

Validación y Verificación de Software

**Propiedades de Sistemas
Concurrentes y su Análisis**

Propiedades de los sistemas concurrentes

Con frecuencia, los programas concurrentes suelen ser reactivos, y sus características diferentes de las de los programas secuenciales convencionales.

Por esto, las propiedades que en general se desea garantizar de programas concurrentes difieren de las propiedades de programas secuenciales. Algunas de estas son:

- No violación de **invariantes** de sistema

- Ausencia de **starvation**

- Ausencia de **deadlock**

Categorías de propiedades

Una clasificación clásica de las propiedades de los sistemas reactivos incluye las siguientes cuatro categorías:

Propiedades de **alcanzabilidad** (“reachability”)

Propiedades de **seguridad** (“safety”)

Propiedades de **vitalidad** (“liveness”)

Propiedades de **equidad** (“fairness”)

Propiedades de alcanzabilidad (“reachability”)

“Es posible que el sistema llegue a algún estado de un conjunto dado”

Por ejemplo:

- una (o alguna) componente entra a la region crítica

- es posible llegar a un estado de ERROR

En este último caso nos interesa que se cumpla la negación de la propiedad. Es usual que la propiedad de interés sea la negación de la alcanzabilidad.

Propiedades de seguridad ("safety")

"Nunca va a pasar nada malo"

Por ejemplo:

- no es posible llegar a un estado de deadlock

- se garantiza exclusión mutua

- el sistema preserva un invariante dado

Propiedades de seguridad ("safety")

"Nunca va a pasar nada malo"

Por ejemplo:

- no es posible llegar a un estado de deadlock

- se garantiza exclusión mutua

- el sistema preserva un invariante dado

En general, la negación de una propiedad safety es una propiedad de alcanzabilidad

Propiedades de vitalidad ("liveness")

"Siempre es posible que algo bueno ocurra"

Por ejemplo:

un (sub)proceso dado termina su ejecución
es posible alcanzar un estado de estabilidad
si se llama al ascensor, este llegará en algún
momento

Propiedades de Equidad ("fairness")

"Siempre ocurrirá algo de manera frecuente"

Por ejemplo:

siempre que un proceso esté esperando para entrar a una región crítica, entonces logrará entrar

siempre que un proceso solicite periódicamente un recurso finalmente se le será asignado.

un proceso dado no sufre de inanición

Por qué categorizar

Las razones dependen de los distintos puntos de vista:

Metodología de especificación:

Nos ayuda a buscar las preguntas correctas que debemos hacerle a nuestro sistema, por ejemplo:

¿qué necesito para que mi sistema no llegue a una situación no deseada?
(safety)

¿qué se debe cumplir para que mi sistema progrese en su ejecución?
(liveness).

Metodología de modelado:

En la tarea de modelado uno hace más hincapié en ciertos aspectos de acuerdo al objetivo de verificación.

Por ejemplo, se pueden aplicar distintos tipos de simplificaciones según se desee verificar una propiedad de safety o de liveness.

Propiedades como conjuntos de trazas

La semántica de procesos (en realidad, una de las semánticas más simples para procesos) puede definirse como el conjunto de todas sus ejecuciones. Cada ejecución de un proceso puede verse como la sucesión de eventos en los cuales el proceso se involucra (y en el orden en que lo hace).

Ejemplo: Consideremos el siguiente proceso:

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

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

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

La única traza posible para **MAKER** es:

```
make, ready, make, ready, make, ...
```

Propiedades como conjuntos de trazas

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

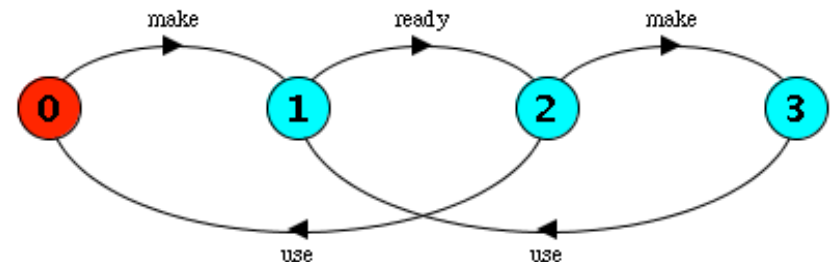
```
USER  = (ready->use->USER).
```

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

Algunas trazas posibles para **MAKER_USER** son:

make, ready, use, make, ready, use, ...

make, ready, make, use, ready, make, ready, use, ...



Propiedades como conjuntos de trazas

Al igual que los procesos, las propiedades tienen una semántica dada en términos de conjuntos de trazas. Una propiedad P se identifica con todas las sucesiones de eventos atómicos que exhiben la propiedad P .

Por ejemplo, la propiedad “**no** ocurren dos producciones (**make**) sucesivas” contiene las siguientes trazas:

use, use, use, use, use, use, use, ...

make, use, make, use, make, use, make, use, ...

ready, ready, ready, ready, ready, ...

Formalización de Propiedades

Lenguajes ω -regulares

Dado un lenguaje finito $A \subseteq \Sigma^*$ definimos

$$A^\omega \stackrel{\text{def}}{=} \{\sigma_1\sigma_2\sigma_3 \dots \mid \forall i \geq 0 : \sigma_i \in A \wedge \sigma_i \neq \epsilon\}$$

es decir, A^ω es el lenguaje conteniendo todas las concatenaciones infinitas de cadenas no vacías de A .

