Ingeniería del Software II
3 - Sincronización de procesos concurrentes

Recursos compartidos

En ámbitos donde la concurrencia es útil, suele ocurrir que diferentes procesos necesitan interactuar mediante el uso de recursos comunes.

Luego de experimentar un poco con programas concurrentes (o modelos de éstos), resulta evidente que el uso de recursos comunes por diferentes procesos puede dar lugar a actualizaciones incorrectas en el estado de estos recursos. Este problema es conocido como **interferencia**.

Para manejar el problema, se debe asegurar que, mientras uno de los procesos utiliza un recurso compartido, el resto no puede acceder al mismo (**exclusion mutua**).

Recursos compartidos

En ámbitos donde la concurrencia es útil, suele ocurrir que diferentes procesos necesitan interactu[illegible]de recursos comunes.

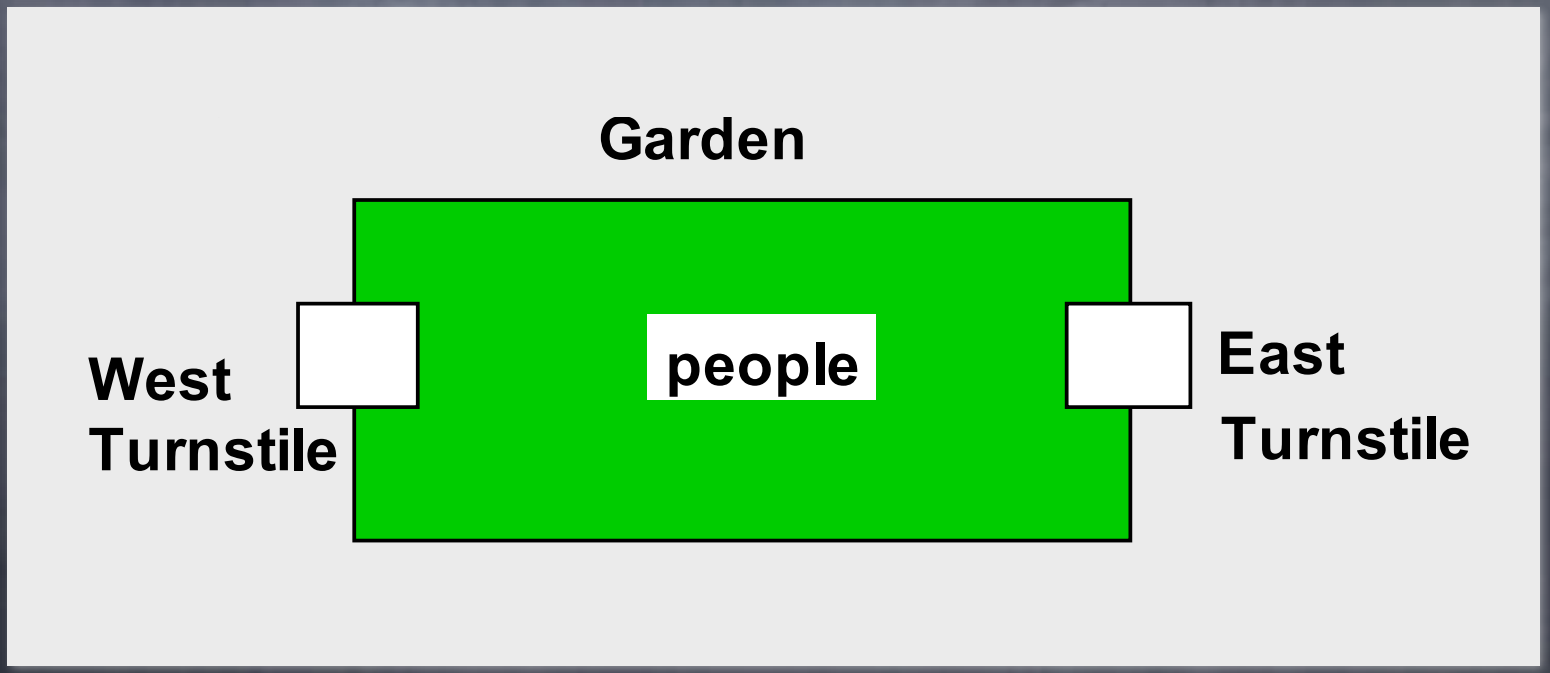
... o **Race Condition**

Luego de experimentar un poco con programas concurrentes (o modelos de éstos), resulta evidente que el uso de recursos comunes por diferentes procesos puede dar lugar a actualizaciones incorrectas en el estado de estos recursos. Este problema es conocido como **interferencia**.

Para manejar el problema, se debe asegurar que, mientras uno de los procesos utiliza un recurso compartido, el resto no puede acceder al mismo (**exclusion mutua**).

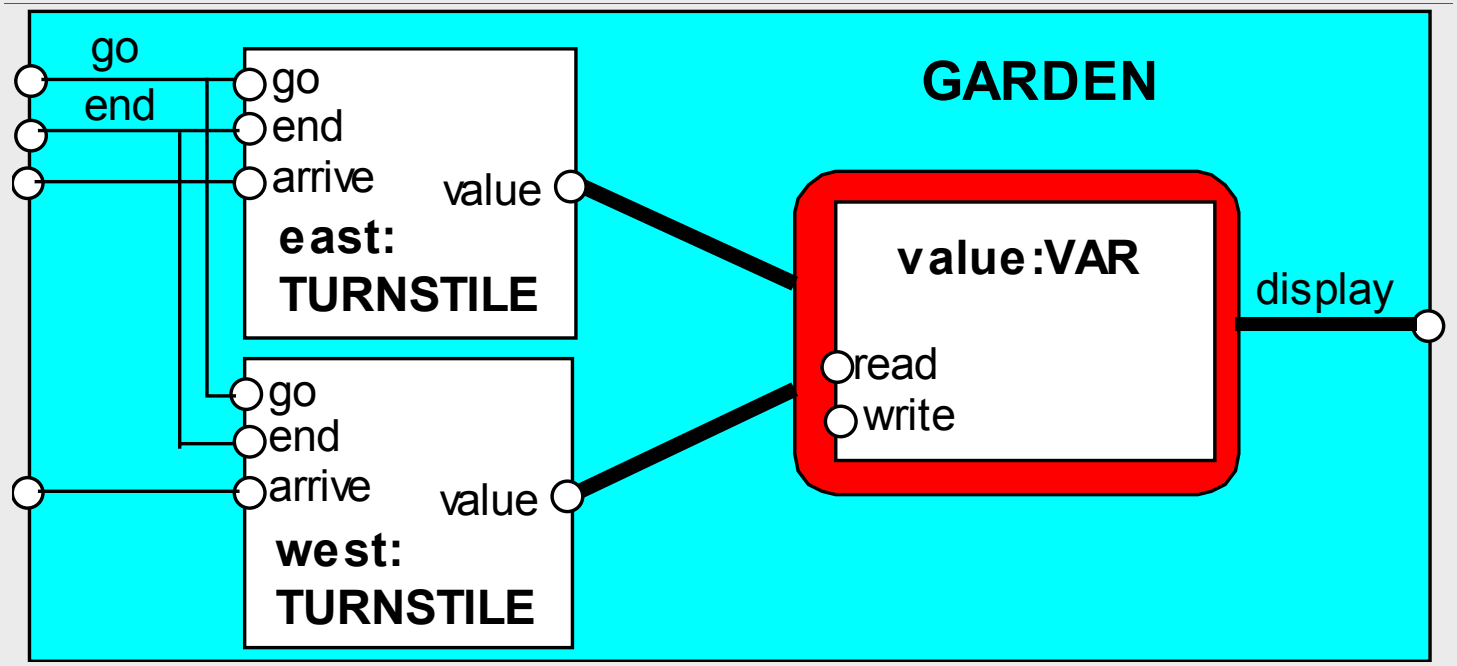
Un ejemplo de interferencia

Consideremos el problema del jardín ornamental. Tenemos un jardín ornamental con dos entradas (este y oeste), con sendas puertas giratorias. Se desea contar el número de personas que visitaron el jardín.



Un ejemplo de interferencia

Un modelo simple del problema consiste de tres procesos: dos para controlar (independientemente) las puertas y un contador para el número de personas (recurso compartido).



```
const N = 4
range T = 0..N
set VarAlpha = {value.{read[T],write[T]}}

VAR       = VAR[0],
VAR[u:T] = (read[u]   ->VAR[u]
           |write[v:T]->VAR[v]).

TURNSTILE = (go     -> RUN),
RUN       = (arrive -> INCREMENT
           |end    -> TURNSTILE),
INCREMENT = (value.read[x:T]
           -> value.write[x+1] -> RUN)
           +VarAlpha.

||GARDEN = (east:TURNSTILE || west:TURNSTILE
           || {east,west,display}::value:VAR)
            /{go /{east,west}.go,
              end/{east,west}.end}.
```

```
const N = 4
range T = 0..N
set VarAlpha = {value.{read[T],write[T]}}

VAR       = VAR[0],
VAR[u:T] = (read[u]    ->VAR[u]
           |write[v:T]->VAR[v]).

TURNSTILE = (go     -> RUN),
RUN       = (arrive -> INCREMENT
           |end    -> TURNSTILE),
INCREMENT = (value.read[x:T]
           -> value.write[x+1] -> RUN)
           +VarAlpha.

||GARDEN = (east:TURNSTILE || west:TURNSTILE
           || {east,west,display}::value:VAR)
            /{go /{east,west}.go,
              end/{east,west}.end}.
```

El alfabeto de la variable compartida se declara explicitamente ...

