

# Diseño y Verificación de Programas Concurrentes

Escuela de Informática - CACIC 2021

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

# Implementación vs Diseño

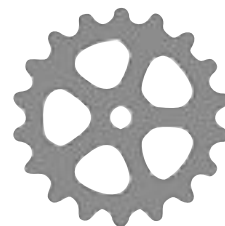
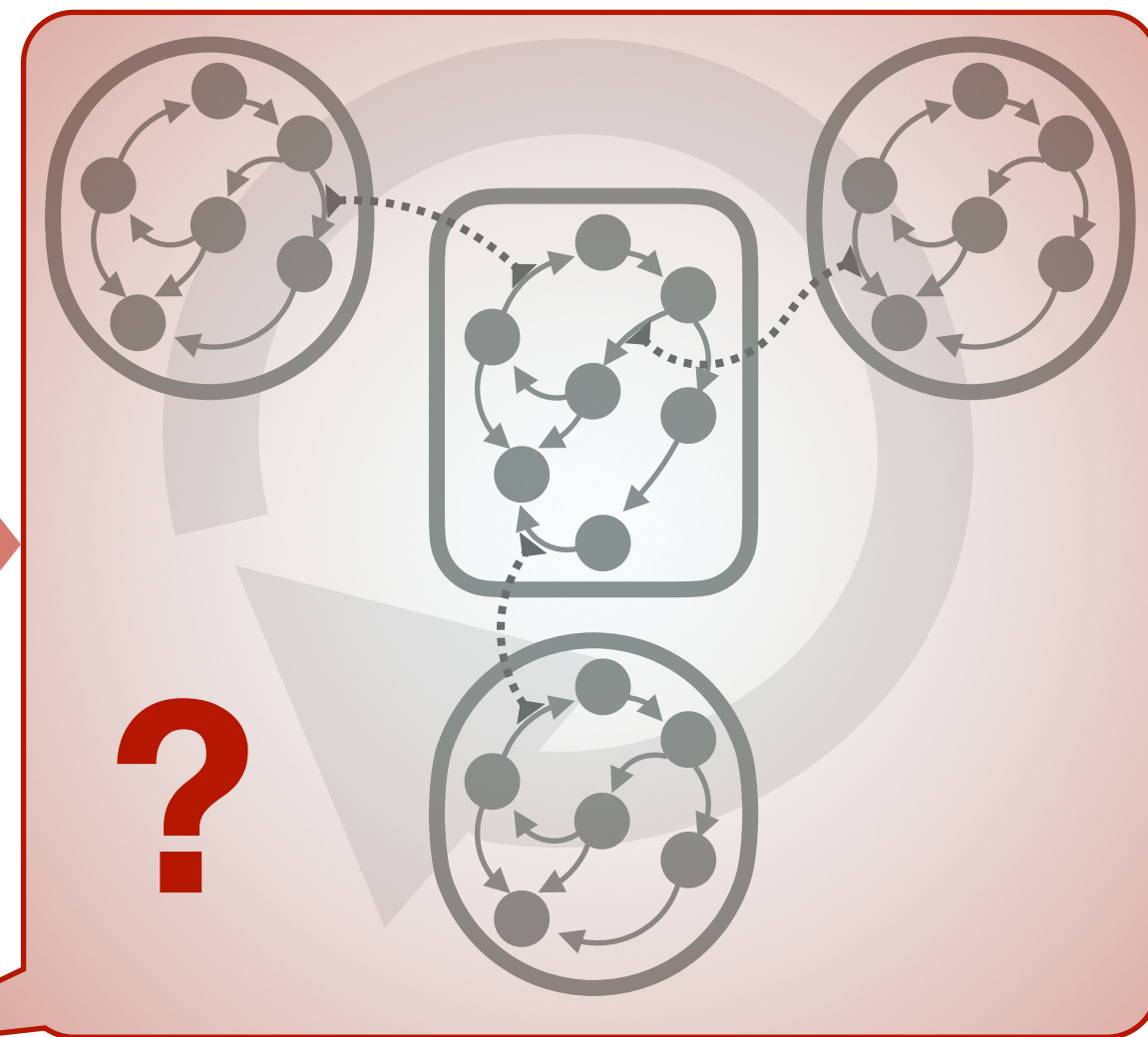
Modelos de programas concurrentes (Abstracción)

## Implementación del Sistema

```
Principal.java x
1 package com.carlos;
2
3 /**
4  * Esta es la clase principal del programa.
5  * @author carlos
6  * @version 1.0
7  * @since 19/08/2019*/
8
9 public class Principal {
10
11     public enum Dias {Lunes, Martes, Miércoles};
12
13     public static void main(String[] args) {
14
15         int num1 = 9;
16         float num2 = 9.5f;
17         float suma = num1 + num2;
18
19         System.out.println(suma);
20     }
21 }
```



