

# Diseño y Verificación de Programas Concurrentes

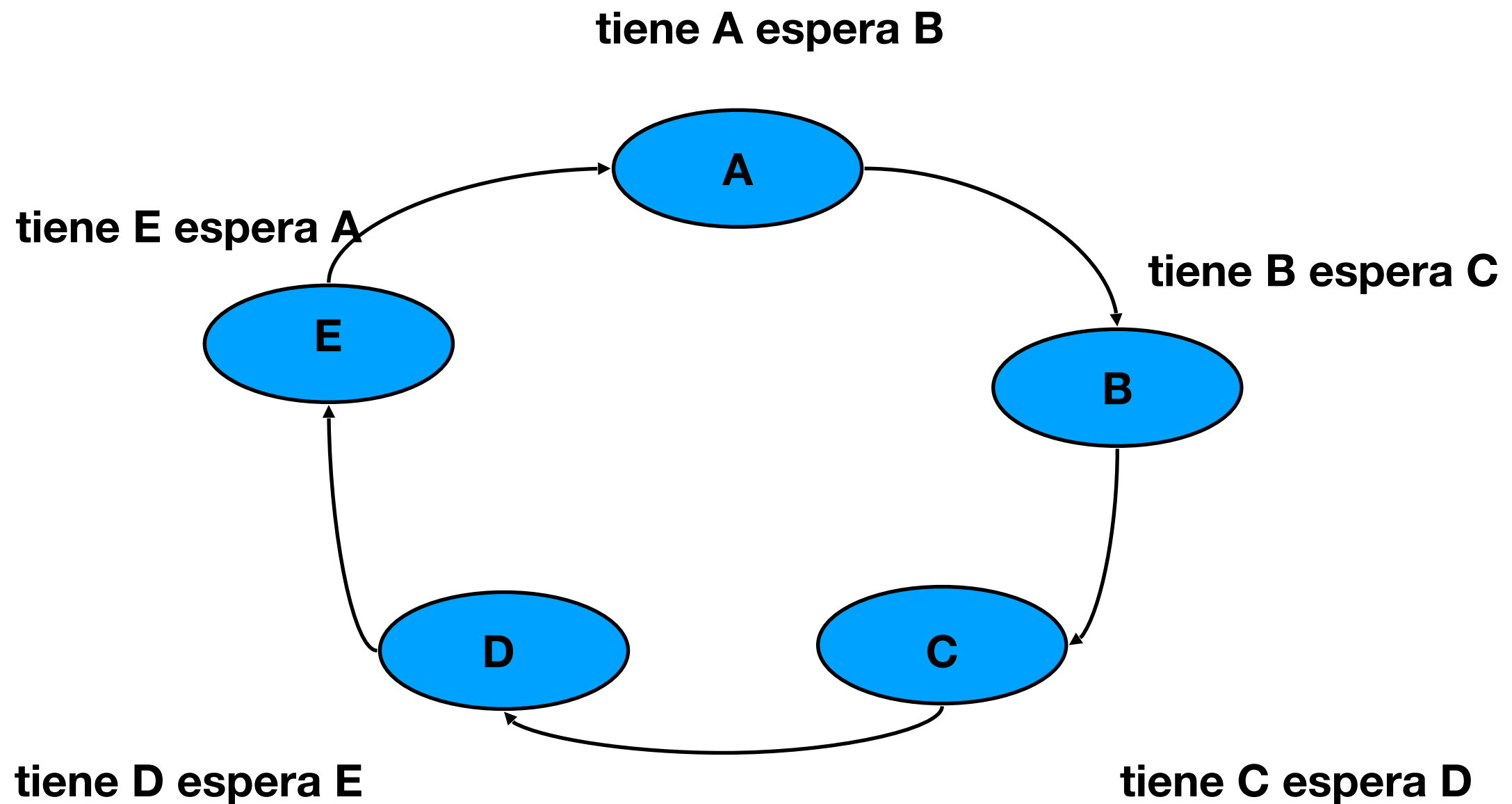
Escuela de Informática - CACIC 2021

Tomado de slides de Concurrency, State Models & Java Programs  
(Magee & Kramer 2006)

# Deadlock: condiciones necesarias y suficientes

- **Recursos reusables serialmente:**  
los procesos involucrados comparten recursos que utilizan de manera mutuamente excluyente.
- **Adquisición incremental:**  
los procesos mantienen la posesión de recursos ya adquiridos mientras esperan por la adquisición de recursos adicionales.
- **No pre-emption:**  
una vez que un proceso adquiere un recurso, el mismo no puede “quitarse” compulsivamente (pre-emption), sino que el proceso debe liberarlo de forma voluntaria.
- **Espera cíclica:**  
se da una cadena de espera circular entre los procesos, en la cual cada proceso preserva la adquisición de un recurso mientras que su sucesor espera que el mismo se libere.

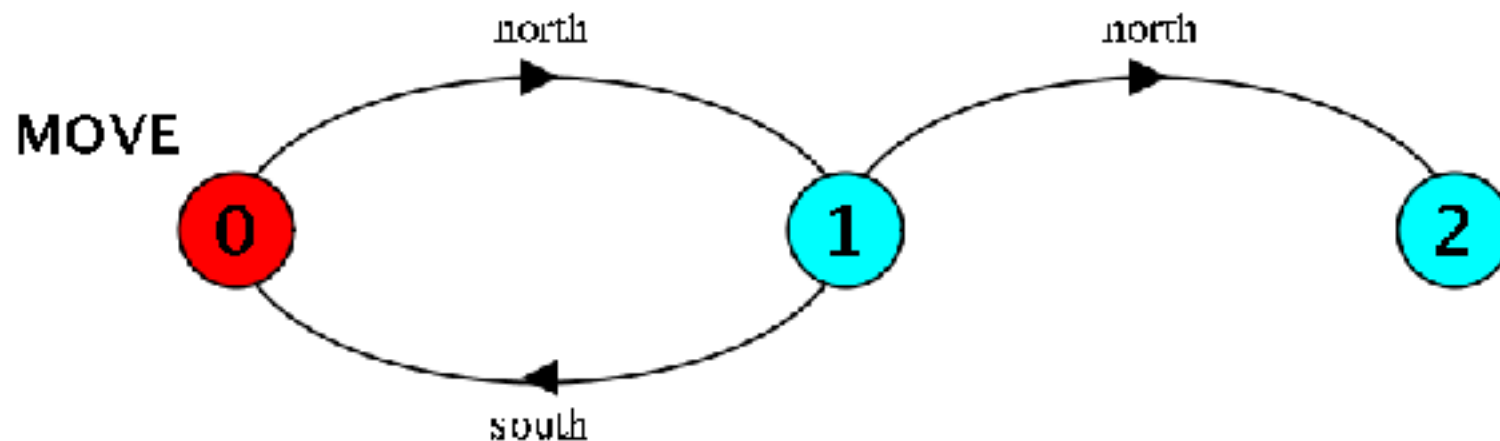
# Espera Circular



# Análisis de Deadlock - procesos primitivos

un estado de deadlock es uno que no tiene transiciones salientes  
en FSP: proceso **STOP**

**MOVE = (north->(south->MOVE | north->STOP)) .**



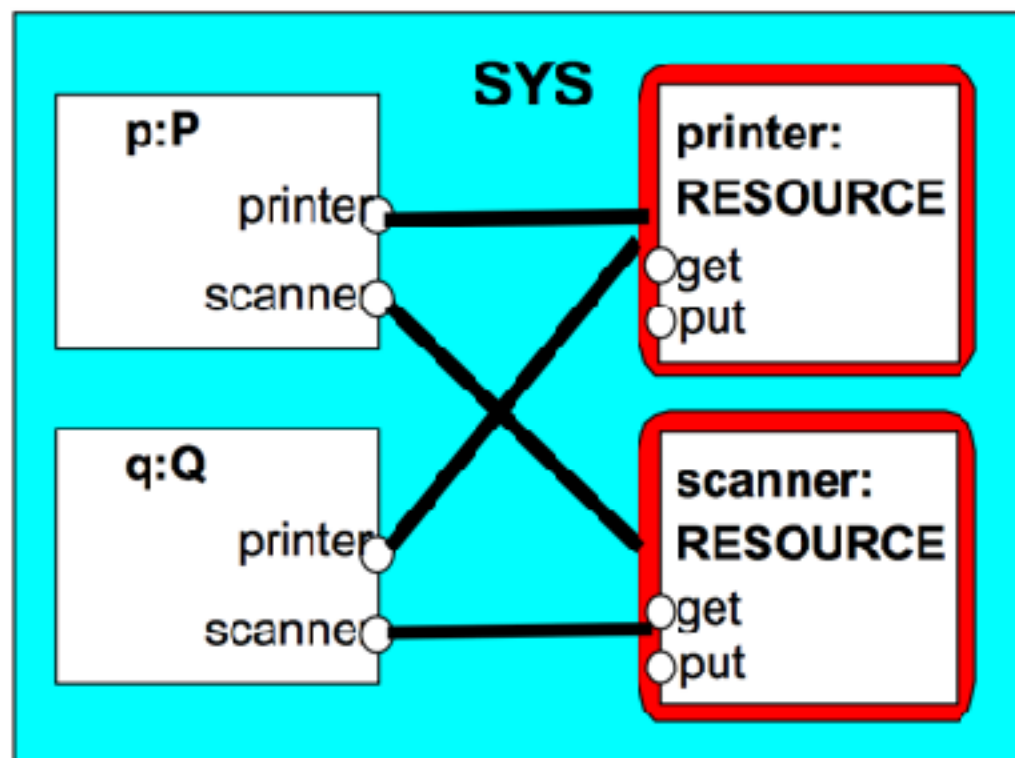
Trace to DEADLOCK:  
north  
north

Se puede usar animación para (intentar) producir una traza

Análisis usando LTSA (traza más corta a STOP)

# Análisis de deadlock ( composición paralela )

En sistemas compuestos por múltiples procesos, puede haber deadlock como resultado de la composición de procesos.



Traza a deadlock?  
Cómo evitarlo?

```
RESOURCE = (get->put->RESOURCE) .  
P = ( printer.get -> scanner.get  
    -> copy  
    -> printer.put -> scanner.put  
    -> P ) .  
Q = ( scanner.get -> printer.get  
    -> copy  
    -> scanner.put -> printer.put  
    -> Q ) .  
||SYS = (p:P || q:Q  
    || {p,q}::printer:RESOURCE  
    || {p,q}::scanner:RESOURCE  
    ) .
```

# Análisis de deadlock - formas de evitarlo

¿ adquisición de recursos en el mismo orden?

