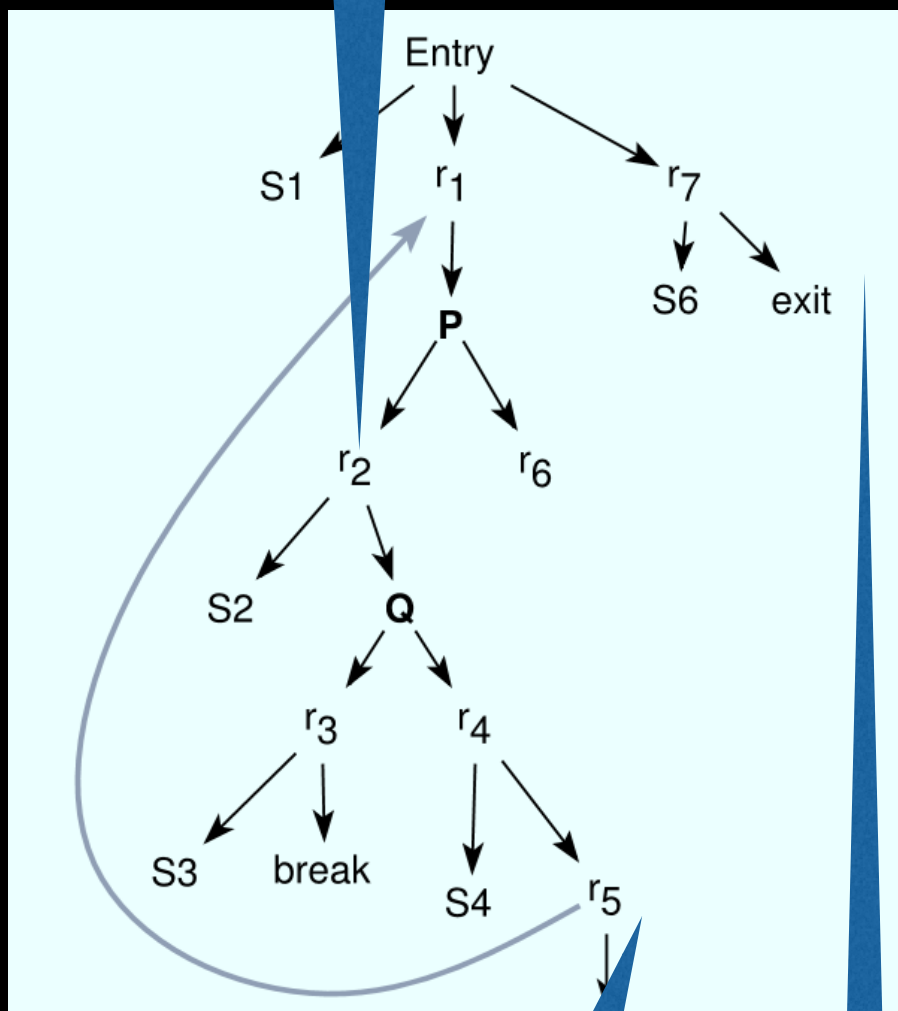
Particulares de programas concurrentes

- Deadlock
- Livelock
- Interferencia
- Fairness
- Atomicidad
- Propiedades de dominio específico
-

Modelos de Programas

Secuencial

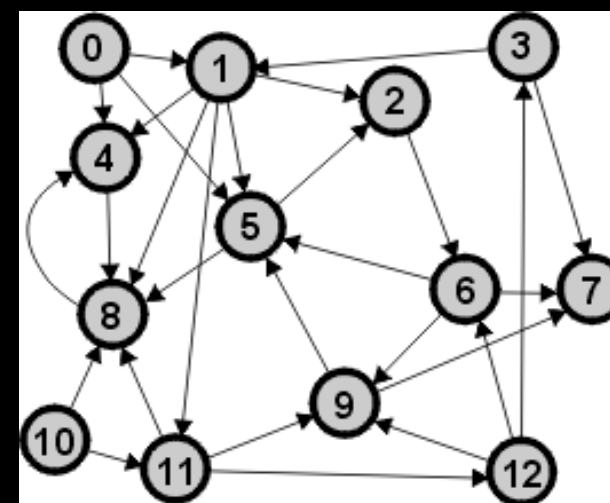
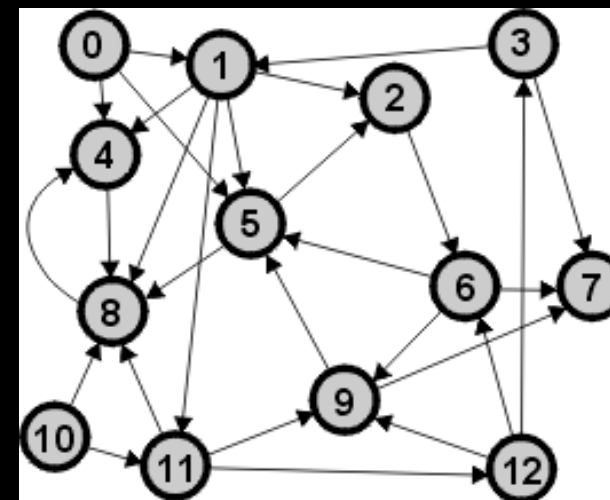
Es alcanzable? Con qué input?



Vale el invariante de ciclo?

Vale la pos-condición?

Concurrente



?

Deadlock

Livelock

Interferencia

Fairness

Algunos temas a tratar...

- ¿Qué modelo de programa es adecuado para **razonar** sobre programas concurrentes?
- ¿Qué lenguaje es adecuado para **formular preguntas** sobre programas concurrentes
- ¿Cómo se **computan automáticamente** respuestas a estas preguntas?

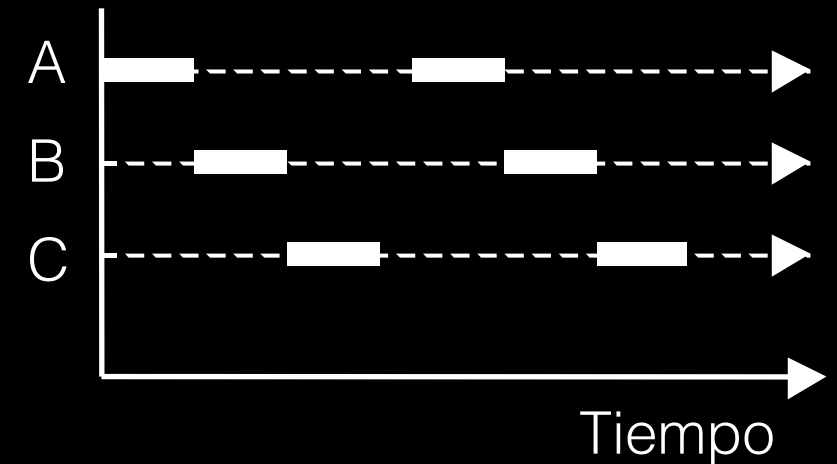
Paralelismo vs Concurrency

Concurrency:

Procesamiento lógicamente simultáneo.
No implica múltiples unidades de procesamiento.
Requiere ejecución "interleaved" en un UP.

Paralelismo

Procesamiento físicamente simultáneo.
Involucra múltiples UPs.



Decisión: Asumir que **no existe** capacidad de cómputo paralelo.

Consecuencias: Modelos más simples, requiere trucos para razonar sobre paralelismo, no se puede asumir nada sobre velocidad relativa de procesadores

Comunicación de Procesos

Memoria Compartida

- Semáforos
- Locks
- Monitores
- Pipes

Estructuras de Kripke

Mas adecuado para HW y programas de bajo nivel

Mensajes

- Handshakes y Rendezvous
- Go, Smalltalk, Scala
- SOAP
- MPI

Sistemas de Transición Etiquetados

Más adecuado para modelar sistemas distribuidos

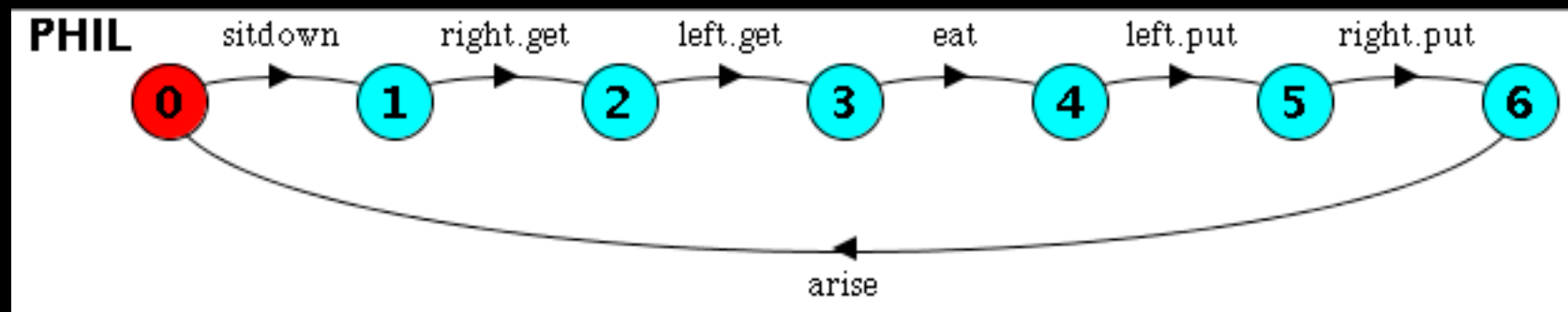
Sistemas de Transición Etiquetados

- Las etiquetas modelan **interacciones** de un proceso con su entorno (inputs y outputs)
- Hay una acción especial “tau” que modela cómputo interno del proceso, que no es observable desde el entorno.
- Hay una cantidad **finita** de estados y transiciones

Definición. (*LTS*) Sea *Estados* el universo de estados, *Act* el universo de acciones observables, y $Act_\tau = Act \cup \{\tau\}$. Un LTS es una tupla $P = (S, A, \Delta, s_0)$, donde $S \subseteq Estados$ es un conjunto finito, $A \subseteq Act_\tau$ es un conjunto de etiquetas, $\Delta \subseteq (S \times A \times S)$ es un conjunto de transiciones etiquetadas, y $s_0 \in S$ es el estado inicial. Definimos el alfabeto de comunicacion de P como $\alpha P = A \setminus \{\tau\}$.