Formalización de Propiedades

Lenguajes ω -regulares

Dado un lenguaje finito $A \subseteq \Sigma^*$ definimos

$$A^\omega \stackrel{\text{def}}{=} \{\sigma_1\sigma_2\sigma_3 \dots \mid \forall i \geq 0 : \sigma_i \in A \wedge \sigma_i \neq \epsilon\}$$

También las denominaremos
fragmentos de trazas

es decir, A^ω es el lenguaje conteniendo todas las concatenaciones infinitas de cadenas no vacías de A .

Lenguajes ω -regulares

Un lenguaje L se dice ω -regular si existen lenguajes regulares A_i y B_i , $0 \leq i \leq k$, tal que $\epsilon \notin B_i \neq \emptyset$ y

$$L = \bigcup_{0 \leq i \leq k} A_i \cdot B_i^\omega$$

donde \cdot denota la concatenación habitual de lenguajes.

Propiedad: Los lenguajes ω -regulares son cerrados por union, intersección y complemento.

Lenguajes ω -regulares

no ocurren dos producciones (**make**)
sucesivas

Ejemplo:

La propiedad P sobre el proceso **MAKER_USER** puede escribirse como

Lenguajes ω -regulares

no ocurren dos producciones (**make**)
sucesivas

Ejemplo:

La propiedad P sobre el proceso **MAKER_USER** puede escribirse como

$$\left((\text{make} + \text{ready} + \text{use})^* \text{make make} (\text{make} + \text{ready} + \text{use})^\omega \right)$$

Formalización de propiedades safety

“Nunca va a pasar nada malo”

Observar:

Si una traza viola una propiedad de safety, lo hace en un “instante” finito.

Si un prefijo de una traza (infinita) viola una propiedad de safety, no hay forma de remediarlo (cualquiera sea la forma en que se continúe, la traza no dejará de violar la propiedad).

Formalización de propiedades safety

Un conjunto de trazas $P \subseteq \Sigma^\omega$ es una propiedad de safety si cumple

$$\forall \sigma : \sigma \notin P \Leftrightarrow \exists i \geq 0 : \forall \beta : \sigma[..i]\beta \notin P$$

o equivalentemente:

$$\forall \sigma : \forall i \geq 0 : \exists \beta : \sigma[..i]\beta \in P \Leftrightarrow \sigma \in P$$

donde $\sigma[..i]$ denota el prefijo i -ésimo de σ .

Formalización de propiedades liveness

“Siempre es posible que algo bueno ocurra”

Observar:

Ningún fragmento de traza (finito) viola una propiedad de liveness: Si el evento que se esperaba que ocurriera aún no lo hizo, puede suceder más adelante.

Es decir:

Un conjunto de trazas $P \subseteq \Sigma^*$ es una propiedad de liveness si cumple

$$\left((\text{make} + \epsilon) (\text{ready} + \text{use})^+ \right)^\omega$$

Ejemplos: $((\text{ready} + \text{use})^* \text{make} (\text{make} + \text{ready})^* \text{use})^\omega$

Formalización de propiedades liveness

“Siempre es posible que algo bueno ocurra”

Observar:

Ningún fragmento de traza (finito) viola una propiedad de liveness: Si el evento que se esperaba que ocurriera aún no lo hizo, puede suceder más adelante.

Esta propiedad de safety es equivalente a:

Es decir:

$$\frac{}{((\text{make} + \text{ready} + \text{use})^* \text{make} \text{make} (\text{make} + \text{ready} + \text{use})^\omega)}$$

Un conjunto de trazas $P \subseteq \Sigma^*$ es una propiedad de liveness si cumple

“Nunca ocurren dos eventos **make** seguidos”

$$\left((\text{make} + \epsilon) (\text{ready} + \text{use})^+ \right)^\omega$$

Ejemplos:

$$\left((\text{ready} + \text{use})^* \text{make} (\text{make} + \text{ready})^* \text{use} \right)^\omega$$

Formalización de propiedades liveness

“Siempre es posible que algo bueno ocurra”

Observar:

Ningún fragmento de traza (finito) viola una propiedad de liveness: Si el evento que se esperaba que ocurriera aún no lo hizo, puede suceder más adelante.

Esta propiedad de safety es equivalente a:

Es decir:

$$((\text{make} + \text{ready} + \text{use})^* \text{make} \text{make} (\text{make} + \text{ready} + \text{use})^\omega)$$

Esta propiedad de liveness es equivalente a

Un conjunto de trazas $P \subseteq \Sigma^*$ es una propiedad de liveness si cumple

“Nunca ocurren dos eventos **make** seguidos”

$$((\text{make} + \text{ready} + \text{use})^* \text{make} (\text{make} + \text{ready} + \text{use})^* \text{use})^\omega$$

“**make** y **use** ocurren infinitamente a lo largo de la ejecución”

$$((\text{make} + \epsilon) (\text{ready} + \text{use})^+)^{\omega}$$

Ejemplos:

$$((\text{ready} + \text{use})^* \text{make} (\text{make} + \text{ready})^* \text{use})^\omega$$