... para extender el alfabeto del proc. `TURNSTILE` y así asegurar que `VAR` no realiza acciones autonomamente (ej. `value.write[0]`). Todas las acciones de la var. compartida deben ser controladas

Detección de errores

Para poder comprobar si el modelo anterior funciona de la manera esperada o no, podemos combinarlo con un proceso que detecte la actualización incorrecta del recurso compartido:

```
TEST        = TEST[0],
TEST[v:T]   = (when (v<N){east.arrive,west.arrive}->TEST[v+1]
              | end->CHECK[v]
              ),
CHECK[v:T] = (display.value.read[u:T] ->
                (when (u==v) right -> TEST[v]
                |when (u!=v) wrong -> ERROR)
              )+{display.VarAlpha}.
```

Detección de errores

Para poder comprobar si el modelo anterior funciona de la manera esperada o no, podemos combinarlo con un proceso que detecte la actualización incorrecta del recurso compartido:

```
TEST         = TEST[0],
TEST[v:T]    = (when (v<N){east.arrive,west.arrive}->TEST[v+1]
                | end->CHECK[v]
                ),
CHECK[v:T]   = (display.value.read[u:T] ->
                  (when (u==v) right -> TEST[v]
                  |when (u!=v) wrong -> ERROR)
                )+{display.VarAlpha}.
```

Detección de errores

```
Trace to property violation in TEST
  go
  east.arrive
  east.value.read.0
  west.arrive
  west.value.read.0
  east.value.write.1
  west.value.write.1
  end
  display.value.read.1
  wrong
```

Modelado de la exclusión mutua

Una forma simple de manejar exclusion mutua es mediante una "traba":

```
LOCK = (acquire -> release -> LOCK).
```

Para el caso de nuestro ejemplo, esta llave puede usarse para garantizar el acceso exclusivo al recurso compartido VAR:

```
...
LOCK = (acquire->release->LOCK).
||LOCKVAR = (LOCK || VAR).

TURNSTILE = (go    -> RUN),
RUN       = (arrive-> INCREMENT
             |end   -> TURNSTILE),
INCREMENT = (value.acquire
              -> value.read[x:T] -> value.write[x+1]
              -> value.release -> RUN
             )+VarAlpha.
...
```

Modelado de la exclusión mutua

Una forma simple de manejar exclusion mutua es mediante una "traba":

```
LOCK = (acquire -> release -> LOCK).
```

Para el caso de nuestro ejemplo, esta llave puede usarse para garantizar el acceso exclusivo al recurso compartido VAR:

```
...
LOCK = (acquire->release->LOCK).
||LOCKVAR = (LOCK || VAR).

TURNSTILE = (go    -> RUN),
RUN       = (arrive-> INCREMENT
             |end   -> TURNSTILE),
INCREMENT = (value.acquire
              -> value.read[x:T] -> value.write[x+1]
              -> value.release -> RUN
             )+VarAlpha.
...
```

Monitores y otros TAD de sincronización

Los **monitores** [Brinch Hansen 72, Hoare 74] encapsulan datos que solo pueden modificarse y observarse a través de procedimientos.

Además, garantizan que sólo uno de tales procedimientos a la vez accede a los datos ocultos en estas estructuras.

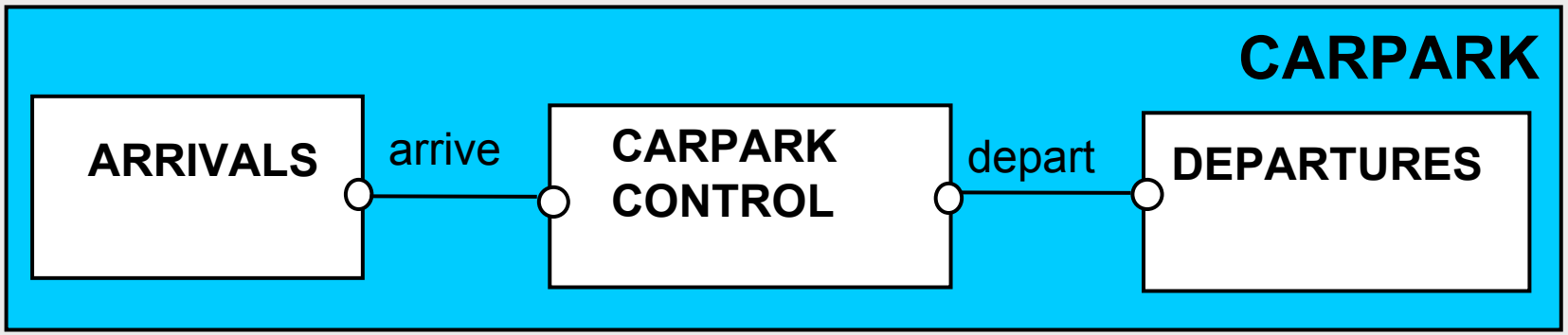
Otros TADs para sincronizar procesos son:

- los **semáforos** y
- los **buffers**.

Estos podrán verse como monitores particulares (pero no necesariamente lo son).

Sincronización condicional

Ejemplo: Se requiere un controlador para un estacionamiento de automóviles que sólo permita entrar un auto si el estacionamiento no está lleno y (obviamente) no permita salir autos si no hay ninguno estacionado.



Invariante de un monitor

Un **invariante para un monitor** es una aserción sobre las variables que éste encapsula.

Esta aserción **debe** valer siempre que ningún hilo esté ejecutando algún procedimiento del monitor.

- ¿Cuál es el invariante del controlador?
- ¿Para qué sirve este invariante?

Invariante de un monitor

El invariante del monitor nos permite derivar las condiciones de sincronización.

Invariante: $0 \leq espacio\_libre \leq capacidad$

En el modelo, `N` es la capacidad del estacionamiento.

Ir or

En el modelo son **i** y **N** respectivamente.

El invariante del monitor no permite derivar las condiciones de sincronización.

Invariante: $0 \leq espacio\_libre \leq capacidad$

```
CARPARKCONTROL(N=4) = SPACES[N],
SPACES[i:0..N] = (when(i>0) arrive->SPACES[i-1]
                 |when(i<N) depart->SPACES[i+1]
                 ).

ARRIVALS    = (arrive->ARRIVALS).

DEPARTURES = (depart->DEPARTURES).

||CARPARK = (ARRIVALS||CARPARKCONTROL(4)||DEPARTURES).
```

En el modelo, **N** es la capacidad del estacionamiento.

Relación con los verdaderos monitores

```
monitor carparkcontrol {
  const N=4
  int i
  condition free_space
  condition car_parked

  procedure arrive() {                    // when(i>0) arrive->SPACES[i-1]
    if not(i>0) then wait(free_space)
    i := i-1
    notify(car_parked)
  }

  procedure depart() {                    // when(i<N) depart->SPACES[i+1]
    if not(i<N) then wait(car_parked)
    i := i+1
    notify(free_space)
  }
}
```

Semáforos [Dijkstra 68]

- Es uno de los primeros mecanismos propuestos para sincronización de procesos.
- Un semáforo es un TAD que consta de una variable protegida que toma valores enteros no negativos y que sólo se puede acceder a través de dos operaciones: down y up (o P y V):

  - *down*(s): `when` $s > 0$ `do` s`--`
  - *up*(s): s`++`

(Importante: la operación **`when...do...`** es bloqueante.)

¿Cuál es el invariante de un semáforo?

Modelo del semáforo

```
const Max = 3
range Int = 0..Max

SEMAPHORE(N=0) = SEMA[N],
SEMA[v:Int]    = (up->SEMA[v+1]
                 |when(v>0) down->SEMA[v-1]
                 ).
```

Invariante: $0 \leq s$

Es conveniente considerar un invariante mas fuerte dado que usualmente los semáforos están asociados a un tipo de recurso y cuentan cuantos de estos quedan disponibles.

Modelo del semáforo (cont.)

```
const Max = 3
range Int = 0..Max

SEMAPHORE(N=0) = SEMA[N],
SEMA[v:Int]    = (up->SEMA[v+1]
                  |when(v>0) down->SEMA[v-1]
                  ),
SEMA[Max+1]    = ERROR.
```

Invariante: $0 \leq s \leq \texttt{Max}$

Notar que la especificación explícita del estado **ERROR** en FSP no es necesaria dado que éste (y en consecuencia, el compilador a través de la herramienta LTSA) mapea todo estado indefinido al estado **ERROR**. Por consiguiente, la especificación previa bastará.

Modelo del semáforo (cont.)

```
const Max = 3
range Int = 0..Max

SEMAPHORE(N=0) = SEMA[N],
SEMA[v:Int]    = (up->SEMA[v+1]
                 |when(v>0) down->SEMA[v-1]
                 ),
SEMA[Max+1]    = ERROR.
```

Invariante: $0 \leq s \leq \text{Max}$

Produce un estado erróneo y LTSA avisa de ello.

Notar que la especificación explícita del estado **ERROR** en FSP no es necesaria dado que éste (y en consecuencia, el compilador a través de la herramienta LTSA) mapea todo estado indefinido al estado **ERROR**. Por consiguiente, la especificación previa bastará.

Buffers acotados

Es una estructura de datos utilizada para regular la velocidad de transferencia entre el productor de la información y el consumidor de esta.

- Ejemplos de uso?
- Invariante?

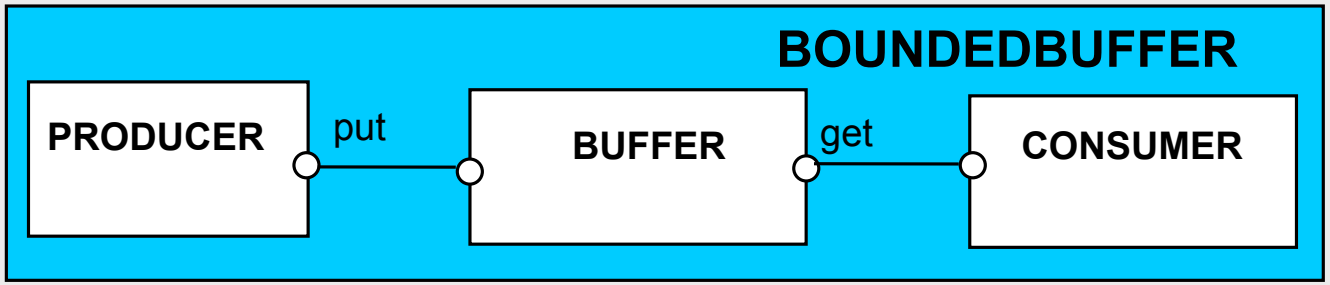
Ejemplo: Productor/Consumidor (ya lo conocen)

Observación para un modelado apropiado: El comportamiento del productor, del consumidor y del buffer mismo no es afectado por el valor de los datos manipulados; es decir, ninguna de las componentes testea o controla estos datos para su funcionamiento correcto.