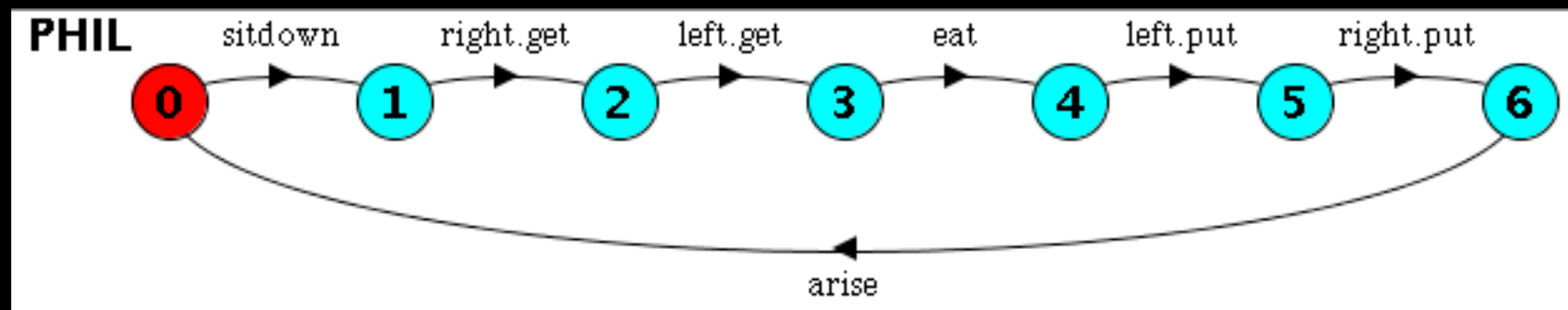
Un LTS

PHIL = $\langle \{0, 1, 2, 3, 4, 5, 6\},$
 $\{\text{sitdown}, \text{right.get}, \text{left.get}, \text{eat}, \text{left.put}, \text{right.put}, \text{arise}\},$
 $\{(0, \text{sitdown}, 1), (1, \text{right.get}, 2), (2, \text{left.get}, 3), (3, \text{eat}, 4),$
 $(4, \text{left.put}, 5), (5, \text{right.put}, 6), (6, \text{arise}, 0)\},$
 $0\}$



LTS - Ejecuciones

PHIL = $\langle \{0, 1, 2, 3, 4, 5, 6\},$
 $\{\text{sitdown}, \text{right.get}, \text{left.get}, \text{eat}, \text{left.put}, \text{right.put}, \text{arise}\},$
 $\{(0, \text{sitdown}, 1), (1, \text{right.get}, 2), (2, \text{left.get}, 3), (3, \text{eat}, 4),$
 $(4, \text{left.put}, 5), (5, \text{right.put}, 6), (6, \text{arise}, 0)\},$
 $0 \rangle$

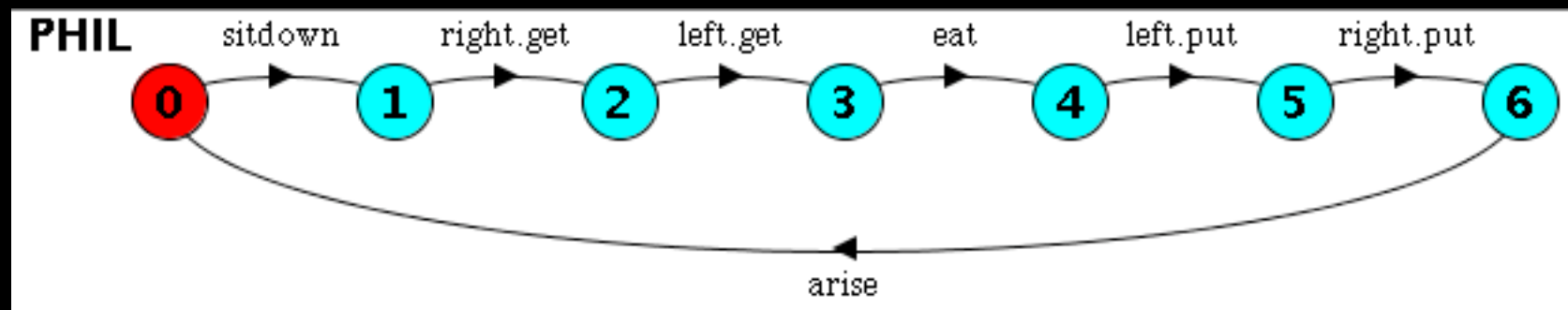


Definición. (*Ejecuciones y Trazas*) Una ejecución de un LTS $P = (S, A, \Delta, s_0)$ es una secuencia $s_0, \ell_0, s_1, \ell_1, \dots$, donde para cada $i \geq 0$ tenemos $(s_i, \ell_i, s_{i+1}) \in \Delta$.

0, sitdown, 1, right.get, 2, left.get, 3, eat, 4, left.put, 5, right.put, 6, arise, 0, sitdown, ...

LTS - Trazas

PHIL = $\langle \{0, 1, 2, 3, 4, 5, 6\},$
 $\{\text{sitdown}, \text{right.get}, \text{left.get}, \text{eat}, \text{left.put}, \text{right.put}, \text{arise}\},$
 $\{(0, \text{sitdown}, 1), (1, \text{right.get}, 2), (2, \text{left.get}, 3), (3, \text{eat}, 4),$
 $(4, \text{left.put}, 5), (5, \text{right.put}, 6), (6, \text{arise}, 0)\},$
 $0 \rangle$

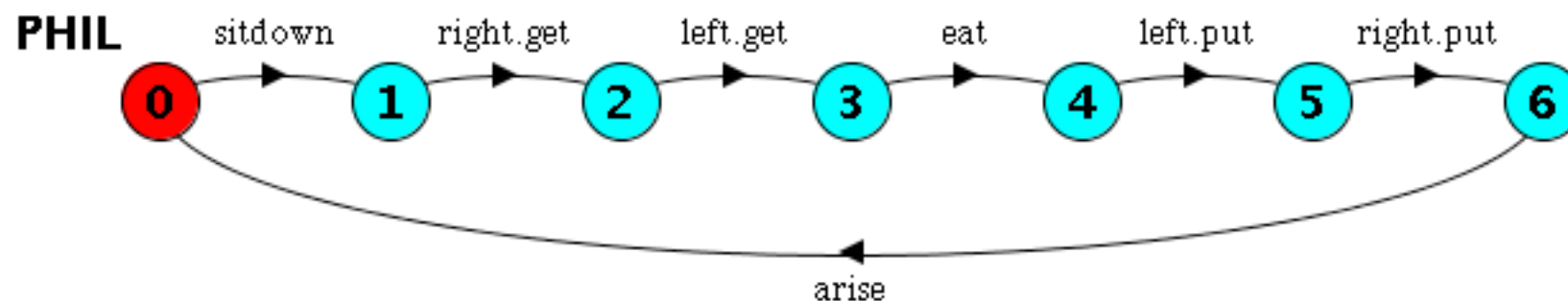


Definición 2.1.4. (Trazas) Una *traza* de un LTS E es una secuencia de etiquetas $\pi = \ell_0, \ell_1, \dots$ de las cuales existe una secuencia de estados s_0, s_1, \dots tal que s_0 es el estado inicial de E y $\forall i \geq 0 \cdot \ell_i \in \Delta_E(s_i)$.

sitdown, right.get, left.get, eat, left.put, right.put, arise, sitdown, ...

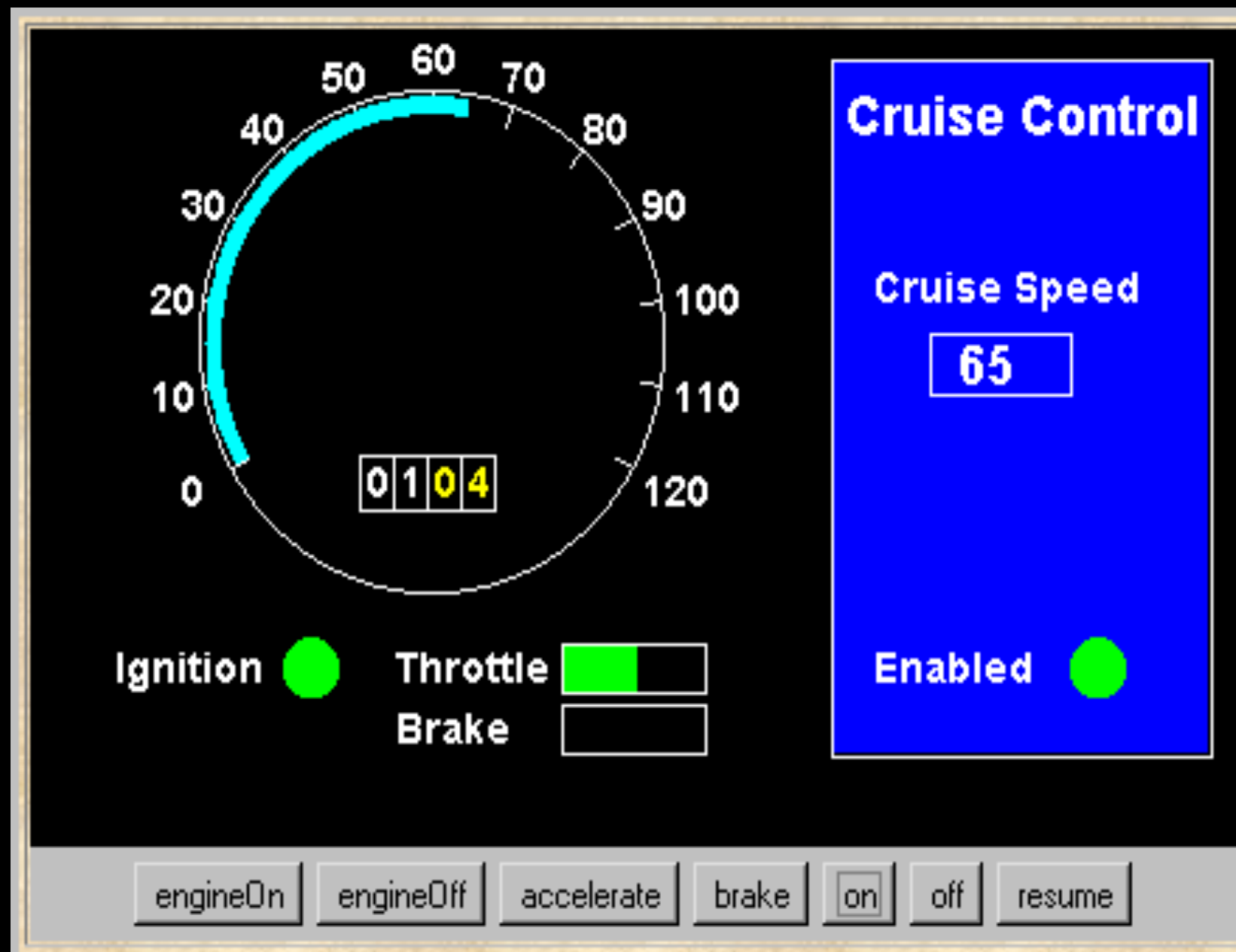
LTS - Transitar

Definición. (*Transitar*) Dado un LTS $P = (S, L, \Delta, s)$, decimos que P transita con la acción $\ell \in A$ a un LTS P' , ($P \xrightarrow{\ell} P'$), si $P' = (S, L, \Delta, s')$, donde $(s, \ell, s') \in \Delta$. Usamos $P \xrightarrow{\ell}$ para decir que existe un P' tal que $P \xrightarrow{\ell} P'$