Análisis de propiedades en FSP

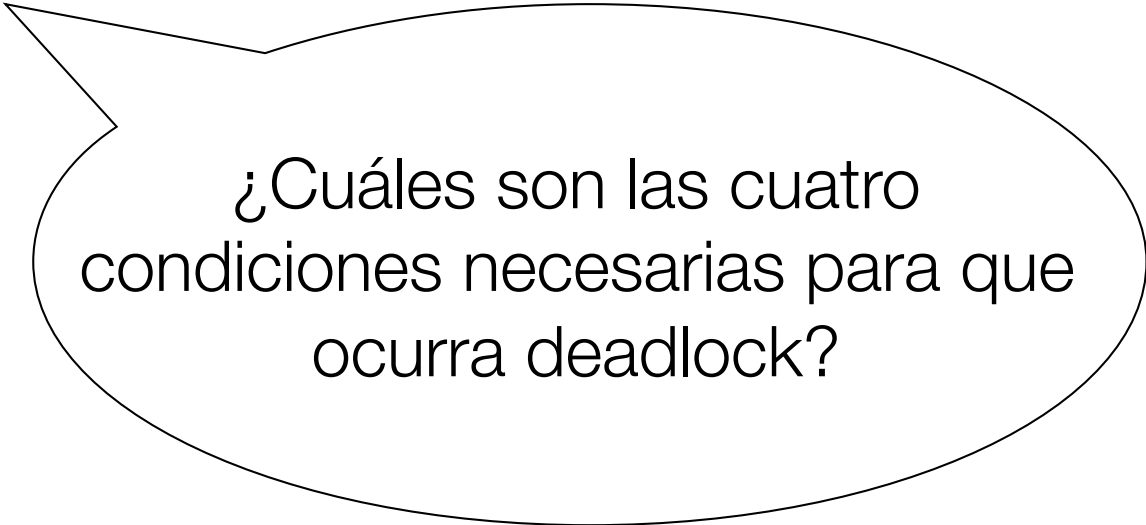
Los distintos tipos de propiedades en FSP se verifican y/o modelan de manera distinta.

Deadlock

Análisis de propiedades en FSP

Los distintos tipos de propiedades en FSP se verifican y/o modelan de manera distinta.

Deadlock



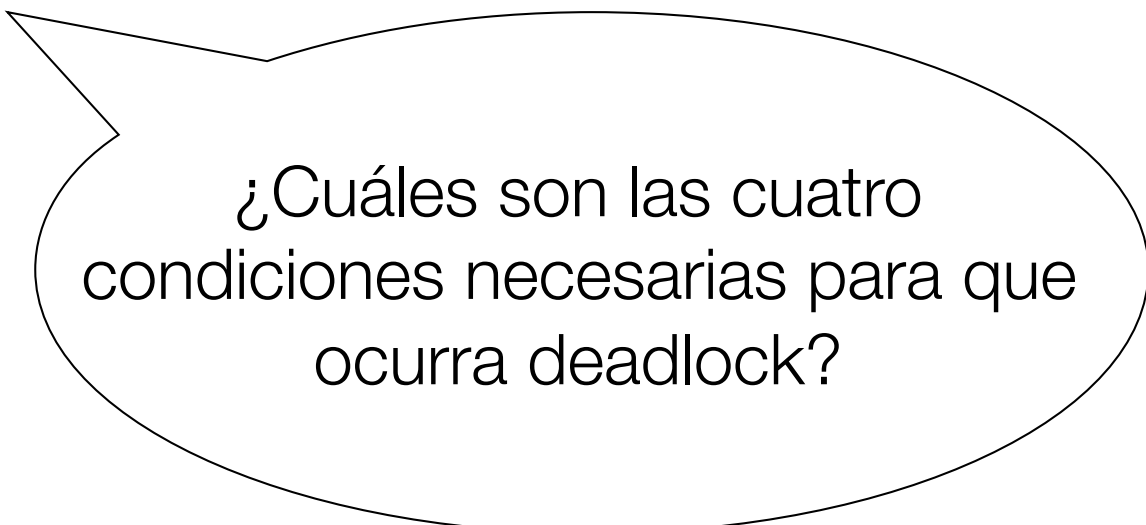
¿Cuáles son las cuatro condiciones necesarias para que ocurra deadlock?

Análisis de propiedades en FSP

Los distintos tipos de propiedades en FSP se verifican y/o modelan de manera distinta.

exclusión mutua

Deadlock



¿Cuáles son las cuatro condiciones necesarias para que ocurra deadlock?

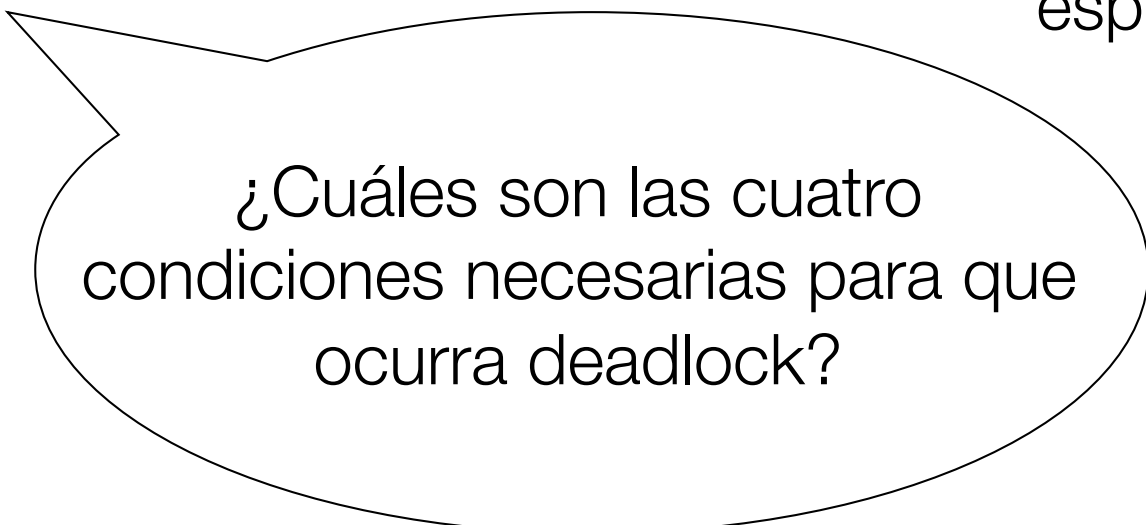
Análisis de propiedades en FSP

Los distintos tipos de propiedades en FSP se verifican y/o modelan de manera distinta.

exclusión mutua

espera y retención

Deadlock



¿Cuáles son las cuatro condiciones necesarias para que ocurra deadlock?