## Modelos Abstracto del Sistema

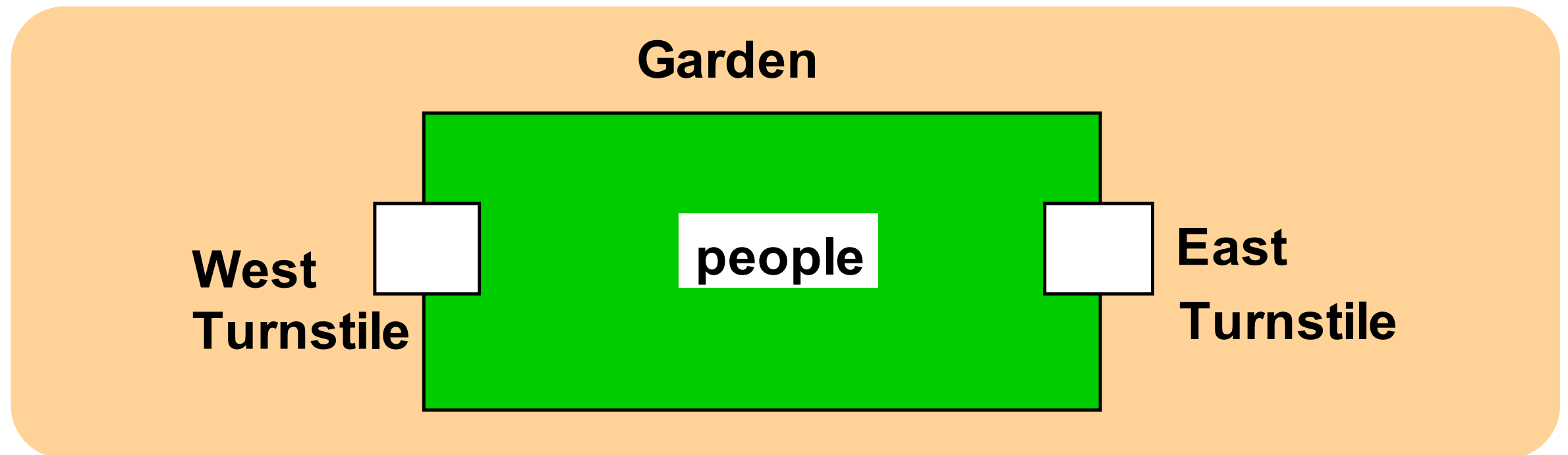


**Análisis automático  
(Model checking)**

# Interferencia

Ejemplo: **Problema del Jardín ornamental**

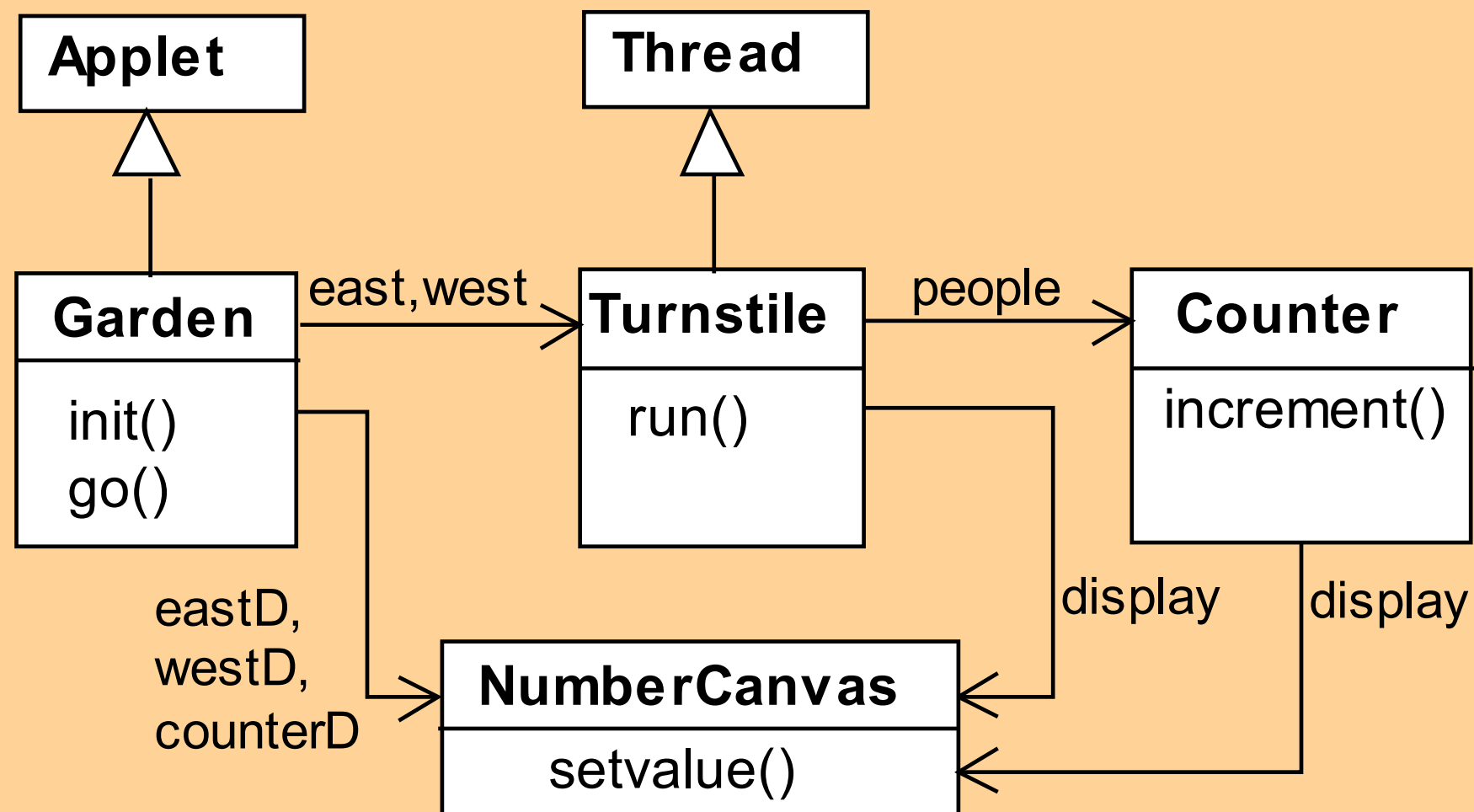
Las personas entran en un jardín ornamental a través de cualquiera de los dos molinillos. La Administración desea saber cuántas personas hay en el jardín en cualquier momento.



Una implementación del problema consiste en **dos threads** concurrentes y un **objeto contador compartido**.

# Programa: Jardín Ornamental

## Diagrama de Clases



**El thread Turnstile (molinillo) simula el arribo periódico de un visitante al jardín cada un segundo. Esto lo realiza durmiendo el thread (1 segundo) y luego invocando el método `increment()` del objeto `counter`.**

# Programa: Jardín Ornamental

El objeto **Counter** y los threads de los **Turnstile** (molinillos) son creados por el método **go()** del applet **Garden**:

```
private void go() {  
    counter = new CounterApplet(counterD);  
    west= new TurnstileApplet(westD, counter);  
    east= new TurnstileApplet(eastD, counter);  
    west.start();  
    east.start();  
}
```

counterD, westD y eastD  
son objetos **NumberCanvas**.

# La clase Turnstile

```
class Turnstile extends Thread {
    NumberCanvas display;
    CounterApplet people;

    Turnstile(NumberCanvas n, CounterApplet c)
    { display = n; people = c; }

    public void run() {
        try{
            display.setvalue(0);
            for (int i=1;i<=Garden.MAX;i++){
                Thread.sleep(500); //0.5 second
                display.setvalue(i);
                people.increment();
            }
        } catch (InterruptedException e) {}
    }
}
```

El método `run()` finaliza y el thread termina después de que `Garden.MAX` visitantes hayan entrado.

# La clase Counter

```
class Counter {  
  
    int value=0;  
    NumberCanvas display;  
  
    CounterApplet(NumberCanvas n) {  
        display=n;  
        display.setvalue(value);  
    }  
  
    void increment() {  
        int temp = value;    //read[v]  
        Simulate.HWinterrupt();  
        value=temp+1;        //write[v+1]  
        display.setvalue(value);  
    }  
}
```

Las interrupciones del Hardware pueden ocurrir en **cualquier momento** (aleatoriamente).

El contador simula una interrupción de hardware en **increment()**, entre la obtención y la modificación del valor del contador. El método Interrupt invoca de manera aleatoria a **Thread.sleep()** forzando un cambio de thread.

# Programa: Jardín ornamental

## Display



Luego de que los threads correspondientes a los molinillos East (Este) y West (Oeste) incrementan su contador 20 veces, el contador de las personas en el jardín **no es la suma** de dichos contadores. Algunos incrementos se perdieron. ¿ Por qué ?