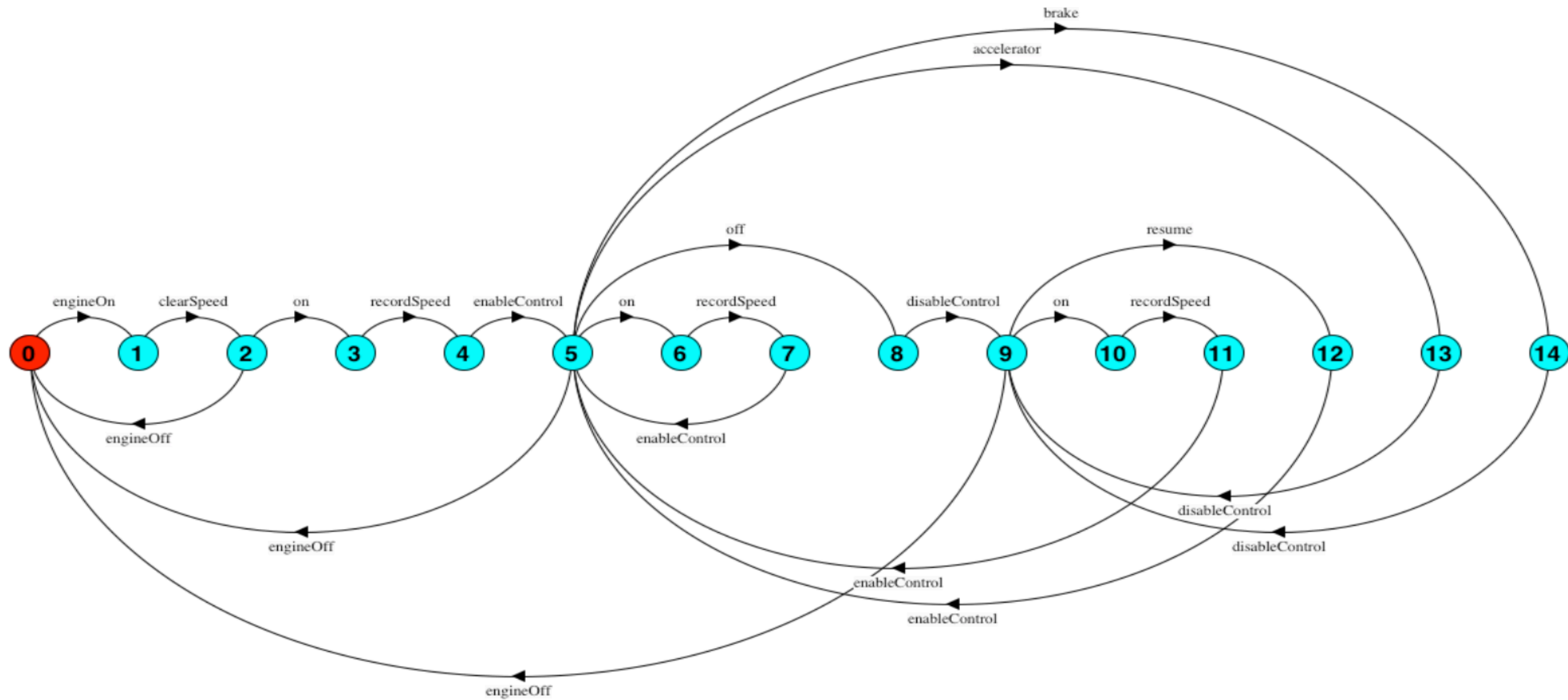
Puede PHIL transitar? A qué?

Sistema de “Autocruzero”



Prototipo basado en un modelo

LTS del Cruise Controller



FSP

Para poder describir LTS en forma compacta, utilizamos un lenguaje de texto llamado Procesos de Estados Finitos (finite state processes (FSP))

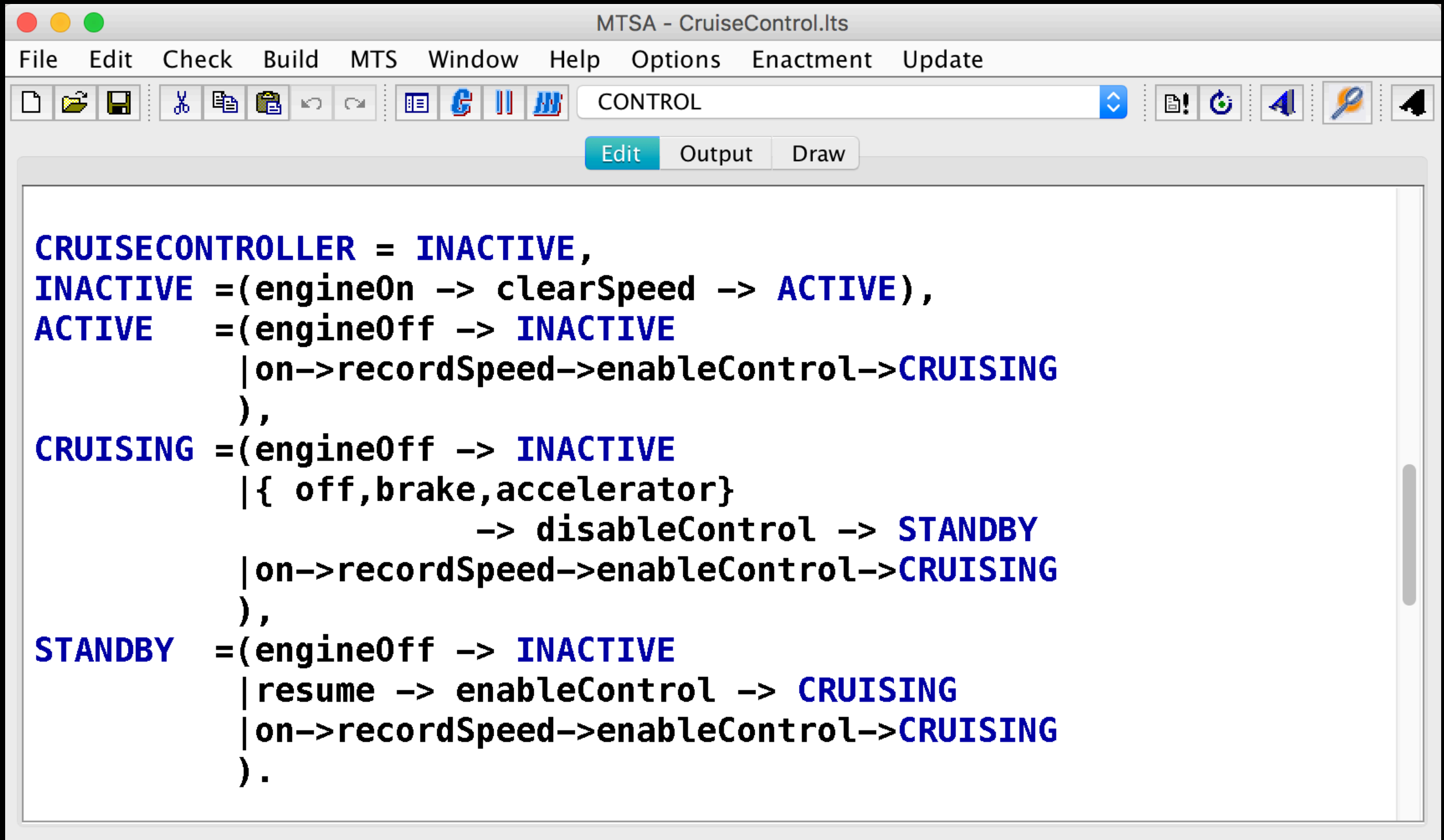
Sintaxis:

Procesos básicos y operadores que definen procesos mas complejos.

Semántica:

Con una función de términos FSP a LTS
 $lts(FSP) \rightarrow LTS$

FSP del Cruise Controller



The screenshot shows a software window titled "MTSA - CruiseControl.Its". The menu bar includes "File", "Edit", "Check", "Build", "MTS", "Window", "Help", "Options", "Enactment", and "Update". Below the menu bar is a toolbar with various icons for file operations and editing. A dropdown menu is currently set to "CONTROL". Below the toolbar are three tabs: "Edit" (selected), "Output", and "Draw". The main editing area contains the following FSP code:

```
CRUISECONTROLLER = INACTIVE,  
INACTIVE =(engineOn -> clearSpeed -> ACTIVE),  
ACTIVE   =(engineOff -> INACTIVE  
           |on->recordSpeed->enableControl->CRUISING  
           ),  
CRUISING =(engineOff -> INACTIVE  
           |{ off,brake,accelerator}  
           -> disableControl -> STANDBY  
           |on->recordSpeed->enableControl->CRUISING  
           ),  
STANDBY  =(engineOff -> INACTIVE  
           |resume -> enableControl -> CRUISING  
           |on->recordSpeed->enableControl->CRUISING  
           ).
```

FSP Syntax

<https://www.doc.ic.ac.uk/~jnm/LTSdocumentation/FSP-Syntax.html>

Primitive Process

```
primitive_process ::= upper_identifier [ "(" parameter_list ")" ] "=" primitive_process_body
primitive_process_body ::= process_body { "," local_process_defn } [ alphabet_extension ] [ relabels ] [ label_visibility ] "."
local_process_name ::= upper_identifier [ index ] | STOP | ERROR
process_body ::= "(" choices ")" | local_process_name | conditional
choices ::= choice { "|" choice }
choice ::= [ when boolean_expression ] action_label_part "->" process_body
action_label_part ::= action_label | action_label_set
conditional ::= if boolean_expression then process_body [ else process_body ]
local_process_defn ::= upper_identifier [ "@" ] [ index | index_range ] "=" process_body
parameter_list ::= parameter { "," parameter }
parameter ::= upper_identifier "=" integer_value
```

Composite Process

```
composite_process ::=
"|" upper_identifier [ "(" parameter_list ")" ] "=" composite_body [ relabels ] [ label_visibility ] "."
composite_body ::= process_instance | parallel_list | composite_conditional | composite_replicator
composite_replicator ::= forall index_range composite_body
composite_conditional ::= if boolean_expression then composite_body [ else composite_body ]
parallel_list ::= "(" composite_body { "|" composite_body } ")"
process_instance ::= [ action_label_set "::" ] [ action_label ":" ] upper_identifier
[ "(" actual_parameter_list ")" ] [ relabels ]
actual_parameter_list ::= expression { "," expression }
```

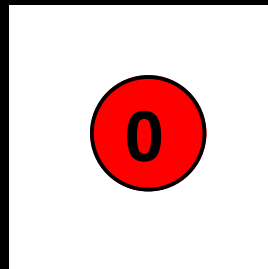
...

Vamos a introducir parte de la sintaxis de a poco y simultánea e informalmente su semántica a LTS.

FSP - El proceso Deadlock

STOP es un proceso (y palabra reservada) que describe el proceso que es incapaz de interactuar con su entorno.