Este comportamiento se dice **independiente de los datos**

=> abstraer los datos

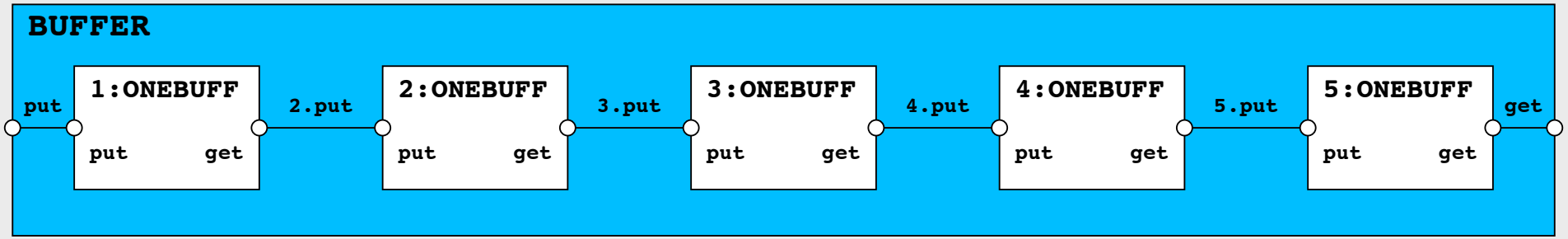
Buffers acotados




```
BUFFER(N=5) = COUNT[0],
COUNT[i:0..N]
      = (when (i<N) put->COUNT[i+1]
        |when (i>0) get->COUNT[i-1]
        ).

PRODUCER = (put->PRODUCER).
CONSUMER = (get->CONSUMER).

||BOUNDEDBUFFER = (PRODUCER||BUFFER(5)||CONSUMER).
```

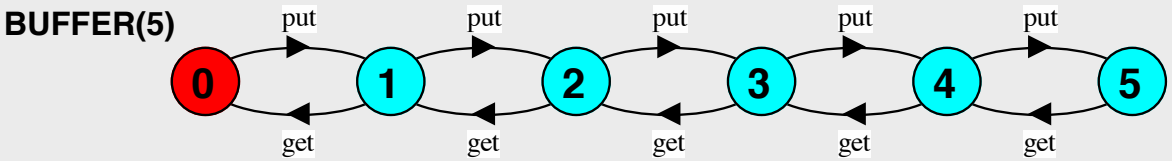
Notar las condiciones respecto del invariante.

Buffers acotados

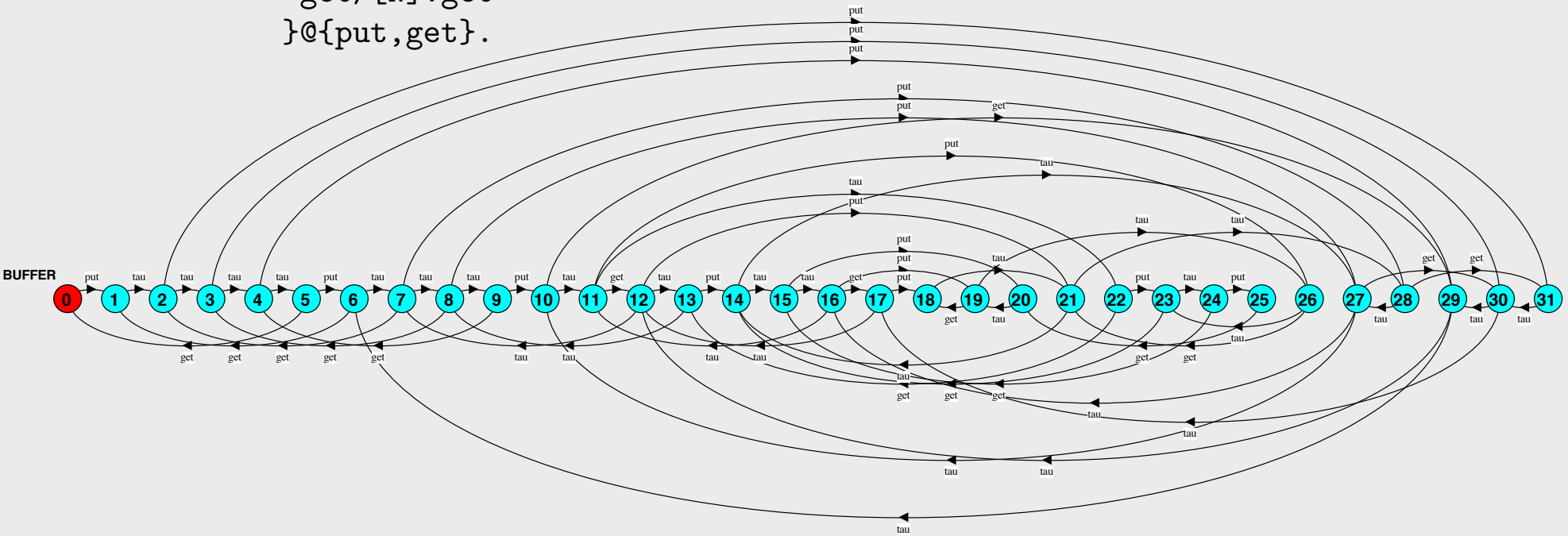



```
ONEBUFF = (put -> get -> ONEBUFF).

||BUFFER(N=5) = ([1..N]:ONEBUFF
                )/{put/[1].put,
                   [i:2..N].put/[i-1].get,
                   get/[N].get
                  }@{put,get}.

PRODUCER = (put->PRODUCER).
CONSUMER = (get->CONSUMER).

||BOUNDEDBUFFER = (PRODUCER||BUFFER(5)||CONSUMER).
```

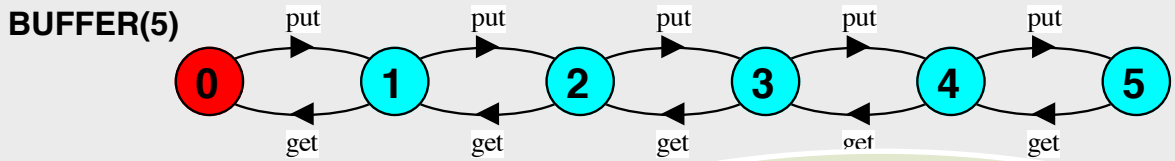
```
BUFFER(N=5) = COUNT[0],
COUNT[i:0..N]
      = (when (i<N) put->COUNT[i+1]
        |when (i>0) get->COUNT[i-1]
        ).
```



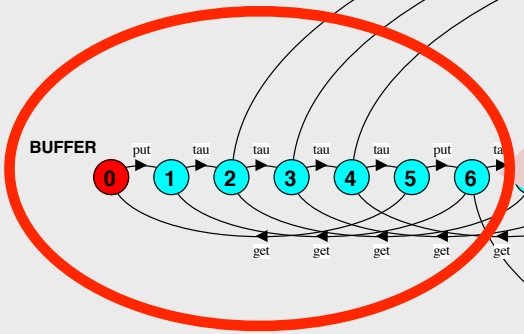

```
ONEBUFF = (put -> get -> ONEBUFF).

||BUFFER(N=5) = ([1..N]:ONEBUFF
               )/{put/[1].put,
                  [i:2..N].put/[i-1].get,
                  get/[N].get
                 }@{put,get}.
```

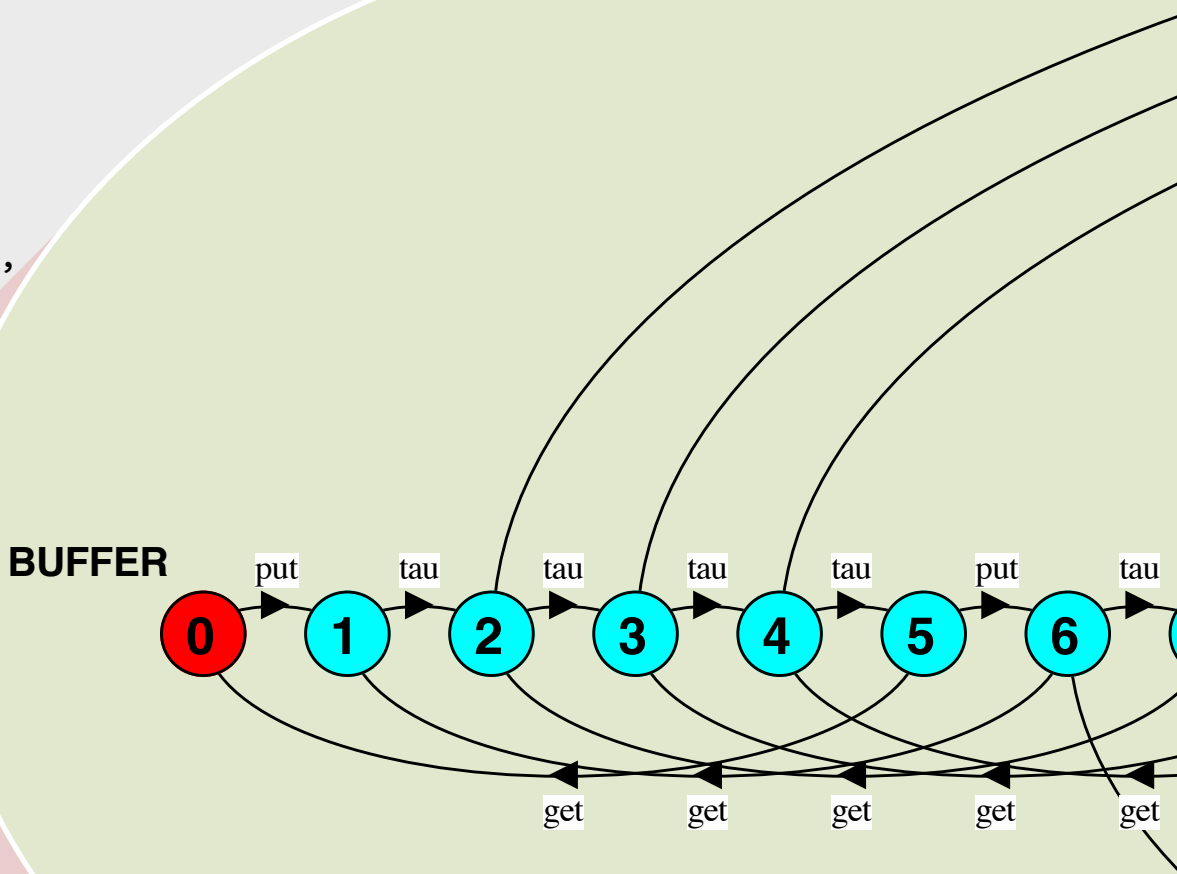


```
BUFFER(N=5) = COUNT[0],
COUNT[i:0..N]
      = (when (i<N) put->COUNT[i+1]
        |when (i>0) get->COUNT[i-1]
        ).
```

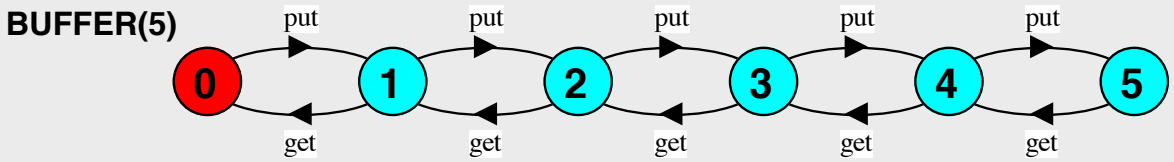



```
ONEBUFF = (put -> get -> ONEBUFF).

||BUFFER(N=5) = ([1..N]:ONEBUFF
               )/{put/[1].put,
                  [i:2..N].put/[i-1].get,
                  get/[N].get
                 }@{put,get}.
```