# Activación de métodos concurrentes

En Java **las activaciones de los métodos no son atómicas.**

En este caso particular, los threads East y West pueden ejecutar el código correspondiente al método **increment()** al **mismo tiempo.**

# Modelo del Jardín ornamental

```
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.

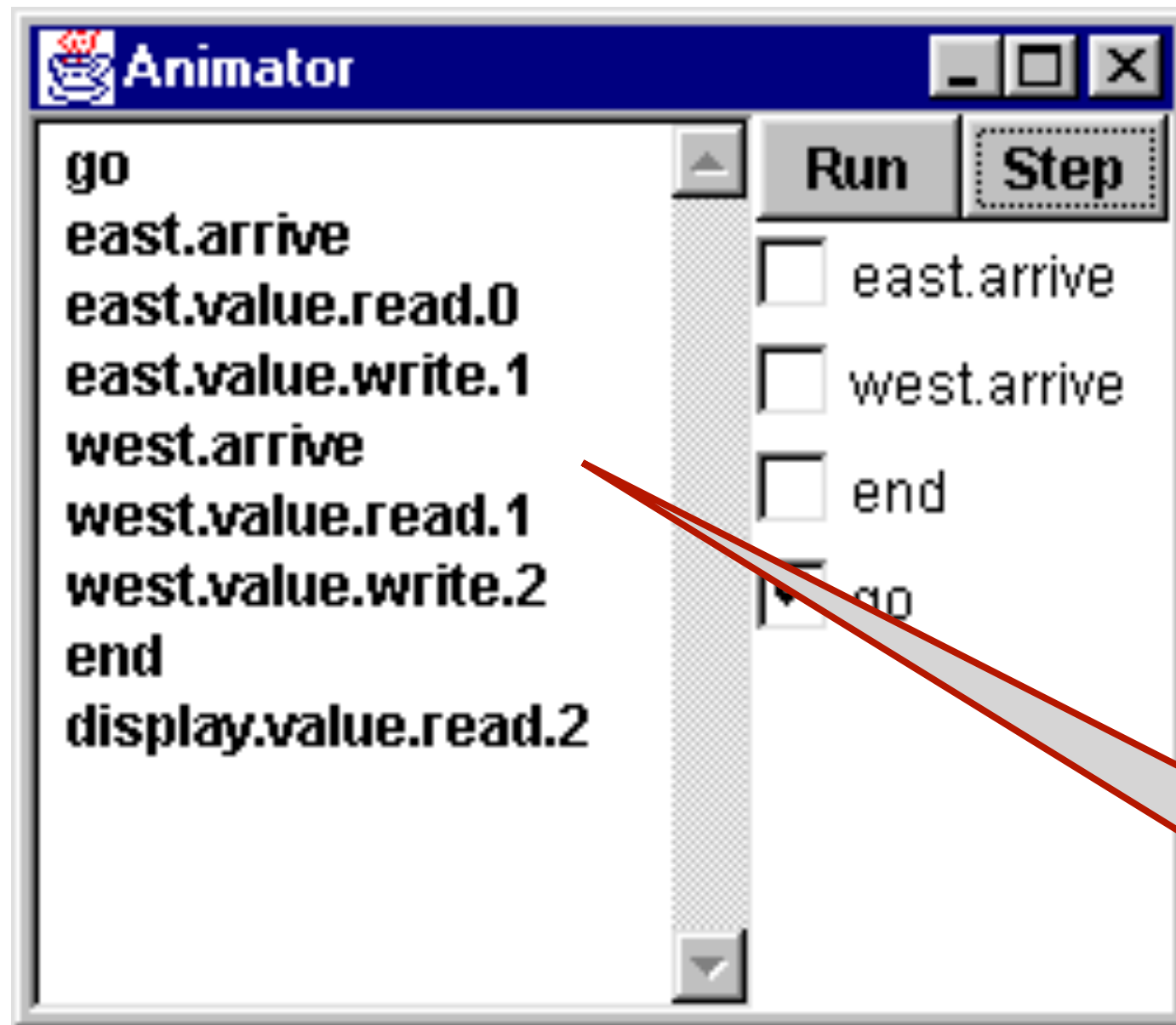
|| GARDEN = (east:TURNSTILE || west:TURNSTILE
|| { east,west,display}::value:VAR)
/{ go /{east,west}.go,
  end/{east,west}.end} .
```

El alfabeto del proceso compartido **VAR** es declarado explícitamente como el conjunto de constantes **VarAlpha**.

El alfabeto de **TURNSTILE** es extendido con **VarAlpha** para asegurar la inexistencia de acciones libres no deseadas en **VAR** e.g. **value.write[0]**. Todas las acciones en **VAR** deben estar controladas (compartidas) por algún **TURNSTILE**.

# Buscando errores

## Animación



Comprobando escenarios. Usar el **simulador** para producir una traza.

Esta traza, ¿ es correcta ?

# Buscando errores

## Análisis exhaustivo

**Verificación Exhaustiva:** componer el modelo con un proceso TEST que sume los arribos y que contraste el resultado con el valor del display:

```
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}.
```

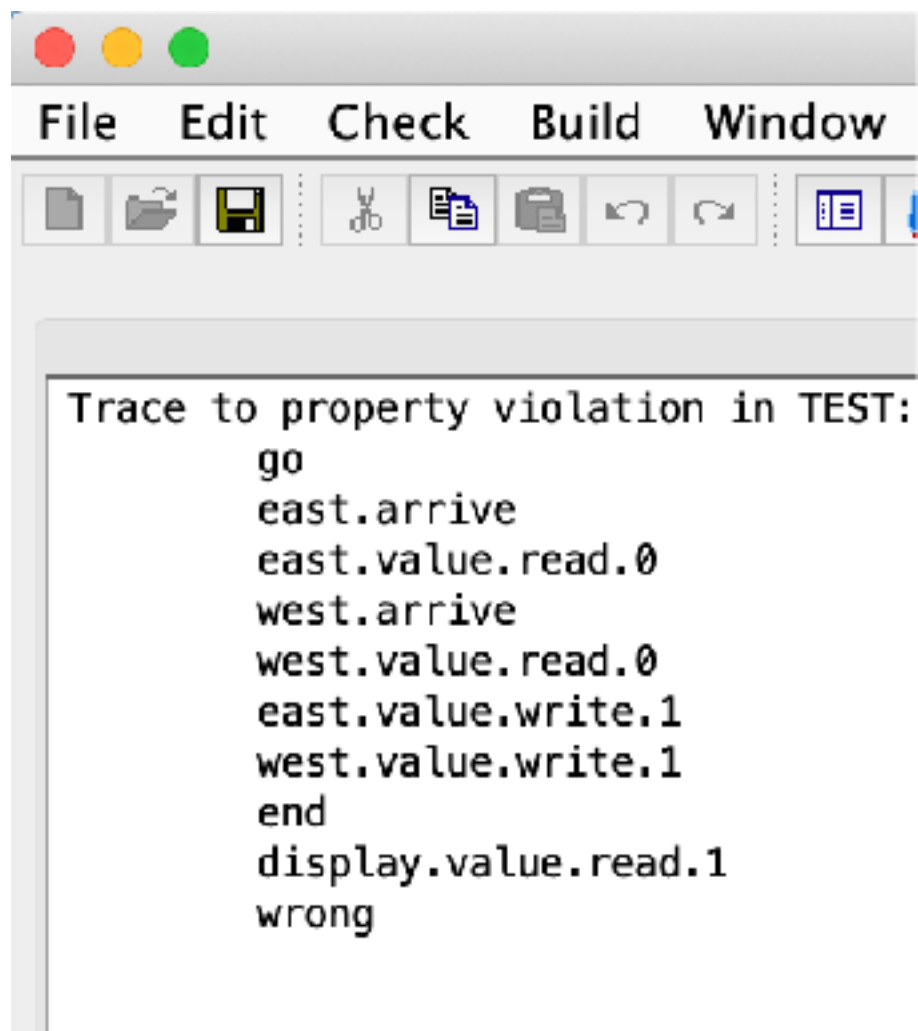
Al igual que **STOP**, **ERROR** representa un estado predefinido de FSP, enumerado con **-1**.