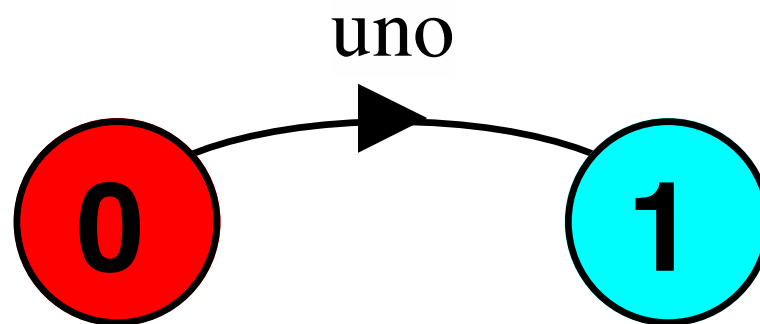
`NadaDeNada = STOP.`


$$lts(\text{STOP}) = \langle \{s\}, \{\tau\}, \{ \}, s \rangle$$

FSP - action prefix

Si x es una acción y P un proceso entonces $(x \rightarrow P)$ describe un proceso que inicialmente **interactúa** a través de la acción x y luego se comporta como P .

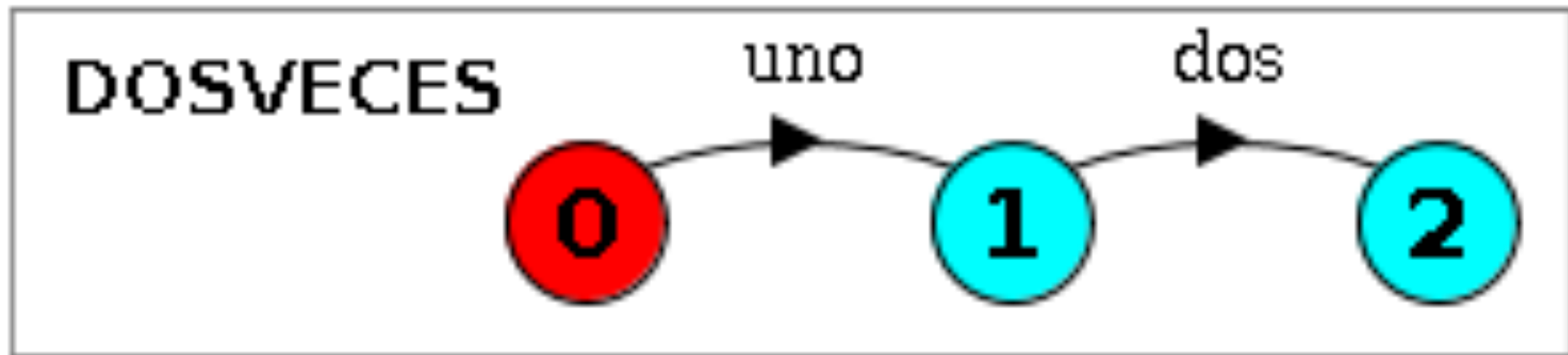
`UnicaVez = (uno -> STOP) .`



If $lts(E) = \langle S, A, \Delta, q \rangle$ and E is not ERROR
then $lts(a \rightarrow E) = \langle S \cup \{p\}, A \cup \{a\}, \Delta \dot{\cup} \{(p, a, q)\}, p \rangle$, where $p \notin S$.
 $lts(a \rightarrow \text{ERROR}) = \langle \{p, \pi\}, \{a\}, \{(p, a, \pi)\}, p \rangle$, where $p \neq \pi$.

FSP - action prefix

Dosveces = (uno \rightarrow dos \rightarrow STOP).



FSP - Recursión

Repetición de comportamiento se modela con recursión:

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

$$\frac{lts(E[X \leftarrow rec(X = E)]) \xrightarrow{a} P}{lts(rec(X = E)) \xrightarrow{a} P}$$

$lts(\text{rec}(\text{SWITCH} = \text{on} \rightarrow \text{off} \rightarrow \text{SWITCH})) \sim$

$lts(\text{on} \rightarrow \text{off} \rightarrow \text{rec}(\text{SWITCH} = \text{on} \rightarrow \text{off} \rightarrow \text{SWITCH})) \sim$

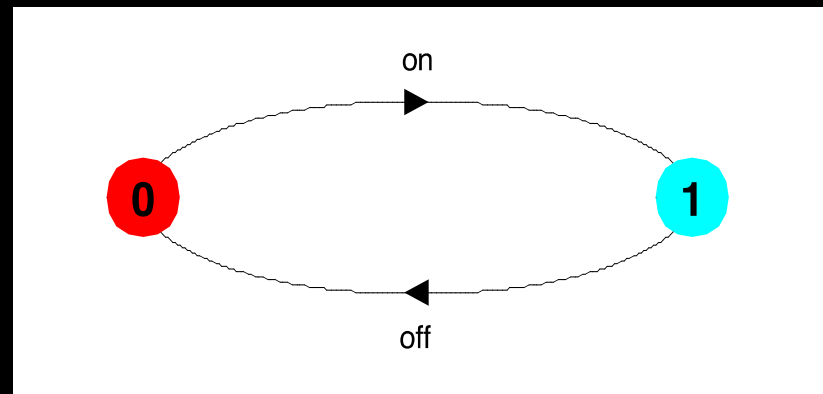
$lts(\text{on} \rightarrow \text{off} \rightarrow \text{on} \rightarrow \text{off} \rightarrow \text{rec}(\text{SWITCH} = \text{on} \rightarrow \text{off} \rightarrow \text{SWITCH})) \sim$

FSP - Recursión

Repetición de comportamiento se modela con recursión:

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

¿Cuál es el LTS más chico que cumple la recursión?



Sub-Procesos

a.k.a. Definiciones locales a un Proceso

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

La coma introduce un subproceso.

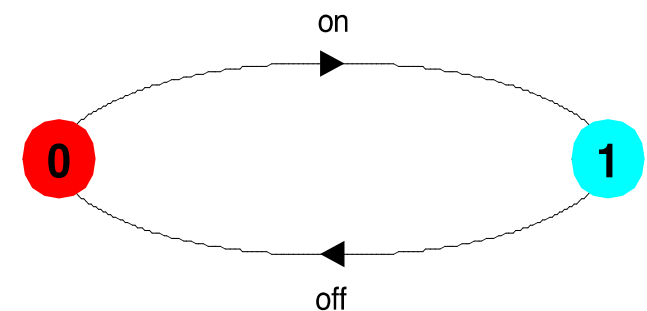
```
SWITCH = OFF,  
OFF    = (on -> (off->OFF)) .
```

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

El punto termina la definición de un proceso

Convención:

Acciones se escriben en minúscula,
Procesos se escriben en mayúscula



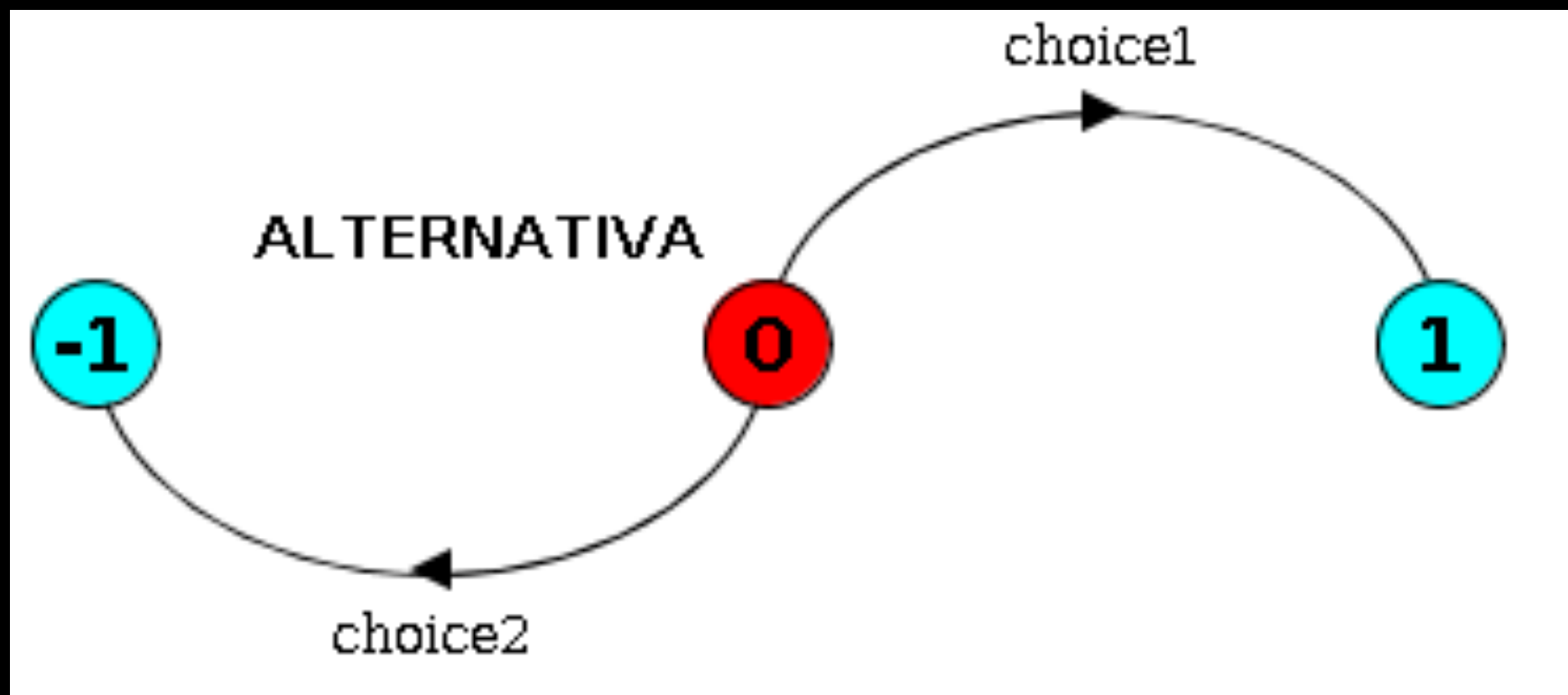
FSP - Alternativas

Si x e y son acciones entonces $(x \rightarrow P \mid y \rightarrow Q)$ describe un proceso que inicialmente es capaz de interactuar a través de las acciones x o y . El proceso pasa a comportarse como P o Q según ocurra x o y .

Let $1 \leq i \leq n$, and $lts(E_i) = \langle S_i, A_i, \Delta_i, q_i \rangle$,
then $lts(a_1 \rightarrow E_1 \mid \dots \mid a_n \rightarrow E_n)$
 $= \langle S \cup \{p\}, A \cup \{a_1 \dots a_n\}, \Delta \cup \{(p, a_1, q_1 \dots (p, a_n, q_n, p) \rangle$,
where $p \notin S_i$, $S = \bigcup_i S_i$, $A = \bigcup_i A_i$, $\Delta = \bigcup_i \Delta_i$.
If E_i is ERROR then $A_i = \{ \}$.

