INTRODUCCIÓN AL TESTING BASADO EN MODELOS

SEMANA DE LA CIENCIA Y DE LA INGENIERÍA.

UNIVERSIDAD DE CÁDIZ.

Manuel Núñez - Universidad Complutense de Madrid

WARNING!

El uso que haré del castellano en esta presentación puede ofender a alguno de los asistentes!

test,
testear, testeador,
implementación,
conformance,
input, output,
y alguna falta de ortografía como móbil o provar.

Contenidos

Conceptos generales en testing (formal).

Conformance testing: conceptos básicos.

IOCO: Una relacion de implementación para sistemas con inputs y outputs.

Validación versus Verificación

Validación es el proceso de evaluar el sistema al final de su desarrollo para asegurarse de que tiene las características deseadas.

¿Hemos construido el software correcto?
(¿es lo que el cliente quiere?)

Verificación es el proceso de evaluar el sistema para determinar si el producto desarrollado al principio de una fase cumple la condiciones que nos planteamos al principio de dicha fase.

¿Hemos construido el software correctamente?
(¿hace lo que dice la especificación?)

Testing de Software (I)

El principal propósito del **testing de software** consiste en **detectar fallos** en un sistema bajo test para que puedan corregirse con posterioridad.

Buscamos fallos **aplicando tests** a un sistema y determinando si el **comportamiento** observado es el **esperado**.

Testing de Software (II)

El testing **exhaustivo** es **impractible** o incluso imposible. En general, no podemos asegurar la ausencia de errores.

Un programa que sume dos números de 32 bits da lugar a 2^64 tests

Otro ejemplo *motivador*….

Ejemplo

- Consideremos la siguiente **especificación**

```
x in {0,1}
while true do begin
   read(x); write(x)
end
```

- ¿Podemos demostrar mediante testing que un programa no tiene errores?

```
begin
        for i:=1 to 10000000 do begin
           read(x); write(x)
        end;
        read(x); write('Soy erróneo');
        while true do begin
           read(x); write(x)
        end
 end.
```

Problema: *¿Cuántas veces damos valores?*

Testing de Software (II)

El testing **exhaustivo** es **impractible** o incluso imposible. En general, no podemos asegurar la ausencia de errores.

Un programa que sume dos números de 32 bits da lugar a 2^64 tests

La *solución* consiste en **seleccionar** el **subconjunto de tests** que tienen una mayor probabilidad de detectar la mayoría de los errores.

Conseguir métodos para realizar esta selección es un área extensísima de trabajo.

Testing de Software (II)

El testing **exhaustivo** es **impractible** o incluso imposible. En general, no podemos asegurar la ausencia de errores.

Un programa que sume dos números de 32 bits da lugar a 2^64 tests

La *solución* consiste en **seleccionar** el **subconjunto de tests** que tienen una mayor probabilidad de detectar la mayoría de los errores.

Conseguir métodos para realizar esta selección son **muchas** áreas de trabajo (*mutación*, *heurísticas*, metamórfico) y quedan fuera del ámbito de esta presentación.

Testing de caja negra vs caja blanca

Testing de caja negra: Los tests se derivan a partir de descripciones externas al software tales como especificaciones y requisitos.

Testing de caja blanca: Los tests se derivan a partir del código, específicamente, incluyendo ramas, condiciones individuales y sentencias.

Especificación de requisitos

Una descripción completa del comportamiento del sistema que vamos a desarrollar.

- **Requisitos funcionales.** ¿Qué debe hacer el sistema?
- **Requisitos no funcionales.** ?Cómo hace el sistema las cosas que hace?

Observación y análisis del comportamiento

El aplicar un test a un sistema, el testeador debe usar un **oráculo** para comprobar la corrección del comportamiento observado.

Los oráculos pueden ser especificaciones, versiones anteriores del mismo producto, expectativas del usuario o cliente, estándares relevantes, otros programas.

Observación y análisis del comportamiento

El aplicar un test a un sistema, el testeador debe usar un **oráculo** para comprobar la corrección del comportamiento observado.

Los oráculos pueden ser **especificaciones**, versiones anteriores del mismo producto, expectativas del usuario o cliente, estándares relevantes, otros programas.