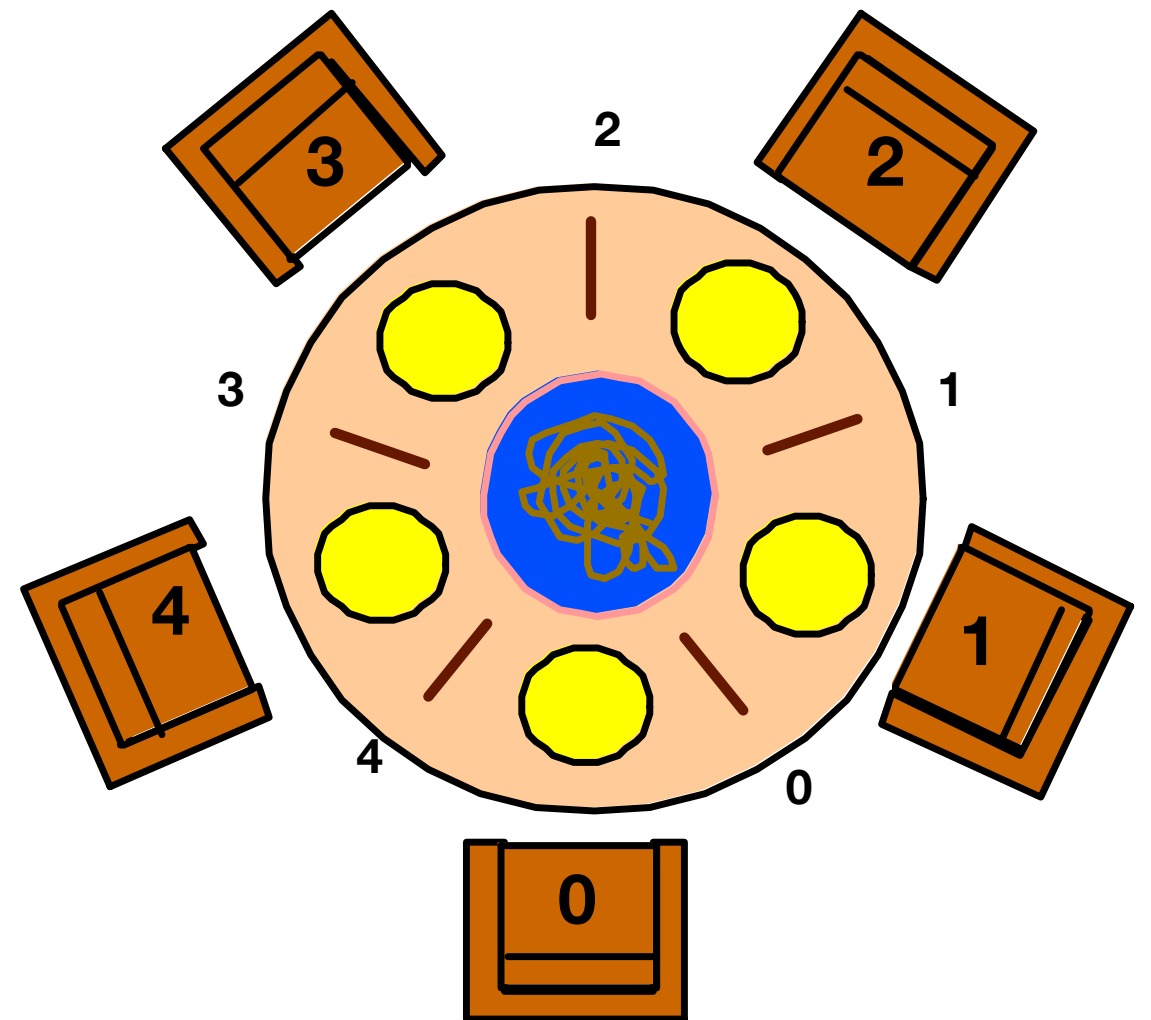
**Timeout:**

```
P          = (printer.get-> GETSCANNER),  
GETSCANNER = (scanner.get->copy->printer.put  
              ->scanner.put->P  
              |timeout -> printer.put->P  
              ).  
  
Q          = (scanner.get-> GETPRINTER),  
GETPRINTER = (printer.get->copy->printer.put  
              ->scanner.put->Q  
              |timeout -> scanner.put->Q  
              ).
```

**Deadlock? Progreso?**

# Filósofos Comensales

Cinco filósofos están sentados alrededor de una mesa redonda. Cada filósofo pasa su vida alternando entre pensar y comer. En el centro de la mesa hay un bowl de spaghetti. Un filósofo necesita dos tenedores para comer.

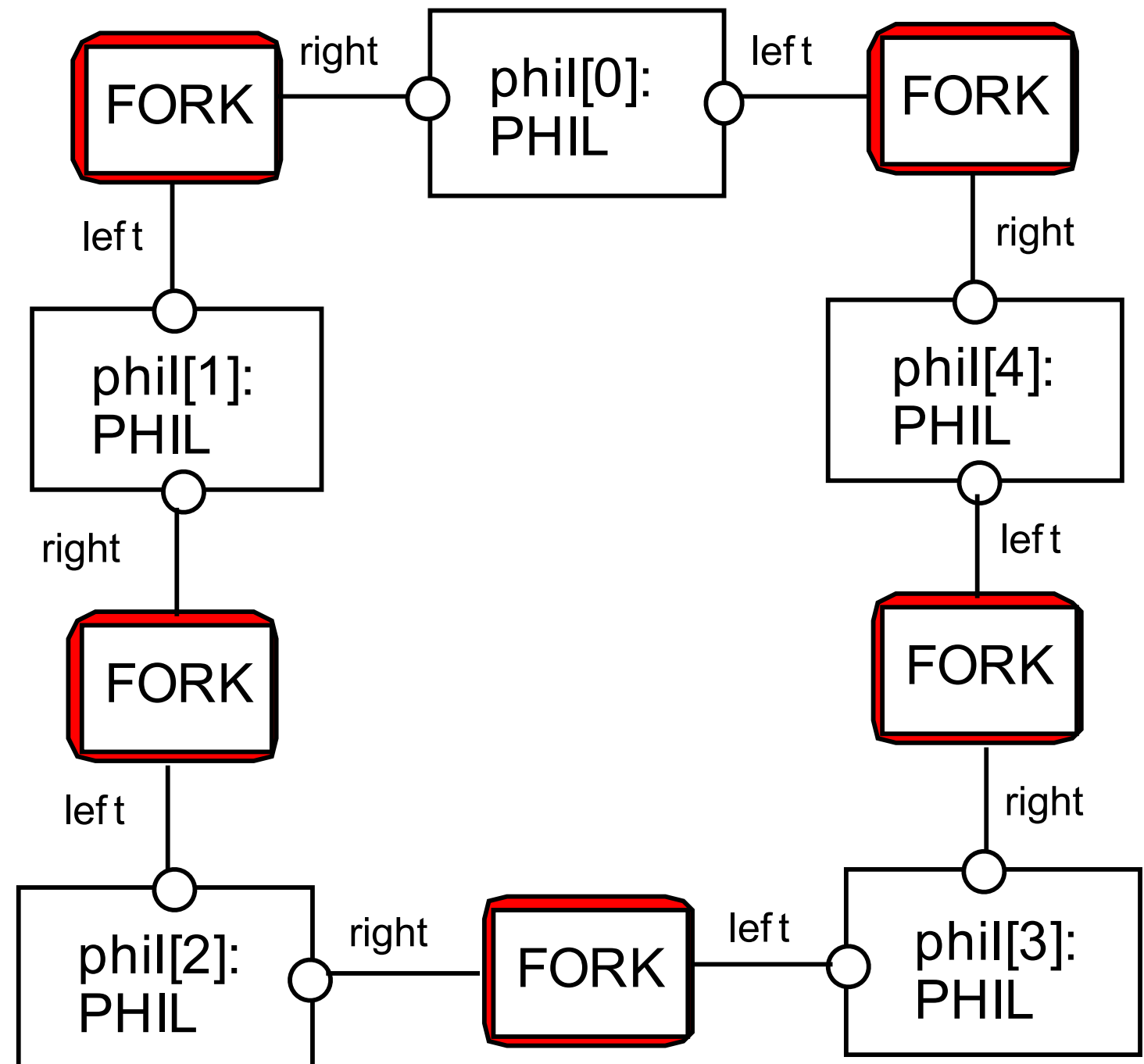


Cada tenedor se ubica entre un par de filósofos, los cuales acuerdan en que usarán sólo los tenedores inmediatamente a la derecha e izquierda de cada uno.

# Filósofos comensales - diagrama de estructuras

**Cada FORK es un recurso compartido, con acciones get y put.**

**Cada PHIL debe, cuando tiene hambre, obtener (get) sus FORK de derecha e izquierda, para poder comenzar a comer.**





# Filósofos comensales - Modelo

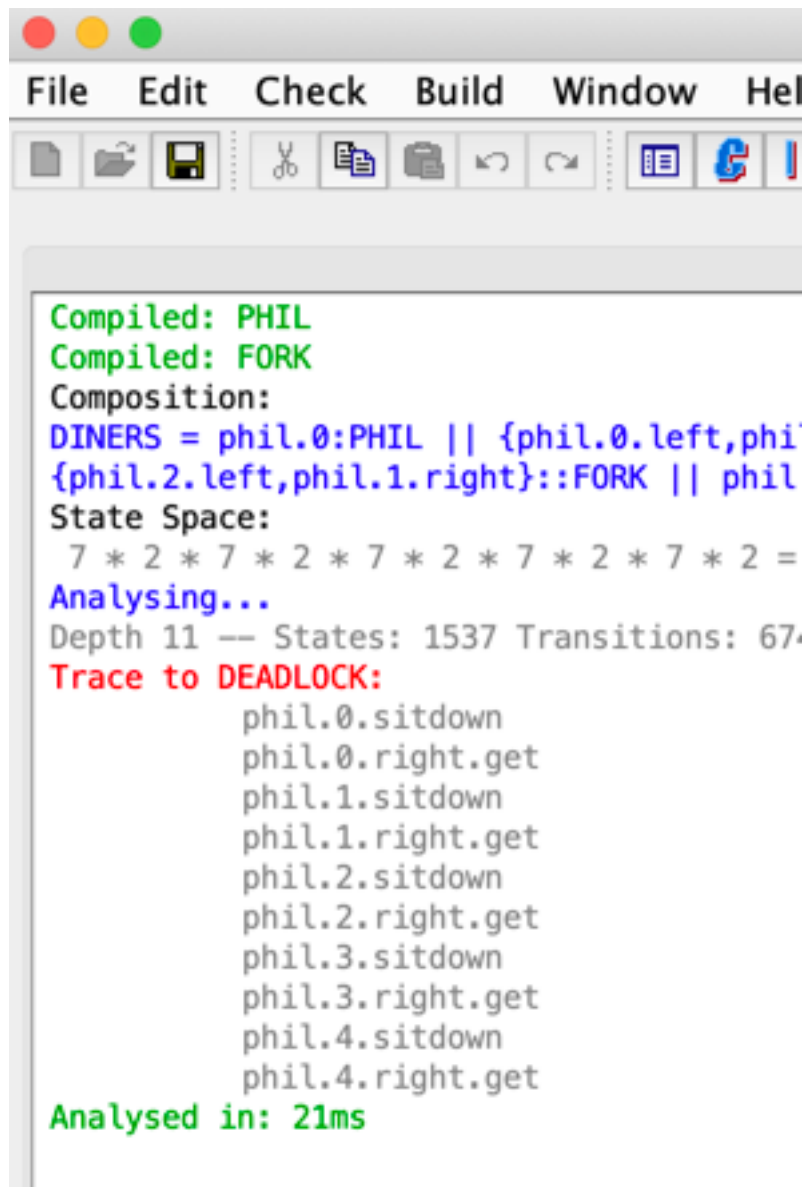
```
FORK = (get -> put -> FORK) .  
PHIL = (sitdown -> right.get -> left.get  
         -> eat  
         -> right.put -> left.put  
         -> arise -> PHIL) .
```