# Modelo Jardín ornamental

## Buscando errores

**|| TESTGARDEN = (GARDEN || TEST) .**

Usar LTSA para realizar una búsqueda exhaustiva de **ERROR**.



The screenshot shows the LTSA software interface. It has a menu bar with 'File', 'Edit', 'Check', 'Build', and 'Window'. Below the menu is a toolbar with icons for file operations and execution. The main window displays a trace of events leading to a property violation in a test named 'TEST'.

```
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
```

LTSA produce la traza **más corta**  
para alcanzar el estado **ERROR**

# Interferencia y Exclusión Mutua

Las **actualizaciones** de valores **destructivas** causadas por la ocurrencia arbitraria (interleaving) de las acciones read y write, se denomina **interferencia**.

Los errores (bugs) causados por **interferencia** son extremadamente **difíciles** de detectar y localizar.

La solución general a este tipo de problemas es realizar las acciones sobre **objetos compartidos** de **manera exclusiva** (acceso exclusivo). La **exclusión mutua** se puede modelar a través de **acciones atómicas**.

# Exclusión mutua en Java

En Java, las activaciones concurrentes de métodos puede realizarse de **manera exclusiva** prefijando en la declaración del método la palabra reservada **synchronized**, la cual produce un bloqueo en la utilización del objeto.

Corregimos la clase COUNTER generando una nueva que herede de ella y declarando el método de incremento como sincronizado:

```
class SynchronizedCounter extends Counter {  
  
    SynchronizedCounter(NumberCanvas n)  
    {super(n);}  
  
    synchronized void increment() {  
        super.increment();  
    }  
}
```

Bloquea el objeto

Desbloquea el  
objeto

# Exclusión Mutua

El Jardín ornamental



Java **asocia** un **lock** (bloqueo) a cada objeto. El compilador de Java inserta el **código** de la **adquisición** del lock correspondiente **antes** de ejecutar el método y su respectiva **liberación** una vez que el método ha **retornado**. Los **threads concurrentes** sobre el mismo objeto compartido son **bloqueados** hasta que el lock es liberado.



# La sentencia `synchronized` en Java

El acceso a un objeto puede hacerse de manera **exclusiva** mediante la utilización de la sentencia **`synchronized`**:

**`synchronized (object) { statements }`**

Una forma menos elegante de corregir el ejemplo podría ser modificando el método **`Turnstile.run()`**:

```
...  
synchronized (people) {people.increment();}  
...
```

# La sentencia `synchronized` en Java

El acceso a un objeto puede hacerse de manera **exclusiva** mediante la utilización de la sentencia **`synchronized`**:

**`synchronized (object) { statements }`**

Una forma menos elegante de corregir el ejemplo podría ser modificando el método **`Turnstile.run()`**:

```
...  
synchronized (people) {people.increment();}  
...
```

¿ Por qué es  
“menos elegante” ?

# Modelando exclusión mutua

Para agregar bloqueo al modelo, **definir** un **LOCK**, componerlo junto con la variable compartida **VAR** en el jardín, y modificar el conjunto de alfabeto:

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

```
||LOCKVAR = (LOCK || VAR).
```

```
set VarAlpha = {value.{read[T], write[T], acquire, release}}
```

Modificar el proceso TURNSTILE para adquirir y liberar el lock:

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

# Modelo del Jardín ornamental revisado

Buscando errores

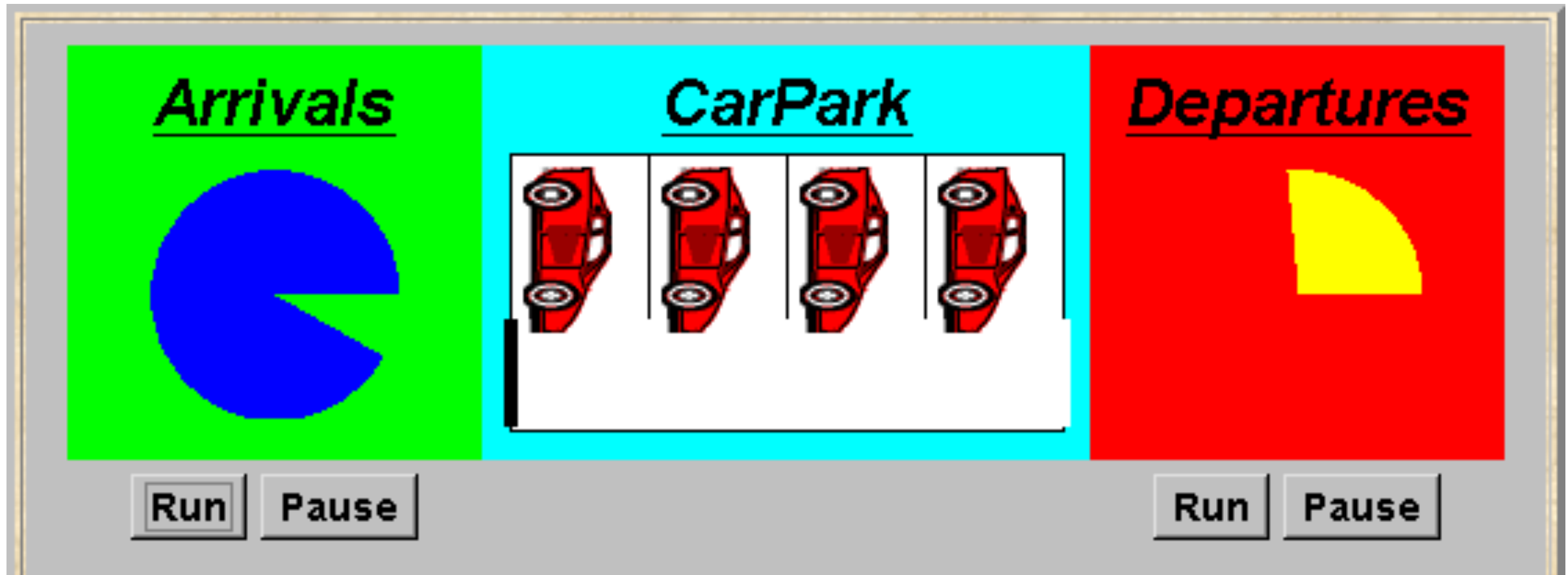
Un ejemplo del simulador de trazas de ejecución

Usar TEST y LTSA para realizar una verificación exhaustiva.

¿ Se cumple TEST ?