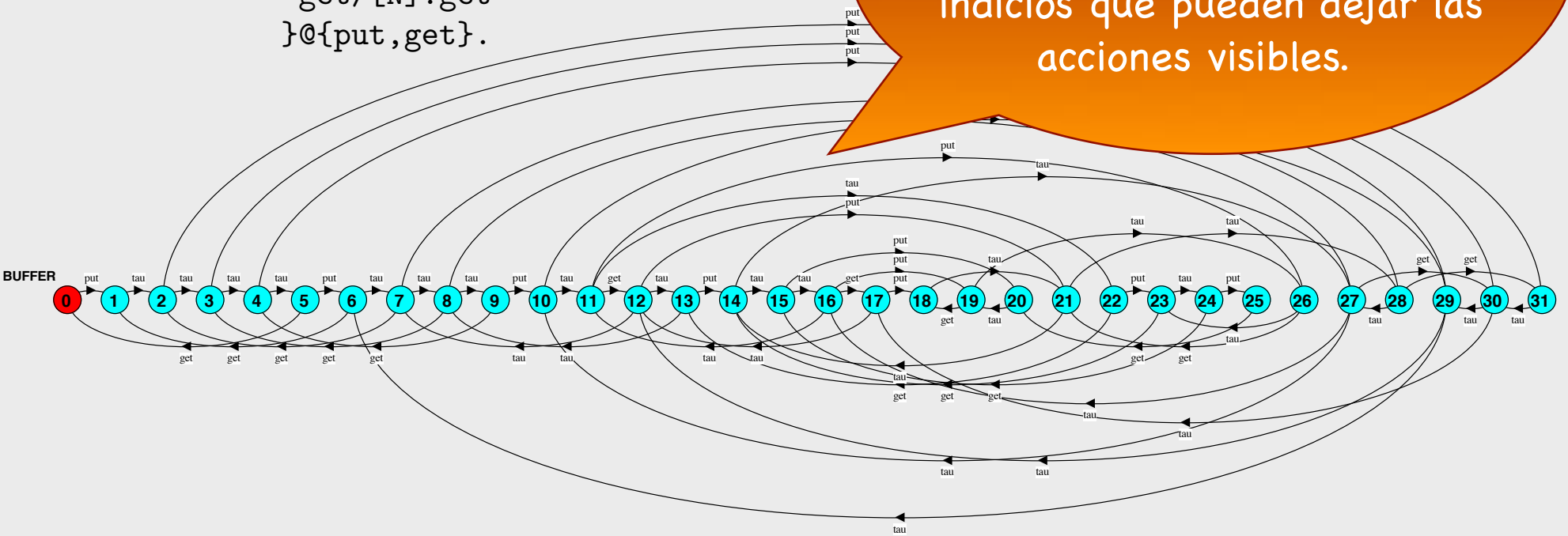


```
BUFFER(N=5) = COUNT[0],
COUNT[i:0..N]
      = (when (i<N) put->COUNT[i+1]
        |when (i>0) get->COUNT[i-1]
        ).
```

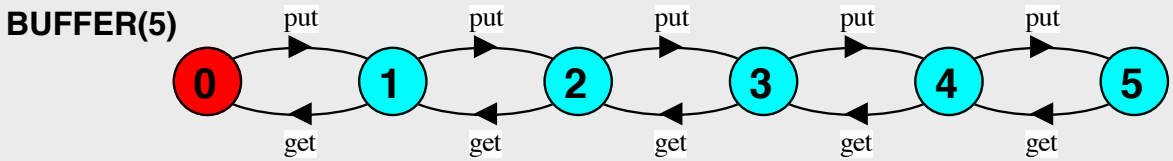



```
ONEBUFF = (put -> get -> ONEBUFF).

||BUFFER(N=5) = ([1..N]:ONEBUFF
               )/{put/[1].put,
                  [i:2..N].put/[i-1].get,
                  get/[N].get
                 }@{put,get}.
```

Recordar que las acciones tau son eventos internos que no pueden observarse sino a través de indicios que pueden dejar las acciones visibles.

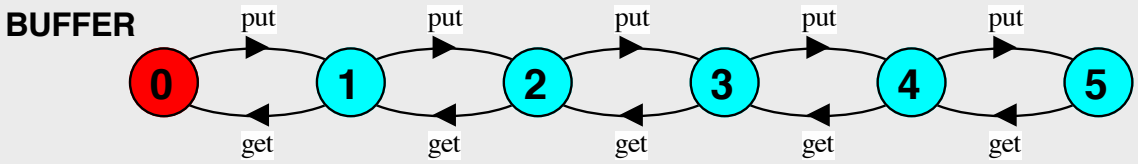


```
BUFFER(N=5) = COUNT[0],
COUNT[i:0..N]
      = (when (i<N) put->COUNT[i+1]
        |when (i>0) get->COUNT[i-1]
        ).
```



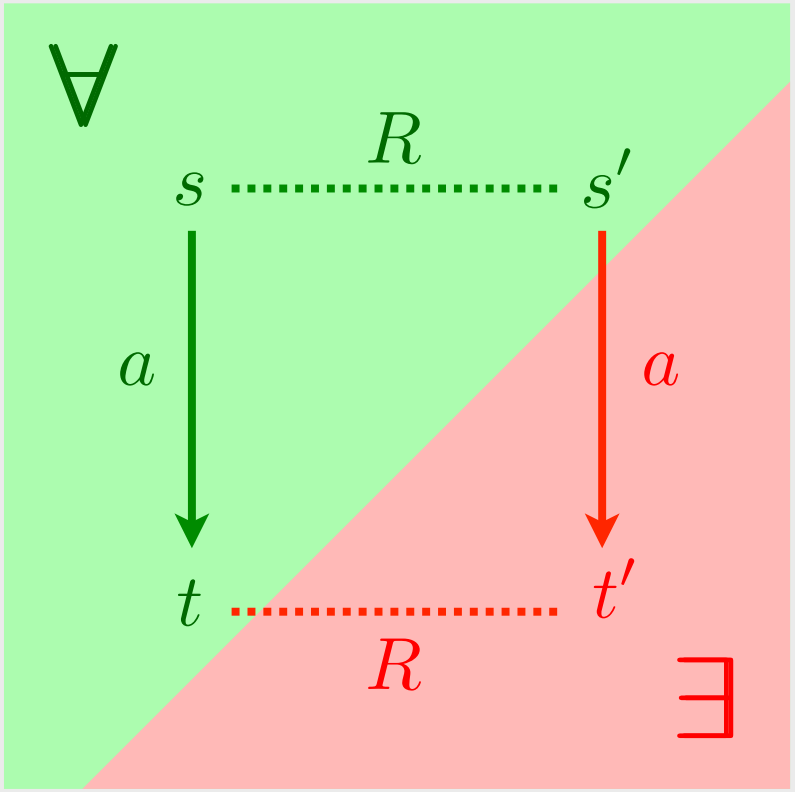

```
ONEBUFF = (put -> get -> ONEBUFF).

||BUFFER(N=5) = ([1..N]:ONEBUFF
               )/{put/[1].put,
                  [i:2..N].put/[i-1].get,
                  get/[N].get
                 }@{put,get}.
```

Luego de **abstraer** estos eventos internos, el LTS resulta isomorfo a la especificación anterior!!!

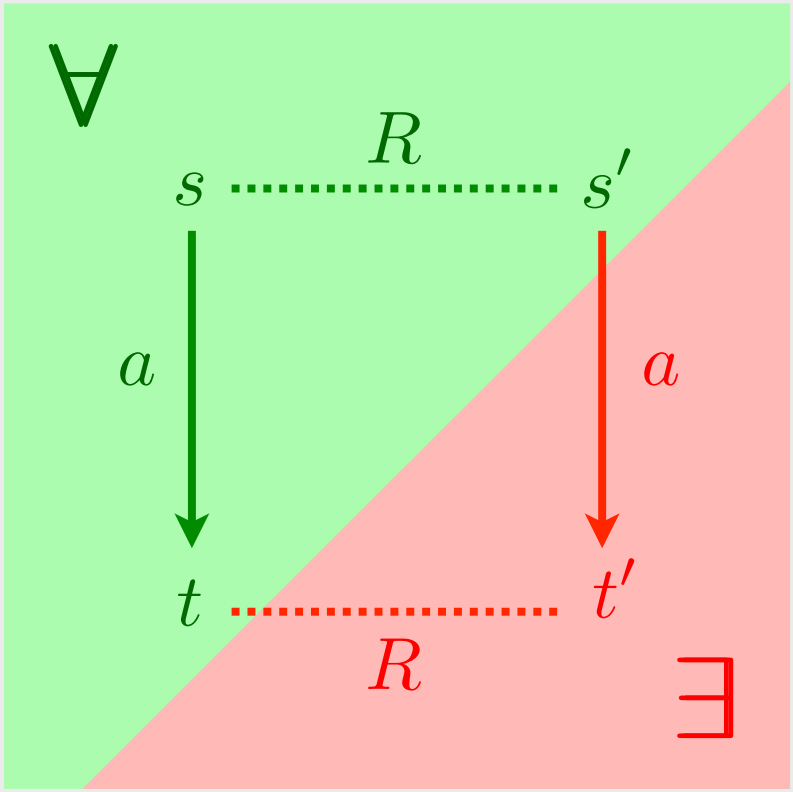


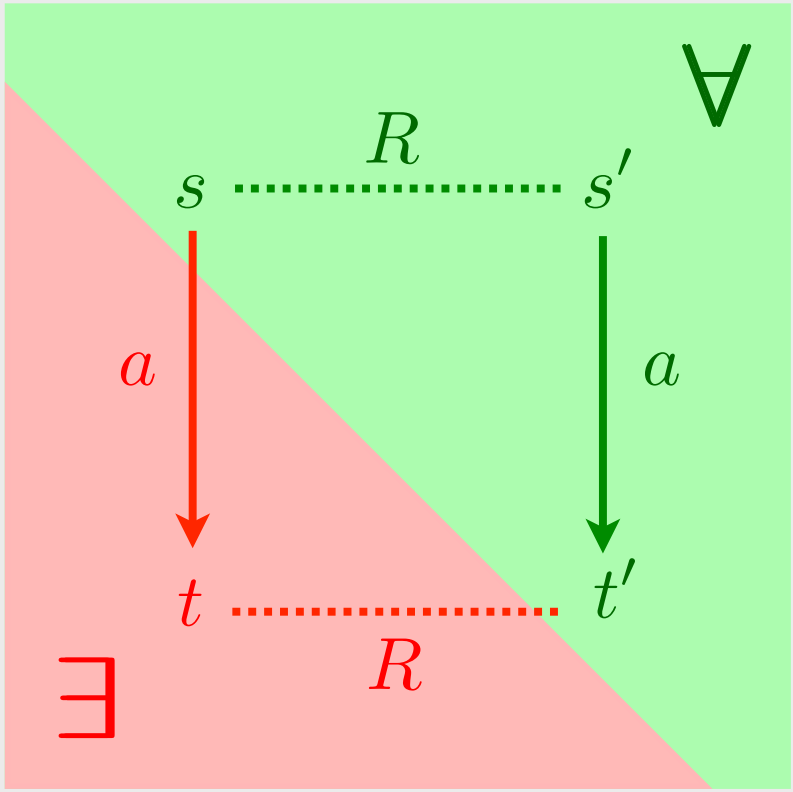
En LTSA, tal abstracción es una **minimización** que preserva la relación de equivalencia denominada **bisimulación débil**.