Observación y análisis del comportamiento

El aplicar un test a un sistema, el testeador debe usar un **oráculo** para comprobar la corrección del comportamiento observado.

Los oráculos pueden ser especificaciones, versiones anteriores del mismo producto, expectativas del usuario o cliente, estándares relevantes, otros programas.

Oráculos: Programas

Los oráculos pueden ser también programas diseñados para comprobar el comportamiento de otros programas.

Por ejemplo, podríamos usar un programa para determinar si un programa que invierte matrices produce el resultado correcto.

Construcción de oráculos

La construcción de oráculos automáticos, tales como el anterior que comprobaba la inversa de una matriz, requieren de una descripción precisa de la relación entre inputs y outputs.

En general, la construcción de estos oráculos es muy compleja y, en algunos casos imposible:

Metamorphic Testing

Esta charla en ~~dos líneas~~ seis líneas

Presentar un marco formal para validar el comportamiento funcional de sistemas dados como una caja negra y que han sido construidos a partir de una especificación formal que cumple los requisitos iniciales. Aplicaremos tests al sistema y usaremos la especificación como oráculo.

CONFORMANCE TESTING: UN MARCO PARA TESTING BASADO EN MODELOS

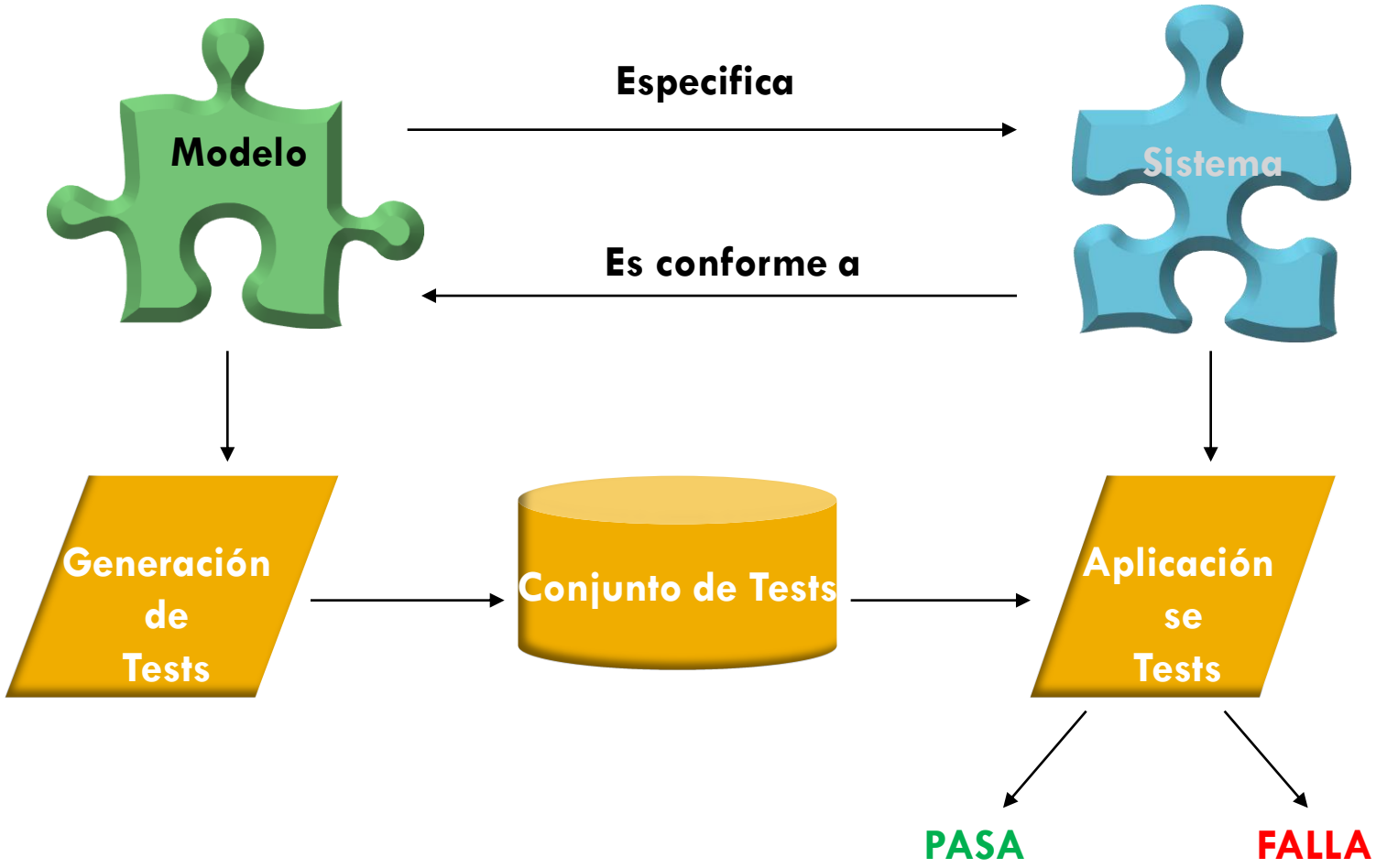
Conceptos Generales

Conformance Testing

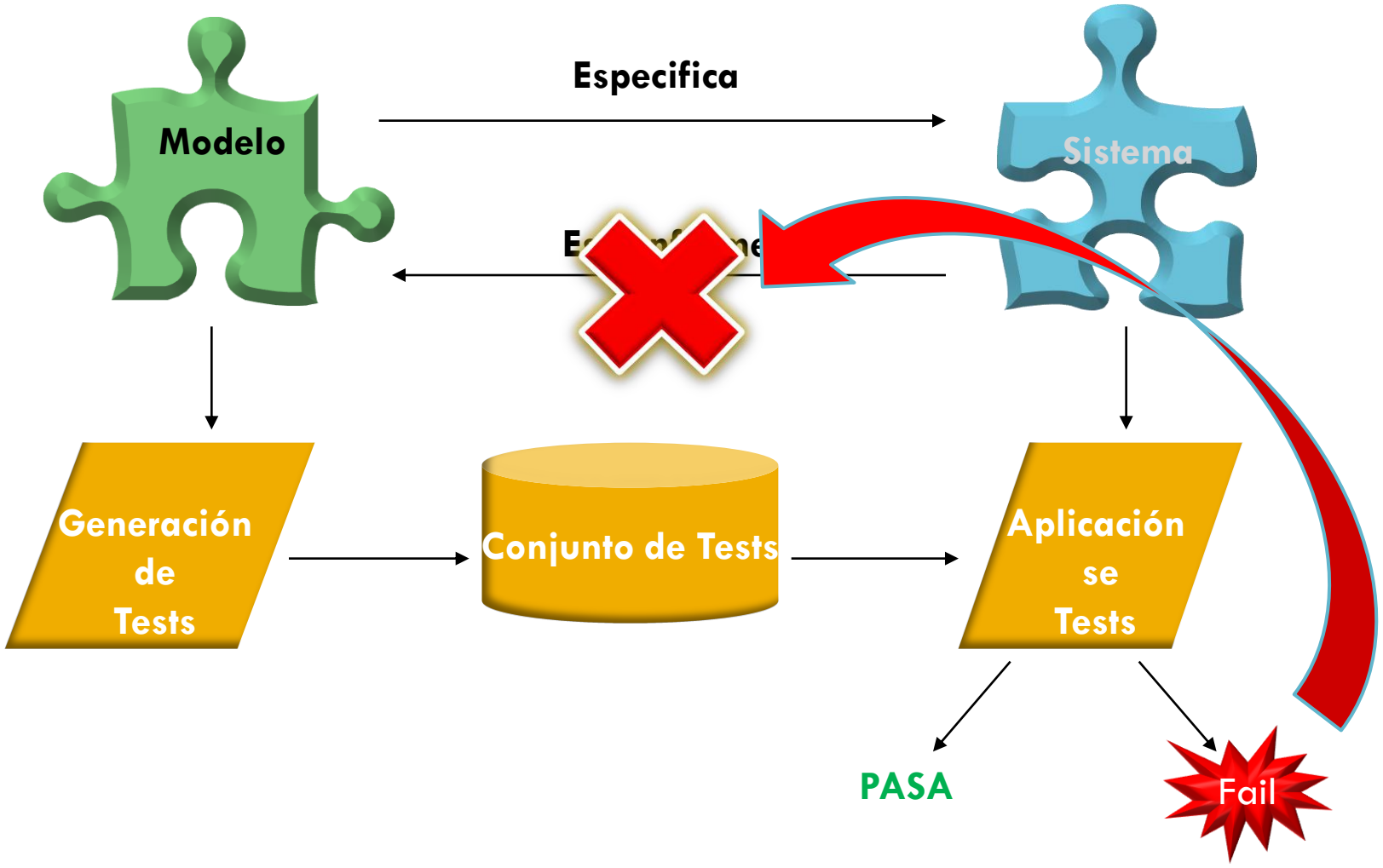
Dado una **especificación *S*** (de la que disponemos, expresada formalmente) y una **implementación bajo test *I*** (de la que podemos observar el comportamiento) queremos *testear* si