Análisis de propiedades en FSP

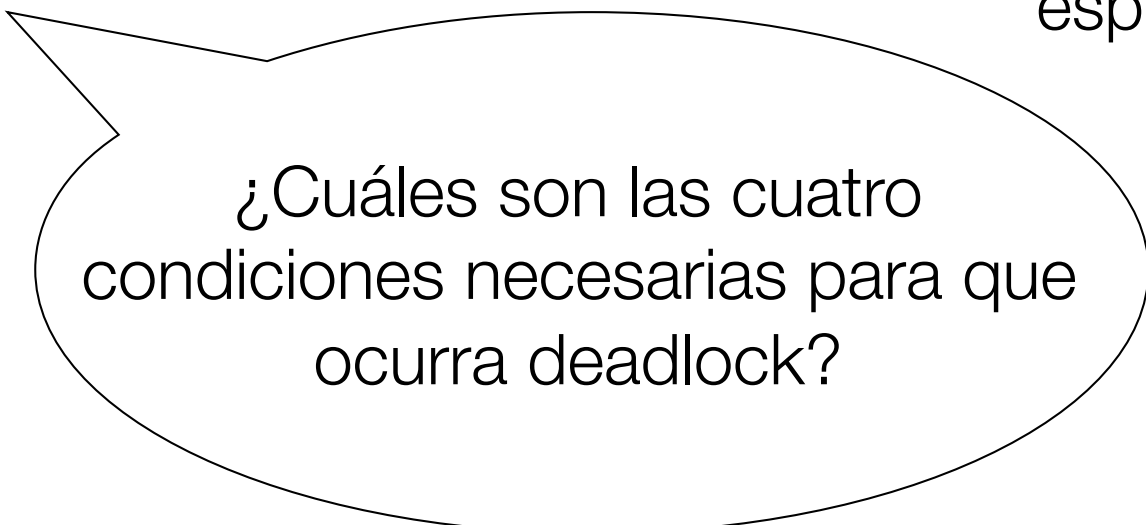
Los distintos tipos de propiedades en FSP se verifican y/o modelan de manera distinta.

Deadlock

exclusión mutua

espera y retención

no quita de recursos

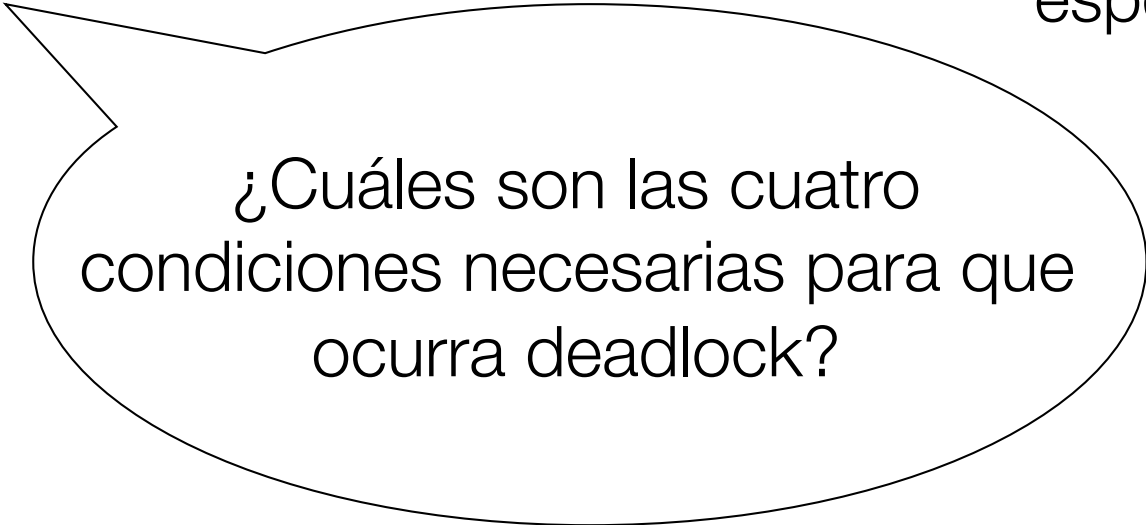


¿Cuáles son las cuatro condiciones necesarias para que ocurra deadlock?

Análisis de propiedades en FSP

Los distintos tipos de propiedades en FSP se verifican y/o modelan de manera distinta.