## Mesa de filósofos:

```
|| DINERS(N=5) = forall [i:0..N-1] ( phil[i]:PHIL ||  
    {phil[i].left, phil[((i-1)+N%N).right]}::FORK ).
```

**Tiene deadlock este sistema?**

# Filósofos comensales - análisis del modelo



```
File Edit Check Build Window Hel
[Icons]

Compiled: PHIL
Compiled: FORK
Composition:
DINERS = phil.0:PHIL || {phil.0.left,phi
{phil.2.left,phil.1.right}::FORK || phil
State Space:
7 * 2 * 7 * 2 * 7 * 2 * 7 * 2 * 7 * 2 =
Analysing...
Depth 11 -- States: 1537 Transitions: 67
Trace to DEADLOCK:
    phil.0.sitdown
    phil.0.right.get
    phil.1.sitdown
    phil.1.right.get
    phil.2.sitdown
    phil.2.right.get
    phil.3.sitdown
    phil.3.right.get
    phil.4.sitdown
    phil.4.right.get
Analysed in: 21ms
```

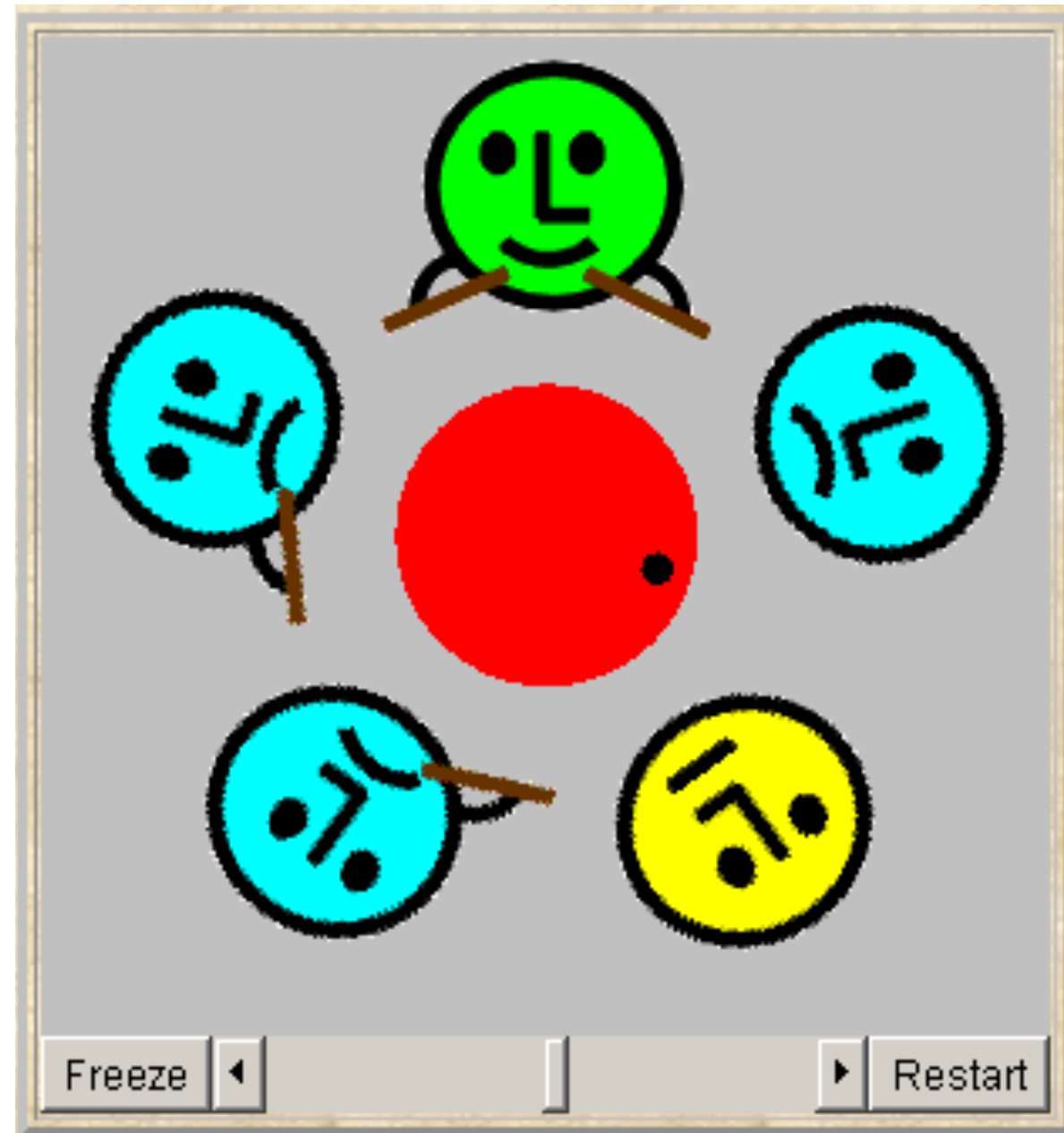
**En esta situación, todos los filósofos tienen hambre al mismo tiempo, se sientan a la mesa y cada filósofo alcanza a tomar el tenedor de su derecha.**

**El sistema no puede avanzar más, dado que cada filósofo se queda esperando por la liberación del tenedor de su izquierda, sin liberar el que tiene.**

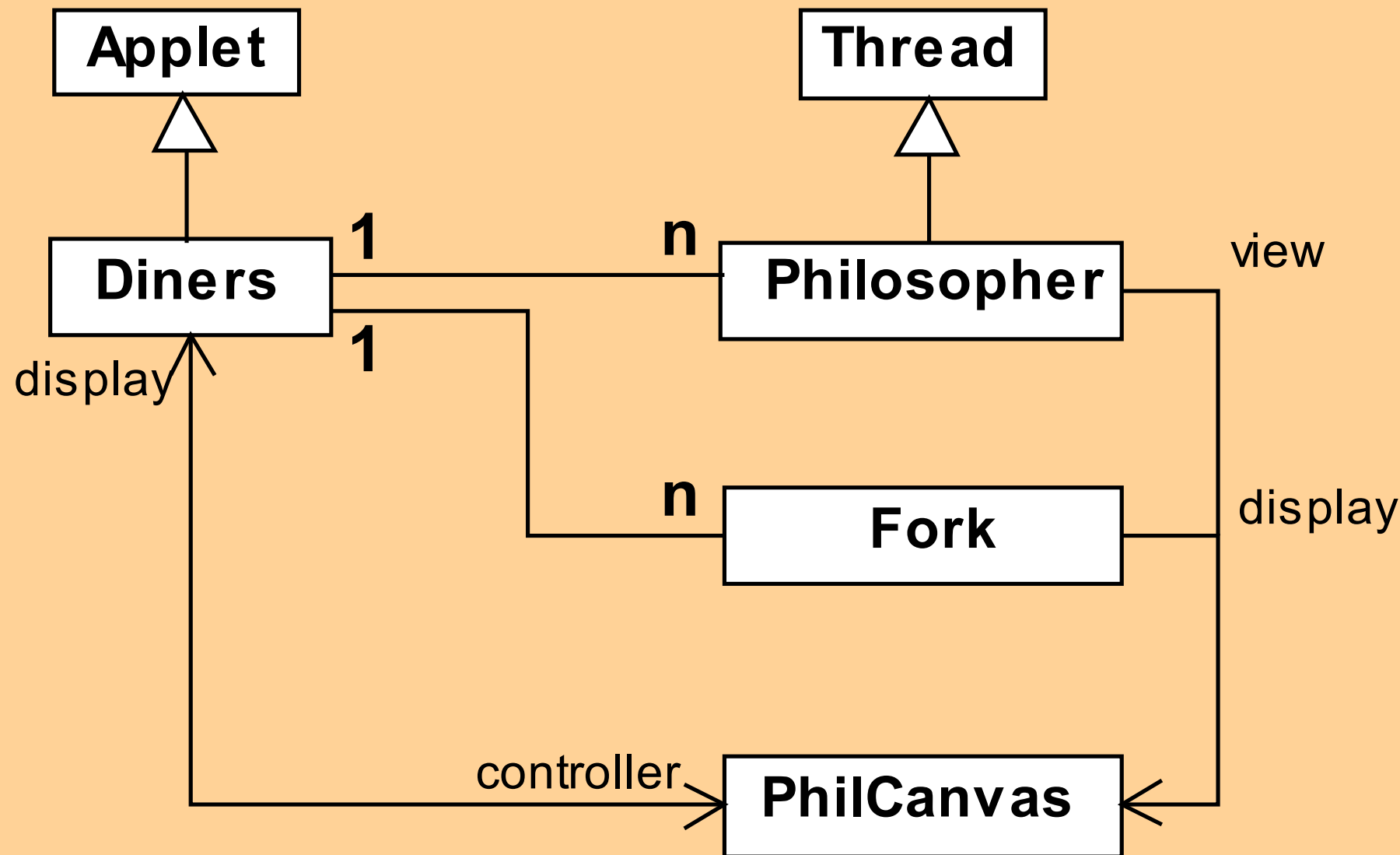
# Filósofos comensales

**La situación de deadlock es detectada fácilmente en nuestro modelo.**

**Cuán fácil es detectar la posibilidad de deadlock en una implementación?**



# Filósofos comensales - Implementación en Java



**filósofos:**

entidades activas,  
se implementan  
como threads

**tenedores:**

entidades  
compartidas  
pasivas, se  
implementan  
como monitores  
display

# Filósofos comensales - Tenedor como Monitor

```
public class Fork {  
  
    private boolean taken=false;  
    private PhilCanvas display;  
    private int identity;  
  
    Fork(PhilCanvas disp, int id)  
        { display = disp; identity = id;}  
  
    synchronized void put() {  
        taken=false;  
        display.setFork(identity,taken);  
        notify();  
    }  
  
    synchronized void get()  
        throws java.lang.InterruptedException {  
        while (taken) wait();  
        taken=true;  
        display.setFork(identity,taken);  
    }  
}
```

**taken** codifica el estado del  
tenedor

# Filósofos comensales - Implementación de los filósofos

```
class Philosopher extends Thread {
    private int identity;
    private PhilCanvas view;
    private Diners controller;
    private Fork left;
    private Fork right;
...
    public void run() {
        try {
            while (true) {
                //thinking
                view.setPhil(identity, view.THINKING);
                sleep(controller.sleepTime());
                //hungry
                view.setPhil(identity, view.HUNGRY);
                right.get();
                //gotright chopstick
                view.setPhil(identity, view.GOTRIGHT);
                sleep(500);
                left.get();
                //eating
                view.setPhil(identity, view.EATING);
                sleep(controller.eatTime());
                right.put();
                left.put();
            }
        } catch (InterruptedException e) {}
    }
}
```

se sigue del modelo (las acciones para sentarse y levantarse de la mesa se omitieron).