# **Monitores y Sincronización Condicional**

# Sincronización Condicional



Se necesita un controlador para el estacionamiento. Éste sólo debe permitir la entrada de autos cuando el estacionamiento no está lleno, y la salida de los mismos, sólo cuando no está vacío. La llegada y salida de autos es simulada por diferentes threads.

# Modelo de Chochera

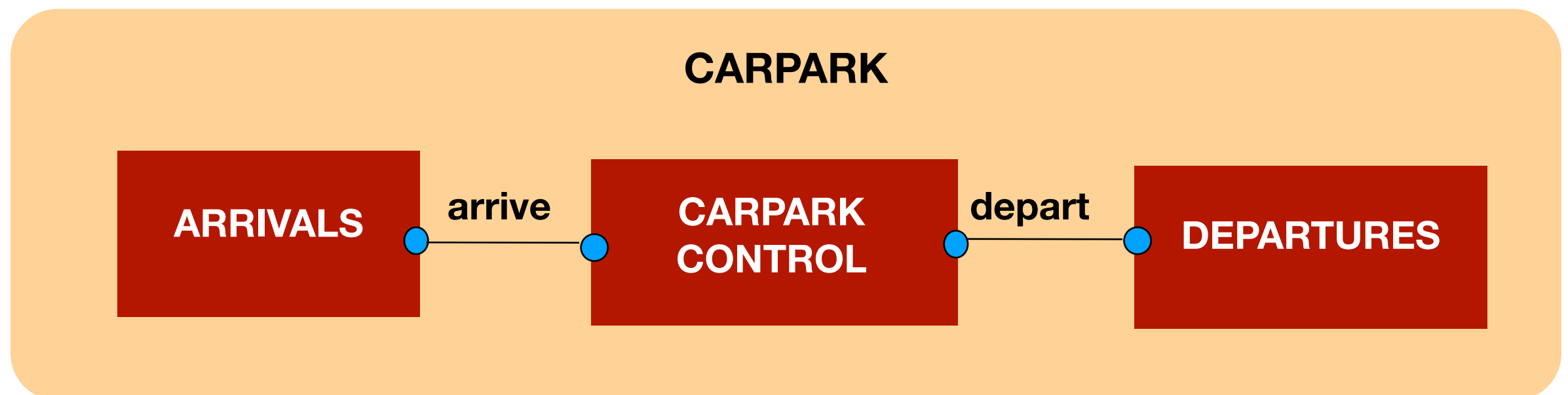
- ¿Qué eventos o acciones nos interesan?

arrive y depart

- Identificar procesos.

Ilegadas, salidas y control del estacionamiento.

- Definir cada proceso y sus interacciones (estructura).



# Modelo de Estacionamiento

```
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).
```

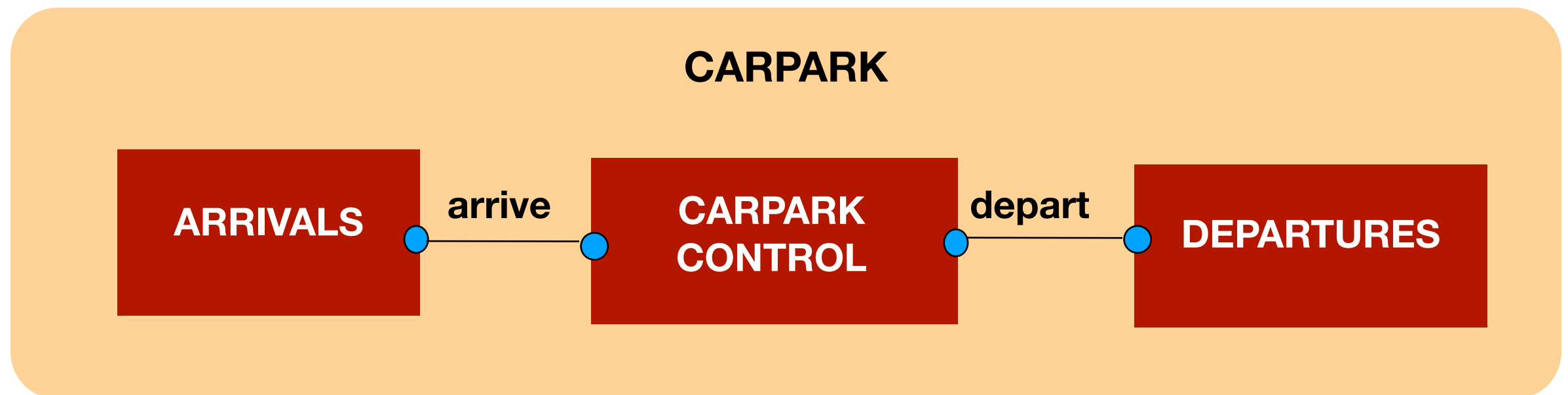
```
||CARPARK = (ARRIVALS || CARPARKCONTROL(4) || DEPARTURES).
```

Las acciones con guardas son utilizadas para controlar la llegada (si no está lleno) y salida (si no está vacío) de autos.



# Programa del Estacionamiento

- **Modelo** - toda identidad son procesos que interactúan mediante acciones
- **Programa** - necesitamos identificar threads y monitores
  - thread - **entidad activa** que inicia acciones (output)
  - monitor - **entidad pasiva** que responde a acciones (input).



# Programa Estacionamiento

`Arrivals` y `Departures` implementan a `Runnable`, `CarParkControl` provee el control (sincronización condicional).

Las instancias de éstos son creadas por el método `start()` del applet `CarPark`:

```
public void start() {  
    CarParkControl c = new DisplayCarPark(carDisplay, Places);  
    arrivals.start(new Arrivals(c));  
    departures.start(new Departures(c));  
}
```

# Programa Estacionamiento

## Threads Arrivals y Departures

```
class Arrivals implements Runnable {  
  
    CarParkControl carpark;  
  
    Arrivals(CarParkControl c) {  
        carpark = c;  
    }  
  
    public void run() {  
        try {  
            while(true) {  
                ThreadPanel.rotate(330);  
                carpark.arrive();  
                ThreadPanel.rotate(30);  
            }  
        } catch (InterruptedException e) {}  
    }  
}
```

De manera similar  
Departures invoca a  
`carpark.depart()`.

¿ Cómo implementamos el control de CarParkControl?

# Programa Estacionamiento

## Monitor CarParkControl

```
class CarParkControl {  
  
    protected int spaces;  
    protected int capacity;  
  
    CarParkControl(int n) {  
        capacity = spaces = n;  
    }  
  
    synchronized void arrive() throws InterruptedException {  
        while (spaces==0) wait();  
        --spaces;  
        notifyAll();  
    }  
  
    synchronized void depart() throws InterruptedException {  
        while (spaces==capacity) wait();  
        ++spaces;  
        notifyAll();  
    }  
}
```

Exclusión mutua  
mediante métodos  
sincronizados

¿ sincronización  
condicional ?