I implementa correctamente (es conforme a) S

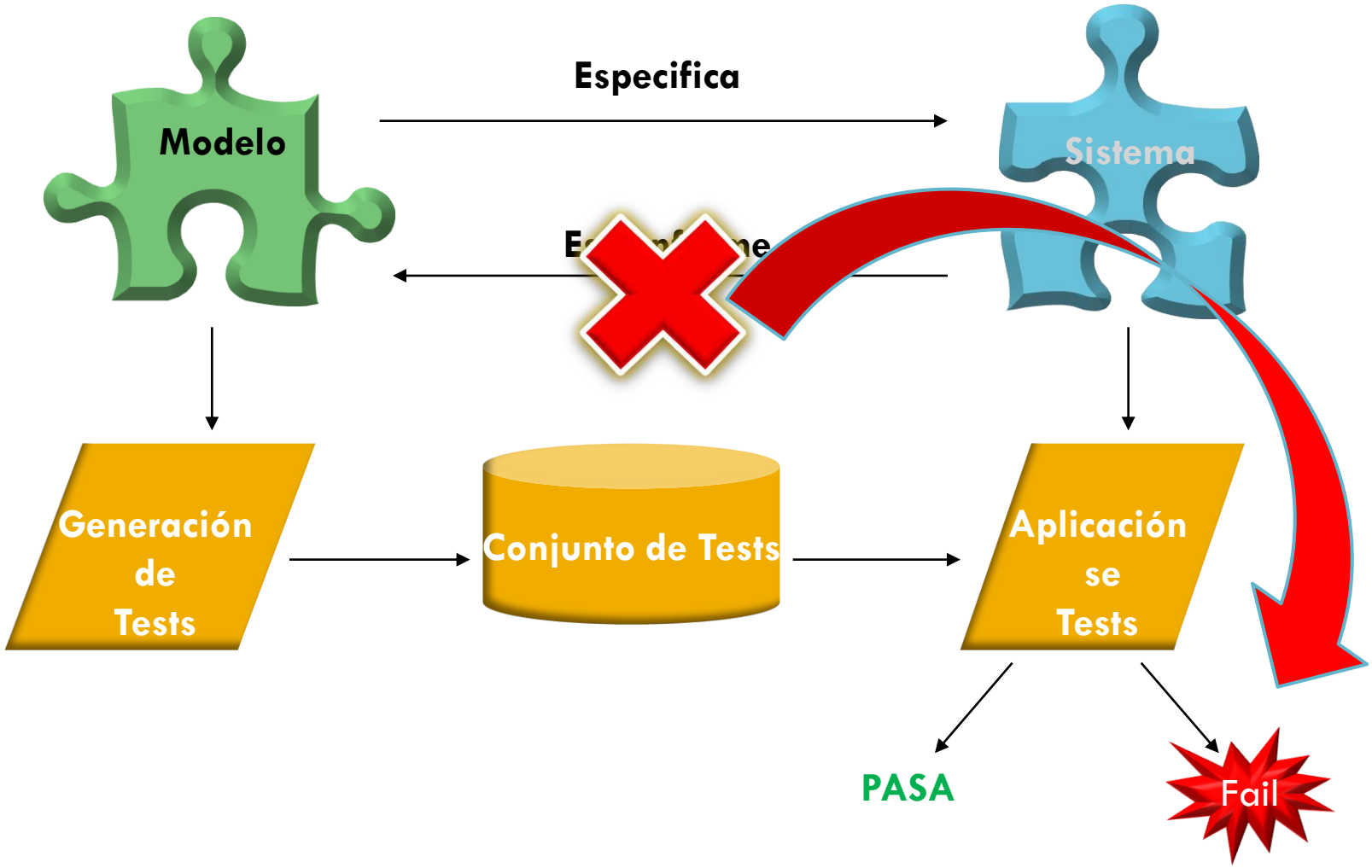
Conformance Testing



Corrección del Conformance Testing



Completitud del Conformance Testing



Corrección y Completitud

Queremos marcos de testing que sean correctos y completos!

> *Testing can never be sound and complete!*
>
> Edsger W. Dijkstra

Como ya hemos dicho, el número de tests necesarios sería usualmente infinitio (o muy, muy grande).

Por tanto, ¿Qué hacer?

Asumiremos una serie de hipótesis sobre los sistemas que estamos testeando y encontraremos conjuntos de tests (a partir del modelo) que nos permitan probar que la implementación es *buena*.

Durante el resto de la presentación veremos una de estas técnicas: **relaciones de implementación**.

IOCO: UNA RELACION DE IMPLEMENTACIÓN PARA SISTEMAS CON INPUTS Y OUTPUTS

ioco: Input-Output Conformance

Esta parte de la charla está basada en el trabajo de **Jan Tretmans,** iniciado en su tesis doctoral, terminada en 1991, en la definición de **ioco**.

Jan Tretmans: Model Based Testing with Labelled Transition Systems. *Formal Methods and Testing, An Outcome of the FORTEST Network*, LNCS 4949, páginas 1-38, Springer, 2008.

Input y Outputs