Simulación en LTS



Simulación en LTS





R^{-1} también debe satisfacer esta propiedad

Simulación en LTS

Sea $\mathcal{S} = (S, s_0, E, \rightarrow)$ un *sistema de transiciones etiquetadas*, i.e., $\rightarrow \subseteq S \times E \times S$.

Una relación $R \subseteq S \times S$ es una *simulación* si para todo par $(s,t) \in R$ y evento $a \in E$,

$$\forall s' : \; s \xrightarrow{a} s' \;\text{ implica }\; \exists t' : \; t \xrightarrow{a} t' \;\text{ y }\; (s', t') \in R$$

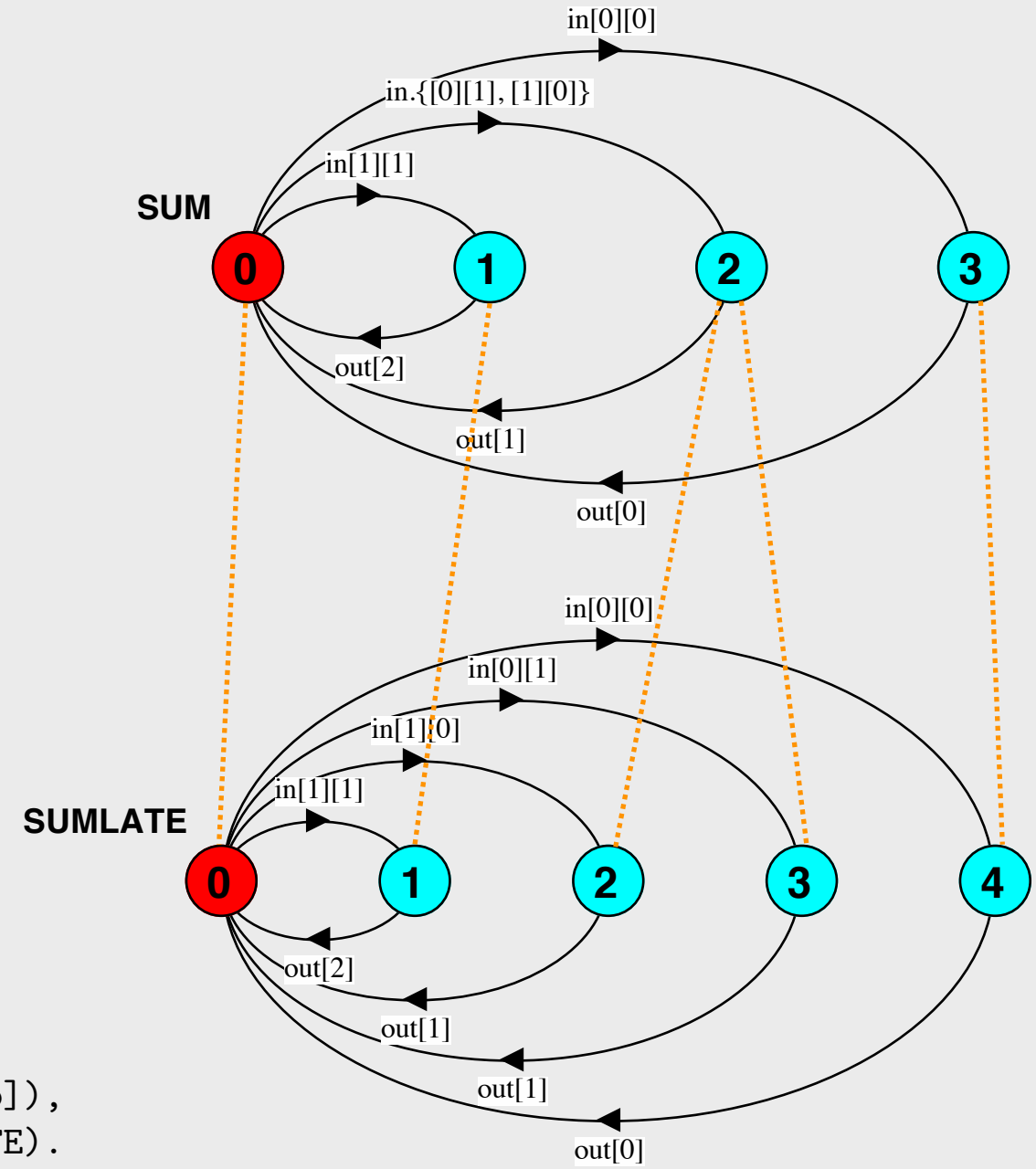
$R \subseteq S \times S$ es una *bisimulación* si R y R^{-1} son ambas simulaciones.

Bisimulación en LTS

```
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).
```

indica la relación

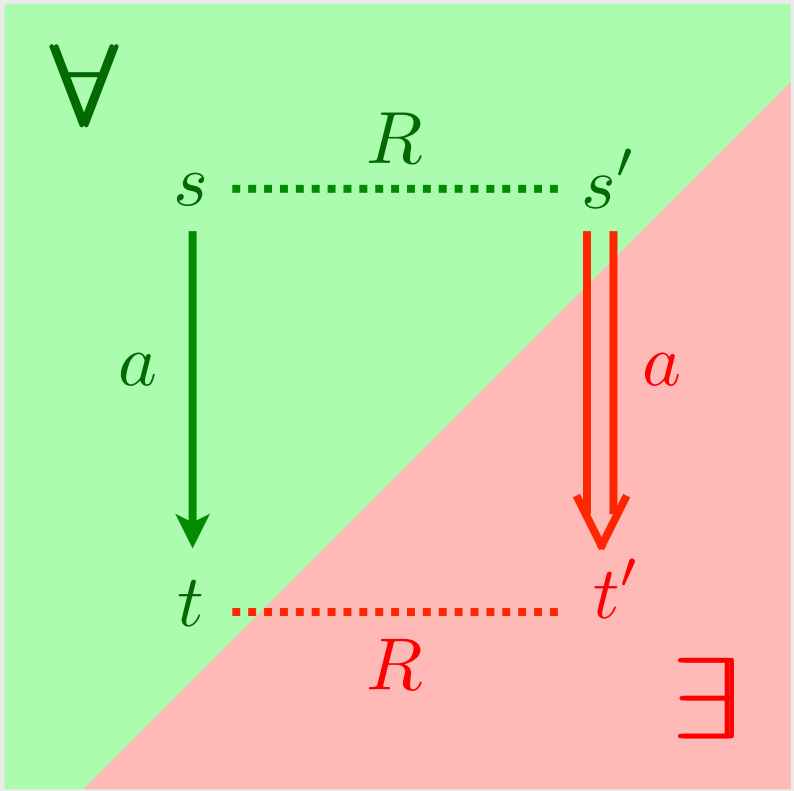
R de bisimulación

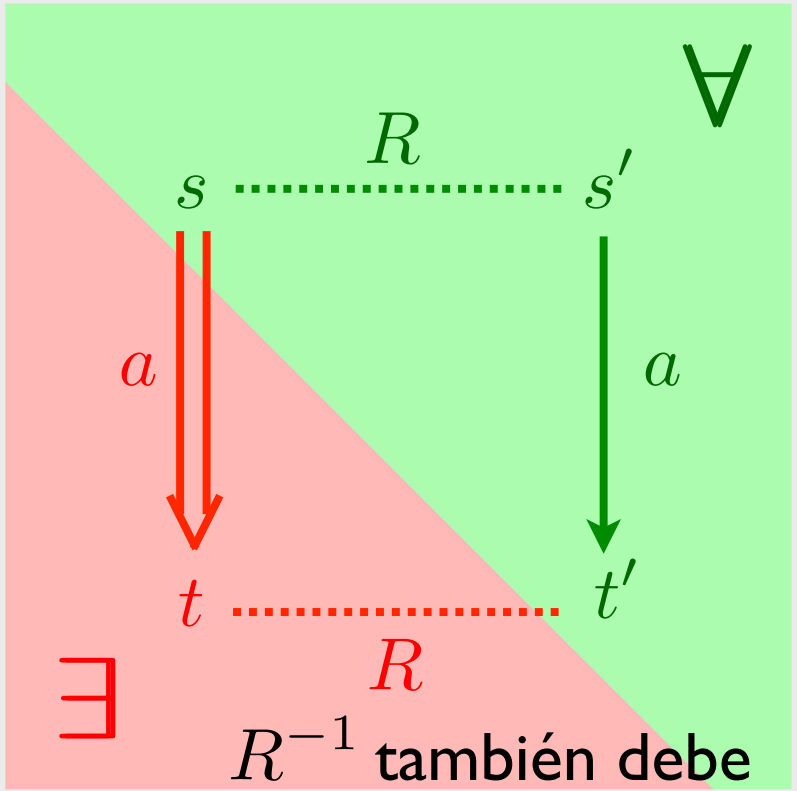



```
const N = 1
range T = 0..N

SUMLATE = (in[a:T][b:T] -> CARRY[a][b]),
CARRY[s:T][t:T] = (out[s+t] -> SUMLATE).
```

Bisimulación débil en LTS

$s \overset{a}{\Longrightarrow} t$ se define como $\begin{cases} s \ (\overset{\texttt{tau}}{\longrightarrow})^* \ t & \text{si } a = \texttt{tau} \\ s \ (\overset{\texttt{tau}}{\longrightarrow})^* \overset{a}{\longrightarrow} (\overset{\texttt{tau}}{\longrightarrow})^* \ t & \text{si } a \neq \texttt{tau} \end{cases}$