¿ bloquear si está lleno  
? (spaces==0)

¿ bloquear si está vacío  
? (spaces==N)

# Sincronización Condicional en Java

Java provee un **conjunto de threads en espera** por monitor (en general por objeto) con los siguientes métodos:

**public final void notify()**

Notifica (despierta) a **un** thread que se encuentra esperando en el conjunto de espera del objeto.

**public final void notifyAll()**

Notifica (despierta) a **todos** los threads que se encuentra esperando en el conjunto del objeto.

**public final void wait() throws InterruptedException**

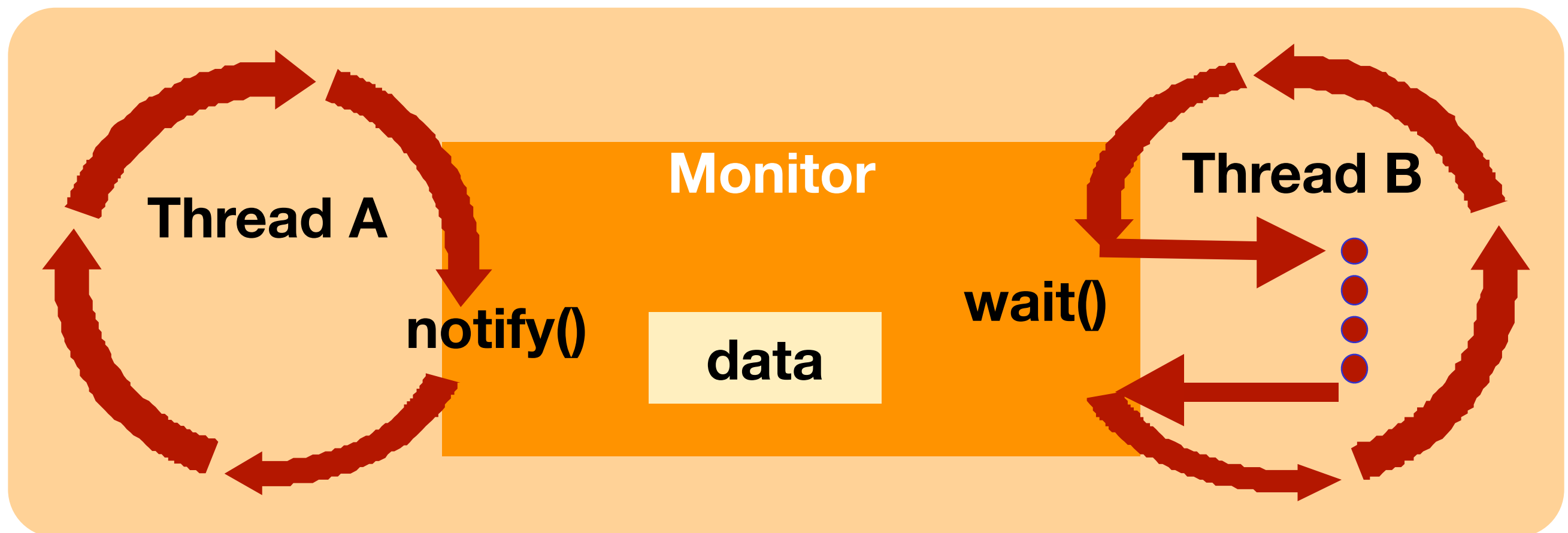
**Espera** ser notificado por otro thread. El thread en espera libera el bloqueo asociado al monitor.

Cuando es **notificado** (despertado) el thread debe esperar a adquirir el monitor de antes seguir con su ejecución.

# Sincronización Condicional en Java

Denominamos a un thread **entrante** a un monitor cuando adquiere el bloqueo exclusivo asociado al monitor y **saliente**, cuando libera el bloqueo del monitor.

**Wait()** - ocasiona la salida del monitor, permitiendo que otros threads entre en el mismo.



# Sincronización Condicional en Java

**FSP:** `when cond act -> NEWSTAT`

**Java:** `public synchronized void act() throws InterruptedException{  
 while (!cond) wait();  
 // modify monitor data  
 notifyAll()  
}`

El ciclo `while` es necesario para comprobar la condición `cond` para asegurar que es satisfecha cuando reingrese al monitor.

`notifyall()` es necesario para despertar otros thread(s) que puede estar esperando entrar al monitor, ahora que los datos han cambiando.

# CarParkControl

## Sincronización Condicional

```
class CarParkControl {  
  
    protected int spaces;  
    protected int capacity;  
  
    CarParkControl(int n) {  
        capacity = spaces = n;  
    }  
  
    synchronized void arrive() throws InterruptedException {  
        while (spaces==0) wait();  
        --spaces;  
        notifyAll();  
    }  
  
    synchronized void depart() throws InterruptedException{  
        while (spaces==capacity) wait();  
        ++spaces;  
        notifyAll();  
    }  
}
```

¿ es correcto usar aquí notify() en vez de notifyAll() ?



# Modelos a Monitores

## Resumen

Las entidades **activas** (las cuales inician acciones) son implementadas como threads. Las entidades **pasivas** (que responden a acciones) son implementados como monitores.

Cada acción con guarda en el modelo de un monitor es implementada como un método sincronizado, el cual usa un ciclo y **wait()** para implementar la la guarda. La **condición** de la guarda es la **negación** de la correspondiente a la del modelo.

Los **cambios** en el monitor son **avisados** a los threads en espera usando **notify()** o **notifyAll()**.

# Semáforos

Los **semáforos** son muy utilizados para lidiar con la **sincronización entre procesos** en sistemas operativos. Un semáforo **s** es una variable entera que sólo puede tener valores positivos.

Las **operaciones** permitidas sobre **s** son **up(s)** y **down(s)**. Los procesos bloqueados se mantienen en una cola **FIFO**.