Deadlock



¿Cuáles son las cuatro condiciones necesarias para que ocurra deadlock?

exclusión mutua

espera y retención

no quita de recursos

espera circular

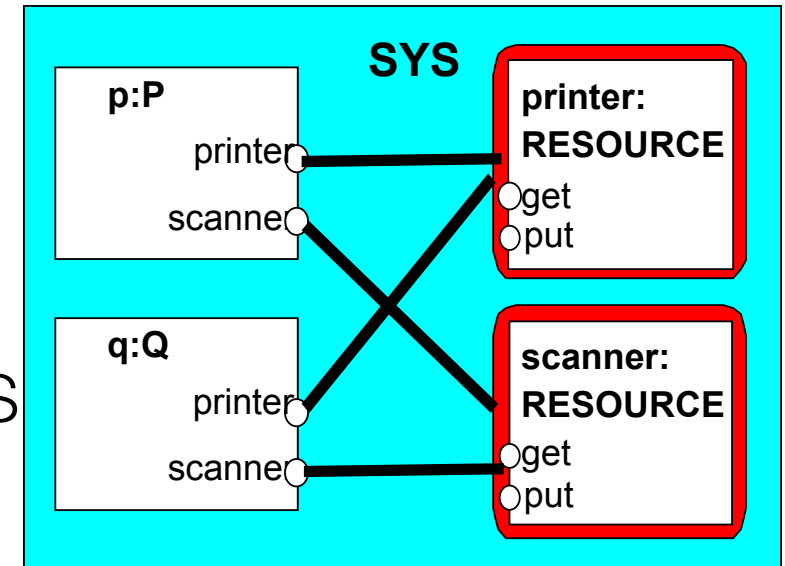
Análisis de propiedades en FSP

Los distintos tipos de propiedades en FS
 $\text{RESOURCE} = (\text{get} \rightarrow \text{put} \rightarrow \text{RESOURCE})$.
 modelan de manera distinta.

$P = (\text{printer.get} \rightarrow \text{scanner.get} \rightarrow \text{copy}$
Deadlock $\rightarrow \text{printer.put} \rightarrow \text{scanner.put} \rightarrow P)$.

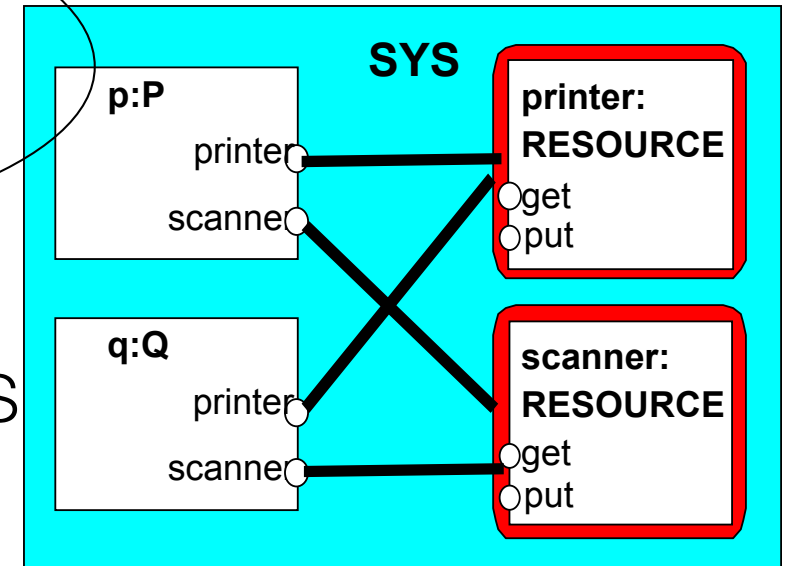
$Q = (\text{scanner.get} \rightarrow \text{printer.get} \rightarrow \text{copy}$
 $\rightarrow \text{printer.put} \rightarrow \text{scanner.put} \rightarrow Q)$.

$||\text{SYS} = (\text{p:P} || \text{q:Q}$
 $|| \{p,q\}::\text{printer:RESOURCE}$
 $|| \{p,q\}::\text{scanner:RESOURCE}$
 $)$.



Análisis de propiedades en FSP

LTSA puede chequear deadlock. Lo hace mediante una búsqueda exhaustiva.



Los distintos tipos de propiedades en FS modelan de manera distinta.

$P = (\text{printer.get} \rightarrow \text{scanner.get} \rightarrow \text{copy} \rightarrow \text{printer.put} \rightarrow \text{scanner.put} \rightarrow P).$

Deadlock

$Q = (\text{scanner.get} \rightarrow \text{printer.get} \rightarrow \text{copy} \rightarrow \text{printer.put} \rightarrow \text{scanner.put} \rightarrow Q).$

$||\text{SYS} = (\text{p:P} || \text{q:Q} || \{p,q\}::\text{printer:RESOURCE} || \{p,q\}::\text{scanner:RESOURCE}).$

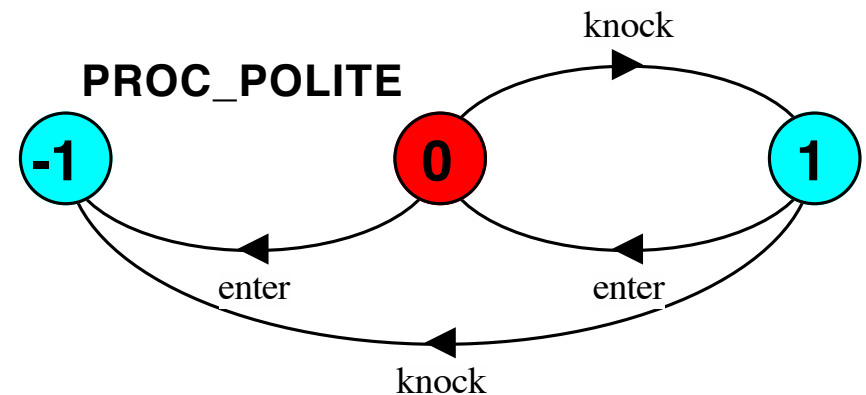
Análisis de propiedades en FSP

Safety

En FSP podemos expresar las propiedades de safety mediante procesos haciendo uso del estado de **ERROR** para indicar cuáles son las trazas que violan esta propiedad (notar que alcanza con indicar la traza parcial, i.e. un prefijo).

