Diseño y Verificación de Programas Concurrentes

Escuela Internacional de Informática - CACIC 2021

Cómo contactarnos fuera de las clases

- mail-> gregis@dc.exa.unrc.edu.ar

- Slack ->

Objetivos de Curso

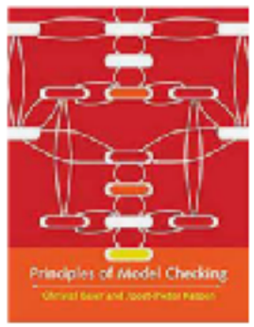
- En esta materia estudiaremos fundamentos de programación concurrente, problemas que afectan a la concurrencia, y diferentes técnicas de programación y diseño de estos sistemas.

- Estudiaremos además formas de modelar y analizar sistemas concurrentes, soportados por herramientas de software.

- El curso será teórico-práctico, y experimentaremos en el uso de programas concurrentes y herramientas para su análisis.

Qué necesitamos para comenzar

- Bibliografía:



- Software:
  - LTSA (Labelled Transition System Analyzer)
  - JAVA 8 o superior
  - Preferentemente algún IDE para JAVA, Eclipse, IntelliJ (CE), NetBeans …

El problema de la corrección del software

Poder garantizar la corrección del software que construimos es una tarea crucial, en particular en algunos dominios:

- Software para equipamiento médico
- Software para el control de vehículos
- Software para el control de procesos
- Algunas aplicaciones financieras
- …

A estos sistemas, cuyas fallas pueden ocasionar daños de importancia (pérdida de vidas humanas, grandes pérdidas financieras, catástrofes nucleares, etc) se denominan sistemas críticos. Muchos de estos sistemas son en efecto sistemas reactivos, concurrentes, distribuidos y/o paralelos

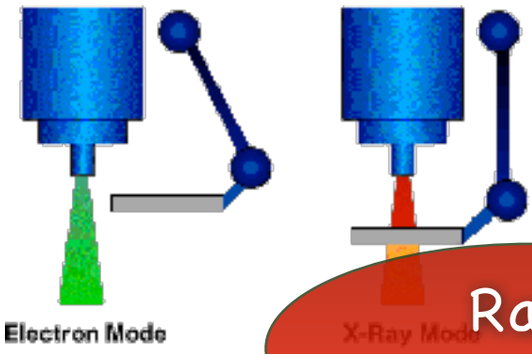
Algunos Ejemplos

Pentium:
FDIV

$$\frac{4195835 * 3145727}{3145727} = 4195579$$

Ariane 5:
64 bits fp
vs 16 bits int

Integración



Therac-25:
one man job

Race Condition

Integración

Mars Climate Orbiter:
Métrico vs Imperial

Voto electrónico:
Integridad/Confidencialidad

Algunos Ejemplos

Pentium:
FDIV

Ariane 5:
64 bits fp
vs 16 bits int

Integración

Muchos más bugs en:

- **The risk digest: http://catless.ncl.ac.uk/Risks/**
- **Otros:**
  - **http://www.cs.tau.ac.il/~nachumd/verify/horror.html**
  - **http://www5.in.tum.de/~huckle/bugse.html**
  - **http://en.wikipedia.org/wiki/List_of_notable_software_bugs**

Integración

Mars Climate Orbiter:
Métrico vs Imperial

Voto electrónico:
Integridad/Confidencialidad

Limitaciones del testing y la simulación

Testing y simulación son dos de las técnicas más efectivas y ampliamente usadas para garantizar corrección de software

- Proveen una serie de entradas o escenarios al software, y estudian el comportamiento del mismo en esos casos.

- Involucran experimentos previos al lanzamiento o uso masivo del software.

- No permiten garantizar la ausencia de errores: “El testing puede confirmar la presencia de errores pero nunca garantizar su ausencia”.

- Particularmente inefectivos para sistemas concurrentes.

Verificación (semi)automática de software

Verificación (semi-)automática de software ofrece fuertes garantías de corrección

- pero demanda conocimientos/habilidades complejas de los desarrolladores.

- tiene limitaciones (e.g., decidir si un programa dado termina o no no es computable).

- bajo algunas restricciones (tanto en las propiedades como en los sistemas), algunas tareas pueden realizarse automáticamente (estudiaremos en detalle algunas, que se aplican a sistemas concurrentes).

Programación Concurrente

¿Qué?

Programación de sistemas compuestos de varios procesos/programas que se ejecutan concurrentemente (o en paralelo, o en forma distribuida) e interactúan entre sí.

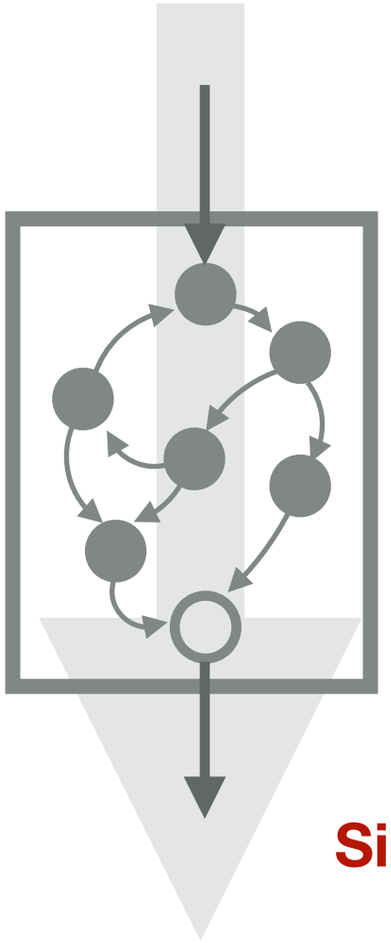
¿Por qué?

- Interacción con el entorno con el fin de observar/controlar dispositivos físicos (ej: dispositivos periféricos de una computadora)
- Mejorar el tiempo de respuesta en la interacción con el usuario (ej: múltiples procesos en un desktop)
- Mejorar velocidad de cálculo
- Comunicación

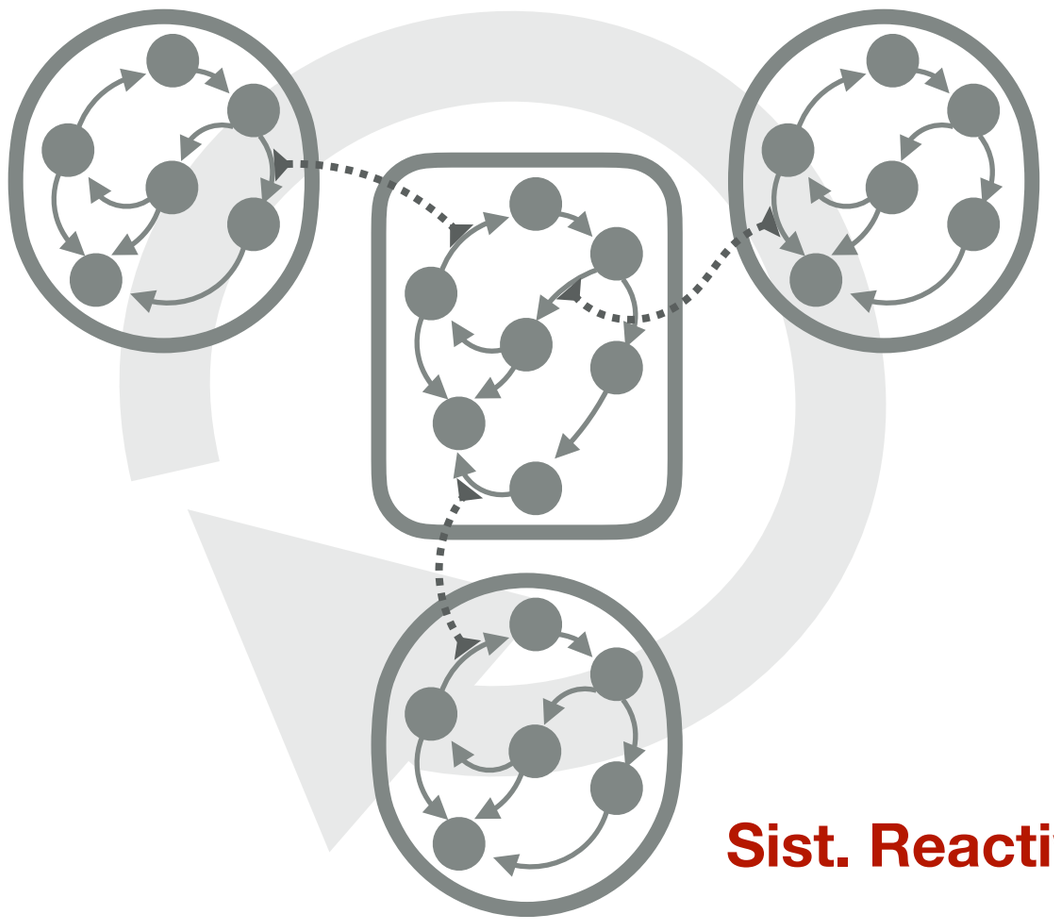
...

Características reactivas de los programas concurrentes

Muchos programas concurrentes suelen ser reactivos, es decir, su funcionalidad involucra la interacción permanente con el ambiente (y otros procesos).



Sist. funcional



Sist. Reactivo

Los sistemas reactivos tienen características diferentes a las de los programas convencionales. En muchos casos, éstos no computan resultados, y suele no requerirse que terminen.

Interacción de programas concurrentes

Los programas concurrentes están compuestos por procesos (o threads, o componentes) que necesitan interactuar. Existen varios mecanismos de interacción entre procesos. Entre éstos se encuentran la memoria compartida y el pasaje de mensajes.

Además, los programas concurrentes deben, en general, colaborar para llegar a un objetivo común, para lo cual la sincronización entre procesos es crucial.

Algunos problemas comunes de los programas concurrentes

- Violación de propiedades universales (invariantes)
- Starvation (inanición): Uno o más procesos quedan esperando indefinidamente un mensaje o la liberación de un recurso
- Deadlock: dos o más procesos esperan mutuamente el avance del otro
- Problemas de uso no exclusivo de recursos compartidos
- Livelock: Dos o más procesos no pueden avanzar en su ejecución porque continuamente responden a los cambios en el estado de otros procesos

```
int res = 0
```

```
1 proc1 {
2 int new_res;
3    for (i=1;i<3;i++){
4       new_res = res;
5       new_res++;
6       res = new_res;
7    }
8 }
```

```
1 proc2 {
2 int new_res;
3    for (i=1;i<3;i++){
4       new_res = res;
5       new_res++;
6       res = new_res;
7    }
8 }
```

```
int res = 0
```

```
1 proc1 {
2 int new_res;
3    for (i=1;i<3;i++){
4       new_res = res;
5       new_res++;
6       res = new_res;
7    }
8 }
```

```
1 proc2 {
2 int new_res;
3    for (i=1;i<3;i++){
4       new_res = res;
5       new_res++;
6       res = new_res;
7    }
8 }
```

```
int res = 0
```

```
1 proc1 {
2 int new_res;
3    for (i=1;i<3;i++){
4       new_res = res;
5       new_res++;
6       res = new_res;
7    }
8 }
```

```
1 proc2 {
2 int new_res;
3    for (i=1;i<3;i++){
4       new_res = res;
5       new_res++;
6       res = new_res;
7    }
8 }
```

Configuración de estado del sistema

(3,0,*1*,3,0,*1*,0)

Estado `proc1`
`PC, new_res, i`

Estado `proc2`
`PC, new_res, i`

Estado `global`
`res`

```
   int res = 0
1  proc1 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

```
1  proc2 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

```
   int res = 0
1  proc1 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

```
1  proc2 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

(3,0,1,3,0,1,0)

```
   int res = 0
1  proc1 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

```
1  proc2 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

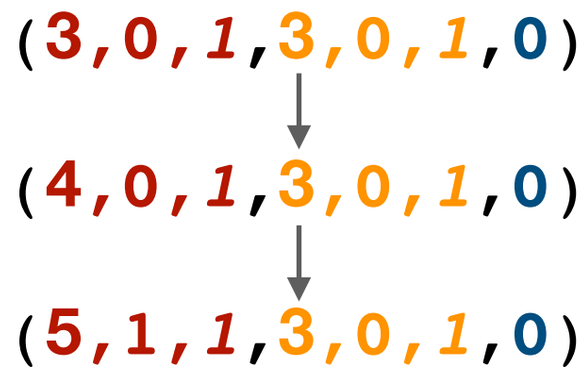
(3,0,1,3,0,1,0)

↓

(4,0,1,3,0,1,0)

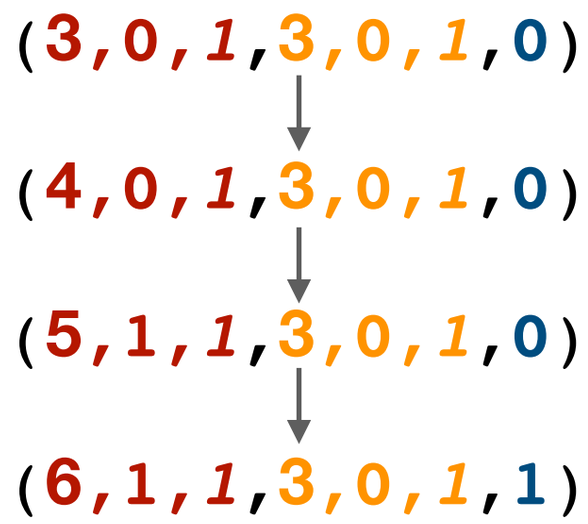
```
   int res = 0
1  proc1 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