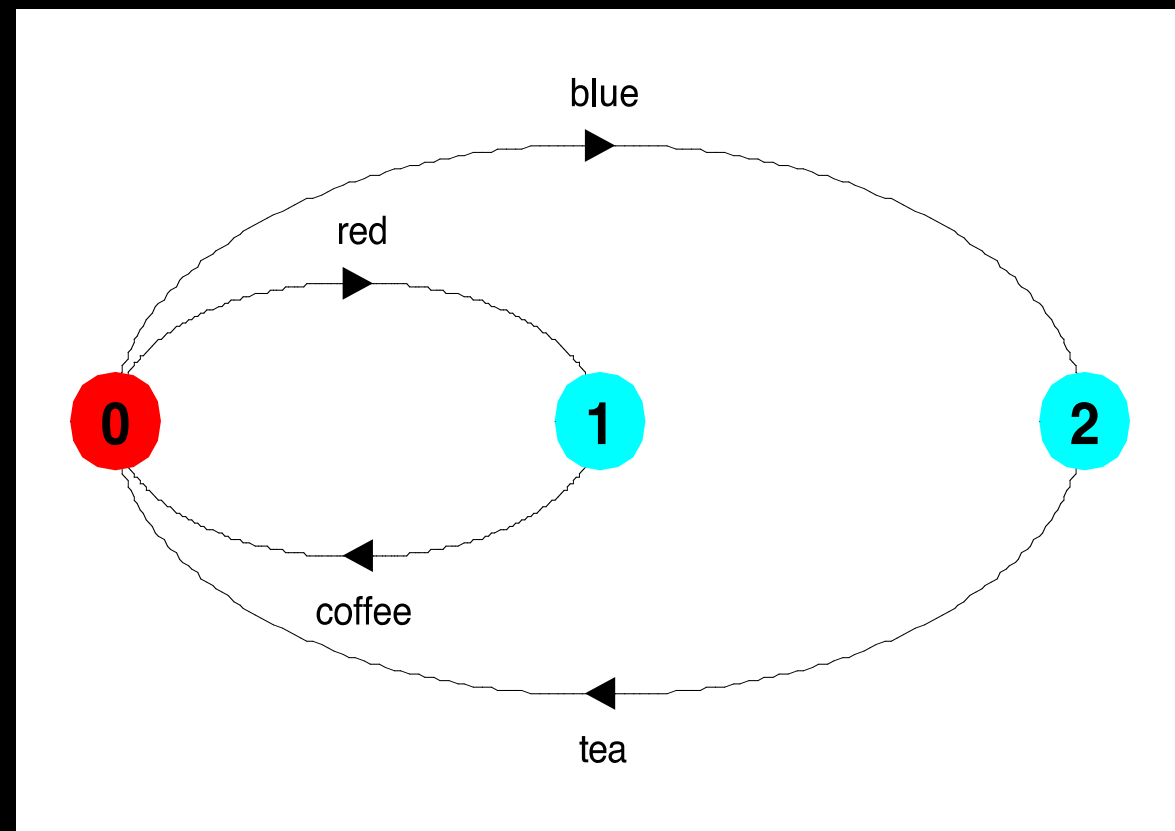
FSP - Alternativas

`ALTERNATIVA = (choice1 -> STOP | choice2 -> ERROR) .`



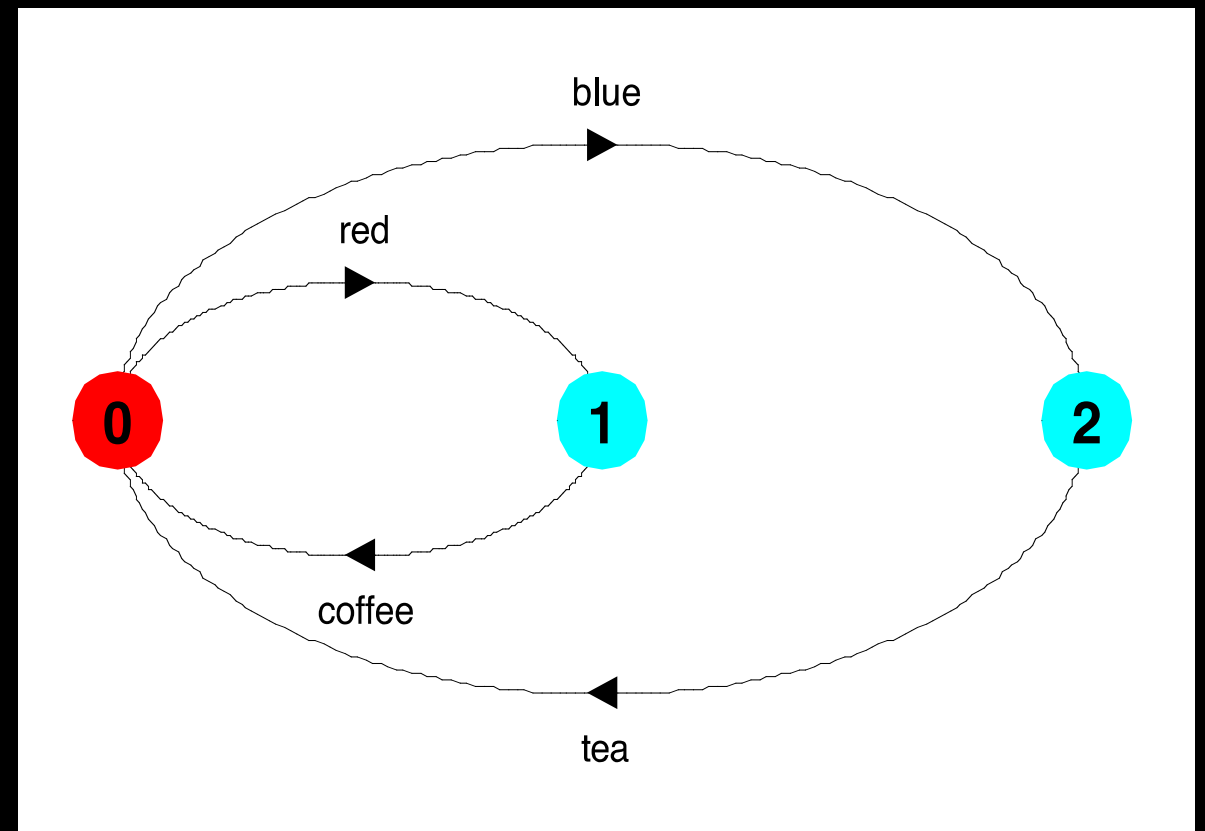
FSP - Alternativas

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



FSP - Alternativas

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



¿Ejecuciones (acciones y estados) y Trazas (solo acciones)?

¿Quién elige que alternativa tomar?

¿No hay una distinción entre output e input?

Elección no-determinística

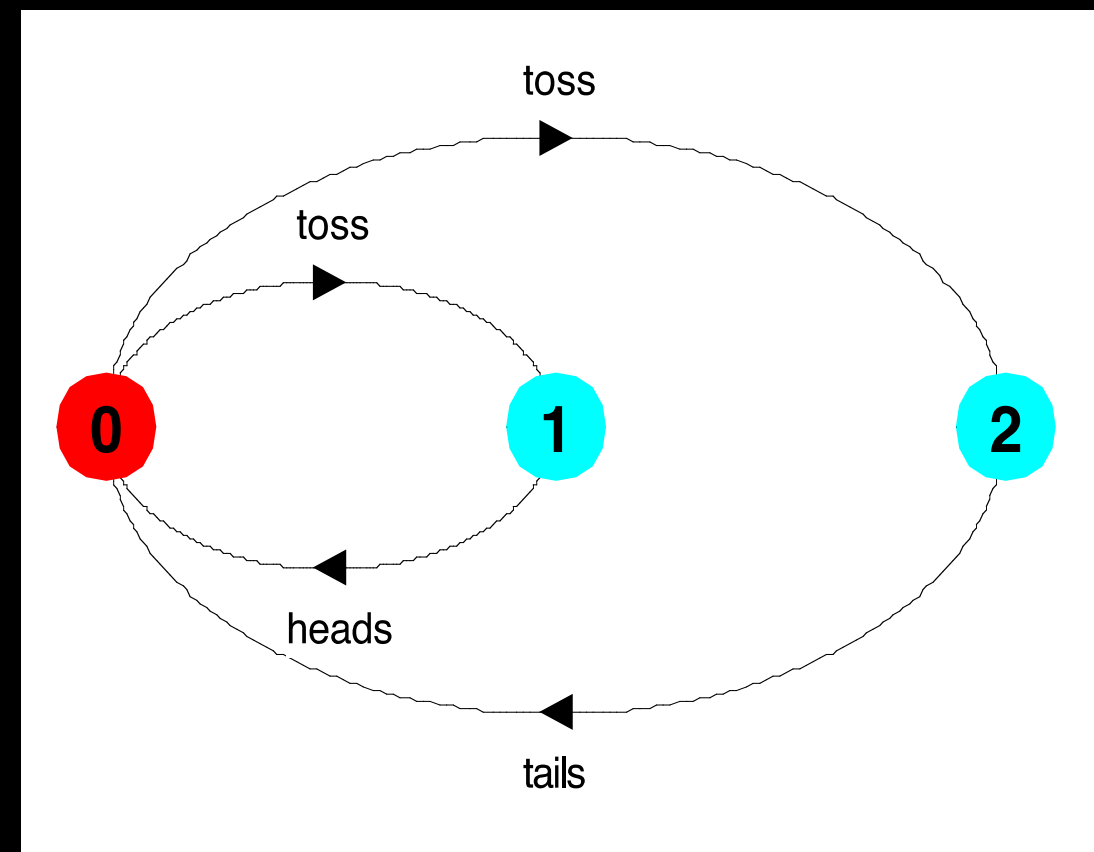
El proceso $(x \rightarrow P \mid x \rightarrow Q)$ describe una elección no determinística entre P or Q .

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

Tirar una moneda...

¿Ejecuciones y Trazas?