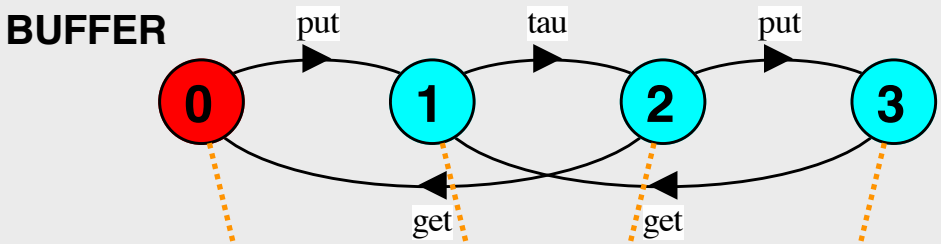


R^{-1} también debe satisfacer esta propiedad

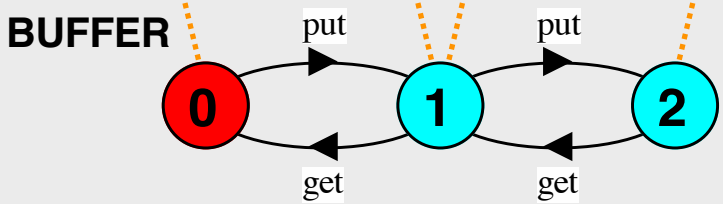
Bisimulación débil en LTS

```
ONEBUFF = (put -> get -> ONEBUFF).

||BUFFER(N=2) = ([1..N]:ONEBUFF
                )/{put/[1].put,
                   [i:2..N].put/[i-1].get,
                   get/[N].get
                  }@{put,get}.
```

LTS de BUFFER original



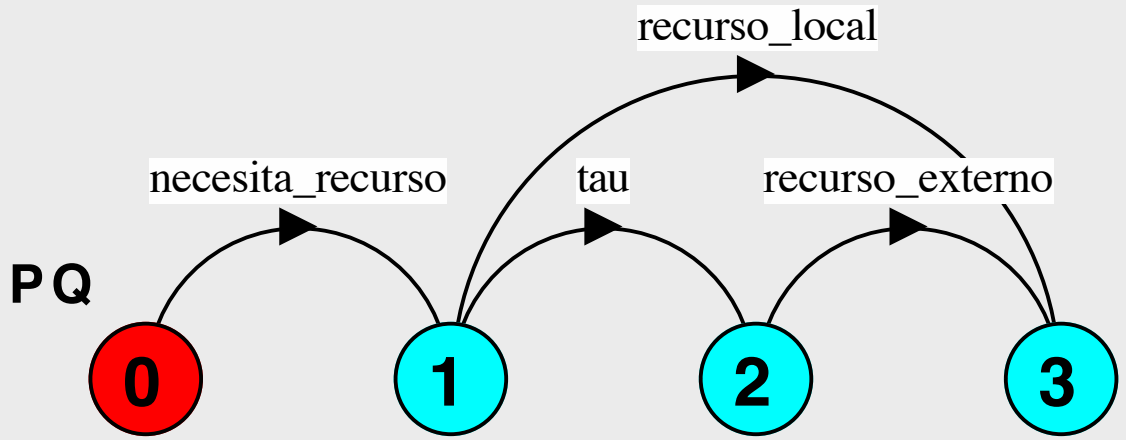
LTS de BUFFER minimizado



Bisimulación débil en LTS

Bajo bisimulación débil, no siempre es posible eliminar las acciones tau

```
P = ( necesita_recurso -> ( recurso_local -> STOP
                         | sync -> STOP)
    ).

Q = ( sync -> recurso_externo -> STOP).

||PQ = (P || Q)\{sync}.
```



El LTS de PQ ya está minimizado

Comunicación mediante pasaje de mensajes

Además de la comunicación mediante recursos compartidos y la sincronización mediante TADs específicos, los procesos pueden comunicarse y/o sincronizarse a través de pasaje de mensajes. Veremos tres esquemas de pasaje de mensajes, a través de canales.

- **Pasaje de mensajes sincrónico**

  En el pasaje de mensajes a través de canales sincrónicos, las operaciones que se consideran son:

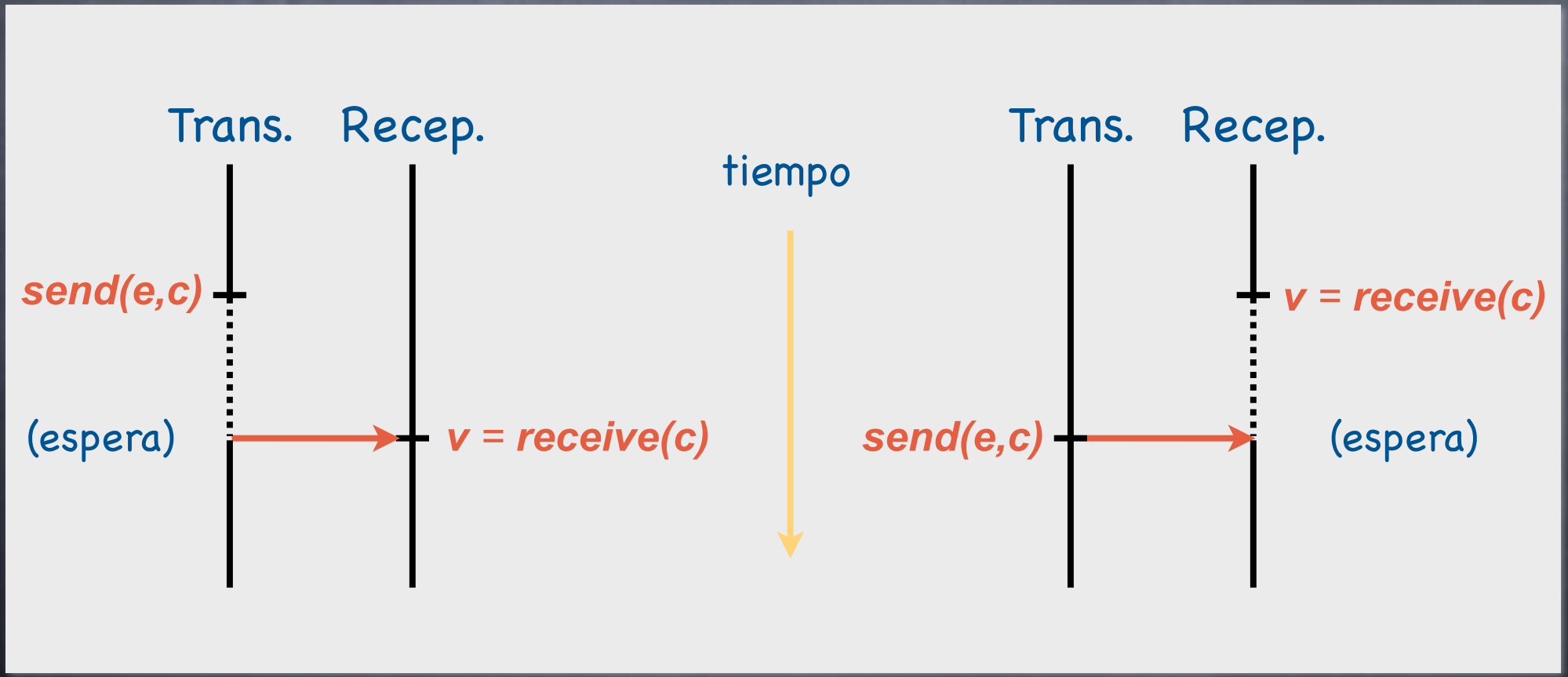
  - ***send(e,c)*** envía el valor de la expresión **e** a través del canal ***c***.
  - ***v = receive(c)*** recibe un dato a través del canal ***c*** y lo almacena en ***v***.

Comunicación mediante pasaje de mensajes

Pasaje de mensajes sincrónico

Cuando un proceso necesita ejecutar una operación de comunicación a través de un canal sincrónico (ya sea de envío o recepción), este se bloquea hasta tanto otro proceso realice la acción complementaria.

Luego de la comunicación, los procesos continúan independientemente.



Comunicación mediante pasaje de mensajes

Pasaje de mensajes sincrónico

Para modelar las operaciones de envio y recepción de mensajes a través de canales sincrónicos en FSP podemos usar acciones indexadas:

- ***send(e,chan)*** se implementa con **`chan.send[e]`**.
- ***v = receive(chan)*** se implementa con **`chan.receive[v:M]`**.

```
range M = 0..9

SENDER = SENDER[0],
SENDER[e:M] = (chan.send[e]->SENDER[(e+1)%10]).

RECEIVER = (chan.receive[v:M]->RECEIVER).

||SyncMsg = (SENDER || RECEIVER)/{chan/chan.{send,receive}}.
```

Comunicación mediante pasaje de mensajes

Pasaje de mensajes sincrónico

Para modelar las operaciones de envio y recepción de mensajes a través de canales sincrónicos en FSP podemos usar acciones indexadas:

- ***send(e,chan)*** se implementa con **`chan.send[e]`**.
- ***v = receive(chan)*** se implementa con **`chan.receive[v:M]`**.

```
range M = 0..9

SENDER = SENDER[0],
SENDER[e:M] = (chan.send[e]->SENDER[(e+1)%10]).

RECEIVER = (chan.receive[v:M]->RECEIVER).

||SyncMsg = (SENDER || RECEIVER)/{chan/chan.{send,receive}}.
```