```
1  proc2 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```



```
   int res = 0
1  proc1 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

```
1  proc2 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```



```
   int res = 0
1  proc1 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

```
1  proc2 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

(3,0,1,3,0,1,0)

↓

(4,0,1,3,0,1,0)

↓

(5,1,1,3,0,1,0)

↓

(6,1,1,3,0,1,1)

↓

(6,1,1,4,1,1,1)

```
   int res = 0
1  proc1 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

```
1  proc2 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

(3,0,*1*,3,0,*1*,0)

↓

(4,0,*1*,3,0,*1*,0)

↓

(5,1,*1*,3,0,*1*,0)

↓

(6,1,*1*,3,0,*1*,1)

↓

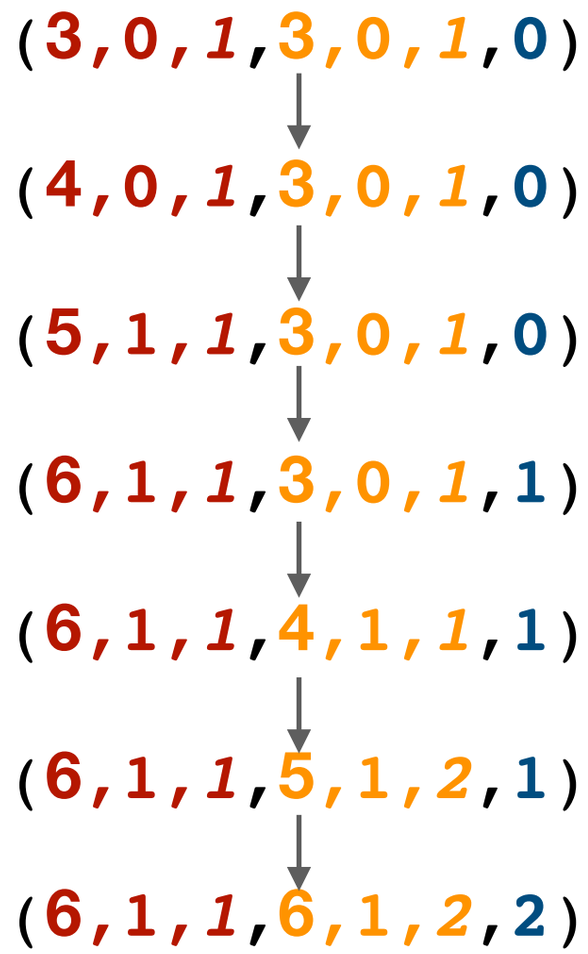
(6,1,*1*,4,1,*1*,1)

↓

(6,1,*1*,5,1,*2*,1)

```
   int res = 0
1  proc1 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

```
1  proc2 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```



```
   int res = 0
1  proc1 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

```
1  proc2 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

(3,0,1,3,0,1,0)

↓

(4,0,1,3,0,1,0)

↓

(5,1,1,3,0,1,0)

↓

(6,1,1,3,0,1,1)

↓

(6,1,1,4,1,1,1)

↓

(6,1,1,5,1,2,1)

↓

(6,1,1,6,1,2,2)

⋮

(7,3,2,7,3,2,4)

```
   int res = 0
1  proc1 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

```
1  proc2 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

```
   int res = 0
1  proc1 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

```
1  proc2 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

(3,0,1,3,0,1,0)

```
   int res = 0
1  proc1 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

```
1  proc2 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

(3,0,1,3,0,1,0)

↓

(4,0,1,3,0,1,0)

```
   int res = 0
1  proc1 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

```
1  proc2 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

(3,0,1,3,0,1,0)

↓

(4,0,1,3,0,1,0)

↓

(4,0,1,4,0,1,0)

```
   int res = 0
1  proc1 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

```
1  proc2 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

(3,0,1,3,0,1,0)

↓

(4,0,1,3,0,1,0)

↓

(4,0,1,4,0,1,0)

↓

(5,1,1,4,0,1,0)

```
   int res = 0
1  proc1 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

```
1  proc2 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

(3,0,1,3,0,1,0)

↓

(4,0,1,3,0,1,0)

↓

(4,0,1,4,0,1,0)

↓

(5,1,1,4,0,1,0)

↓

(5,1,1,5,1,1,0)

```
   int res = 0
1  proc1 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

```
1  proc2 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

(3,0,1,3,0,1,0)

↓

(4,0,1,3,0,1,0)

↓

(4,0,1,4,0,1,0)

↓

(5,1,1,4,0,1,0)

↓

(5,1,1,5,1,1,0)

↓

(6,1,1,5,1,1,1)

```
   int res = 0
1  proc1 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

```
1  proc2 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

(3,0,1,3,0,1,0)

↓

(4,0,1,3,0,1,0)

↓

(4,0,1,4,0,1,0)

↓

(5,1,1,4,0,1,0)

↓

(5,1,1,5,1,1,0)

↓

(6,1,1,5,1,1,1)

↓

(6,1,1,6,1,1,1)

```
   int res = 0
1  proc1 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

```
1  proc2 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

(3,0,1,3,0,1,0)

↓

(4,0,1,3,0,1,0)

↓

(4,0,1,4,0,1,0)

↓

(5,1,1,4,0,1,0)

↓

(5,1,1,5,1,1,0)

↓

(6,1,1,5,1,1,1)

↓

(6,1,1,6,1,1,1)

⋮

(7,3,2,7,3,2,2)

Semántica de programas concurrentes

Una semántica típica para programas concurrentes está basada en **sistemas de transición de estados**. Un sistema de transición de estados es un grafo dirigido en el cual:

(S, s_0, ->)

- **Conjunto de Estados**
- **Estado inicial**
- **Relación de Transición** *(pueden estar etiquetadas)*

Ejemplo

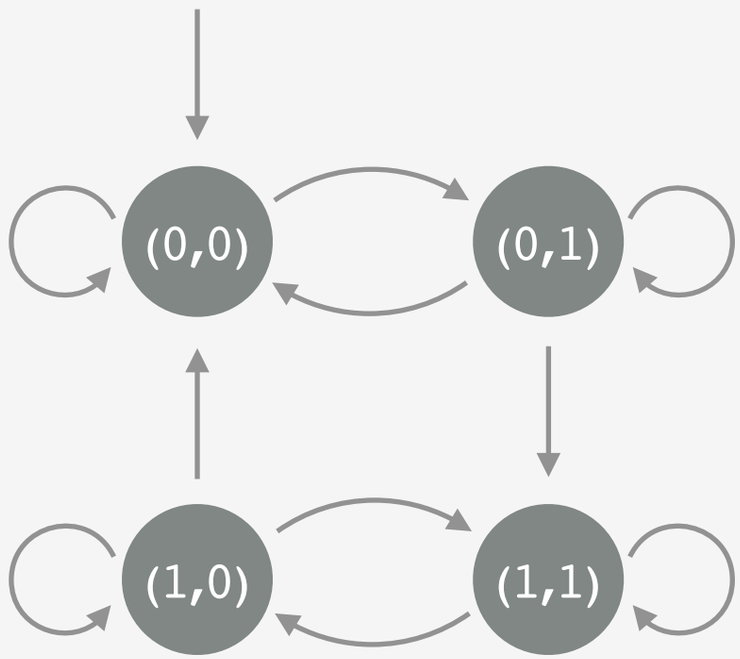
S={0,1}X{0,1}

s_0=(0,0)

(*x*,*y*) -> (*x*,0)

(*x*,*y*) -> (*x*,1)

(*x*,*y*) -> (*y*,*y*)



Ejecuciones de un Programa

Una ejecución es una secuencia de estados $s_0, s_1, s_2, \ldots$, tal que:

s_0 es el estado inicial,

puede llegarse desde s_i a s_{i+1} por alguna sentencia atómica (i.e. transición) del sistema.

El conjunto de todas las ejecuciones determina el comportamiento de un programa concurrente modelado con un sistema de transición de estados.

¿ Cómo se ejecutan los procesos concurrentes ?

De acuerdo al modelo computacional descrito, los procesos concurrentes se ejecutan intercalando las acciones atómicas que los componen. Llamamos a esto, ***interleaving***.

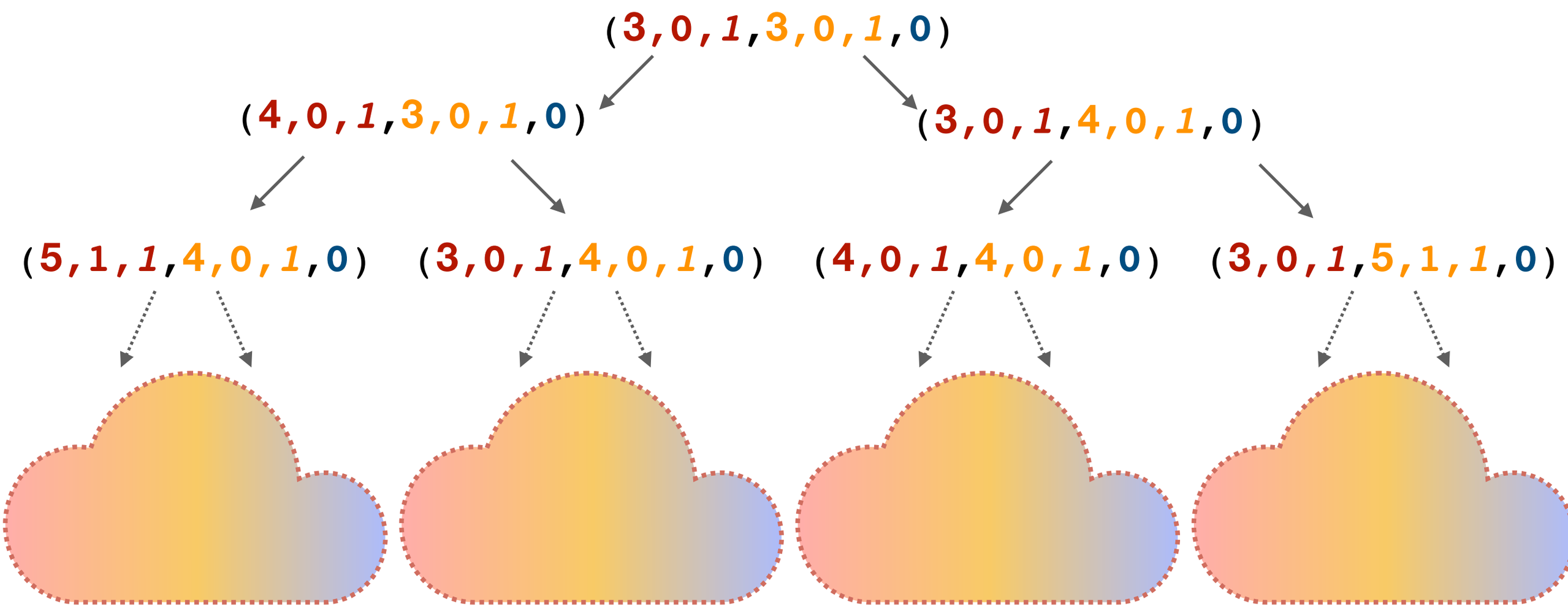
El orden en que se ejecutan las acciones atómicas no puede decidirse en general, y un mismo par de procesos puede tener diferentes ejecuciones debido al no determinismo en la elección de las acciones atómicas a ejecutar.

```
   int res = 0
1  proc1 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

```
1  proc2 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

```
   int res = 0
1  proc1 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```