Ojo, No determinismo no es lo mismo que equi-probable



FSP - Procesos y acciones indexadas

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

equivalente a

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

o usando un parámetro de proceso con un valor por defecto:

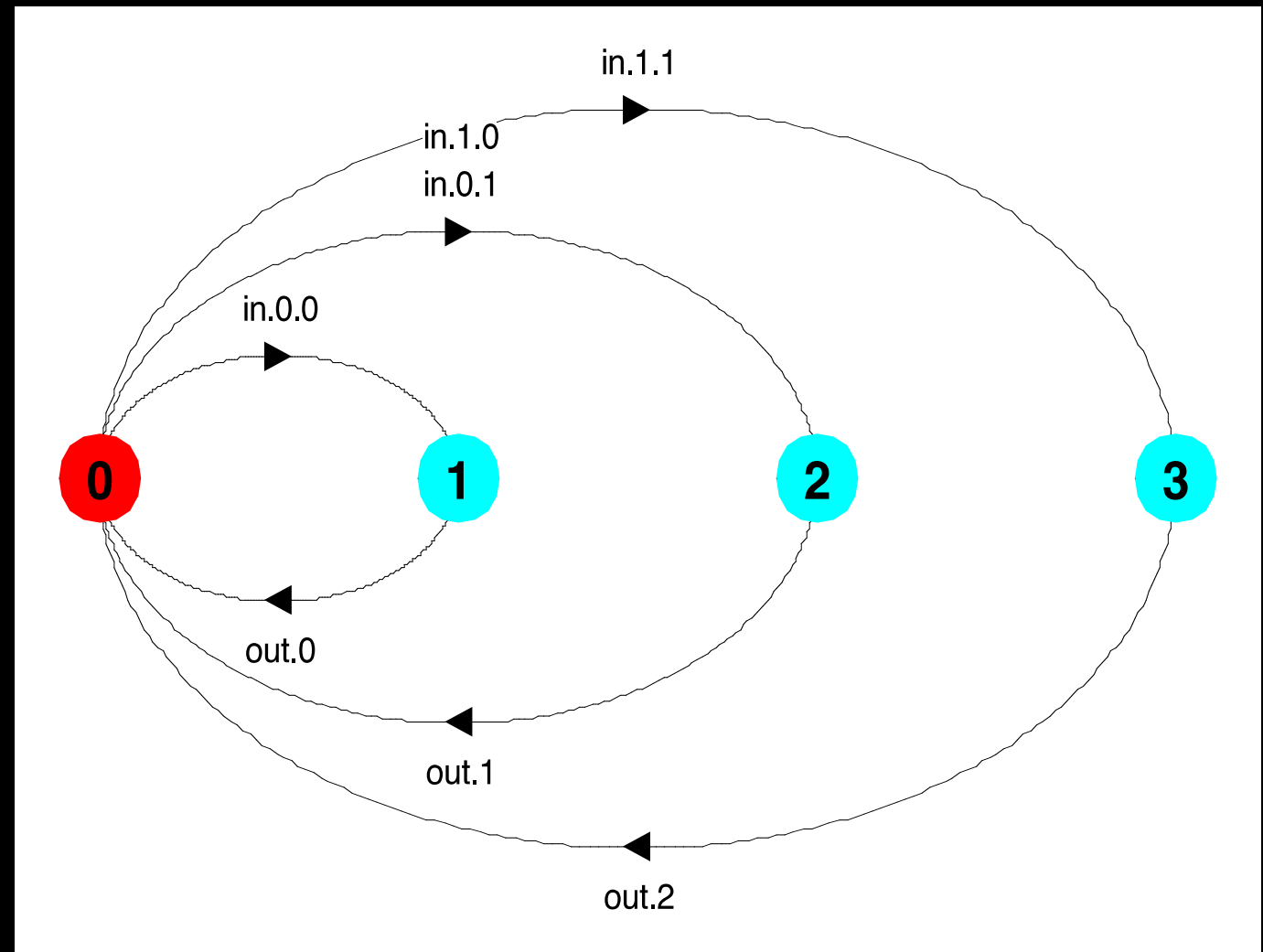
$\text{BUFF}(N=3) = (\text{in}[i:0..N] \rightarrow \text{out}[i] \rightarrow \text{BUFF}) .$

FSP - Declaración de Constantes y Rangos

Indices pueden usarse para modelar cómputo:

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

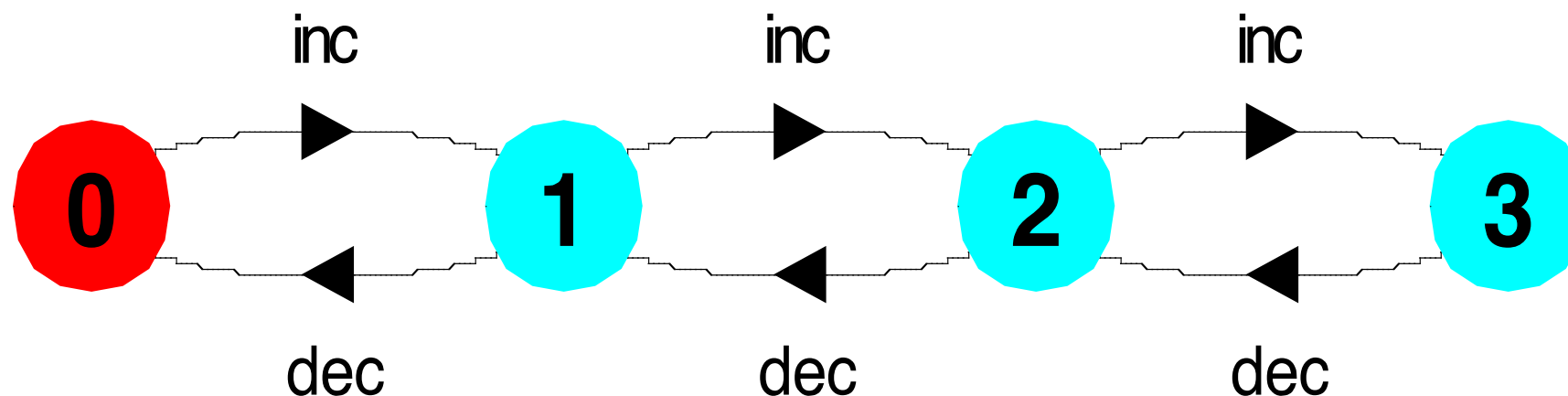
```
SUM          = (in[a:T] [b:T] -> TOTAL[a+b]) ,
TOTAL[s:R]   = (out[s] -> SUM) .
```



FSP - Guardas

La alternativa (**when** **B** **x** \rightarrow **P** | **y** \rightarrow **Q**) significa que cuando la guarda **B** es verdadera entonces tanto **x** como **y** pueden ser elegidas, caso contrario, si **B** es falso entonces la acción **x** no puede ser elegida.