Ejemplo: Una persona educada golpea sólo una vez antes de entrar.

```
PROC_POLITE = (knock->(enter->PROC_POLITE
                    | knock->ERROR
                )
              | enter->ERROR
            ).
```



Análisis de propiedades en FSP

La declaración de un proceso **P** como **property** determina que la omisión de una acción en un estado de **P** interpretado como **proceso** induce una transición errónea en **P** como **propiedad**.

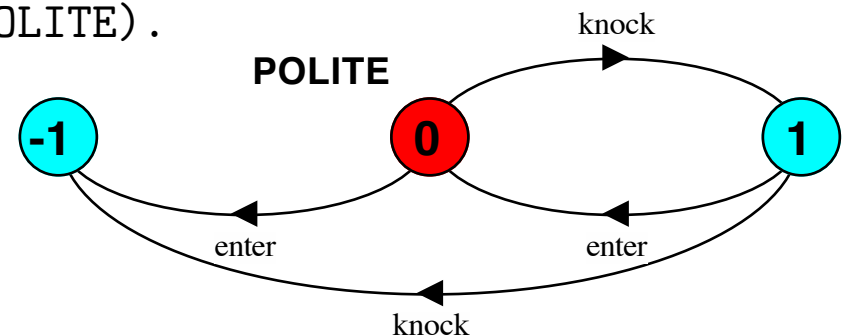
Safety (cont.)

FSP provee una construcción que nos permite obviar el agregado “a mano” de los estados de error:

`property N = P`

indica que los eventos correspondientes al alfabeto de **P** deben ocurrir respetando el patrón de ocurrencias definido por **P** (toda ocurrencia no definida por **P** lleva al estado de **ERROR**).

`property POLITE = (knock -> enter -> POLITE).`



Análisis de propiedades en FSP

```
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]
                  ).
```

Ejemplo

```
LOOP = (mutex.down->enter->exit->mutex.up->LOOP).
```

```
||SEMADEMO = (p[1..3]:LOOP
              || {p[1..3]}::mutex:SEMAPHORE(1)).
```

```
property MUTEX
= (p[i:1..3].enter->p[i].exit->MUTEX).
```

```
||CHECK = (SEMADEMO || MUTEX).
```

Análisis de propiedades en FSP

```
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]
                  ).
```

Ejemplo

```
LOOP = (mutex.down->enter->exit->mutex.up->LOOP).
```

```
||SEMADEMO = (p[1..3]:LOOP
              || {p[1..3]}::mutex:SEMAPHORE(1)).
```

```
property MUTEX
= (p[i:1..3].enter->p[i].exit->MUTEX).
```

```
||CHECK = (SEMADEMO || MUTEX).
```



¿Qué dice esta propiedad?

Análisis de propiedades en FSP

```
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]
                  ).
```

Ejemplo

```
LOOP = (mutex.down->enter->exit->mutex.up->LOOP).
```

```
||SEMADEMO = (p[1..3]:LOOP
              || {p[1..3]}::mutex:SEMAPHORE(1)).
```

```
property MUTEX
= (p[i:1..3].enter->p[i].exit->MUTEX).
```

```
||CHECK = (SEMADEMO || MUTEX).
```

Verificar cambiando la
inicialización del semáforo a
2.

¿Qué dice esta propiedad?

Análisis de propiedades en FSP

LTSA puede chequear safety.
También lo hace mediante búsqueda exhaustiva.

```
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]  
                  ).
```

Ejemplo

```
LOOP = (mutex.down->enter->exit->mutex.up->LOOP).
```

Verificar cambiando la
inicialización del semáforo a
2.

```
||SEMADEMO = (p[1..3]:LOOP  
             || {p[1..3]}::mutex:SEMAPHORE(1)).
```

```
property MUTEX  
= (p[i:1..3].enter->p[i].exit->MUTEX).
```

¿Qué dice esta propiedad?