# Filósofos comensales - implementación en Java

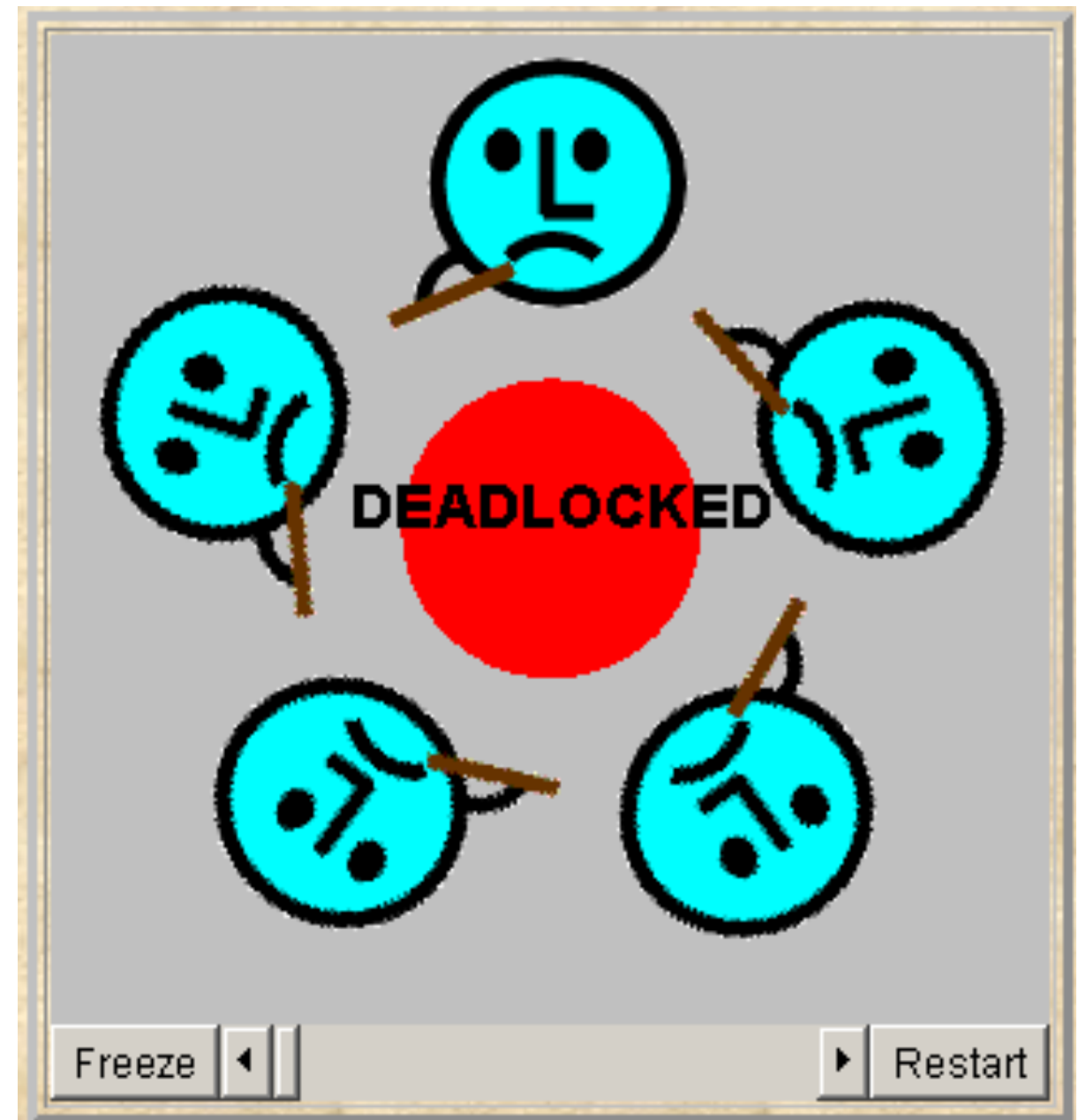
Código para crear los threads de los filósofos y los monitores para los tenedores:

```
public void start() {  
    for (int i = 0; i < display.NUMPHILS; ++i)  
        fork[i] = new Fork(display, i);  
    for (int i = 0; i < display.NUMPHILS; ++i) {  
        phil[i] = makePhilosopher(this, i,  
                                   fork[(i-1+display.NUMPHILS)% display.NUMPHILS], fork[i]);  
        phil[i].start();  
    }  
}
```

# Filósofos comensales

Para asegurar que el deadlock ocurre en algún momento, se puede usar la barra de control deslizante, para reducir el tiempo que cada filósofo dedica a pensar y a comer.

Este "speedup" incrementa la probabilidad de ocurrencia de deadlock.





# Filósofos sin Deadlock

Se puede evitar deadlock asegurando que no puede haber espera circular. Cómo?

Una forma es mediante la introducción de una asimetría en nuestra definición de filósofos.

Se puede usar la identidad  $I$  de cada filósofo para “diferenciarlos”: los filósofos pares toman primero sus tenedores izquierdos, mientras que los impares toman primero sus tenedores derechos.

```
PHIL(I=0)
= (when (I%2==0) sitdown
   ->left.get->right.get
   ->eat
   ->left.put->right.put
   ->arise->PHIL
|when (I%2==1) sitdown
   ->right.get->left.get
   ->eat
   ->left.put->right.put
   ->arise->PHIL
).
```

# Propiedades de Sistemas

## Safety y Liveness

### Conceptos:

propiedades: verdaderas para cualquier ejecución posible.

**safety**: nada malo ocurre.

**liveness**: algo bueno finalmente ocurre.

### Modelos:

**safety**: no existen estados de ERROR/STOP alcanzables.

**progress**: alguna acción finalmente será ejecutada.

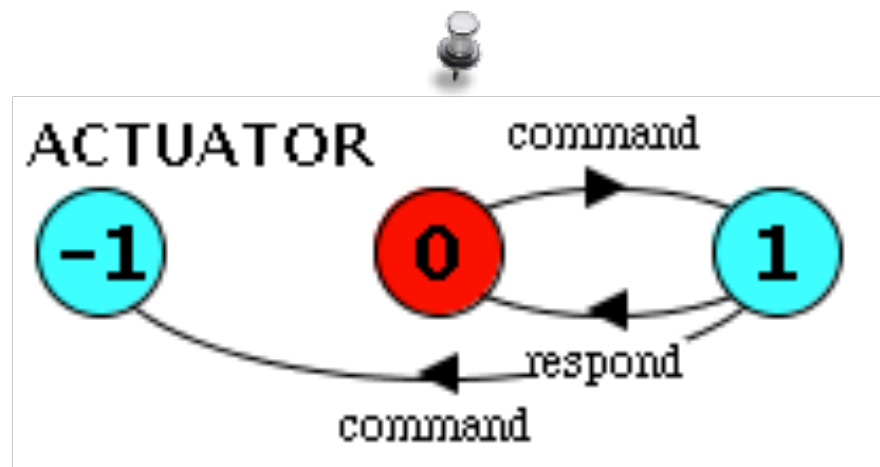
elección justa y prioridad de acciones

# Safety

Una propiedad de **safety** afirma que nunca ocurre algo malo.

STOP o estados de deadlock (sin salidas salientes)

ERROR proceso (-1) para detectar comportamientos erróneos



**ACTUATOR**

**$= (\text{command} \rightarrow \text{ACTION}),$**   
 **$\text{ACTION}$**   
 **$= (\text{respond} \rightarrow \text{ACTUATOR}$**   
 **$|\text{command} \rightarrow \text{ERROR}).$**

**análisis usando LTSA:**  
**( traza más corta)**

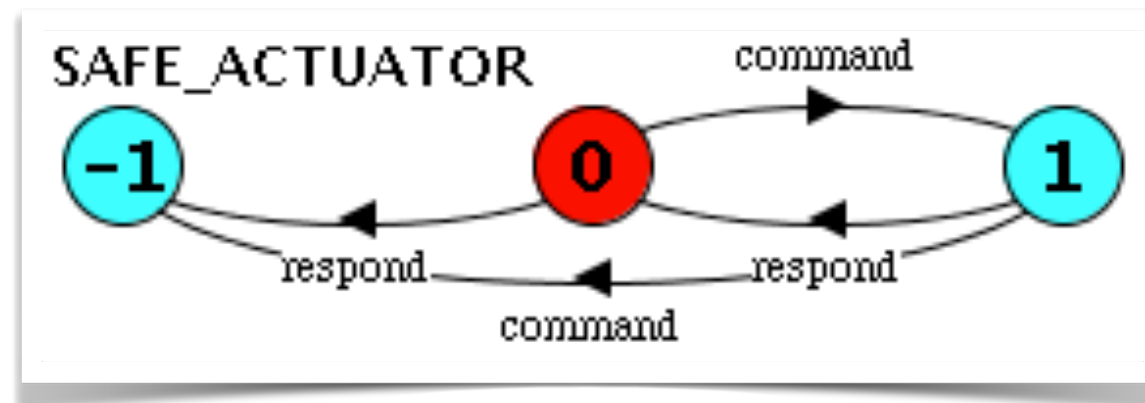
**Trace to ERROR:**  
**command**  
**command**

# Especificación de Propiedades

## Safety

especificar como **ERROR** los estados **no deseados** (cf. excepciones).

en sistemas complejos, usualmente es mejor especificar **qué es requerido**.



**property SAFE\_ACTUATOR = (command -> respond  
-> SAFE\_ACTUATOR ) .**

Análisis utilizando LTSA (como en el caso anterior)

# Propiedades de Safety

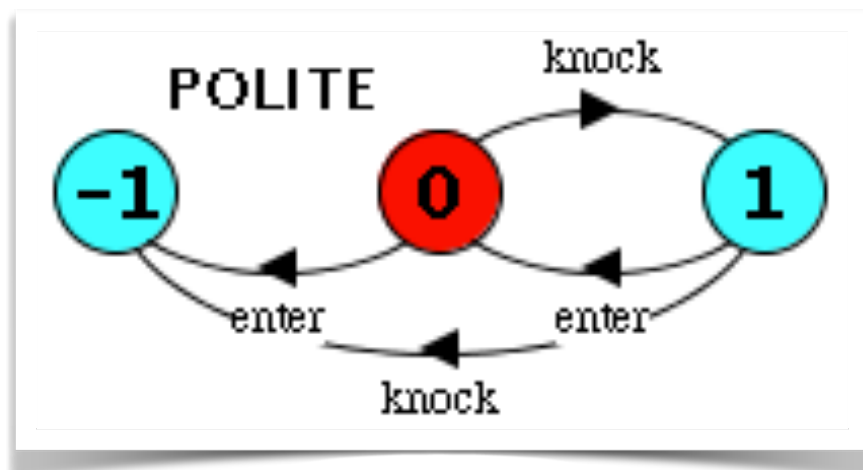
La propiedad: es de buena educación **golpear antes de entrar** en una habitación.

Trazas: **knock -> enter** ☒

**enter** ☒

**knock -> knock**

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



En todos los estados, todas las acciones en el alfabeto de una propiedad son elecciones posibles

# Propiedades de Safety

Una Propiedad de Safety **property** **P** define un proceso determinista que afirma que cualquier traza que incluya acciones en el alfabeto de **P**, es aceptada por **P**.

Si **P** es compuesto con **S**, las trazas de acciones del alfabeto de **S<sub>n</sub>** las del alfabeto de **P**, también deben ser trazas válidas de **P**, sino, el estado **ERROR** es alcanzable.

## Transparencia de propiedades de safety:

Dado que todas las acciones en el alfabeto de la propiedad son elecciones posibles, al componerla con un conjunto de procesos **no afecta el comportamiento** de dichos procesos. No obstante, si un comportamiento que viola la propiedad puede ocurrir, entonces el estado de **ERROR** es alcanzable. Las propiedades deben ser *determinísticas* para ser transparentes.