**down(s):**

if  $s > 0$  then

    decrementar **s**

else

    bloquear la ejecución del proceso  
    invocante

**up(s):**

if proceso bloqueado en **s** then

    despertar uno de ellos

else

    incrementar **s**

# Modelando Semáforos

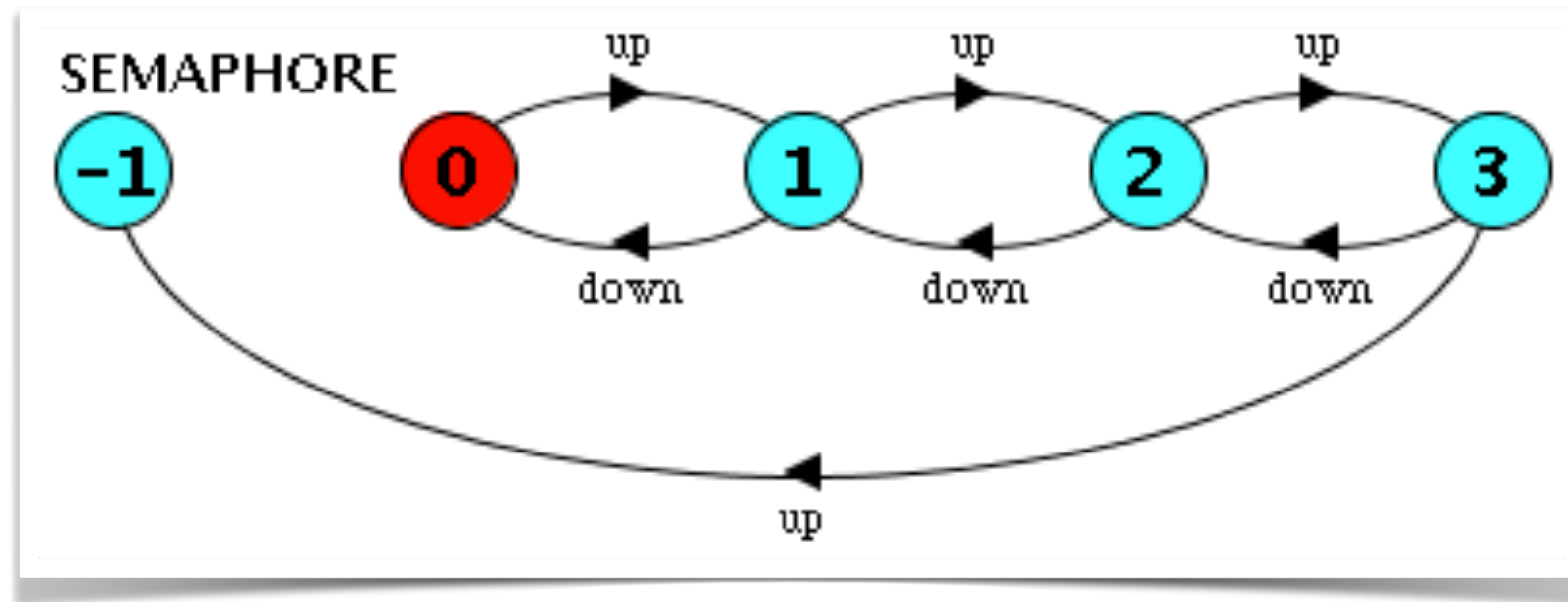
Para poder analizarlos, sólo modelamos semáforos con un **rango finito** de valores. Si este **intervalo es superado**, consideramos la situación como un **ERROR**. N es el valor inicial.

```
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.
```

¿ LTS ?

# Modelando Semáforos



La **acción down** es aceptada sólo cuando el valor  $v$  del semáforo es **mayor que 0**.

La **acción up** no tiene guarda.

Traza de error ( violación):

up -> up -> up -> up

# Ejemplo Semáforo

## Modelo

Tres procesos  $p[1..3]$  usan un semáforo compartido **mutex** para asegurar el acceso exclusivo a algún recurso (acción crítica).

```
LOOP = (mutex.down -> critical -> mutex.up -> LOOP) .  
|| SEMADEMO = (p[1..3]:LOOP  
|| {p[1..3]}::mutex:SEMAPHORE(1)) .
```

# Ejemplo Semáforo

## Modelo

Tres procesos  $p[1..3]$  usan un semáforo compartido **mutex** para asegurar el acceso exclusivo a algún recurso (acción crítica).

```
LOOP = (mutex.down -> critical -> mutex.up -> LOOP) .  
|| SEMADEMO = (p[1..3]:LOOP  
|| {p[1..3]}::mutex:SEMAPHORE(1)) .
```

Para exclusión mutua, el valor inicial del semáforo es 1. ¿ Por qué ?

# Ejemplo Semáforo

## Modelo

Tres procesos  $p[1..3]$  usan un semáforo compartido **mutex** para asegurar el acceso exclusivo a algún recurso (acción crítica).

```
LOOP = (mutex.down -> critical -> mutex.up -> LOOP) .  
|| SEMADEMO = (p[1..3]:LOOP  
|| {p[1..3]}::mutex:SEMAPHORE(1)) .
```

Para exclusión mutua, el valor inicial del semáforo es 1. ¿ Por qué ?  
¿ El estado ERROR es alcanzable en SEMADEMO ?

# Ejemplo Semáforo

## Modelo

Tres procesos  $p[1..3]$  usan un semáforo compartido **mutex** para asegurar el acceso exclusivo a algún recurso (acción crítica).

```
LOOP = (mutex.down -> critical -> mutex.up -> LOOP) .  
|| SEMADEMO = (p[1..3]:LOOP  
|| {p[1..3]}::mutex:SEMAPHORE(1)) .
```

Para exclusión mutua, el valor inicial del semáforo es 1. ¿ Por qué ?

¿ El estado ERROR es alcanzable en SEMADEMO ?

¿ Es suficiente con un semáforo binario (i.e. Max=1) ?



# Ejemplo Semáforo

## Modelo

Tres procesos  $p[1..3]$  usan un semáforo compartido **mutex** para asegurar el acceso exclusivo a algún recurso (acción crítica).

```
LOOP = (mutex.down -> critical -> mutex.up -> LOOP) .  
|| SEMADEMO = (p[1..3]:LOOP  
|| {p[1..3]}::mutex:SEMAPHORE(1)) .
```

Para exclusión mutua, el valor inicial del semáforo es 1. ¿ Por qué ?

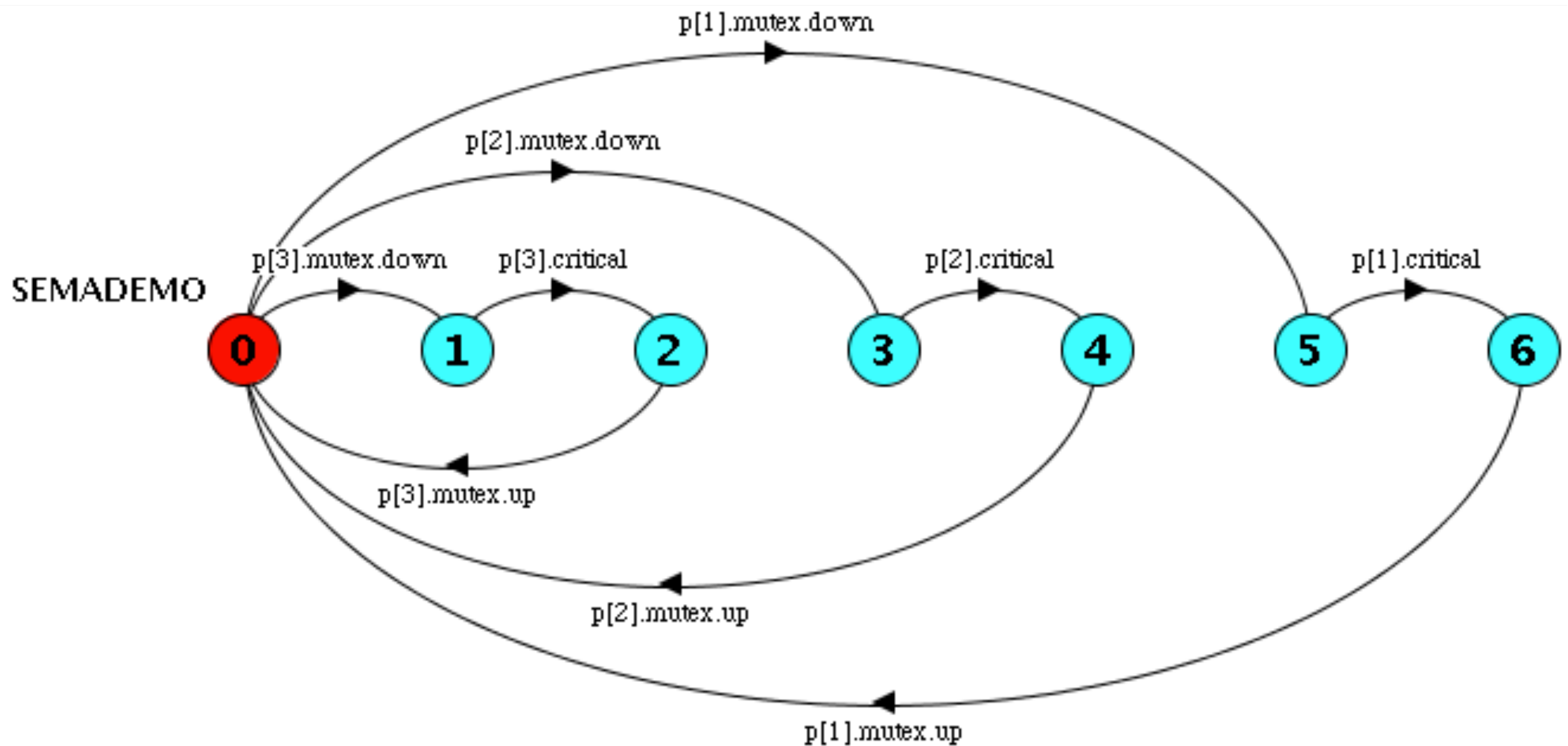
¿ El estado ERROR es alcanzable en SEMADEMO ?