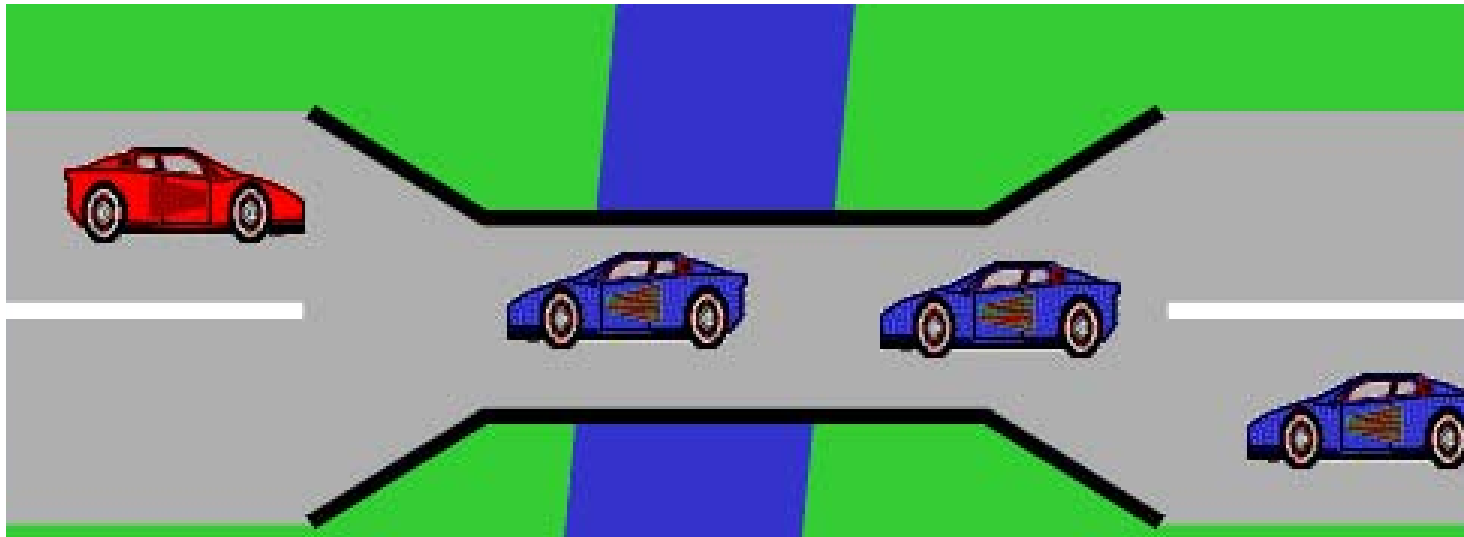
```
||CHECK = (SEMADEMO || MUTEX).
```

Análisis de propiedades en FSP

Ejemplo: Un puente de una sola vía

Un puente sobre un río tiene el ancho suficiente para permitir tráfico en sólo una dirección. Por consiguiente, los autos sólo pueden cruzar el puente concurrentemente si viajan en la misma dirección.

Cuando dos autos que viajan en direcciones opuestas ingresan al puente al mismo tiempo ocurre una violación de seguridad.



```

const N = 3 // number of each type of car
range T = 0..N // type of car count
range ID= 1..N // car identities

```

```

CAR = (enter->exit->CAR).

```

```

/* cars may not overtake each other */
NOPASS1    = C[1],
C[i:ID]    = ([i].enter -> C[i%N+1]).

```

```

NOPASS2    = C[1],
C[i:ID]    = ([i].exit -> C[i%N+1]).

```

||CONVOY = (C[1]:CAR || NOPASS1 || NOPASS2)

```

||CARS = (red:CONVOY || blue:CONVOY).

```

```

BRIDGE = BRIDGE[0][0],
BRIDGE[nr:T][nb:T] =
    (when (nb==0)
        red[ID].enter -> BRIDGE[nr+1][nb]
    |red[ID].exit    -> BRIDGE[nr-1][nb]
    |when (nr==0)
        blue[ID].enter -> BRIDGE[nr][nb+1]
    |blue[ID].exit    -> BRIDGE[nr][nb-1]
    ).

```

```

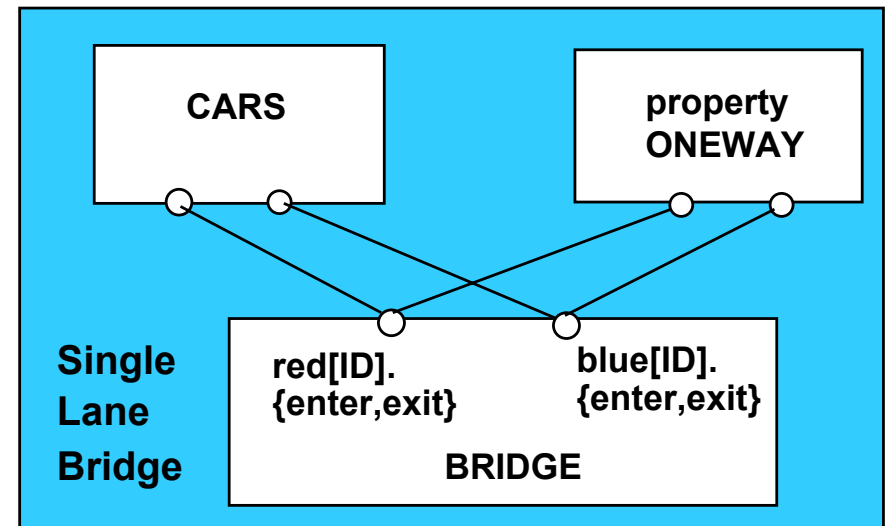
||SingleLaneBridge = (CARS || BRIDGE || ONEWAY ).

```

```

property ONEWAY = ( red[ID].enter -> RED[1]
                    | blue[ID].enter -> BLUE[1]
                    ),
RED[i:ID] = (red[ID].enter -> RED[i+1]
    |when(i==1)red[ID].exit -> ONEWAY
    |when( i>1)red[ID].exit -> RED[i-1]
    ),
BLUE[i:ID] = (blue[ID].enter -> BLUE[i+1]
    |when(i==1)blue[ID].exit -> ONEWAY
    |when( i>1)blue[ID].exit -> BLUE[i-1]
    ).

```



Análisis de propiedades en FSP

Liveness

LTSA solo puede manejar un conjunto reducido de propiedades de liveness:

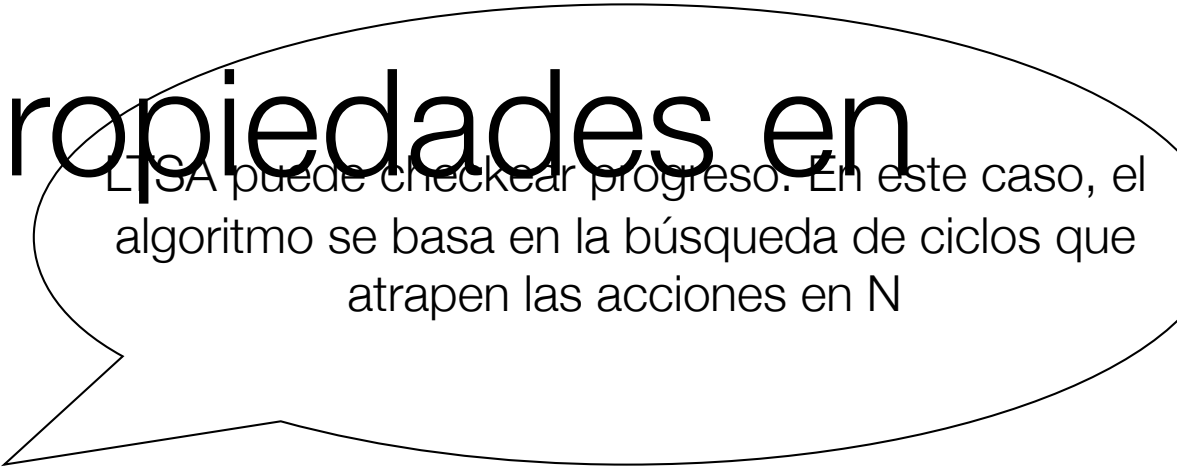
```
progress N = { a1, a2, ... }
```

Esto indica que, en una ejecución infinita de un sistema, alguna de las acciones **a1**, **a2**,... debe ejecutarse infinitas veces

Ejemplo:

```
COIN = ( toss -> heads -> COIN
        | toss -> tails -> COIN ).
progress HEADS = {heads}
```

Análisis de propiedades en FSP



LTSA puede chequear progreso. En este caso, el algoritmo se basa en la búsqueda de ciclos que atrapen las acciones en N

Liveness

LTSA solo puede manejar un conjunto reducido de propiedades de liveness:

```
progress N = { a1, a2, ... }
```

Esto indica que, en una ejecución infinita de un sistema, alguna de la acciones **a1**, **a2**,... debe ejecutarse infinitas veces

Ejemplo:

```
COIN = ( toss -> heads -> COIN
        | toss -> tails -> COIN ).
progress HEADS = {heads}
```


Análisis de propiedades en FSP

LTSA puede chequear progreso. En este caso, el algoritmo se basa en la búsqueda de ciclos que atrapen las acciones en N

Liveness

LTSA solo puede manejar un conjunto reducido de propiedades de liveness:

```
progress N = { a1, a2, ... }
```

Esto indica que, en una ejecución infinita de un sistema, alguna de la acciones **a1**, **a2**,... debe ejecutarse infinitas veces

Ejemplo:

```
COIN = ( toss -> heads -> COIN
        | toss -> tails -> COIN ).
progress HEADS = {heads}
```

LTSA la verifica verdadera

Análisis de propiedades en FSP

LTSA puede chequear progreso. En este caso, el algoritmo se basa en la búsqueda de ciclos que atrapen las acciones en N

Liveness

LTSA solo puede manejar un conjunto reducido de propiedades de liveness:

`progress N = { a1, a2, ... }`
Se asume elección equitativa: si una elección sobre un conjunto de transiciones es ejecutada infinitas veces entonces cada

Esto indica que, en una ejecución infinita de un sistema, alguna de la acciones **a1**, **a2**,... debe ejecutarse infinitas veces

Ejemplo:

```
COIN = ( toss -> heads -> COIN
        | toss -> tails -> COIN ).
progress HEADS = {heads}
```

LTSA la verifica verdadera

Análisis de propiedades en FSP

Liveness

Ejemplo: para el puente de una sola vía

```
progress BLUECROSS = {blue[ID].enter}  
progress REDCROSS = {red[ID].enter}
```

Análisis de propiedades en FSP

Liveness

Ejemplo: para el puente de una sola vía

```
progress BLUECROSS = {blue[ID].enter}  
progress REDCROSS = {red[ID].enter}
```



**LTSA las verifica
correctas!! :-)**

Prioridades entre acciones

LTSA asume en general que la elección es equitativa (fair), por eso no reporta error de progreso en el ejemplo de la moneda o en el puente de una sola vía.

Sin embargo, sabemos que es posible que los autos azules (o los rojos) acaparen el puente haciendo que los rojos (o los azules) esperen por siempre.

Entonces, para detectar problemas de progreso debemos imponer políticas de scheduling en las acciones que modele la situación en la que el puente (o cualquier otro sistema en general) es sometido a usos extremos.

Análisis de propiedades en FSP

Ejemplo: Siguiendo con el puente de una sola vía deseamos modelar que el puente se encuentra congestionado.

Para ello daremos menor prioridad a los eventos de salida del puente. De esta manera se prioriza a la entrada sobre la salida forzando la congestión del puente:

```
||CongestedBridge =  
    SingleLaneBridge >> {red[ID].exit,blue[ID].exit}.
```

Al verificar las propiedades de progreso, LTSA nos indicará ahora una situación de error.

Análisis de propiedades en FSP

Los model checkers más avanzados permiten elegir si se desea realizar la verificación bajo **fairness** o no.

Ejemplo: Siguiendo con el puente de una sola vía deseamos modelar que el puente se encuentra congestionado.

Para ello daremos menor prioridad a los eventos de salida del puente. De esta manera se prioriza a la entrada sobre la salida forzando la congestión del puente:

```
||CongestedBridge =  
    SingleLaneBridge >> {red[ID].exit, blue[ID].exit}.
```

Al verificar las propiedades de progreso, LTSA nos indicará ahora una situación de error.

Bibliografía

capítulos 4, 5, 6 y 7 de Concurrency, State Models and
Java Programs, Magee and Kramer 1999