# Propiedades de Safety

¿ Cómo podemos especificar que una acción “**disaster**”, nunca ocurre?



**property CALM = STOP + {disaster}.**

Una propiedad de safety deber ser especificada de manera tal que incluya **todos** los comportamientos aceptables (válidos) en su alfabeto.

# Safety

## Exclusión Mutua

```
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).
```

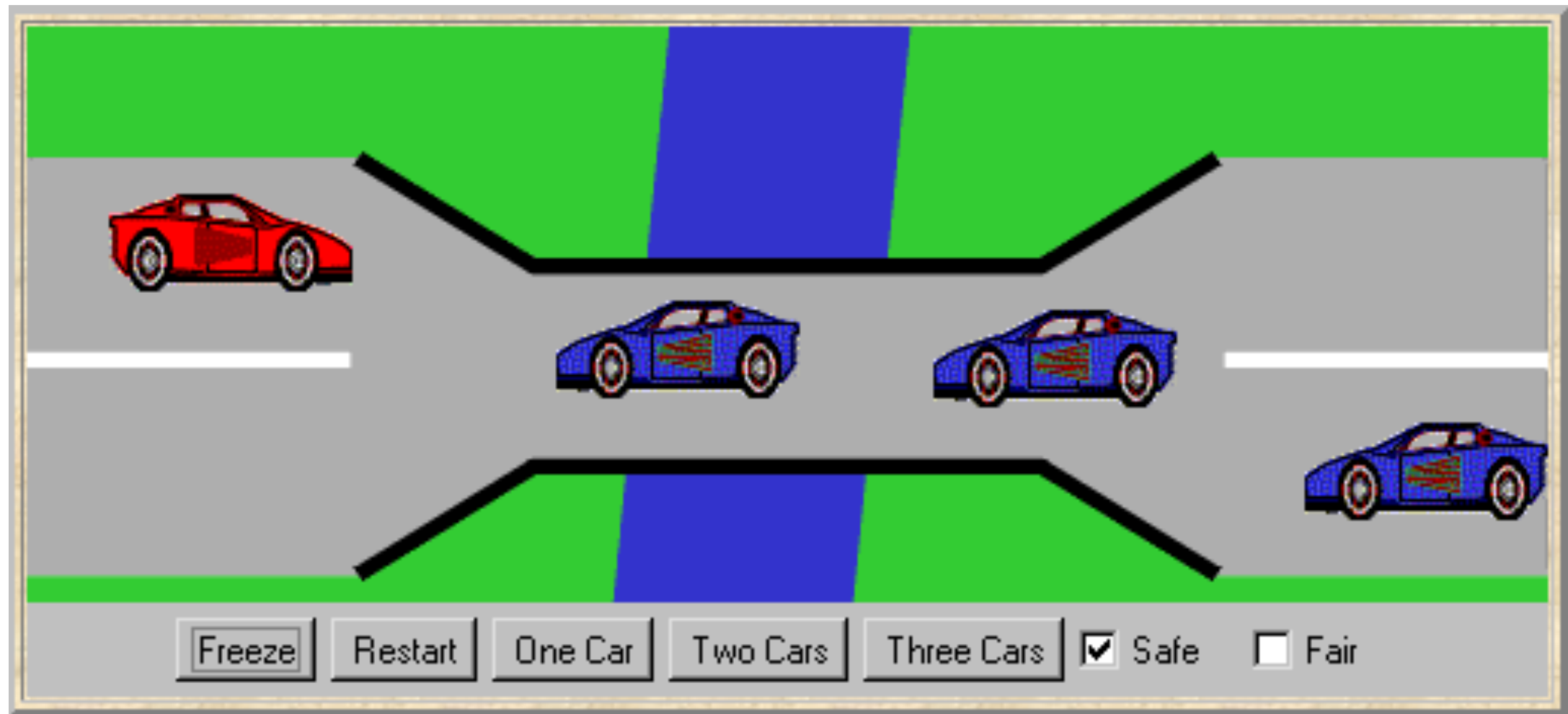
¿ Cómo podemos verificar que el modelo garantiza la exclusión mutua en la **sección crítica**?

Verificar safety usando LTSA.

¿Qué sucede si el semáforo es inicializado en 2?



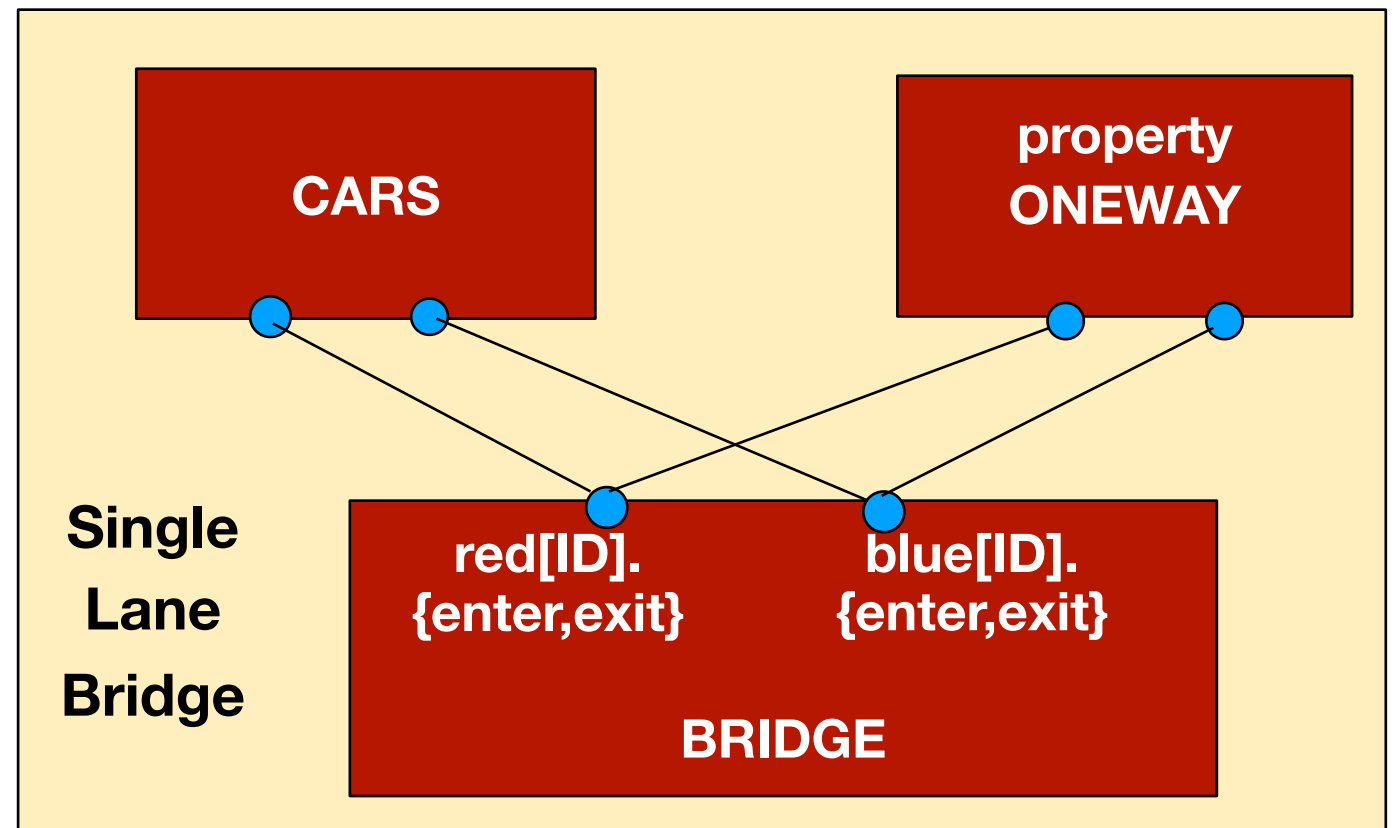
# Problema del puente con único carril (Single Lane Bridge)



Un puente sobre un río sólo permite un carril de tráfico. En consecuencia, los coches sólo pueden moverse al mismo tiempo si se están moviendo en la misma dirección . Una violación de seguridad se produce si dos coches que se mueven en diferentes direcciones entran en el puente al mismo tiempo.

# SLB - Modelo

- ¿ Eventos o acciones de interés ?  
**enter** y **exit**
- Identificar procesos.  
**cars** (autos) y **bridge** (puente)
- Identificar propiedades.  
**oneway**
- Definir cada procesos y sus interacciones (estructura).



# SLB - Modelo de auto

```
const N = 3           // cant. de cada tipo
range T = 0..N        // Rango de cada tipo
range ID= 1..N        // Identificadores

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

Para modelar el hecho de que los autos no pueden pasarse entre sí en el puente, modelamos un CONVOY de autos en la misma dirección. Tendremos un convoy **rojo** y uno **azul** de hasta N autos para cada dirección:

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

# SLB - Modelos CONVOY

```
NOPASS1    = C[1],                                //preserves entry order
C[i:ID]    = ([i].enter-> C[i%N+1]).
NOPASS2    = C[1],                                //preserves exit order
C[i:ID]     = ([i].exit-> C[i%N+1]).

|| CONVOY = ([ID]:CAR || NOPASS1 || NOPASS2).
```

Puede suceder 1.enter -> 2.enter -> 1.exit -> 2.exit  
No puede 1.enter -> 2.enter -> **2.exit** -> 1.exit  
i.e. no se permite sobre paso.

# SLB - Modelo BRIDGE (puente)

Los autos pueden circular de manera concurrente sobre el puente sólo si van en el **mismo sentido**. El puente mantiene la cantidad de autos rojos y azules sobre el puente. Sólo se permite la entrada de autos rojos si la cantidad de azules es cero y vice-versa.

```
BRIDGE = BRIDGE[0][0], //inicialmente vacío
BRIDGE[nr:T][nb:T] =( //nr, nb contador rojo y azul resp.
    when(nb==0) red[ID].enter -> BRIDGE[nr+1][nb] //nb==0
    |
    red[ID].exit -> BRIDGE[nr-1][nb]
    |
    when(nr==0) blue[ID].enter-> BRIDGE[nr][nb+1] //nr==0
    |
    blue[ID].exit -> BRIDGE[nr][nb-1]
).
```

Aún cuando es 0, las acciones “exit” permiten decremental el contador. LTSA asocia estos estado indefinidos a **ERROR**.

# SLB - propiedad de safety ONEWAY

Especificamos una **propiedad de safety** para verificar que los autos **no colisionan**. Mientras haya autos rojos sobre el puente, sólo pueden entrar autos rojos; similar para autos azules. Cuando el puente está vacío, tanto autos rojos como azules pueden entrar.