```
1  proc2 {
2  int new_res;
3     for (i=1;i<3;i++){
4        new_res = res;
5        new_res++;
6        res = new_res;
7     }
8  }
```



¿ Cómo podemos razonar sobre programas concurrentes ?

En general, es muy difícil razonar sobre programas concurrentes. Luego, garantizar que un programa concurrente es correcto es también muy difícil.

Una de las razones tiene que ver con que diferentes *interleavings* de acciones atómicas pueden llevar a diferentes resultados o comportamientos de los sistemas concurrentes.

El número de *interleavings* posibles, por su parte, es en general muy grande (crece exponencialmente), lo que hace que el testing difícilmente pueda brindarnos confianza de que nuestros programas concurrentes funcionan correctamente.

Implementación vs Diseño

Modelos de programas concurrentes (Abstracción)

Una forma de aliviar, en parte, el problema de razonar sobre programas concurrentes es considerar representaciones abstractas de éstos. Estas representaciones abstractas, llamadas modelos, nos permiten concentrarnos en las características particulares que queremos analizar.

Las álgebras de procesos (CSP, CCS, ACP, LOTOS, etc.) permiten construir estos modelos, concentrándose en las propiedades funcionales de sistemas concurrentes. Para esto, es importante considerar los eventos en los cuales cada proceso puede estar involucrado, y los patrones de ocurrencia que éstos siguen.

Implementación vs Diseño

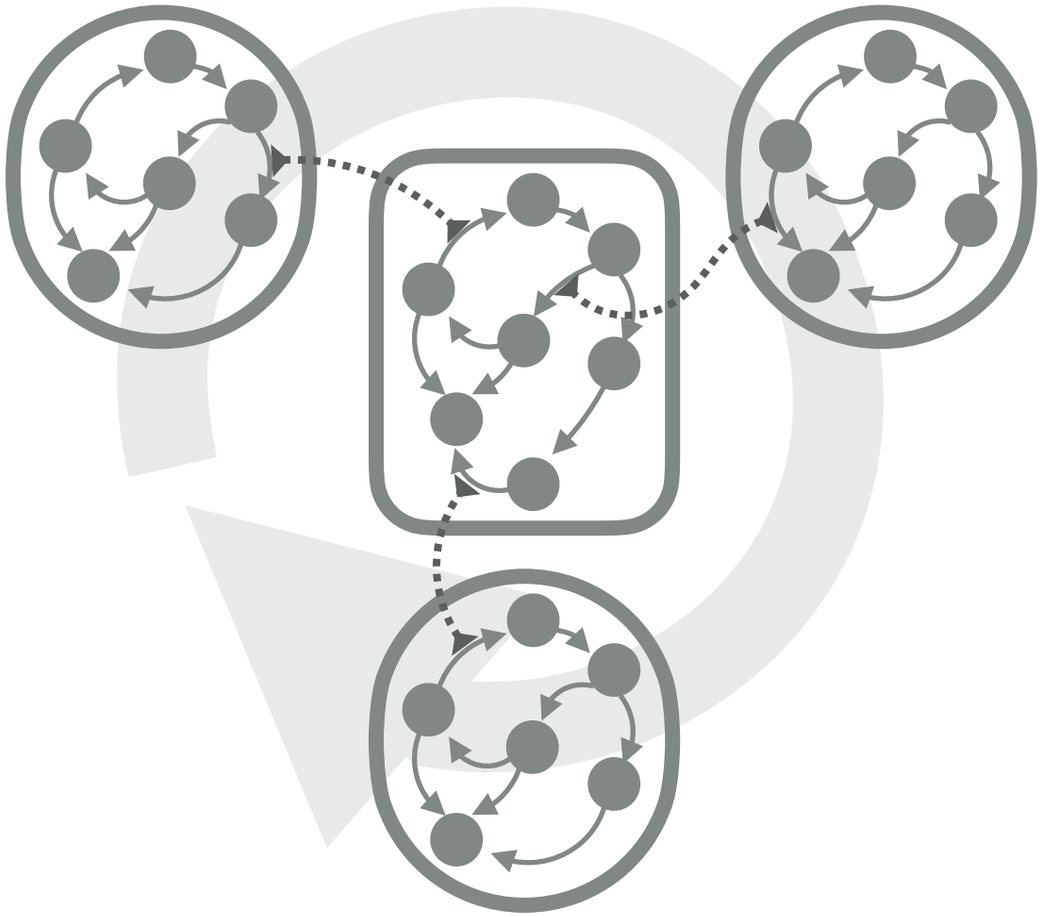
Modelos de programas concurrentes (Abstracción)

Implementación del Sistema

Principal.java

```java
package com.carlos;

/**
 * Esta es la clase principal del programa.
 * @author carlos
 * @version 1.0
 * @since 19/08/2019*/

public class Principal {

    public enum Dias {Lunes, Martes, Miércoles};

    public static void main(String[] args) {

        int num1 = 9;
        float num2 = 9.3f;
        float suma = num1 + num2;

        System.out.println(suma);
    }
}
```

Modelos Abstracto del Sistema



Implementación vs Diseño

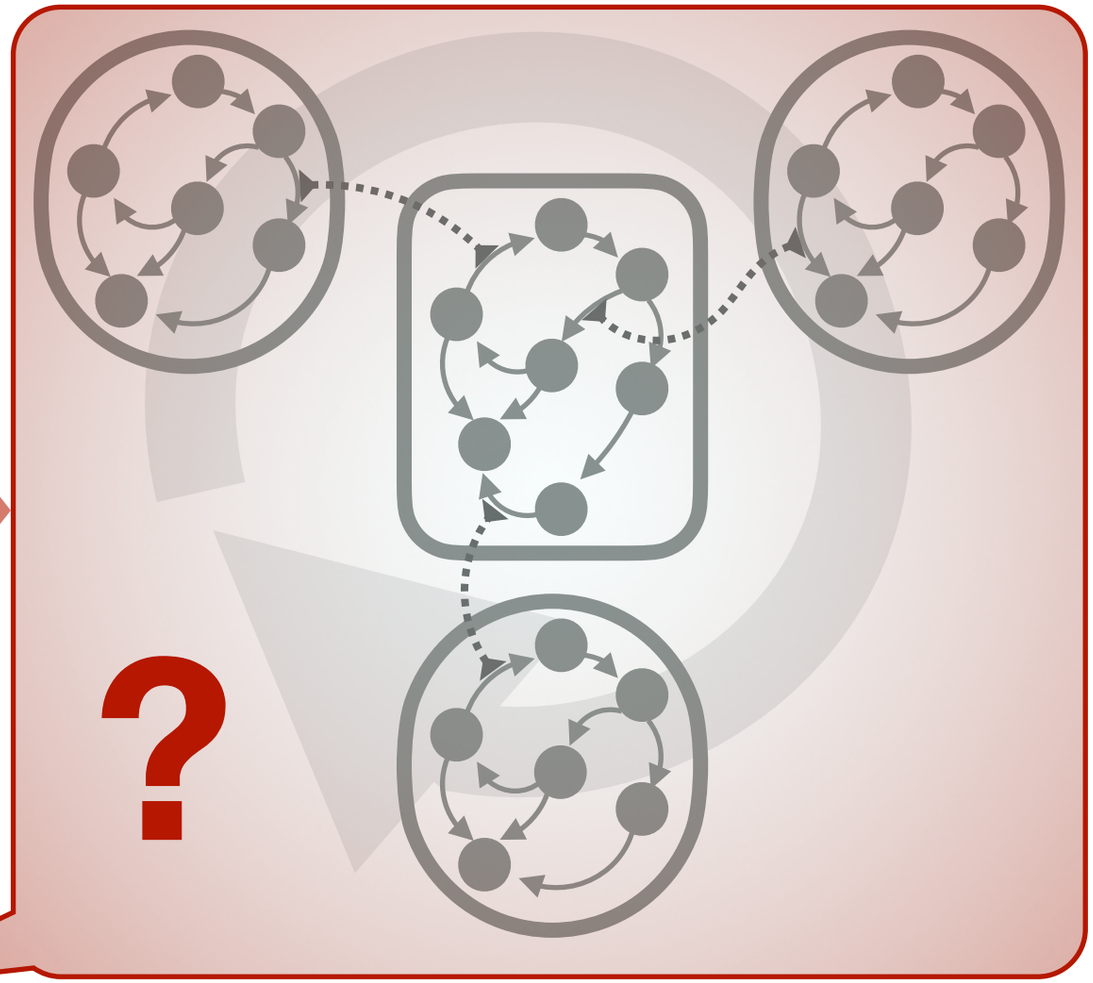
Modelos de programas concurrentes (Abstracción)

Implementación del Sistema

Principal.java

```java
package com.carlos;

/**
 * Esta es la clase principal del programa.
 * @author carlos
 * @version 1.0
 * @since 19/08/2019*/

public class Principal {

    public enum Dias {Lunes, Martes, Miércoles};

    public static void main(String[] args) {

        int num1 = 9;
        float num2 = 9.3f;
        float suma = num1 + num2;

        System.out.println(suma);
    }
}
```

Modelos Abstracto del Sistema

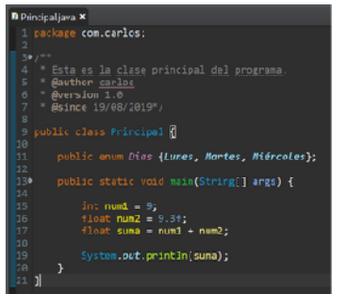


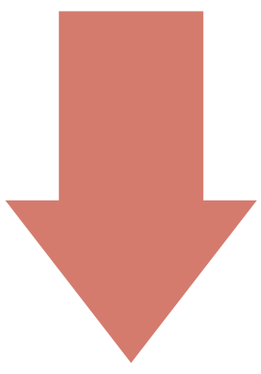
Análisis automático
(Model checking)

Implementación vs Diseño

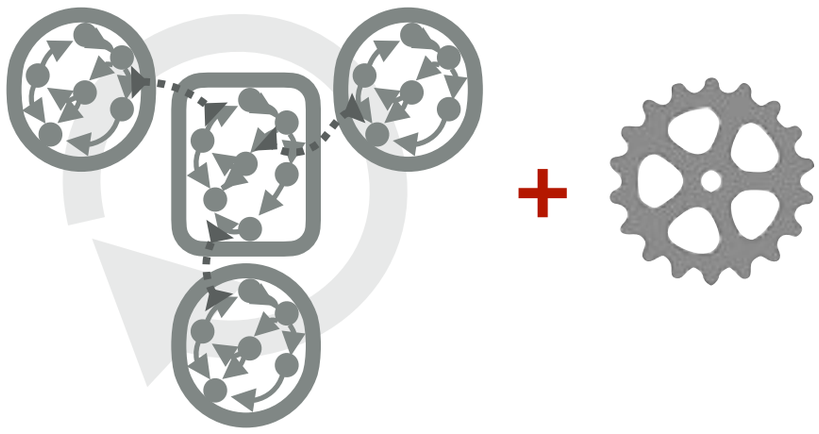
Modelos de programas concurrentes (Abstracción)

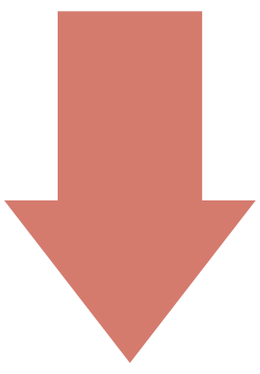
Implementación de Sistemas concurrentes





Modelado y Analisis de Sistemas Concurrentes





LTSA

Labelled Transition System Analyzer

Modelando Sistemas Concurrentes

El lenguaje que utilizaremos para modelado de sistemas concurrentes es FSP (Finite State Processes). FSP es una variante simple de CSP, que incluye, entre otras cosas:

Prefijos de acciones: `x -> P`

Definiciones recursivas: `OFF = on -> off -> OFF`

Elección: `x -> P | y -> Q`

Elección NO determinista: `x -> P | x -> Q`

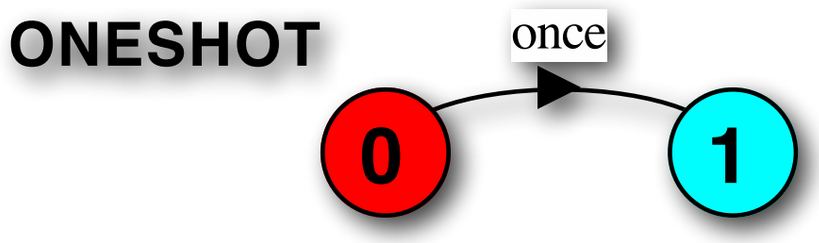
Semántica de los procesos

La semántica de los procesos FSP está dada en términos de sistemas de transición de estados y trazas. En particular, los procesos pueden verse gráficamente como sistemas de transición de estados.

Por ejemplo, el proceso

```
ONESHOT = once -> STOP.
```

puede visualizarse como:



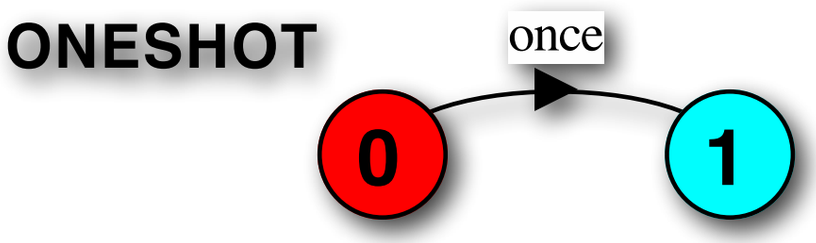
Semántica de los procesos

La semántica de los procesos FSP está dada en términos de sistemas de transición de estados y trazas. En particular, los procesos pueden verse gráficamente como sistemas de transición de estados.

Por ejemplo, el proceso

```
ONESHOT = once -> STOP.
```

puede visualizarse como:



Los gráficos están hechos con LTSA que es la herramienta que soporta

FSP: Prefijos de acciones

Uno puede definir un proceso que, luego de realizar un evento o acción atómica `x`, se comporta exactamente como cierto proceso `P` usando prefijos:

```
(x -> P)
```

Esto es utilizado, por ejemplo, en el proceso `ONESHOT` visto anteriormente.

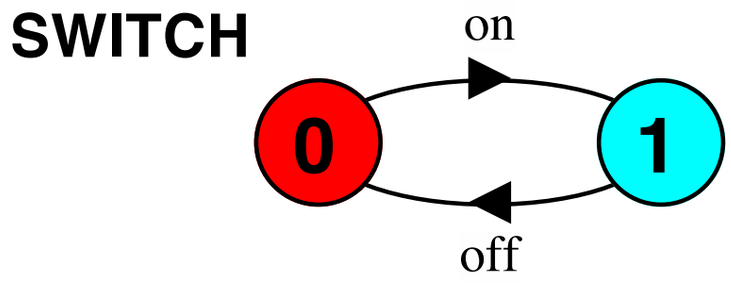
FSP: Recursión

El comportamiento de un proceso también puede definirse usando recursión, donde tenemos algunas restricciones sintácticas:

Ejemplo:

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

Por supuesto, estos procesos pueden verse gráficamente como sistemas de transición de estados (y la herramienta LTSA lo hace por nosotros !).



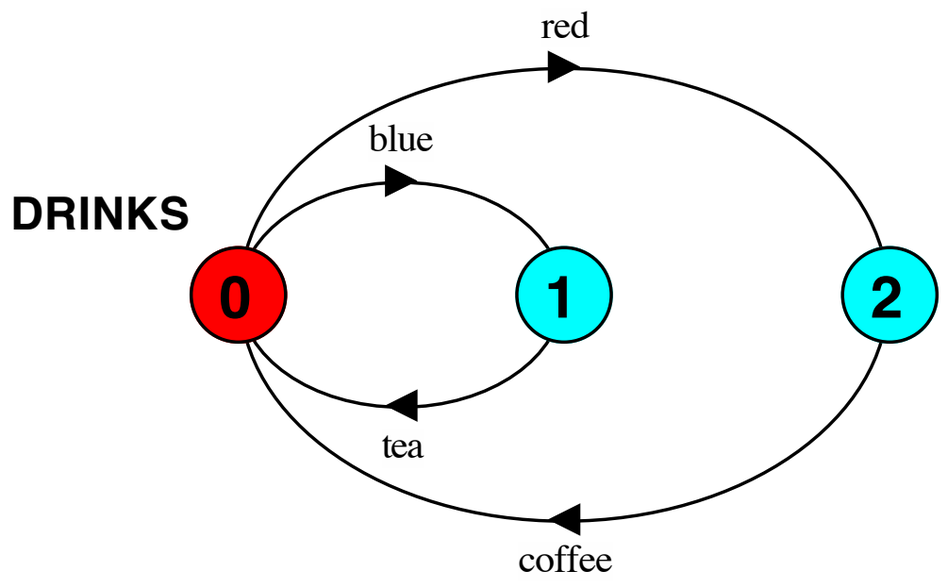
FSP: Elección

La ramificación en el flujo de ejecución de un proceso se describe mediante la elección.

Ejemplo:

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

Y, nuevamente, estos procesos pueden verse gráficamente como sistemas de transición de estados.

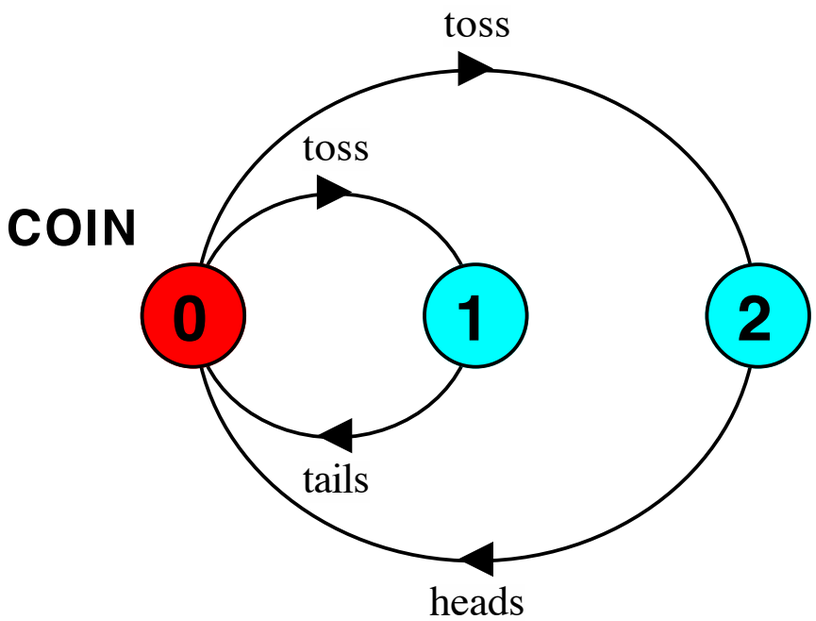


FSP: Elección NO Determinista

Es simplemente un caso particular de elección.

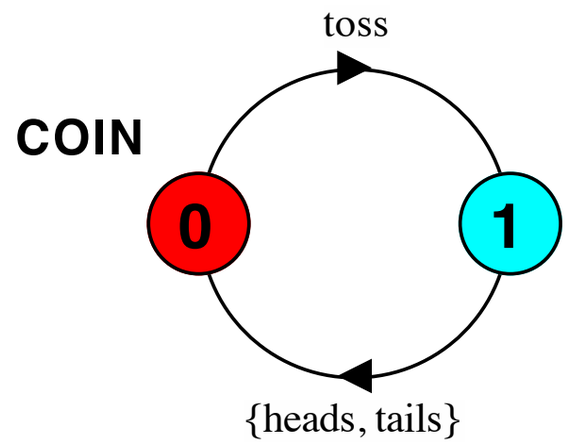
Ejemplo:

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



Comparar:

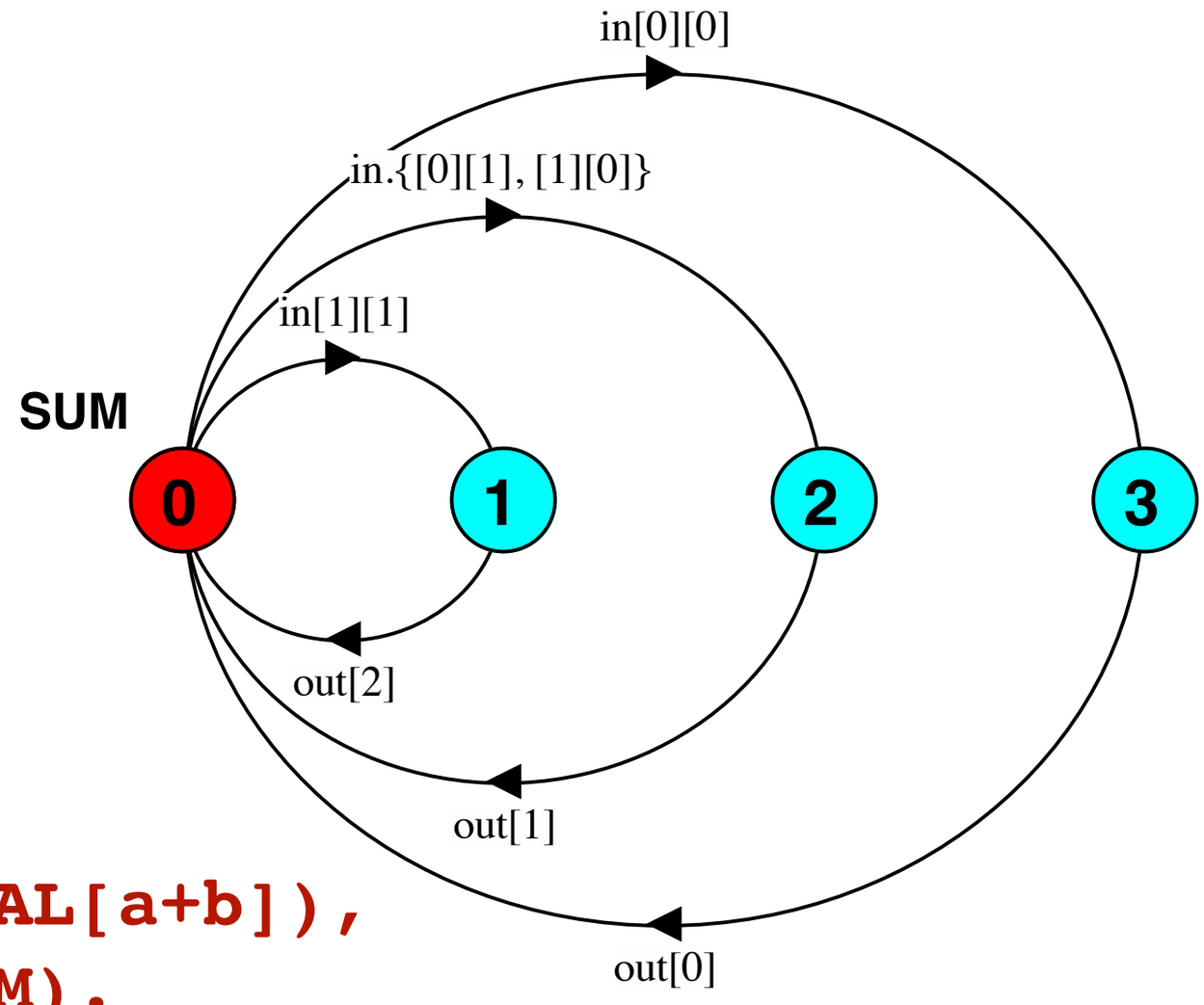
```
COIN = ( toss -> ( heads -> COIN
                 | tails -> COIN
                 )
       ).
```



FSP: Procesos y acciones indexados

Para establecer ciertos valores durante la definición de los modelos, podemos definir constantes y rangos.

Más ejemplos:




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

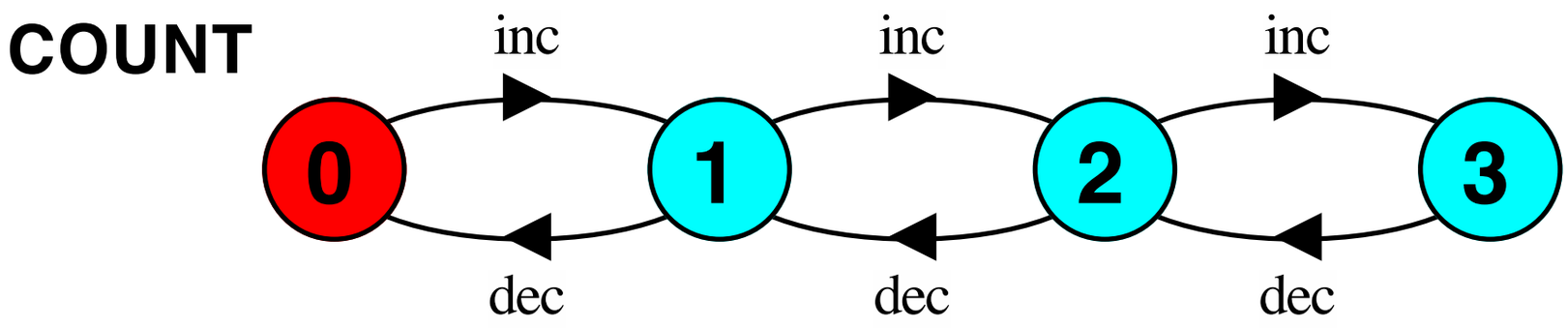
```
SUM = (in[a:T][b:T] -> TOTAL[a+b]),
TOTAL[s:R] = (out[s] -> SUM).
```

FSP: Acciones con guardas

Es en general útil contar con acciones que se ejecuten condicionalmente, con respecto al estado de la máquina o sistema modelados. Esto puede expresarse usando la notación "`when`" en FSP.

Ejemplo:

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



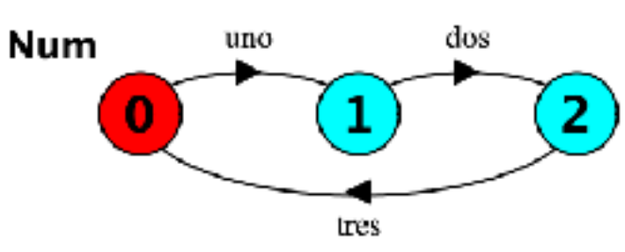
FSP: Composición Paralela

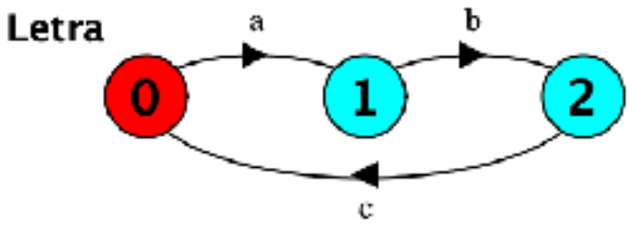
Hasta el momento, ninguna de las construcciones vistas nos permite modelar concurrencia. La construcción que nos permite hacer esto, y las más compleja de comprender, es la composición paralela de procesos.

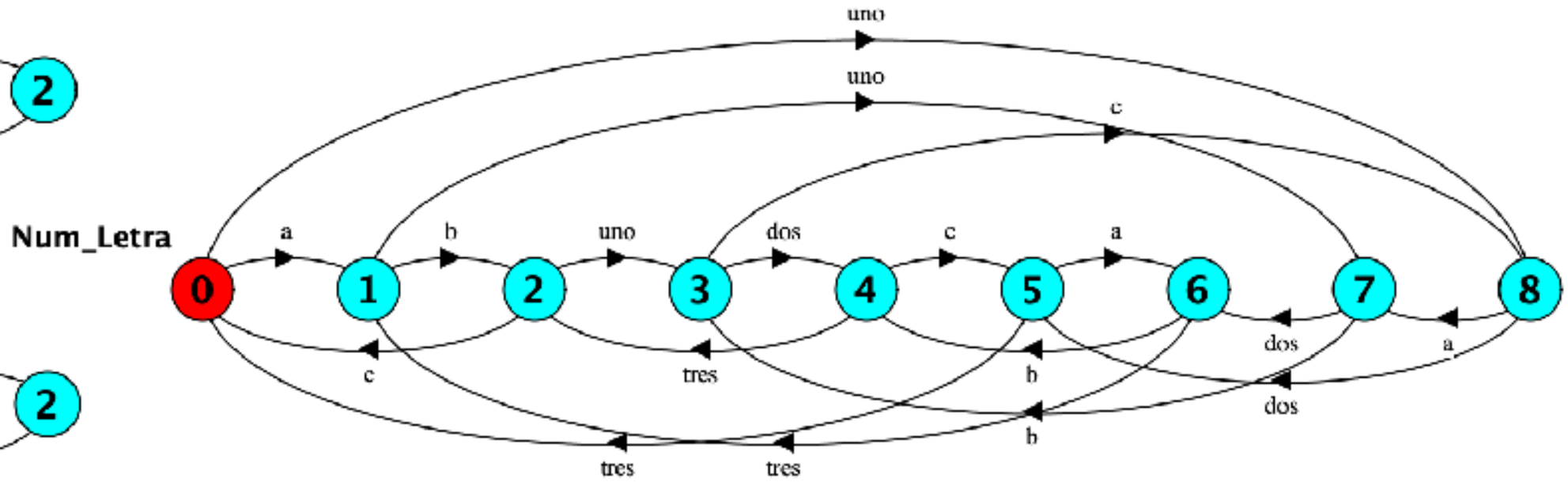
Dados dos procesos **P** y **Q**, **P||Q** denota la composición paralela de estos procesos.

Ejemplo:

```
Num = (uno -> dos -> tres -> Num).
Letra = (a -> b -> c -> Letra).
||Num_Letra = (Num || Letra).
```





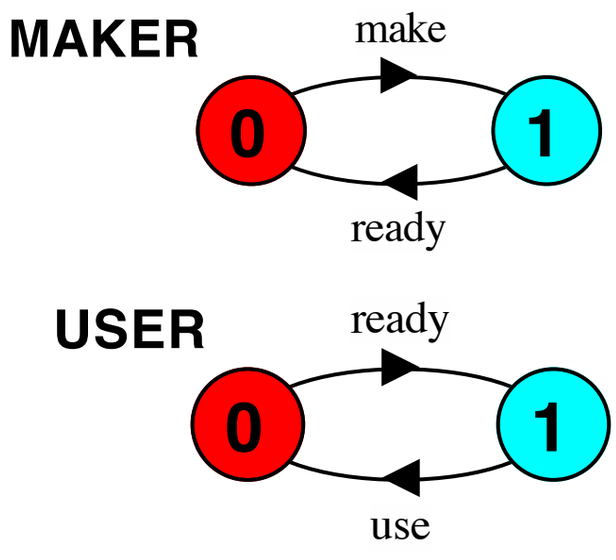


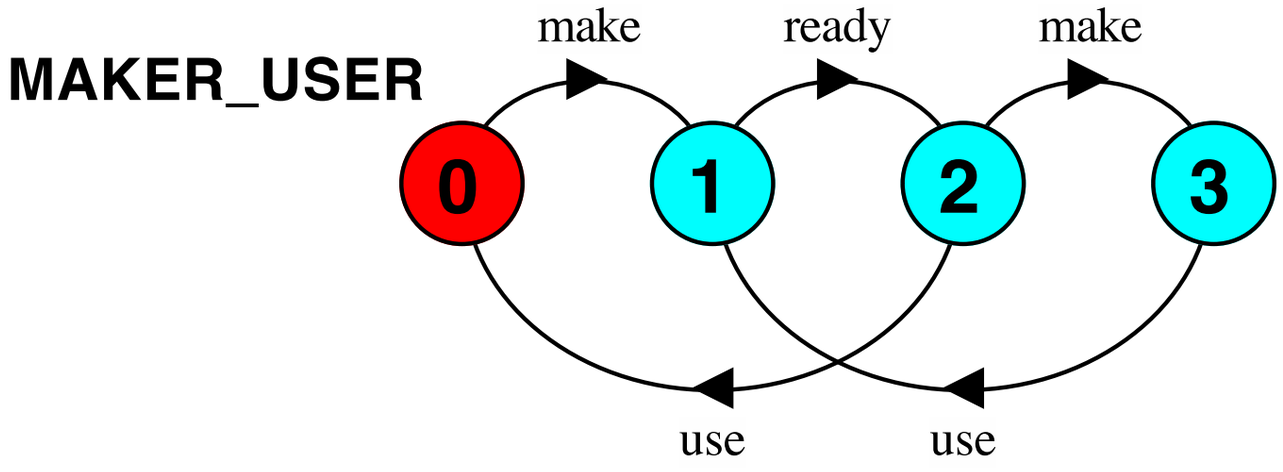
FSP: Composición Paralela (acciones comunes)

Si dos o más procesos comparten (con el mismo nombre) una acción, ésta se debe realizar de manera sincronizada por dichos procesos.

Ejemplo:

```
MAKER = (make -> ready -> MAKER).
USER  = (ready -> use -> USER).
||MAKER_USER = (MAKER || USER).
```



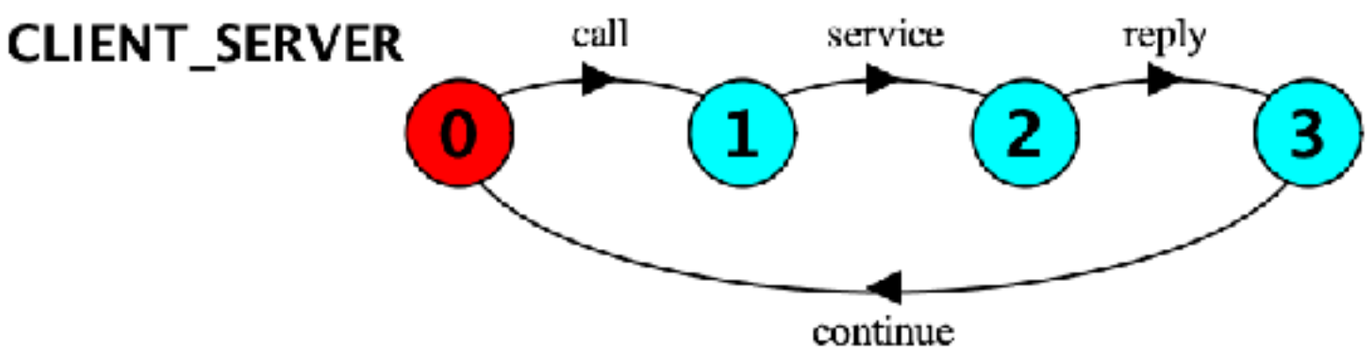


FSP: Reetiquetado

Se utiliza para poder sincronizar procesos en acciones (con diferentes nombres):

```
CLIENT= (call->wait->continue->CLIENT).
SERVER = (request->service->reply->SERVER).
```

```
||CLIENT_SERVER = (CLIENT || SERVER) /{call/request, reply/wait}.
```



FSP: Composición Paralela (Resumen)

Hay varios puntos importantes por recordar con respecto a la composición paralela en FSP:

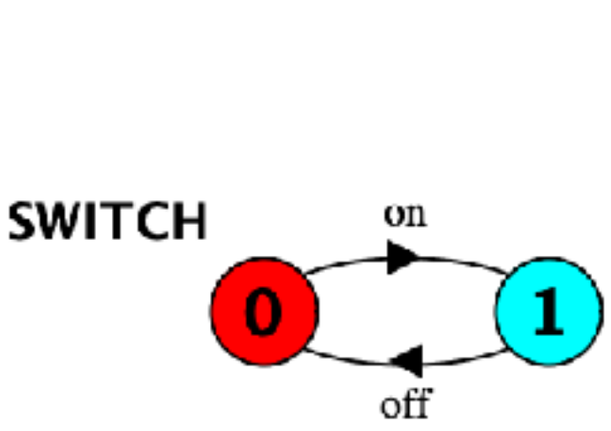
- La sincronización se realiza en las acciones comunes (y puede involucrar a más de dos procesos).
- No se puede identificar explícitamente al proceso “activo” y al proceso “pasivo” en la sincronización de acciones comunes. Esta interpretación corre por cuenta del diseñador (i.e. ustedes).
- El modelo de concurrencia es *interleaving*, donde las acciones atómicas independientes de diferentes procesos pueden ejecutarse en interleavings arbitrarios.

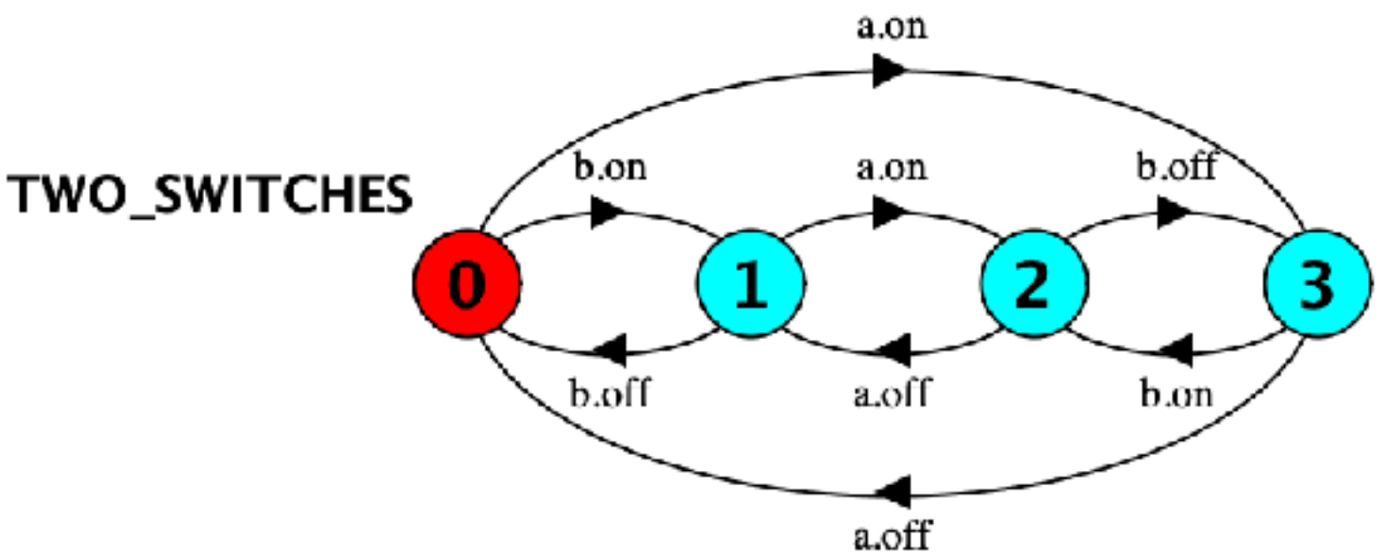
FSP: Etiquetas de procesos

Para poder tener diferentes copias del mismo proceso podemos etiquetarlos:

```
SWITCH = ( on -> off -> SWITCH).
||TWO_SWITCHES = (a:SWITCH || b:SWITCH).
```

Etiqueta (id de la copia)





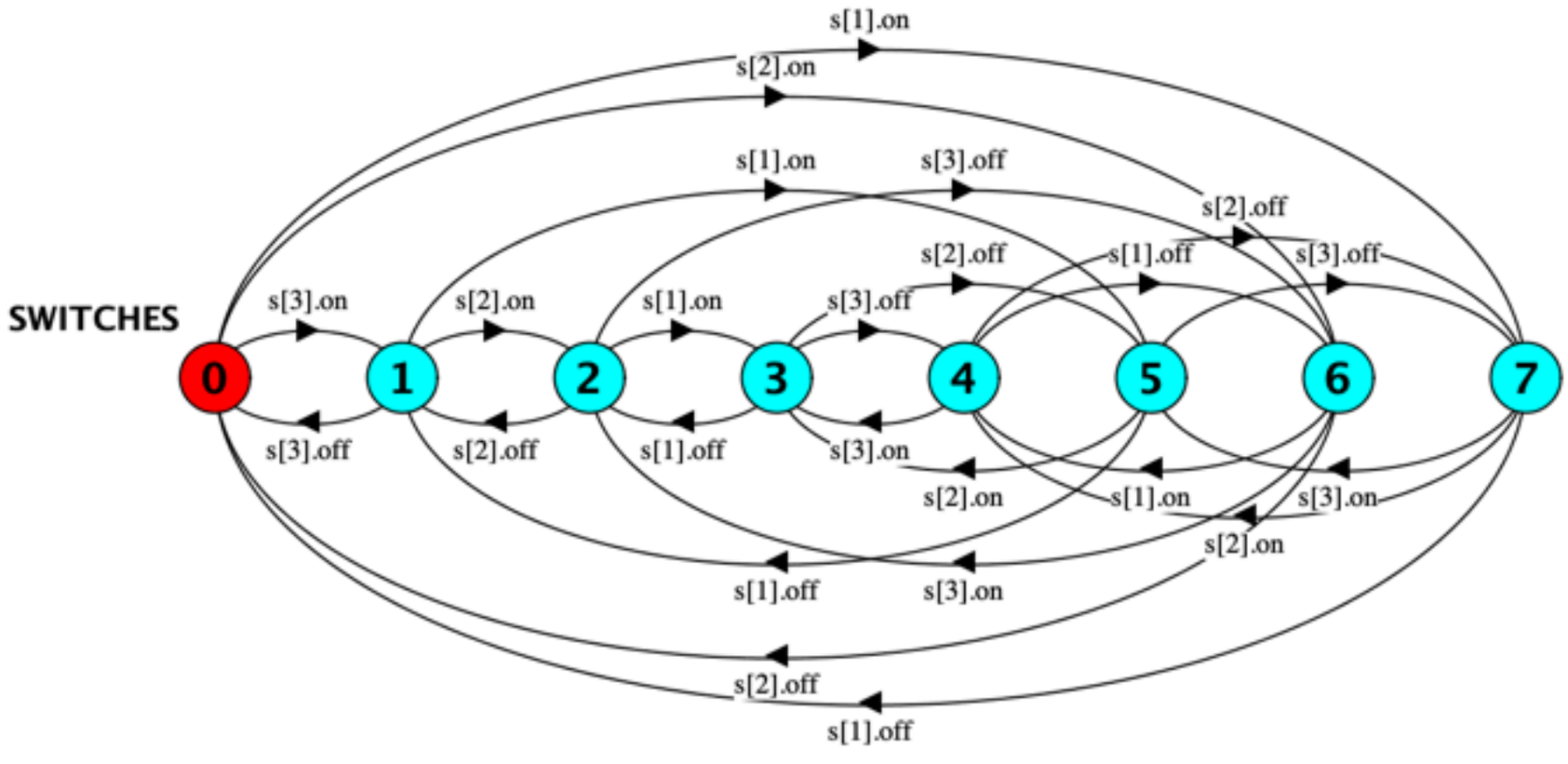
FSP: Etiquetas de procesos (y parámetros)

Existen formas abreviadas de etiquetar procesos, además podemos pasar parámetros a los procesos para utilizar sus valores en su definición:

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

||SWITCHES(N=3) = (s[i:1..N]:SWITCH).
```