Los eventos **send** y **receive** deben sincronizarse apropiadamente

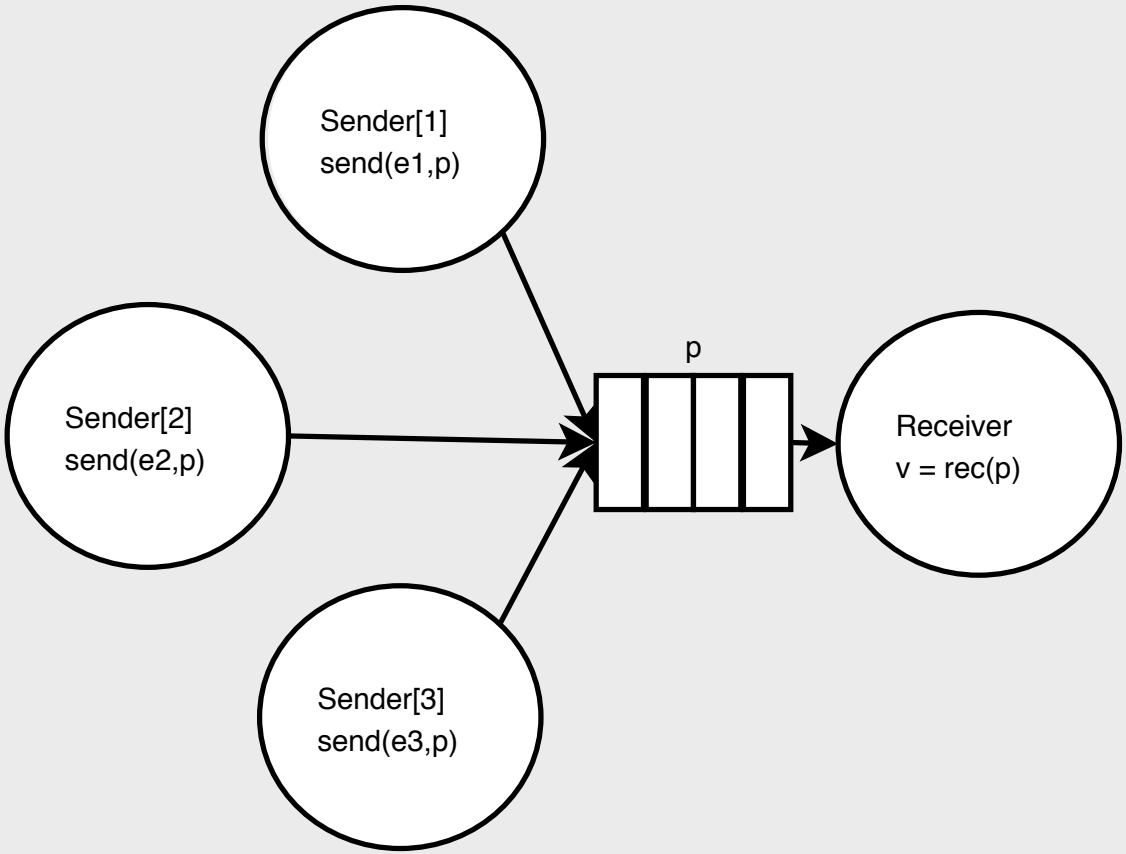
Comunicación mediante pasaje de mensajes
Recepción selectiva

- Permite recibir mensajes que llegan a través de distintos canales o inclusive otros eventos (ej: timeouts).
- Se implementa a través de la **elección** (con condiciones, si es necesario) y corresponde a un select de Ada u Occam.
- En C, la operación select se encuentra implementada en una de las librerías. (Hagan 'man select' en su xterm.)

Comunicación mediante pasaje de mensajes

Pasaje de mensajes asincrónico

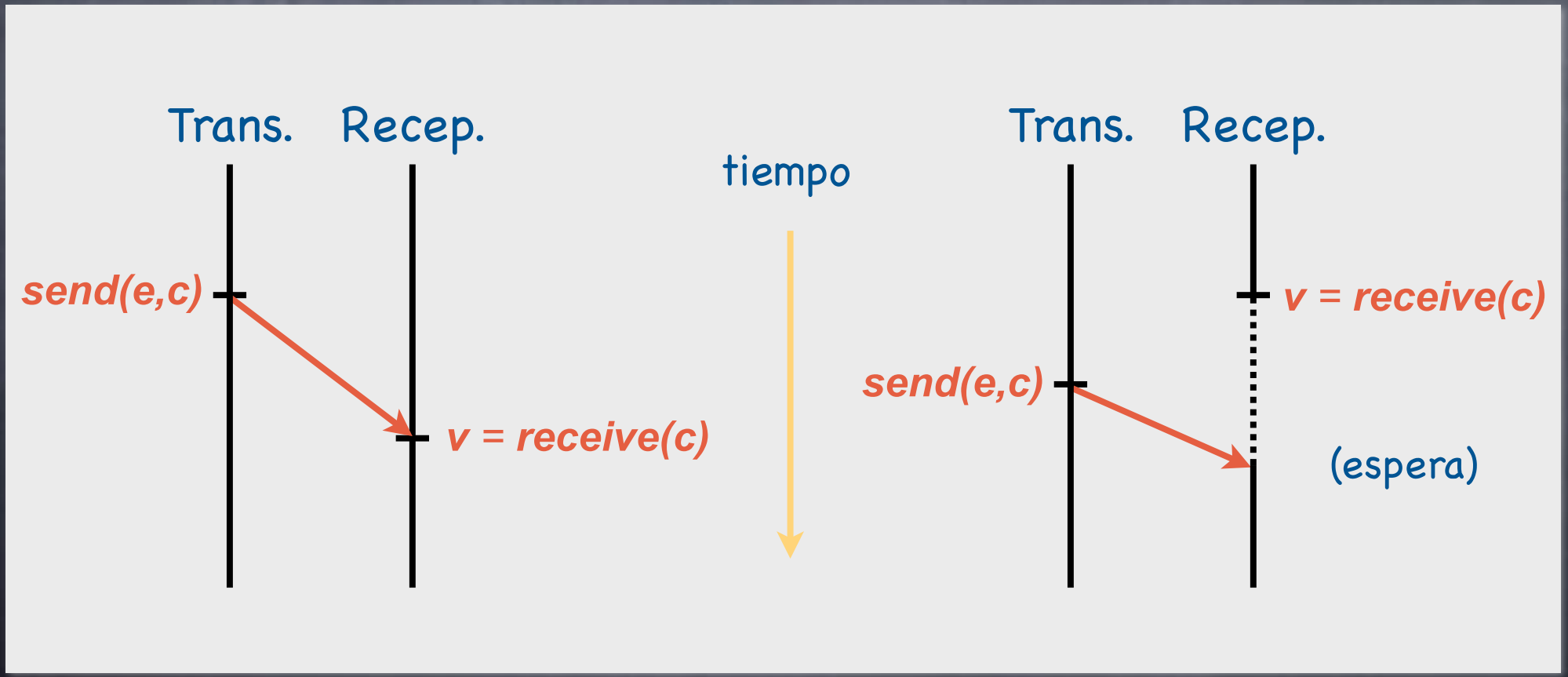
En el pasaje de mensajes asincrónico, la acción de envío no se bloquea, y los mensajes se almacenan en buffers, hasta que las acciones de recepción son ejecutadas.



Comunicación mediante pasaje de mensajes

Pasaje de mensajes asincrónico

Se implementan con las mismas primitivas ***send*** y ***receive*** que para el pasaje de mensaje sincrónico, sólo que ***send*** no es bloqueante.



Comunicación mediante pasaje de mensajes

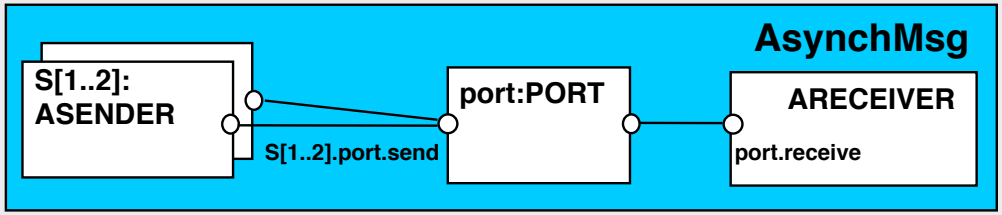
Pasaje de mensajes asincrónico

Para modelar las operaciones de envio y recepción de mensajes a través de canales asincrónicos en FSP debemos considerar buffers acotados (debe preservarse la finitud de los modelos). Para hacer esto puede utilizarse un modelo de puertos:

```
range M = 0..9
set   S = {[M],[M][M]}

PORT              //empty state, only send permitted
  = (send[x:M]->PORT[x]),
PORT[h:M]         //one message queued to port
  = (send[x:M]->PORT[x][h]
    |receive[h]->PORT
    ),
PORT[t:S][h:M]  //two or more  messages queued to port
  = (send[x:M]->PORT[x][t][h]
    |receive[h]->PORT[t]
    ).
```

Comunicación mediante pasaje de mensajes

Pasaje de mensajes asincrónico




```
range M = 0..9
set S = {[M],[M][M]}
PORT              = (send[x:M]->PORT[x]),
PORT[v:M]         = (send[x:M]->PORT[x][v]
                    |receive[v]->PORT),
PORT[s:S][v:M]    = (send[x:M]->PORT[x][s][v]
                    |receive[v]->PORT[s]).

ASENDER = ASENDER[0],
ASENDER[e:M] = (port.send[e]->ASENDER[(e+1)%10]).

ARECEIVER = (port.receive[v:M]->ARECEIVER).

||AsyncMsg = (s[1..2]:ASENDER || ARECEIVER||port:PORT)
             /{s[1..2].port.send/port.send}.

||Abstract = AsyncMsg
             /{s[1..2].port.send/s[1..2].port.send[M],
               port.receive/port.receive[M]
              }.
```

Comunicación mediante pasaje de mensajes

Pasaje de mensajes asincrónico

```
range M = 0..9
set   S = {[M],[M][M]}

PORT            //empty state, only send permitted
  = (send[x:M]->PORT[x]),
PORT[h:M]       //one message queued to port
  = (send[x:M]->PORT[x][h]
    |receive[h]->PORT
    ),
PORT[t:S][h:M]  //two or more  messages queued to port
  = (send[x:M]->PORT[x][t][h]
    |receive[h]->PORT[t]
    ).
```