```
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]
             ), //i es un contador de autos rojos
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]
             ). //i es un contador de autos azules
```

# SLB - Análisis del modelo

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

**¿ Es violada la  
propiedad de safety  
ONEWAY?**

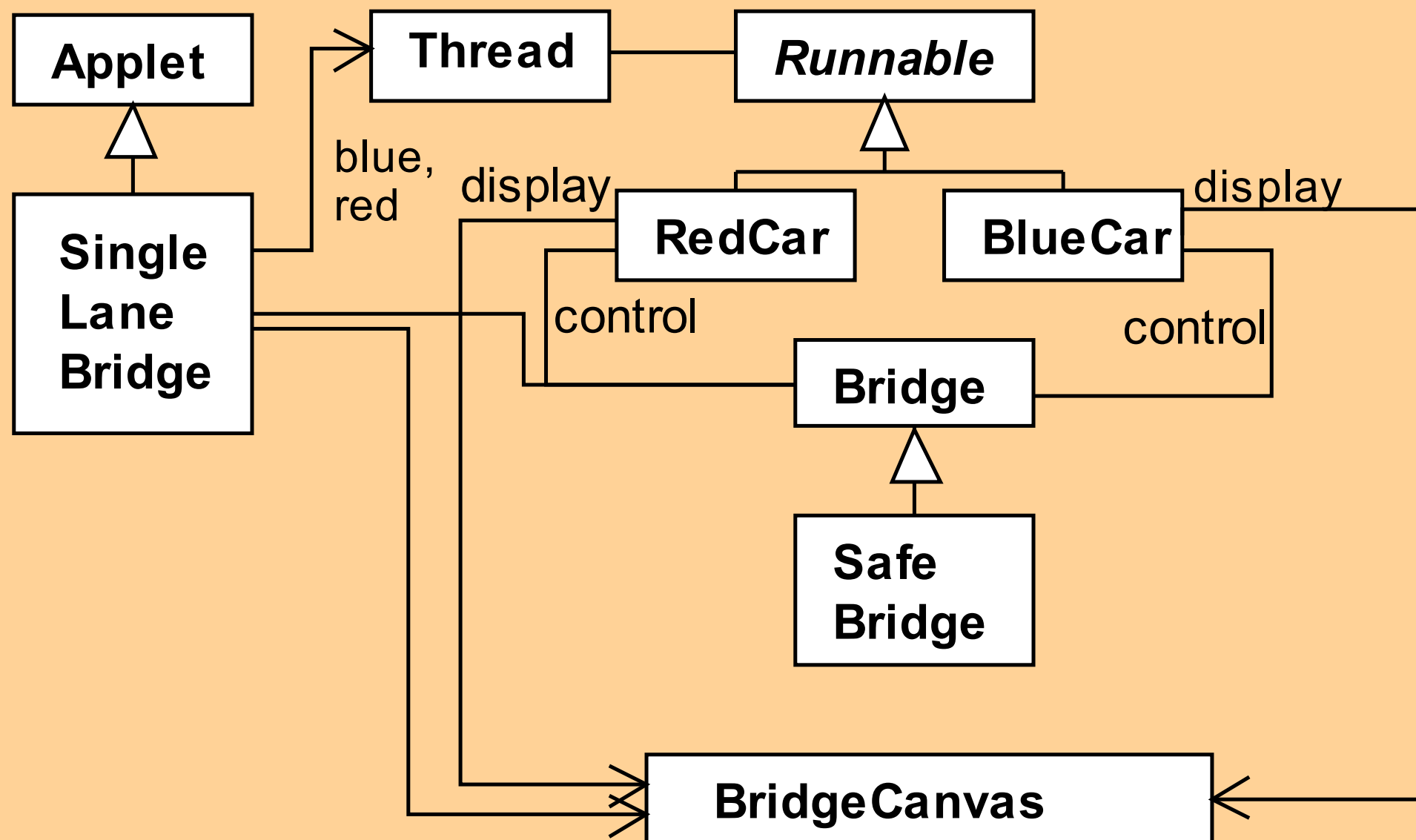
**No deadlocks/errors**

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

**Sin las restricciones  
del BRIDGE, ¿ se  
puede violar la  
propiedad de safety  
ONEWAY?**

**Trace to property violation in ONEWAY:  
red.1.enter  
blue.1.enter**

# SLB - implementación en Java



Las entidades activas (autos) son implementadas como **threads**.  
Las entidades pasivas (puente) como **monitor**.



# SLB - BridgeCanvas

Una instancia de la clase BridgeCanvas es creada por el applet SingleLaneBridge - ref es pasada a cada nuevo objeto creado **RedCar** y **BlueCar**.

```
class BridgeCanvas extends Canvas {
    public void init(int ncars) {...} //establece el número de autos
    //mueve un paso el auto rojo con identidad i
    //devuelve true para el período en el puente , desde poco antes hasta después
    public boolean moveRed(int i)
        throws InterruptedException{...}
    //mueve un paso el auto azul con identidad i
    //devuelve true para el período en el puente , desde poco antes hasta después
    public boolean moveBlue(int i)
        throws InterruptedException{...}
    public synchronized void freeze(){...} // congela display
    public synchronized void thaw(){...}  // descongela display
}
```

# SLB - RedCar

```
class RedCar implements Runnable {

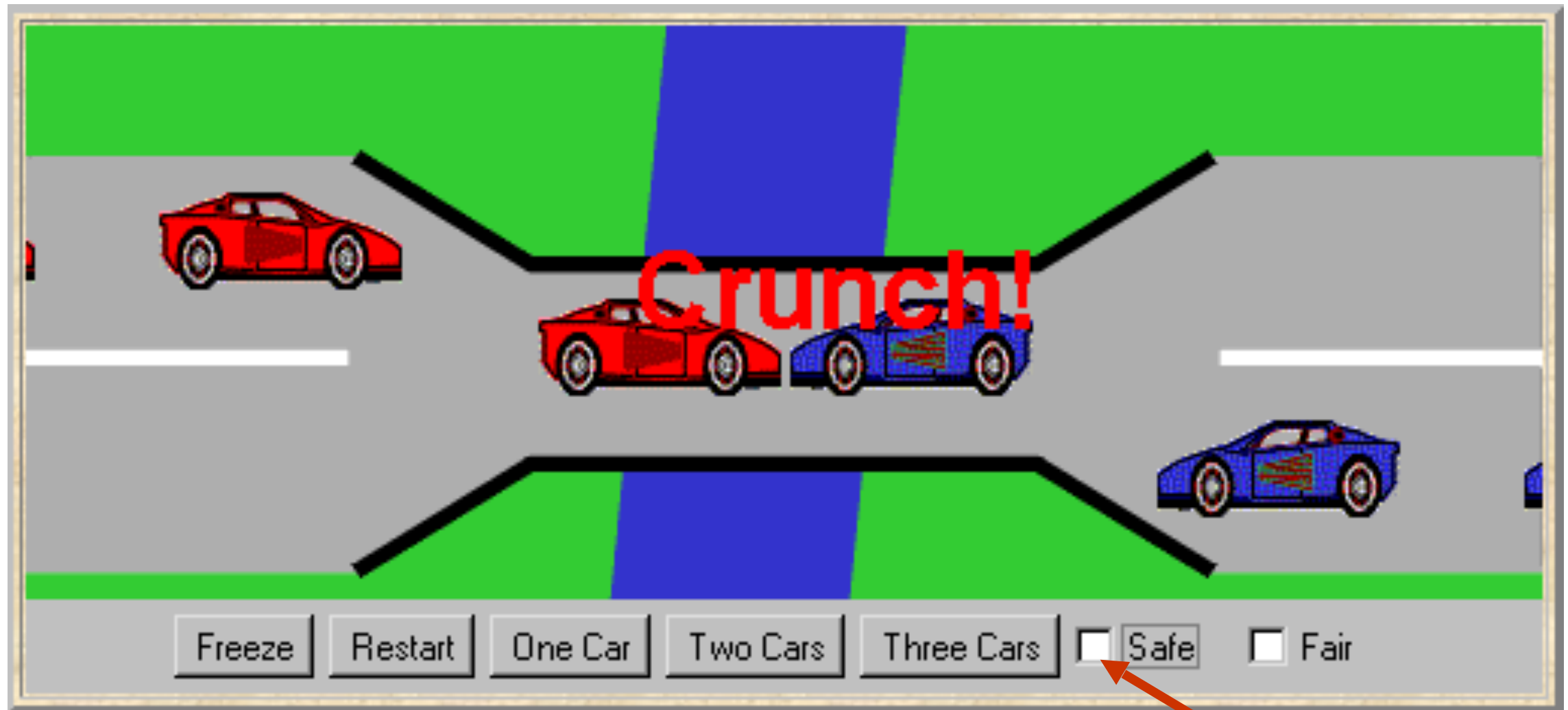
    BridgeCanvas display;
    Bridge control;
    int id;

    RedCar(Bridge b, BridgeCanvas d, int id) {
        display = d;
        this.id = id;
        control = b;
    }

    public void run() {
        try {
            while(true) {
                while (!display.moveRed(id)); // not on bridge
                control.redEnter();
                while (display.moveRed(id)); // move over bridge
                control.redExit();
            }
        } catch (InterruptedException e){}
    }
}
```

**Similar para  
BlueCar**

# Single Lane Bridge



Para garantizar la seguridad , la casilla de verificación "Safe" debe ser elegido con el fin de seleccionar la aplicación **SafeBridge** .

# SLB - SafeBridge

```
class SafeBridge extends Bridge {  
  
    private int nred = 0;  
    private int nblue = 0;  
  
    synchronized void redEnter() throws InterruptedException {  
        while (nblue>0) wait();  
        ++nred;  
    }  
    synchronized void redExit(){  
        --nred;  
        if (nred==0)  
            notifyAll();  
    }  
    synchronized void blueEnter() throws InterruptedException {  
        while (nred>0) wait();  
        ++nblue;  
    }  
    synchronized void blueExit(){  
        --nblue;  
        if (nblue==0)  
            notifyAll();  
    }  
}
```

**Traducción directa del  
modelo BRIDGE**

# SLB - SafeBridge