Parámetro

N copias: S[1], S[2], S[3]

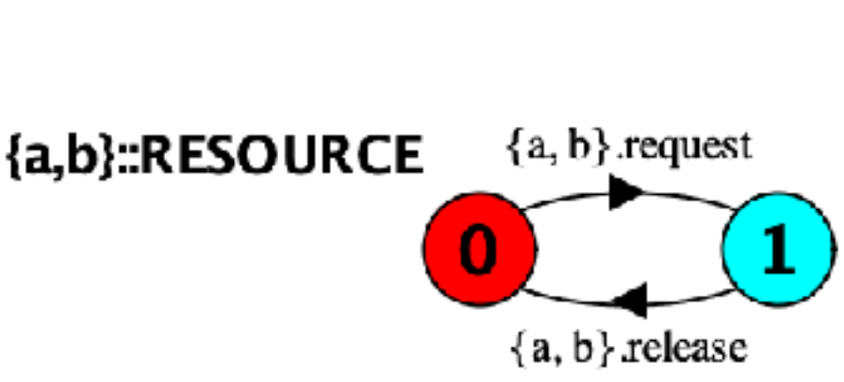


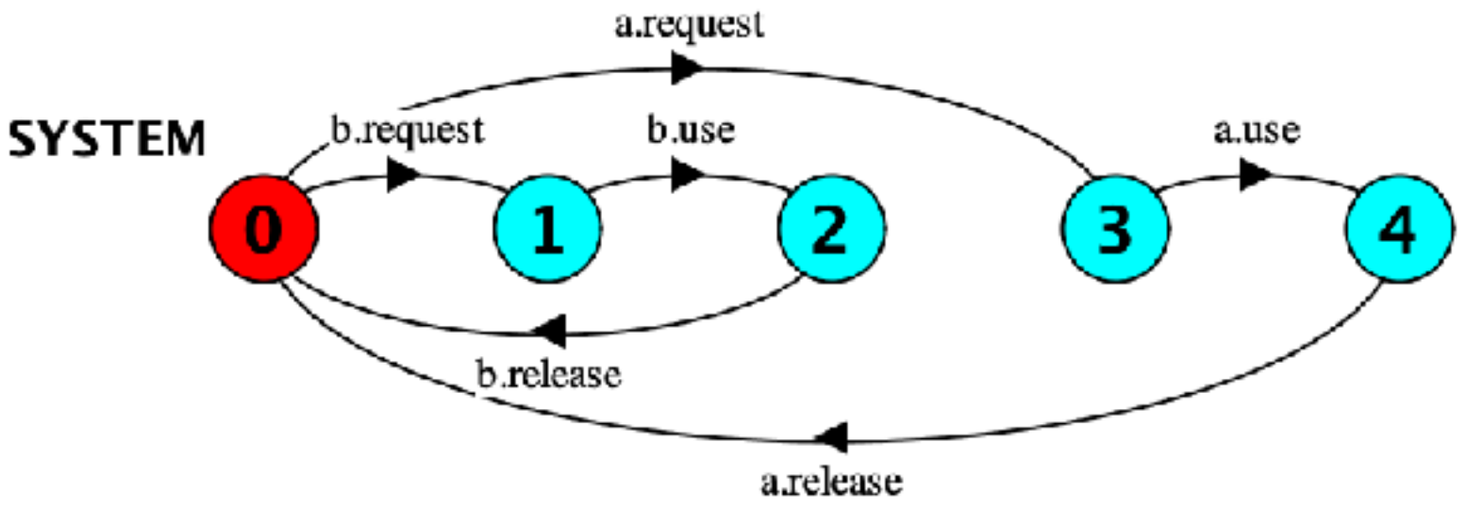
FSP: Etiquetas de procesos (interacción)

Podemos utilizar el etiquetado para prefijar cada acción de un proceso, esto permite interactuar con más de un proceso.

```
USER = ( request -> use -> release -> USER).
RESOURCE = (request -> release -> RESOURCE).

||SYSTEM = (a:USER || b:USER || {a,b}::RESOURCE ).
```

prefijado de cada acción con **a** y con **b**





FSP: Ocultamiento

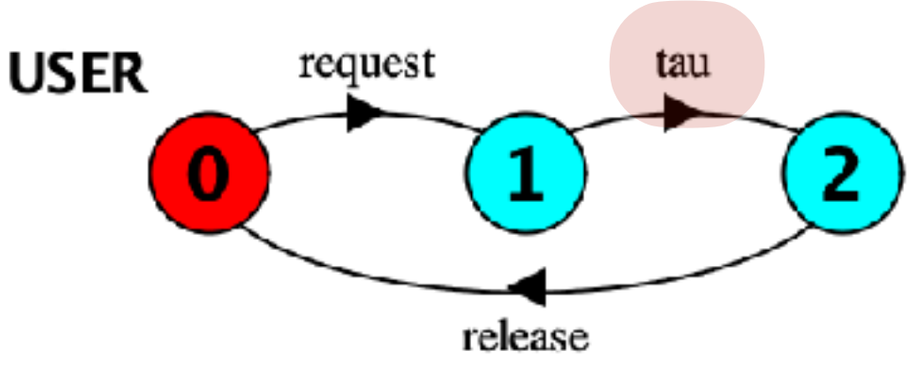
Las acciones ocultadas NO pueden compartirse con otros procesos (desaparecen del alfabeto del proceso). Utiliza una acción distinguida tau (denominada acción sigilosa o invisible) que no tiene permitido sincronizar. Podemos ocultar acciones de dos formas: declarando explícitamente cuáles ocultamos (****) o declarando cuáles publicamos (**@**), aquellas que no están declaradas se ocultan)

```
USER = ( request -> use ->
         release -> USER)/{use}.
```

ocultar **use**

```
USER = ( request -> use ->
         release -> USER)@{request,release}.
```

publicar **request** y **release**



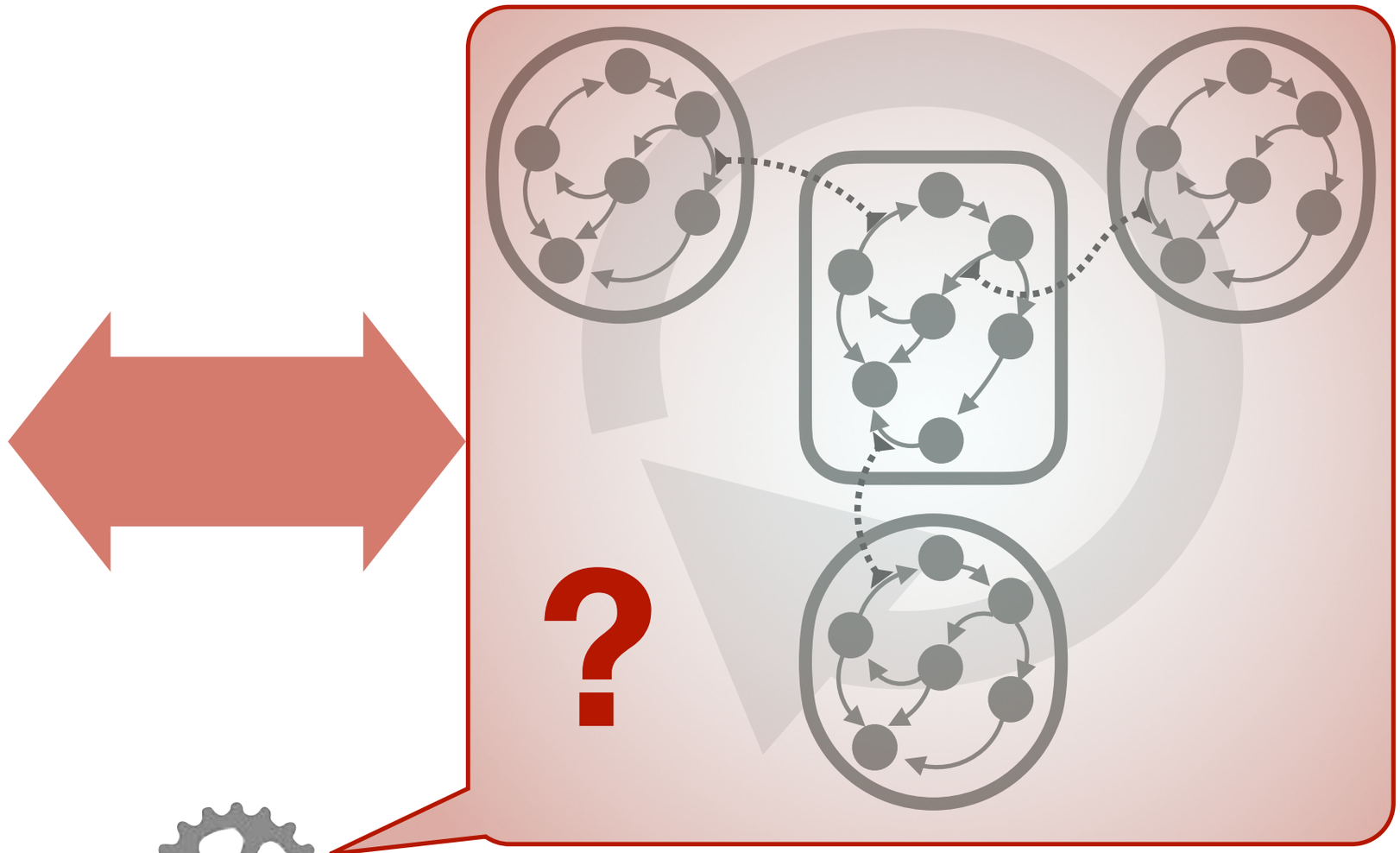
Implementación vs Diseño

Modelos de programas concurrentes (Abstracción)