¿ Es suficiente con un semáforo binario (i.e. Max=1) ?

¿ LTS ?

# Ejemplo Semáforo

## Modelo



# Semáforos in Java

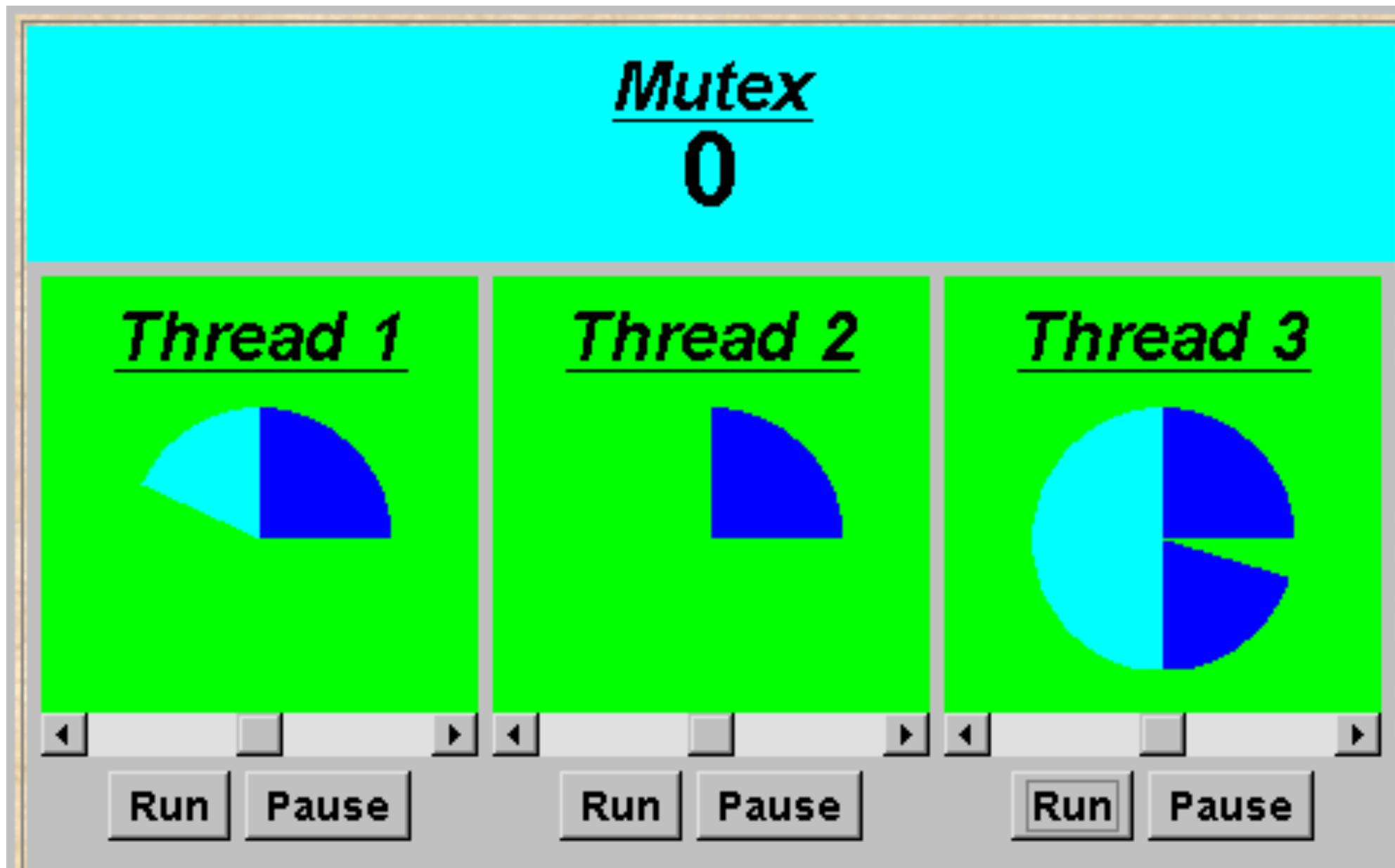
**Semáforos son objetos pasivos, Por lo tanto, implementado como monitores.**

**(En la práctica, los semáforos son un mecanismo de bajo nivel frecuentemente utilizado para implementar monitores de más alto nivel).**

```
public class Semaphore {  
  
    private int value;  
  
    public Semaphore (int initial) {  
        value = initial;  
    }  
  
    synchronized public void up() {  
        ++value;  
        notifyAll();    }  
  
    synchronized public void down() throws InterruptedException  
    {  
        while (value==0) wait();  
        --value;  
    }  
}
```

# SEMADEMO

## visualización



**Valor actual del  
semáforo**

thread 1 está  
ejecutando  
acciones críticas.

thread 2 está  
bloqueado (en  
espera).

thread 3 está  
ejecutando  
acciones no-  
críticas.

# SEMADEMO

¿ Qué sucede si ajustamos el tiempo que utiliza cada thread en la ejecución de su sección crítica ?

- Requerimiento de recursos (grandes) - ¿ más conflictos ?  
(e.g. más del 67% de una rotación)?
- Requerimiento de recursos (pequeños) - ¿ sin conflictos ?  
(e.g. menos que el 33% de una rotación)?

Por lo tanto el **tiempo** de un thread pasa en su **sección crítica** debe ser **lo más corto posible**.