```
synchronized void blueEnter() throws InterruptedException {  
    while (nred>0) wait();  
    ++nblue;  
}  
synchronized void blueExit(){  
    --nblue;  
    if (nblue==0)  
        notifyAll();  
}
```

Para evitar cambios innecesarios de thread, utilizamos notificaciones condicionales para despertar threads sólo cuando el número de autos en el puente es 0 i.e. cuando el último auto salió del puente.

¿ Puede cada auto finalmente cruzar el puente? Esto es una **propiedad de liveness**.

# Liveness

Una propiedad de **Safety** afirma que nada malo sucede.

Una propiedad de **liveness** afirma que algo bueno finalmente ocurre.

**SLB: ¿Tiene cada auto la oportunidad de cruzar el puente?**

**ie. ¿ hay progreso (PROGRESS) ?**

**Una propiedad de progreso asegura que siempre una acción finalmente es ejecutada. Progreso es opuesto a inanición (situación en la cual una acción nunca es ejecutada)**

# Propiedades de Progreso

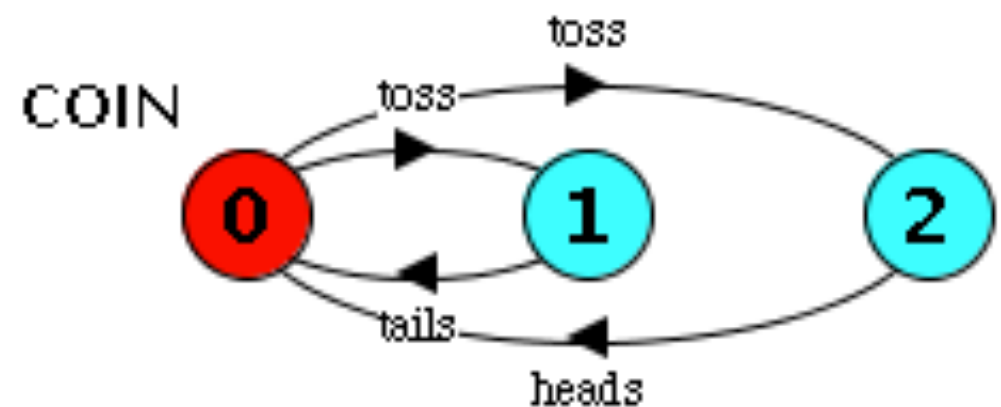
## Elección Justa

**Elección Justa (Fair choice)** : si la elección sobre un conjunto de transiciones es ejecutado infinitas veces, entonces cada transición en el conjunto va a ser ejecutada infinitas veces.

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

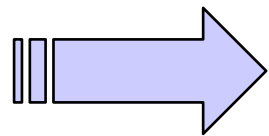
Si lanzamos una moneda infinitas veces, esperamos que tanto cara como cruz puedan ocurrir infinitas veces.

¡ Esto requiere Elección Justa!



# Propiedades de Progreso

**progress  $P = \{a_1, a_2 \dots a_n\}$**  define una propiedad de progreso  **$P$**  la cual afirma que en una ejecución infinita del sistema objeto, al menos una de las acciones  **$a_1, a_2 \dots a_n$**  será ejecutada infinitas veces.



**COIN system:**    progress HEADS = {heads} ?



progress TAILS = {tails} ?



**Verificar progreso con LTSA:**

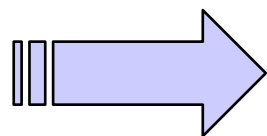
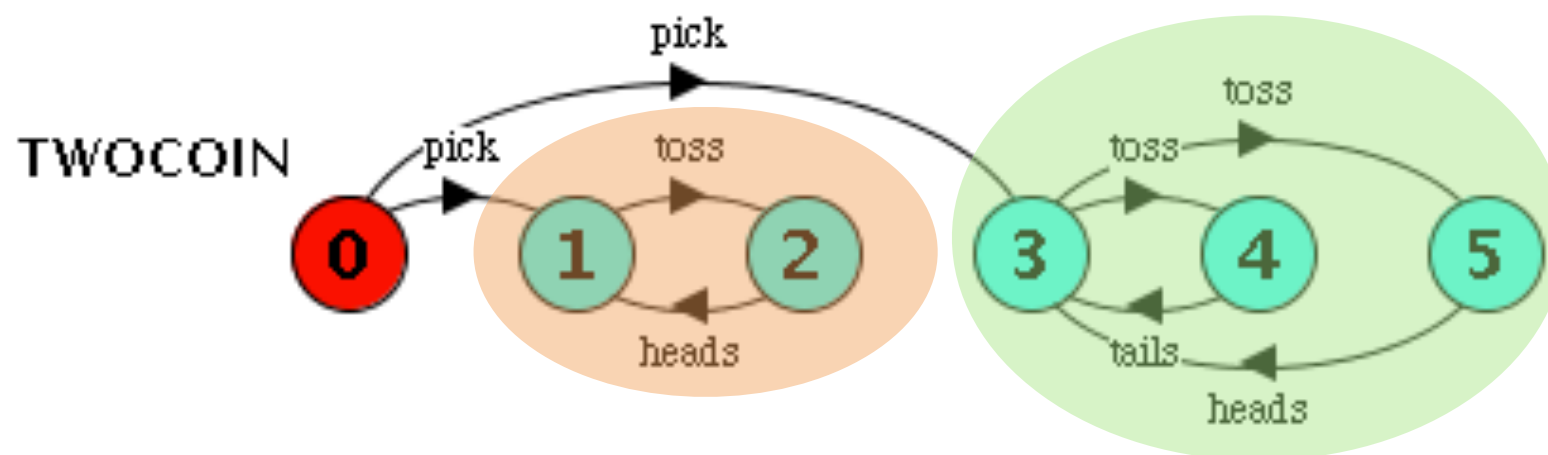
**No progress violations detected.**



# Propiedades de Progreso

Supongamos que hay dos monedas posibles que podrían ser recogidas: una **moneda de truco** y una **moneda normal** .....

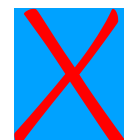
**TWOCOIN** = (**pick**→**COIN**|**pick**→**TRICK**) ,  
**TRICK** = (**toss**→**heads**→**TRICK**) ,  
**COIN** = (**toss**→**heads**→**COIN**|**toss**→**tails**→**COIN**) .



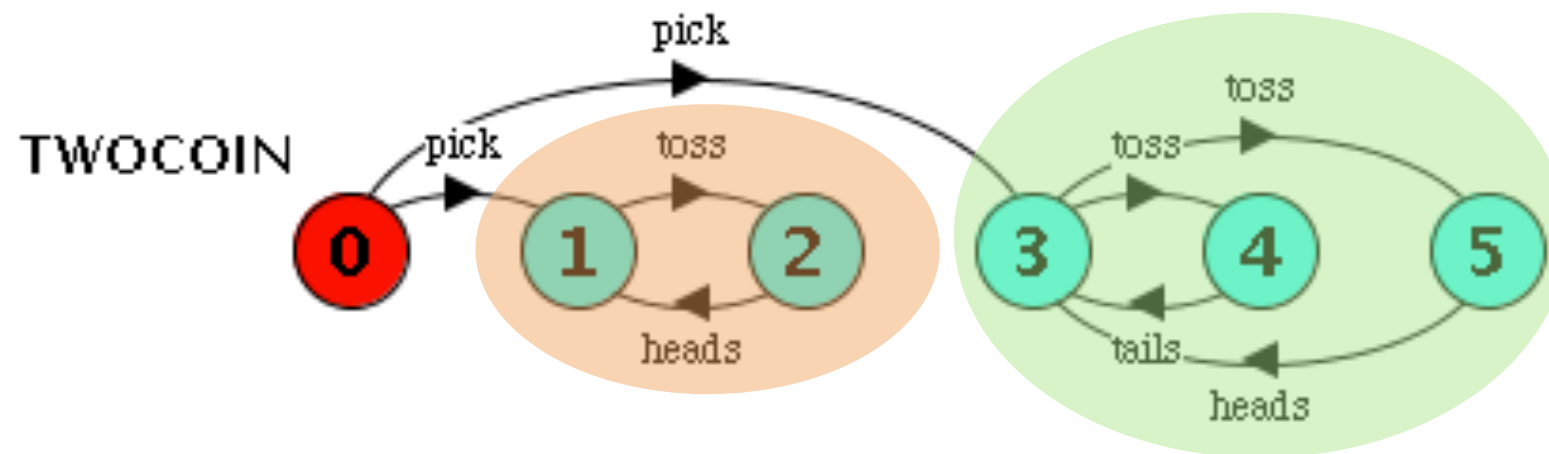
**TWOCOIN: progress HEADS = {heads} ?**



**progress TAILS = {tails} ?**



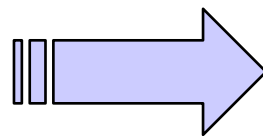
# Propiedades de Progreso



progress HEADS = {heads}

progress TAILS = {tails}

Verificar progreso  
con LTSA