Implementación del Sistema

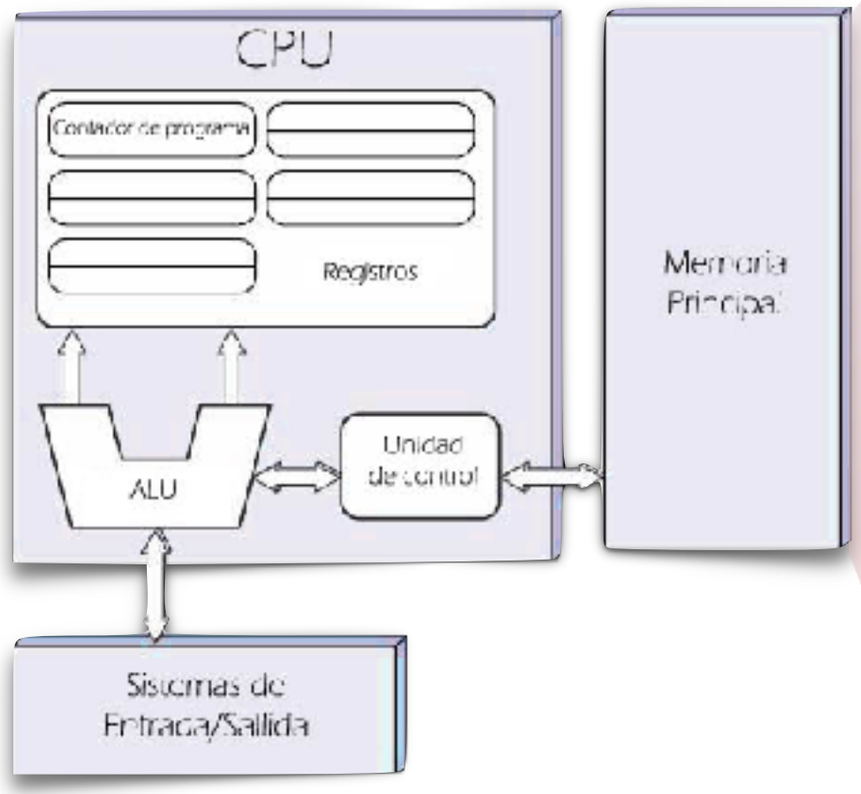
Modelos Abstracto del Sistema

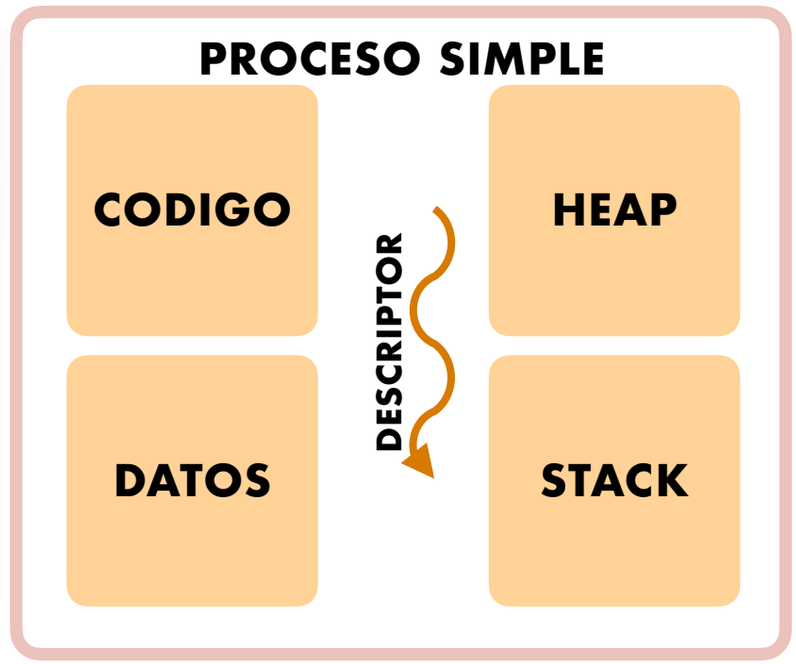
```
Principal.java
1  package com.carlos;
2
3  /**
4   * Esta es la clase principal del programa.
5   * @author carlos
6   * @version 1.0
7   * @since 19/08/2019*/
8
9  public class Principal {
10
11     public enum Dias {Lunes, Martes, Miércoles};
12
13     public static void main(String[] args) {
14
15         int num1 = 9;
16         float num2 = 9.3f;
17         float suma = num1 + num2;
18
19         System.out.println(suma);
20     }
21 }
```

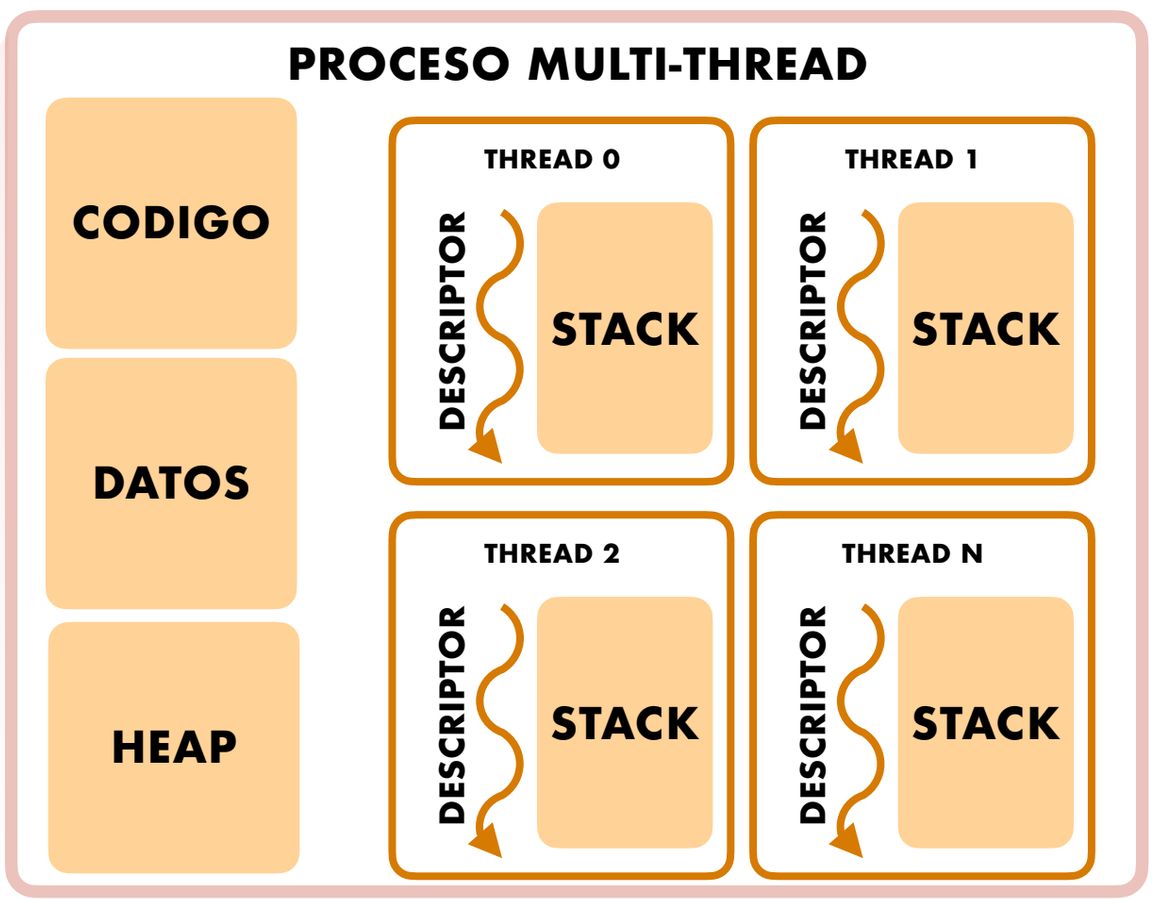


Análisis automático
(Model checking)