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

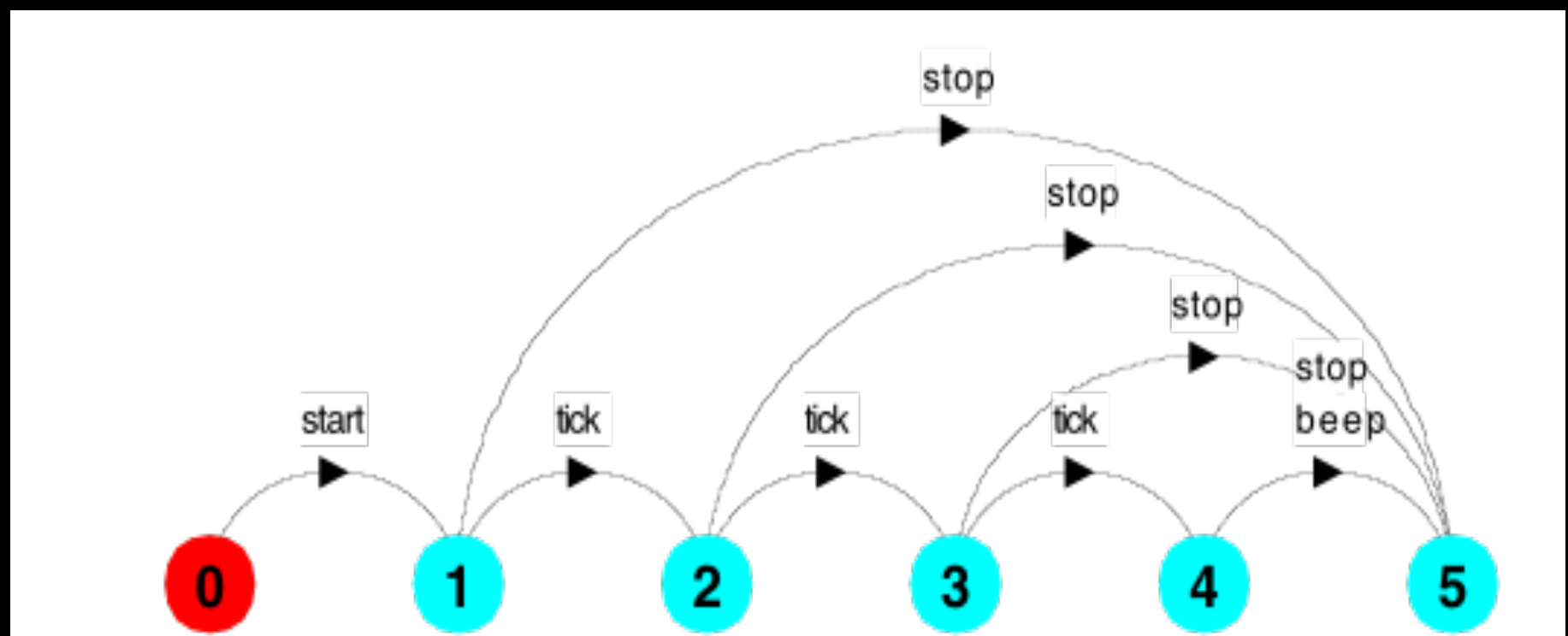


Azúcar
sintáctico

FSP - Guardas

Una alarma que cuenta N ticks y hace beep y que puede ser parado en cualquier momento

```
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 - Guardas

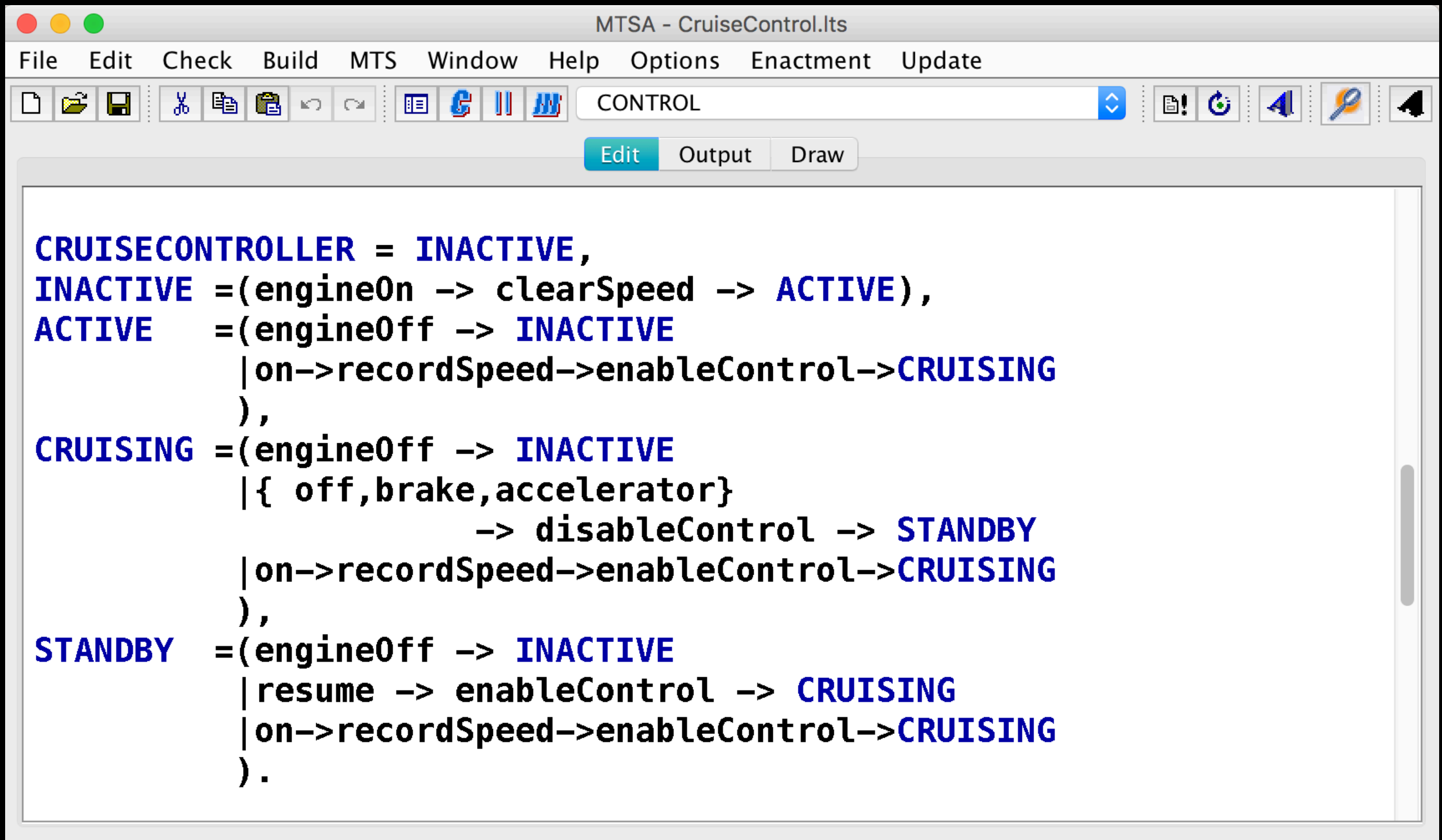
```
const False = 0  
P = (when (False) cualquiera -> P) .
```

P es equivalente a qué proceso?

Respuesta:

STOP

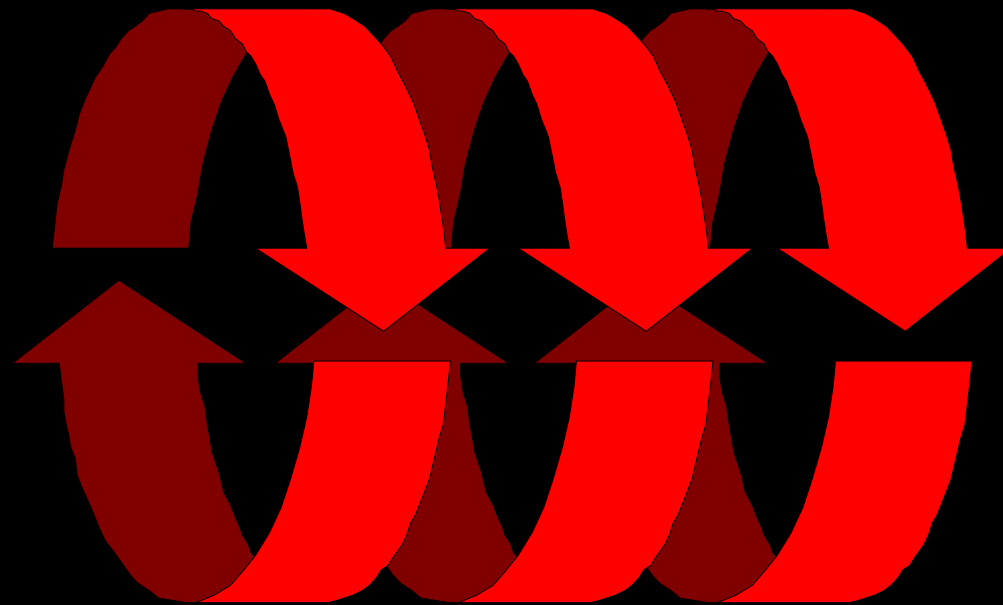
FSP del Cruise Controller



The screenshot shows a software window titled "MTSA - CruiseControl.Its". The menu bar includes "File", "Edit", "Check", "Build", "MTS", "Window", "Help", "Options", "Enactment", and "Update". Below the menu bar is a toolbar with various icons. A dropdown menu is open, showing "CONTROL". Below the toolbar are three buttons: "Edit" (highlighted in blue), "Output", and "Draw". The main text area contains the following FSP code:

```
CRUISECONTROLLER = INACTIVE,  
INACTIVE =(engineOn -> clearSpeed -> ACTIVE),  
ACTIVE   =(engineOff -> INACTIVE  
           |on->recordSpeed->enableControl->CRUISING  
           ),  
CRUISING =(engineOff -> INACTIVE  
           |{ off,brake,accelerator}  
           -> disableControl -> STANDBY  
           |on->recordSpeed->enableControl->CRUISING  
           ),  
STANDBY  =(engineOff -> INACTIVE  
           |resume -> enableControl -> CRUISING  
           |on->recordSpeed->enableControl->CRUISING  
           ).
```

Procesos Concurrentes



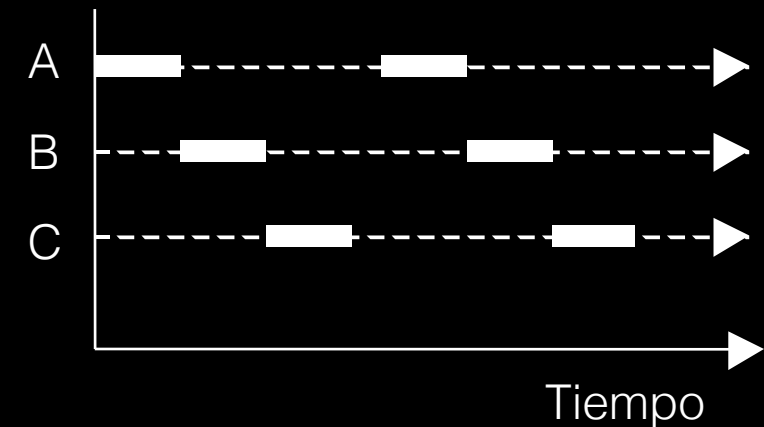
Paralelismo vs Concurrency

Concurrencia:

Procesamiento lógicamente simultáneo.
No implica múltiples unidades de procesamiento.
Requiere ejecución “interleaved” en un UP.

Paralelismo

Procesamiento físicamente simultáneo.
Involucra múltiples UPs.



Decisión: No asumir existencia de capacidad de cómputo paralelo.

Consecuencias: Modelos más simples, requiere trucos para razonar sobre paralelismo, no se puede asumir nada sobre velocidad relativa de procesadores

*Interesados en modelos para paralelismo,
ver redes de petri (matemática subyacente a diagramas de actividad)*

Composición en paralelo

Si P y Q son procesos entonces (P||Q) representa la ejecución concurrente de P and Q. El operador || es el operador de composición en paralela.

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

$$lts(P || Q) = lts(P) || lts(Q)$$

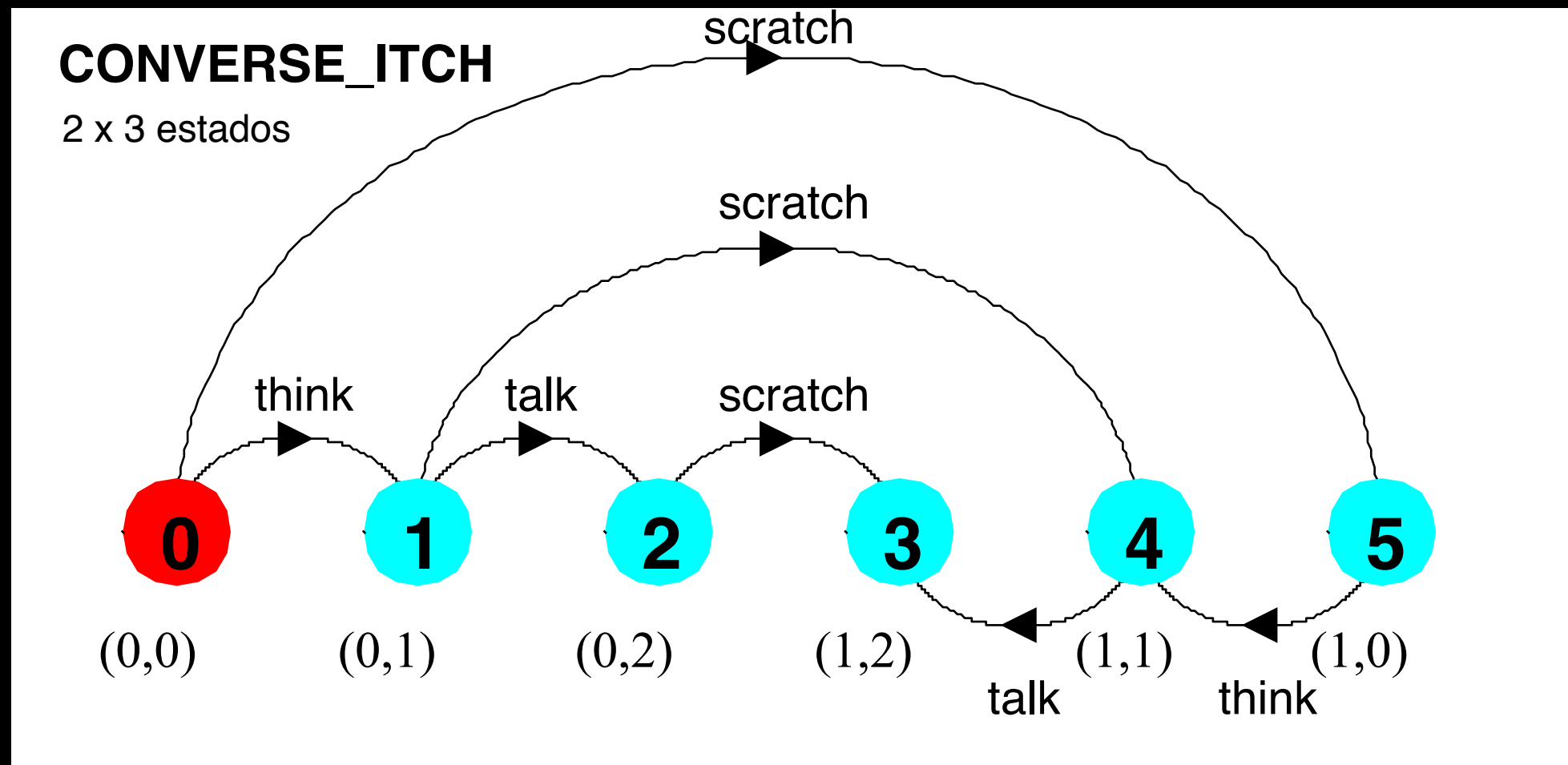
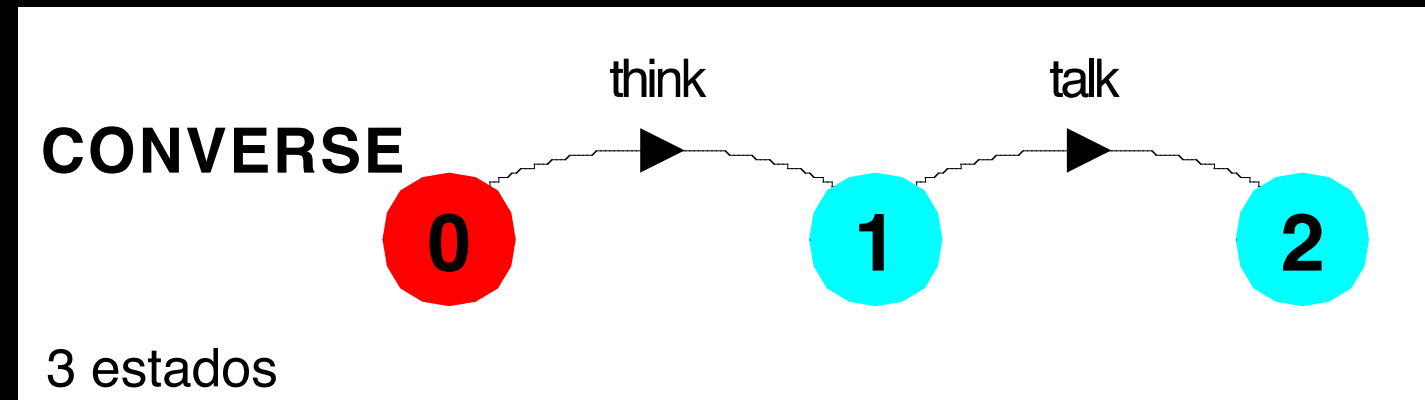
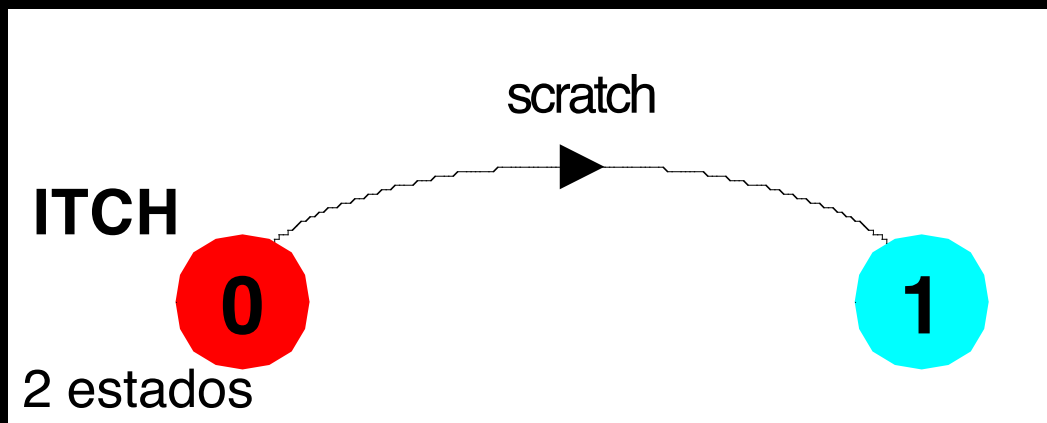
Composición en paralelo de LTS

Definición. (*Composición Paralela*) Sean los LTS $M = (S_M, A_M, \Delta_M, s_0^M)$ y $N = (S_N, A_N, \Delta_N, s_0^N)$. La *composición paralela* \parallel es un operador simétrico tal que $M \parallel N$ es el LTS $(S_M \times S_N, A_M \cup A_N, \Delta, (s_0^M, s_0^N))$, donde Δ es la relación más chica que satisface las siguientes reglas (con $\ell \in A_M \cup A_N$):

$$\frac{(s, \ell, s') \in \Delta_M}{((s, t), \ell, (s', t)) \in \Delta} \ell \notin \alpha N \qquad \frac{(t, \ell, t') \in \Delta_N}{((s, t), \ell, (s, t')) \in \Delta} \ell \notin \alpha M$$

$$\frac{(s, \ell, s') \in \Delta_M, (t, \ell, t') \in \Delta_N}{((s, t), \ell, (s', t')) \in \Delta} \ell \in \alpha M \cap \alpha N$$

Interleaving



Propiedades

Conmutatividad: $(P || Q) = (Q || P)$

Asociatividad: $(P || (Q || R)) = ((P || Q) || R)$
 $= (P || Q || R) .$

$\text{CLOCK} = (\text{tick} \rightarrow \text{CLOCK}) .$

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

$|| \text{CLOCK_RADIO} = (\text{CLOCK} || \text{RADIO}) .$

LTS? Trazas? Número de estados?

Acciones compartidas

Si dos procesos en una composición tienen acciones en común, se dice que acciones son *compartidas*. Acciones compartidas son la manera en que se modela interacción de procesos.

Mientras se hace interleaving de acciones no compartidas, una acción común debe ser ejecutada al mismo tiempo por todos los procesos que participan en esa acción compartida.

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

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