```
Depth 1 -- States: 1 Transitions: 1 Memor
Progress violation: TAILS
Trace to terminal set of states:
    pick
Cycle in terminal set:
    toss
    heads
Actions in terminal set:
    {heads, toss}
Progress Check in: 3ms
```

progress HEADSorTails = {heads,tails}

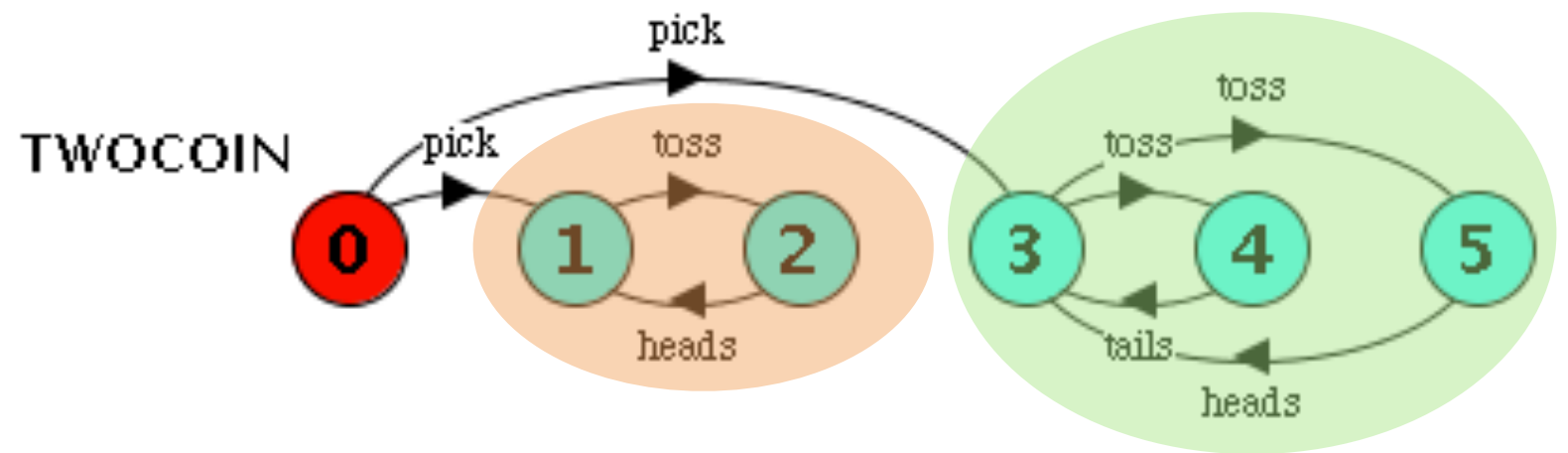
?



# Análisis de Progreso

Un **conjunto terminal de estados** es uno en el cual cada estado es alcanzable desde cualquier otro estado mediante una o más transiciones, y no existen transiciones salientes del conjunto a un estado fuera de él

Terminal sets for  
TWOCOIN:  
 $\{1,2\}$  and  $\{3,4,5\}$

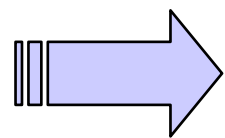


Dada elección justa, cada conjunto terminal representa una ejecución en la cual cada acción utilizada en una transición del conjunto es ejecutada infinitas veces.

Como no existen transiciones salientes de un conjunto terminal, cada acción que **no es utilizada** en el conjunto **puede no ocurrir nunca** en todas las ejecuciones del sistema, y por lo tanto representa una **potencial violación a progreso**

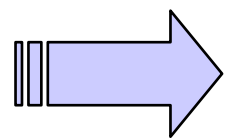
# Análisis de Progreso

Una propiedad de progreso es violada si durante el análisis se encuentra un conjunto terminal de estados en el cual no aparecen ninguna de las acciones indicada en la propiedad.

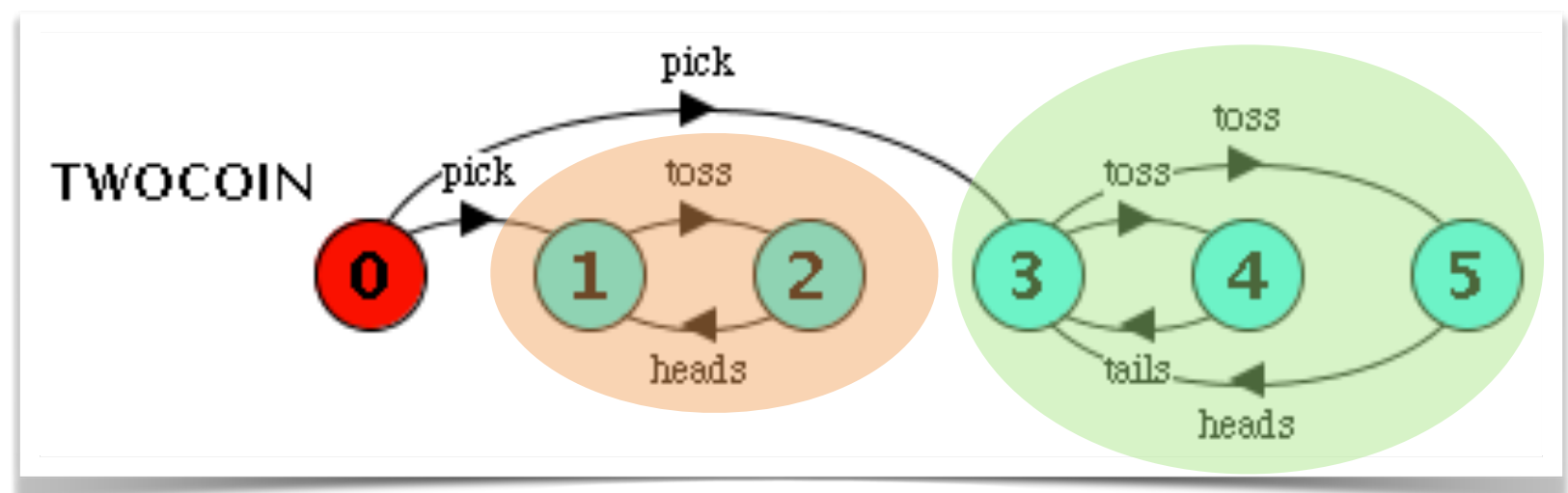


progress TAILS = {tails} en {1,2}

Por defecto: dada **elección justa**, para cada acción en el alfabeto del sistema objeto, dicha acción será ejecutada infinitas veces. Esto es equivalente a especificar una propiedad de progreso separada para cada acción.



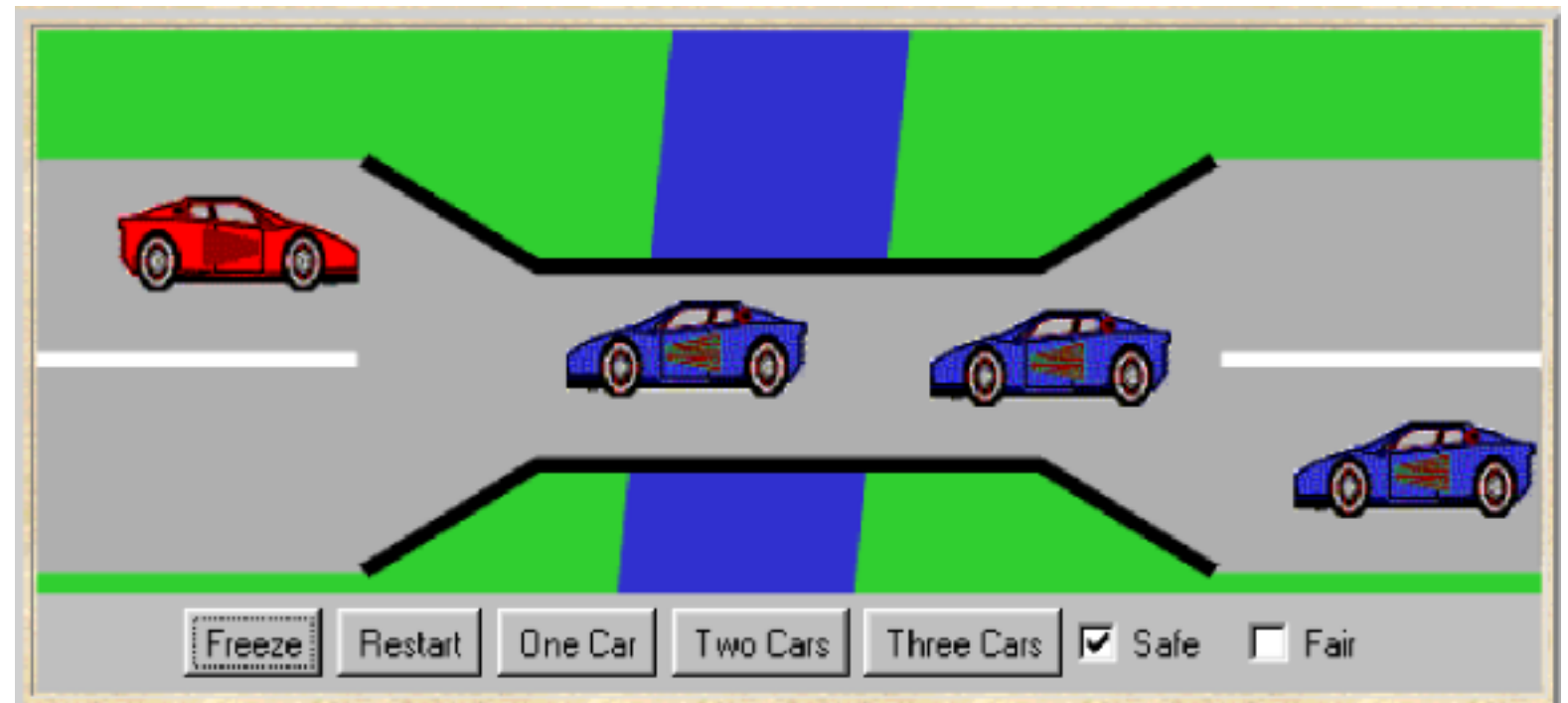
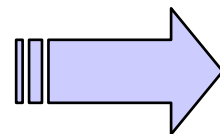
Default analysis  
for TWOCOIN?



# Progreso - SLB

La implementación de SLB puede permitir violaciones a progreso.

Sin embargo, si aplicamos análisis por defecto de progreso, ninguna violación es detectada!  
¿Por qué no?



**progress BLUECROSS = {blue[ID].enter}  
progress REDCROSS = {red[ID].enter}  
No progress violations detected.**

Elección justa significa que finalmente cada posible ejecución ocurre, incluyendo aquellas en las cuales los autos no sufren de inanición. Para detectar problemas de progreso debemos verificar en condiciones adversas. Imponiendo alguna política de planificación de acciones que modele el escenario en el cual el puente está congestionado.

# Progreso - Prioridad de acciones

Las **expresiones de prioridad** de acciones describen propiedades de planificación:

**Alta  
prioridad  
(" << ")**

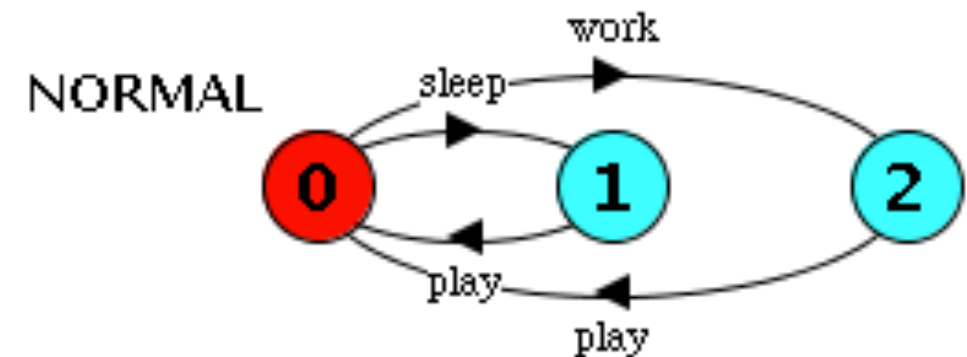
$||C = (P || Q) << \{a_1, \dots, a_n\}$  especifica una composición en la cual las acciones  $a_1, \dots, a_n$  tiene mayor prioridad que cualquier otra acción en el alfabeto de  $P || Q$  incluyendo la acción silenciosa **tau**. En cualquier elección en este sistema que tenga una o mas acciones  $a_1, \dots, a_n$  como etiqueta de una transición, las transiciones etiquetadas con acciones de menor prioridad son descartadas.

**Baja  
Prioridad  
(" >> ")**

$||C = (P || Q) >> \{a_1, \dots, a_n\}$  especifica una composición en la cual las acciones  $a_1, \dots, a_n$  tiene menor prioridad que cualquier otra acción en el alfabeto de  $P || Q$  incluyendo la acción silenciosa **tau**. En cualquier elección en este sistema que tenga con una o mas transiciones no etiquetadas con  $a_1, \dots, a_n$ , las transiciones con  $a_1, \dots, a_n$  son descartadas.

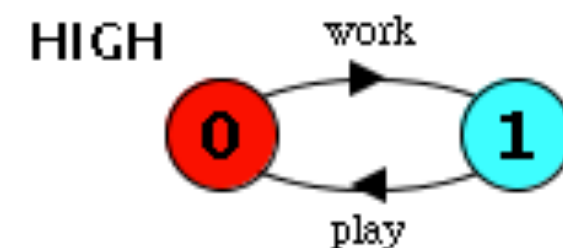
# Progreso - Prioridad de acciones

**NORMAL** = (work → play → NORMAL  
| sleep → play → NORMAL) .

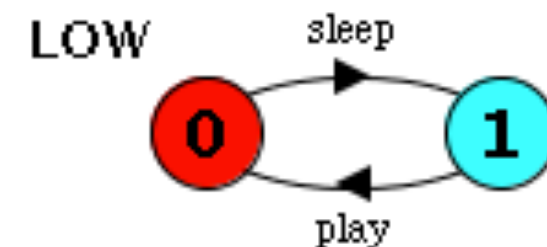


La prioridad sobre acciones simplifica el LTS resultante descartando las acciones con baja prioridad de las elecciones.

**HIGH** = (NORMAL) << {work} .



**LOW** = (NORMAL) >> {work} .





# SLB Congestionado

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

BLUECROSS - Finalmente algún auto azul estará habilitado para entrar

REDCROSS - Finalmente algún auto rojo estará habilitado para entrar

¿Congestión utilizando prioridad sobre acciones?

¿Podríamos dar prioridad a los autos rojos sobre los azules  
(o vice versa) ?

Usualmente ninguno tiene prioridad sobre el otro.

En su lugar, simplemente forzamos la congestión mediante la reducción  
de la prioridad de las acciones de salida de los dos coches desde el  
puente.

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

¿ Análisis de Progreso ? ¿ LTS ?



# Model de SLB congestionado

**Progress violation: BLUECROSS**

**Path to terminal set of states:**

**red.1.enter**

**red.2.enter**

**Actions in terminal set:**

**{red.1.enter, red.1.exit, red.2.enter, red.2.exit,  
red.3.enter, red.3.exit}**

**Progress violation: REDCROSS**

**Path to terminal set of states:**

**blue.1.enter**

**blue.2.enter**

**Actions in terminal set:**

**{blue.1.enter, blue.1.exit, blue.2.enter, blue.2.exit,  
blue.3.enter, blue.3.exit}**

**Este escenario  
se corresponde a  
la situación en la  
cual una vez que  
un auto de color  
entra al puente  
mantiene  
continuamente  
su ocupación.**

# Model de SLB congestionado

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