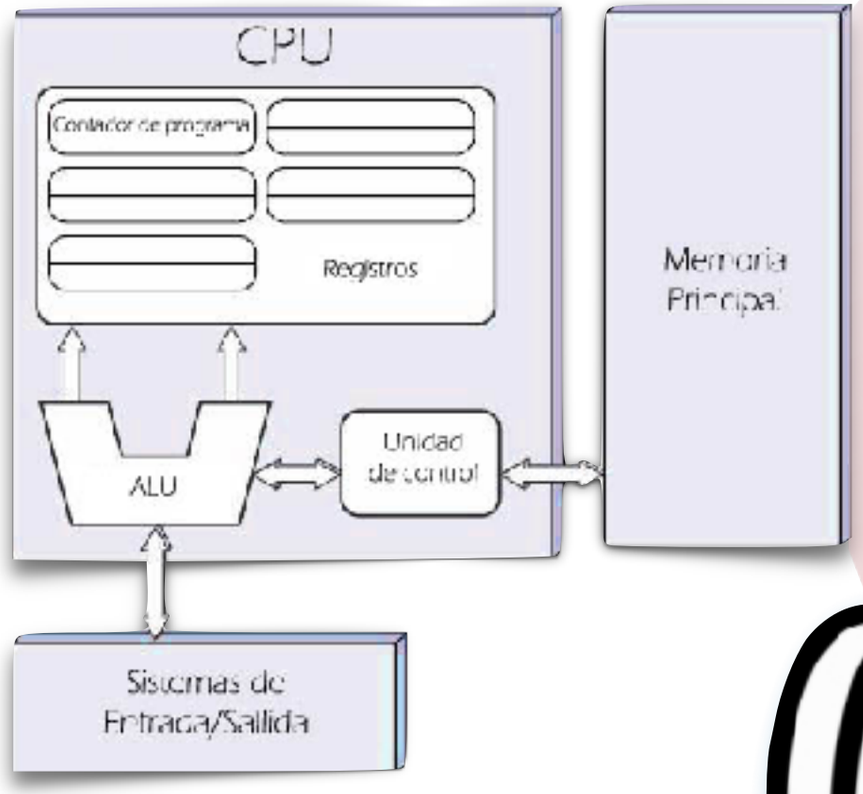
Concurrencia - Procesos y Threads

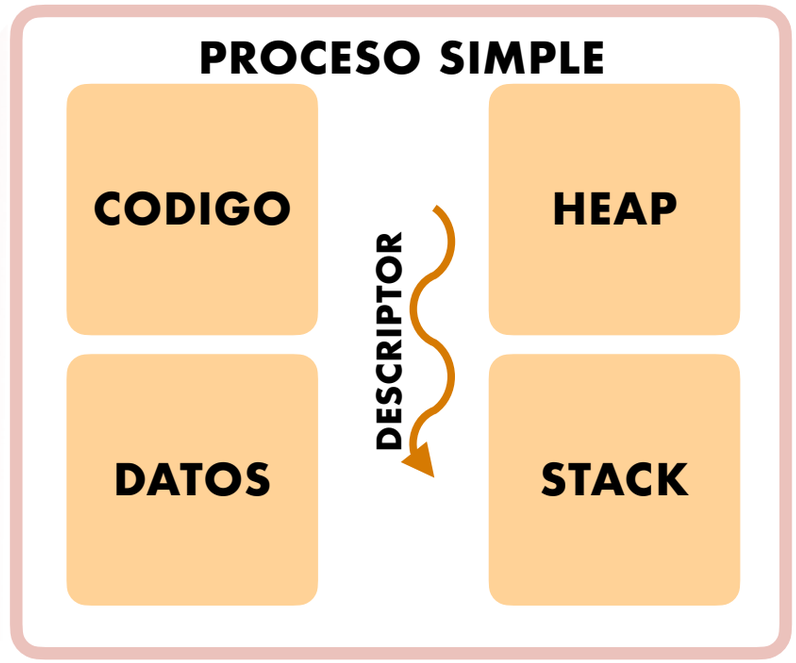


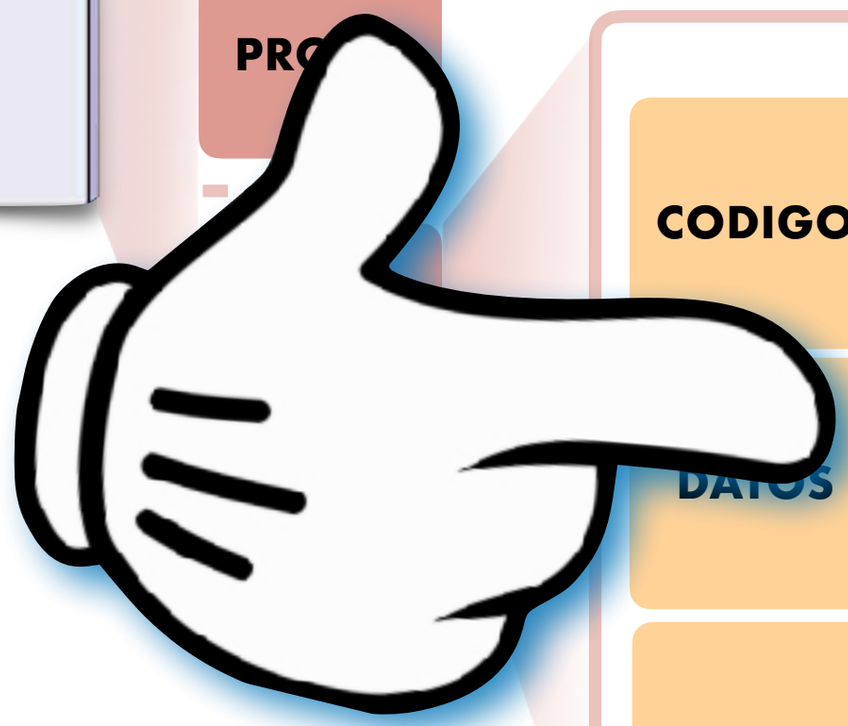


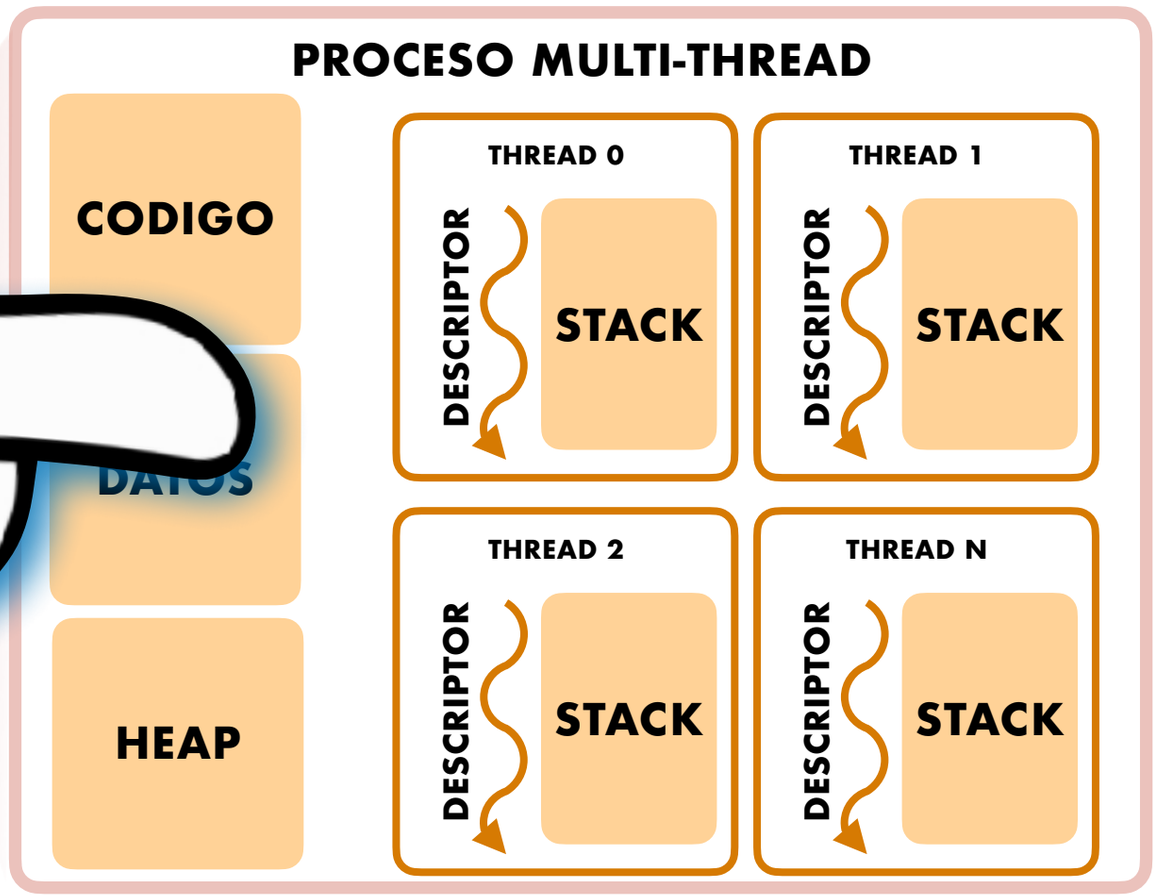


Concurrencia - Procesos y Threads



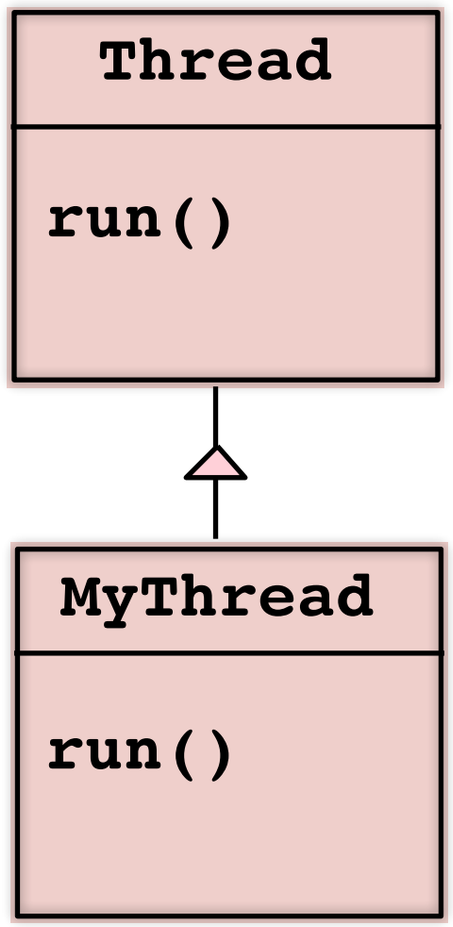






Concurrencia en Java con threads

`Thread` ejecuta instrucciones de su método `run()`. El código a ejecutar dependerá de la implementación específica provista para `run()` (e.g., en una clase derivada).

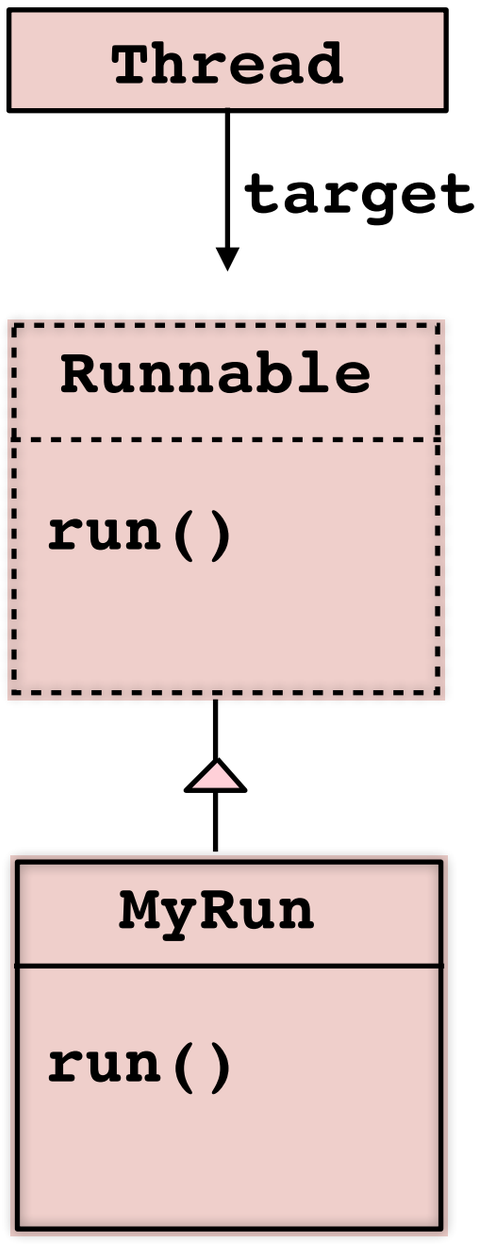



```
class MyThread extends Thread {
    public void run() {
        //......
    }
}
```

Cada instancia de `Thread` lleva adelante un único hilo secuencial de control. Se pueden crear y eliminar threads dinámicamente

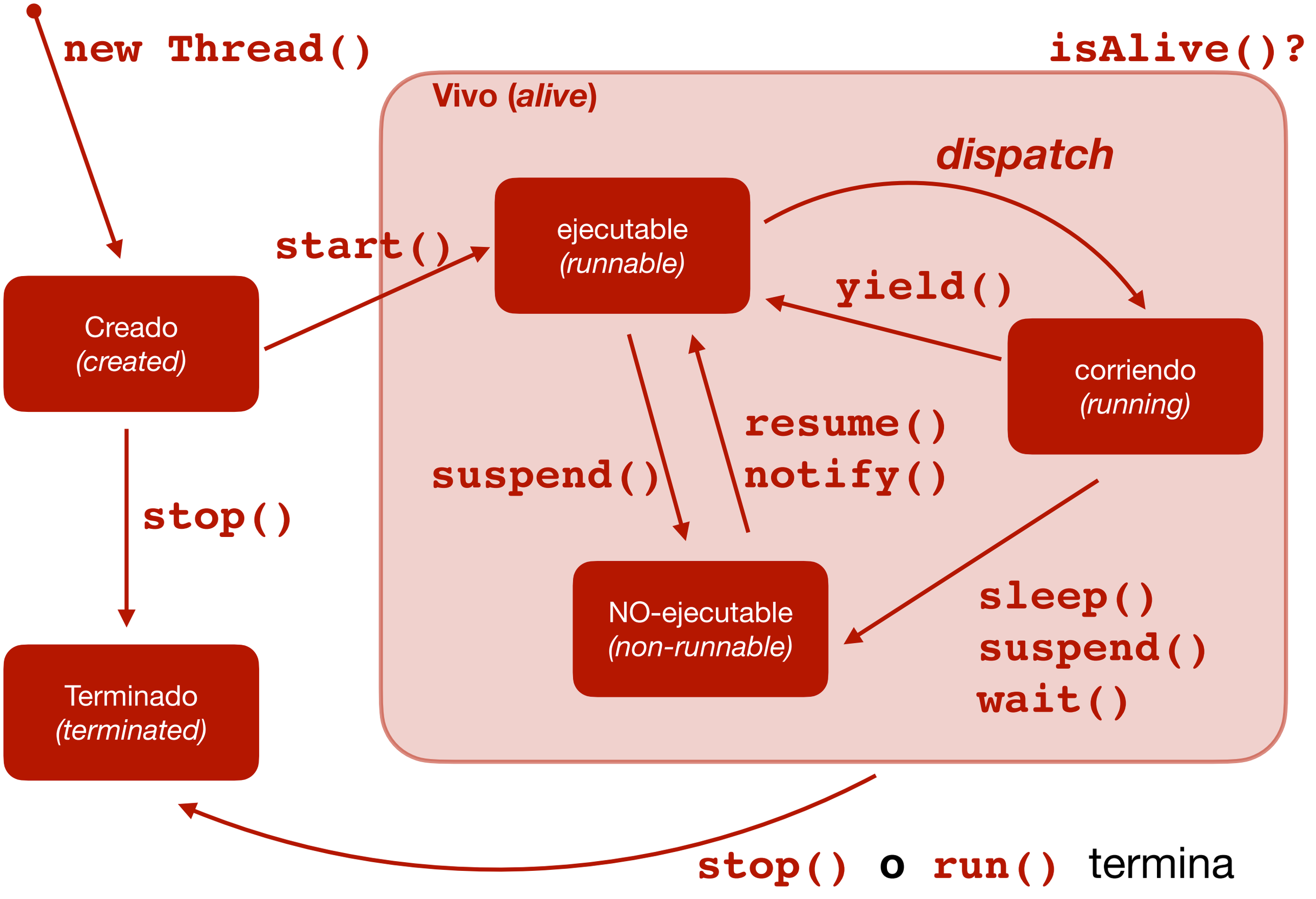
Concurrencia en Java con threads (alternativa)

Dado que Java no permite herencia múltiple, con frecuencia se implementa el método `run()` en una clase que implementa `Runnable`, en lugar de heredar de Thread.



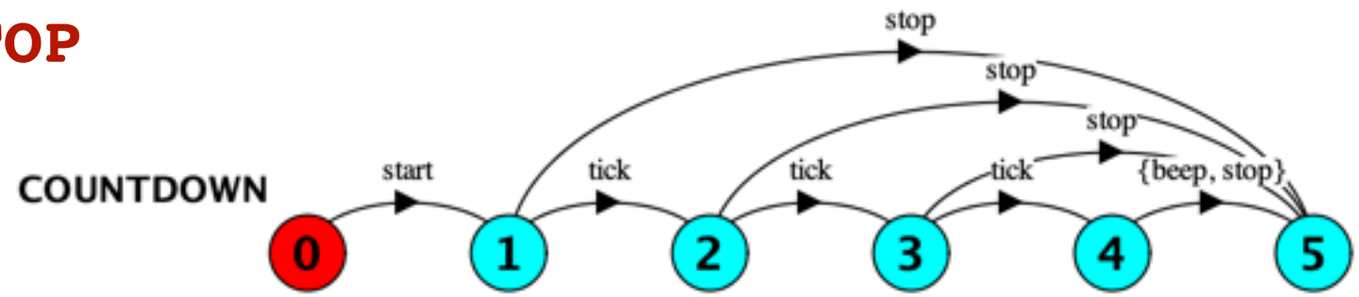

```
class MyRun implements Runnable {
    public void run() {
        //…
    }
}

Thread b = new Thread(new MyRun());
```

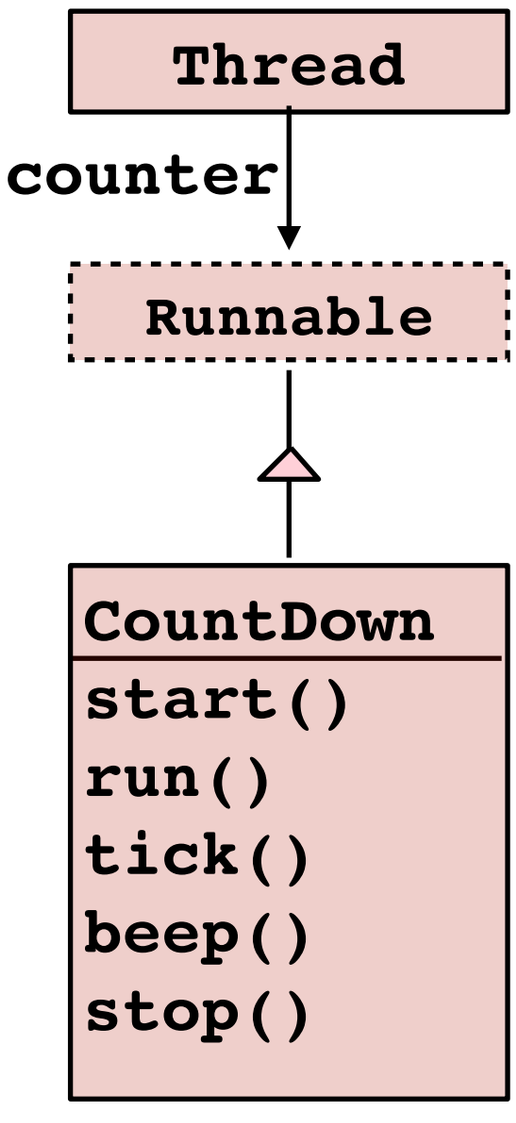
Ciclo de Vida de threads Java



Ejemplo *CountDown Timer*

```
COUNTDOWN (N=3)    = (start->COUNTDOWN[N]),
COUNTDOWN[i:0..N] =
          (when(i>0) tick->COUNTDOWN[i-1]
          |when(i==0)beep->STOP
          |stop->STOP
          ).
```






```java
public class CountDown implements Runnable {
    Thread counter; int i;
    final static int N = 3;


    public void start() {
        counter = new Thread(this);
        i = N; counter.start();
    }

    public void run() {
        while(true) {
            if (i>0)  { tick(); --i; }
            if (i==0) { beep(); return;}
        }
    }
```

Ejemplo *CountDown Timer*