```
|| MAKER_USER = (MAKER || USER) .
```

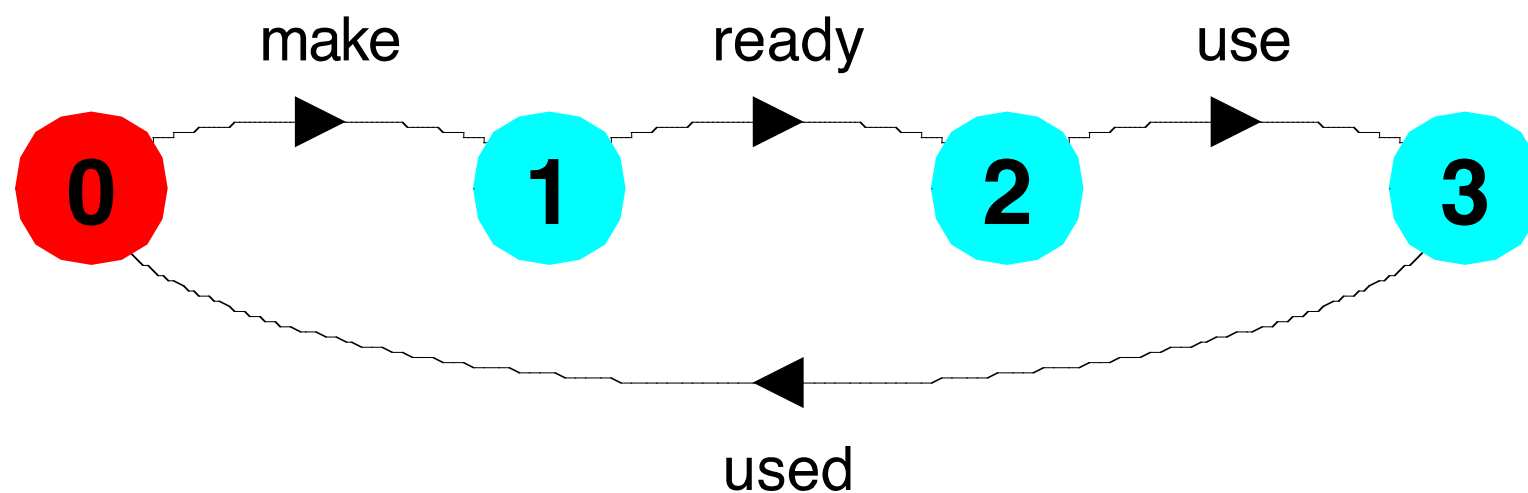
LTS? Trazas? Número de estados?

Modelado de Handshake

```
MAKERv2 = (make->ready->used->MAKERv2) .
```

```
USERv2 = (ready->use->used ->USERv2) .
```

```
|| MAKER_USERv2 = (MAKERv2 || USERv2) .
```

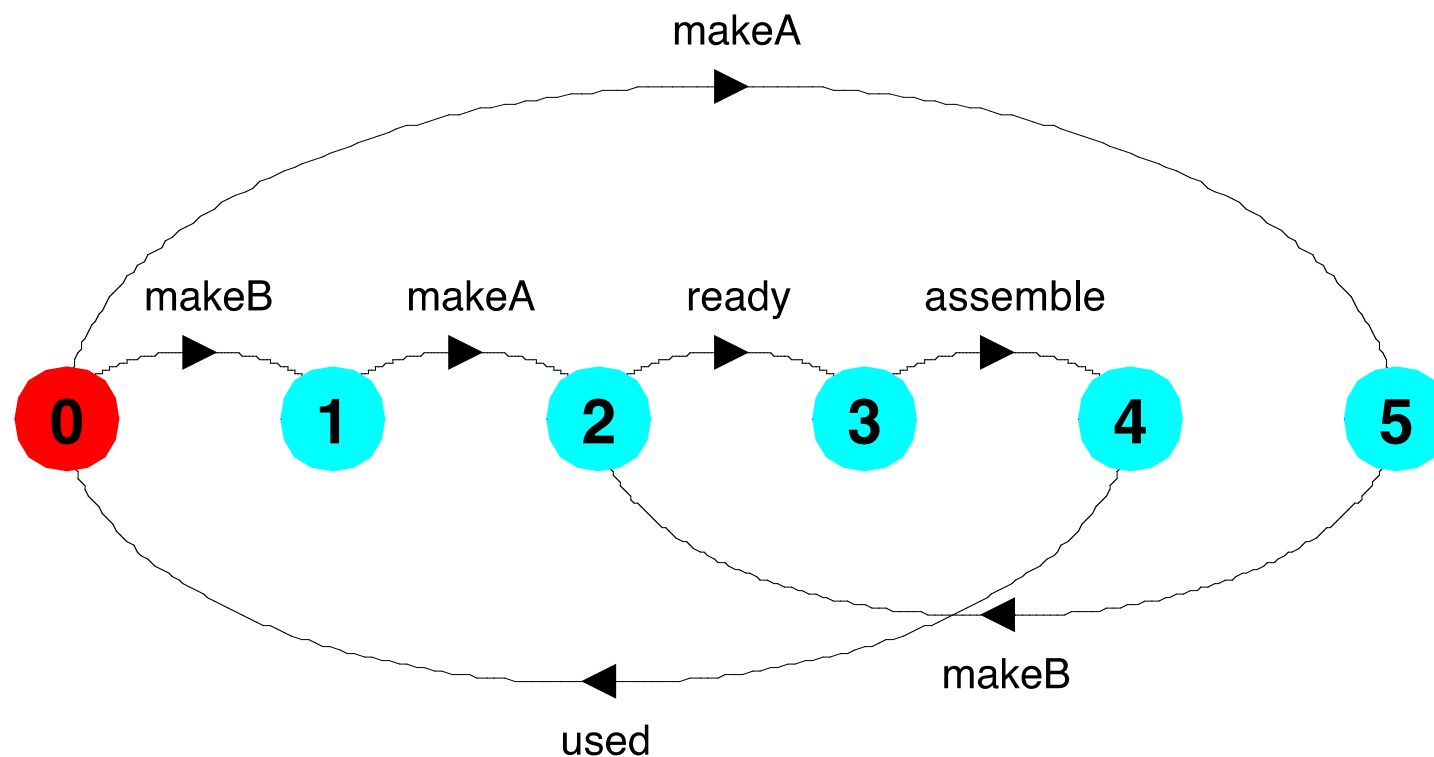


3 x 3 = 9
estados?

La interacción
restringe el
comportamiento
global.

Sincronización Múltiple

```
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) .
```



Procesos compuestos

Un proceso compuesto es la composición en paralelo de procesos. Un proceso compuesto puede usarse en nuevas composiciones.

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

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

Por sustitución y aplicando conmutatividad y asociatividad obtenemos:

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