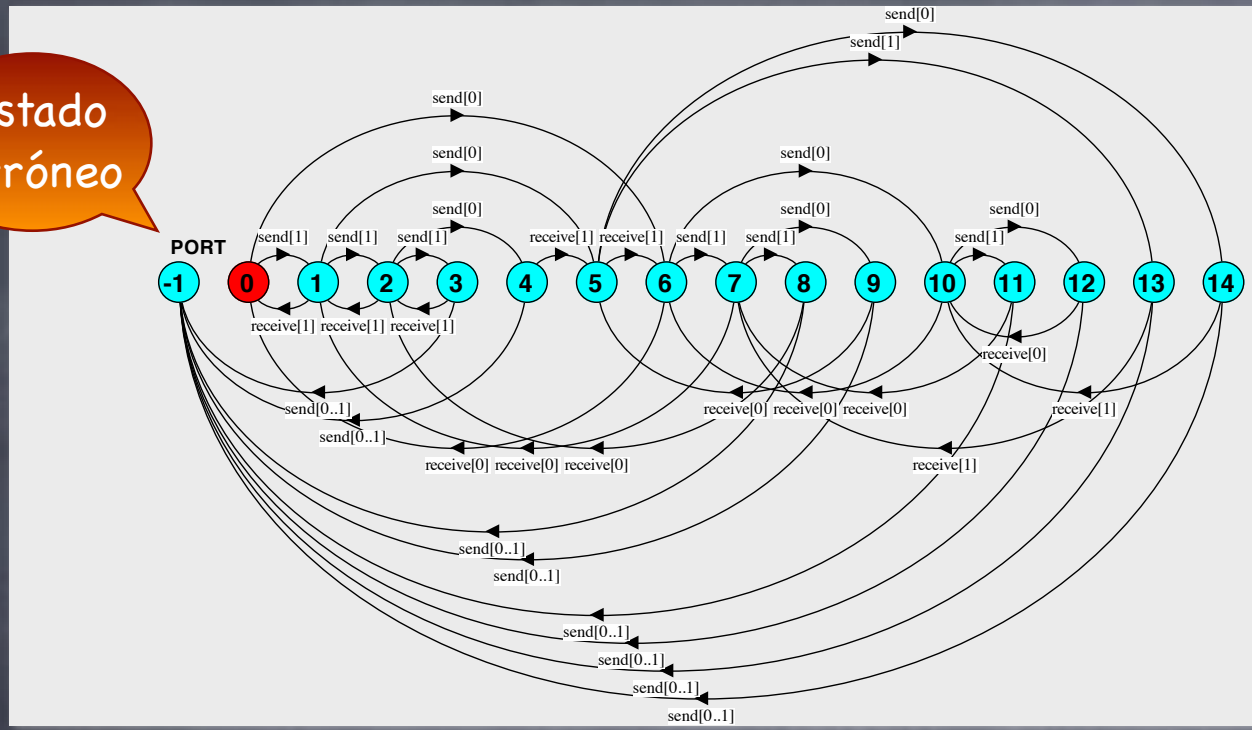
Canal como un buffer con overflow

```
range M = 0..9

ONEBUFF = (put[x:M] -> get[x] -> ONEBUFF).

||BUFFER(N=3) = ([1..N]:ONEBUFF
               )/{put/[1].put,
                  [i:2..N].put/[i-1].get,
                  get/[N].get
                 }@{put,get}.
```

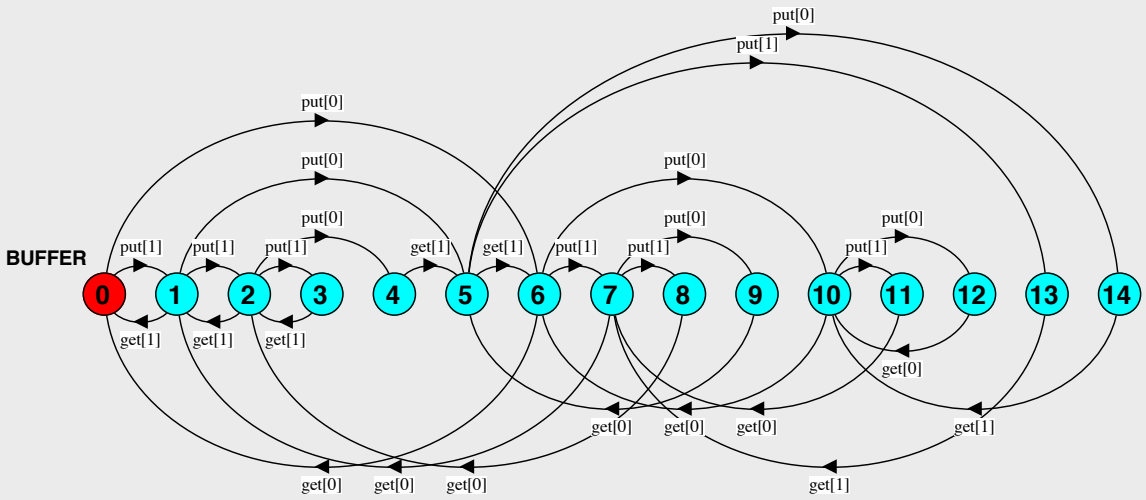
Canal como un buffer bloqueante

Comunicación mediante pasaje de mensajes

Pasaje de mensajes asincrónico



Canal como un buffer con overflow



Canal como un buffer bloqueante

Comunicación mediante pasaje de mensajes
Rendezvous (o Request-Reply)

Es un protocolo de pasaje de mensajes utilizado para soportar interacción cliente-servidor.

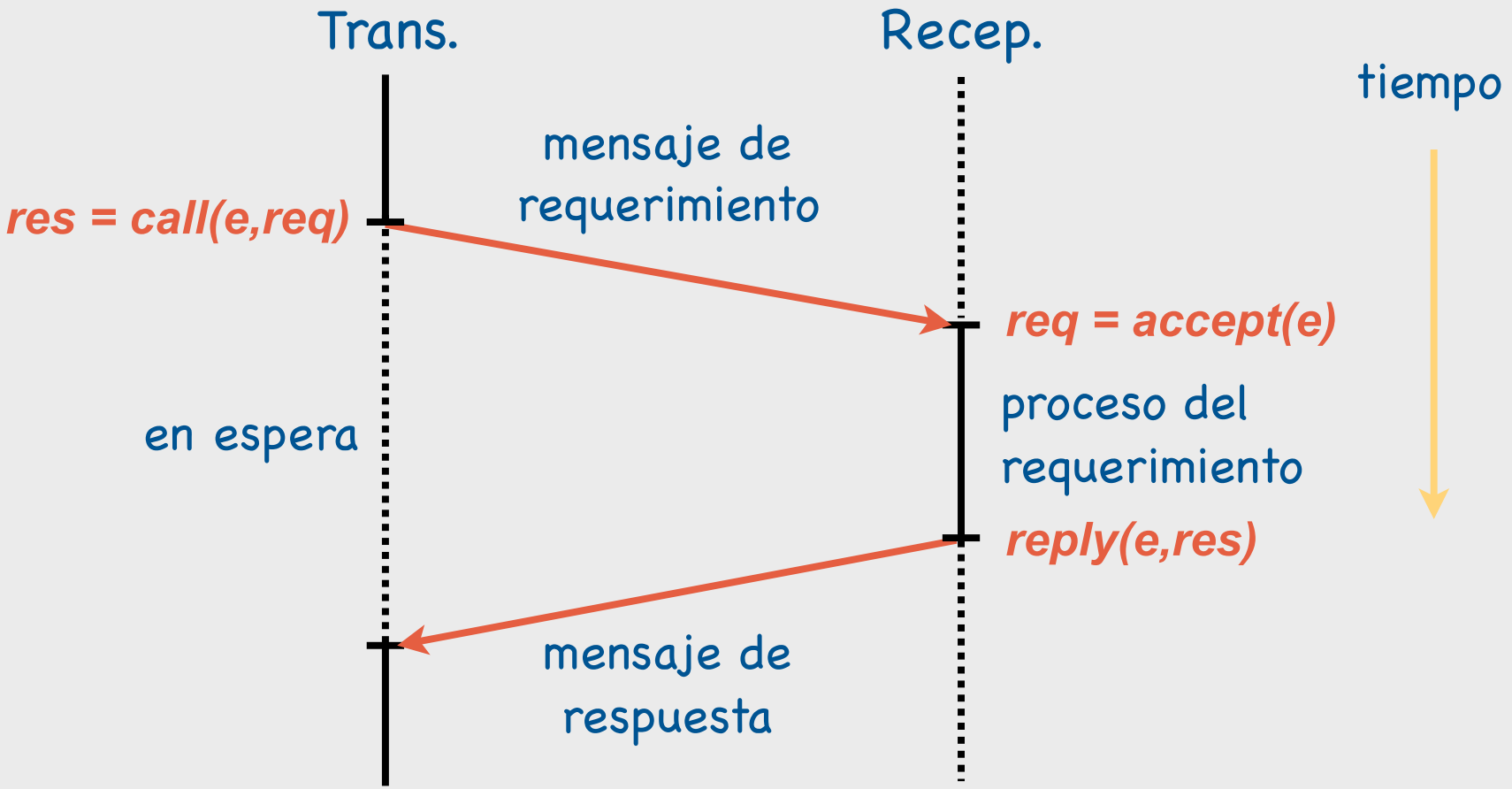
- ***res = call(e,req)*** envía el valor ***req*** estableciendo el requerimiento que se encolará en la entrada ***e***. Luego se bloquea hasta la ocurrencia del mensaje de respuesta, el que será recibido en la variable ***res***.

- ***req = accept(e)*** recibe el valor del requerimiento en ***e*** y lo almacena en ***req***. Si no hay requerimientos encolados en la entrada se bloquea a la espera de estos.

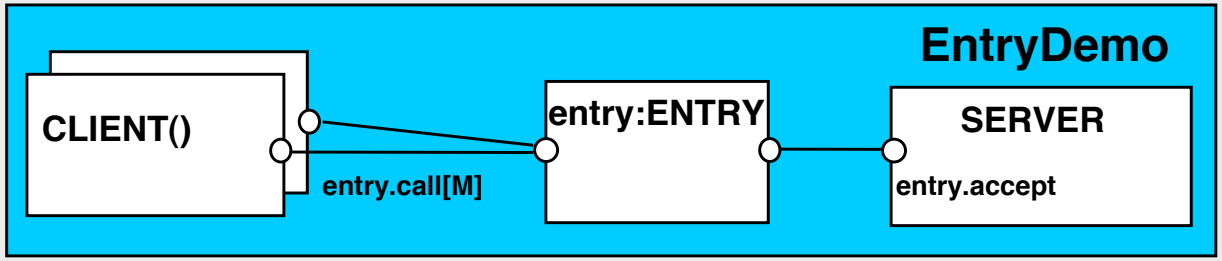
- ***reply(e,res)*** envia el resultado ***res*** como respuesta al mensaje entrado a través de ***e***.

Comunicación mediante pasaje de mensajes

Rendezvous



Comunicación mediante pasaje de mensajes
Rendezvous




```
set M = {replyA,replyB}
set S = {[M],[M][M]}
PORT              = (send[x:M]->PORT[x]),
PORT[v:M]         = (send[x:M]->PORT[x][v]
                    |receive[v]->PORT),
PORT[s:S][v:M]    = (send[x:M]->PORT[x][s][v]
                    |receive[v]->PORT[s]).

||ENTRY = PORT/{call/send, accept/receive}.

CLIENT(CH='reply) = (entry.call[CH]->[CH]->CLIENT).

SERVER = (entry.accept[ch:M]->[ch]->SERVER).

||EntryDemo = (CLIENT('replyA)||CLIENT('replyB)
                || entry:ENTRY || SERVER  ).
```