En la mayoría de los sistemas podemos distinguir entre acciones iniciadas por el entorno y acciones iniciadas por el sistema:

- *Outputs son aquellas acciones iniciadas por el sistema (no podemos controlar cuando se producen).*
- *Inputs son aquellas acciones iniciadas por el entorno(si tenemos control del entorno podemos determinar cuando se aplican).*

Testing Pasivo (e.g. *runtime monitoring*): Útil en sistemas que no podemos controlar.

Sistemas de Transiciones Etiquetadas con Inputs y Outputs

- Un IOLTS es una tupla $M = (S, In, Out, T, S_0)$:
  - S es un conjunto de **estados;** $S0 \in S$ es el estado **inicial.**
  - In y Out son dos conjuntos disjuntos de **inputs** y **outputs,** respectivamente. Etiquetamos inputs con '?' y outputs con '!'.
  - $T \subseteq S \times (In \cup Out \cup \{\tau\}) \times S$ es la **relación de transición.**
  - La acción τ denota una acción **no-observable.**

Sistemas de Transiciones Etiquetadas con Inputs y Outputs: Uso e hipótesis

Suponemos que la **especificación** del sistema que queremos testear viene dada mediante un IOLTS.

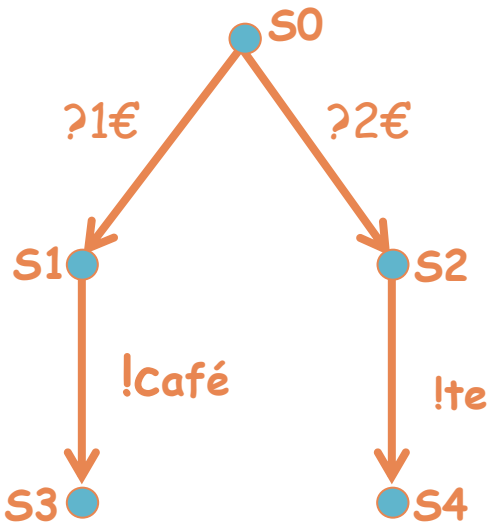
Suponemos que el comportamiento del **sistema** que queremos testear (caja negra) se puede definir mediante una IOLTS.

Importante: **NO** tenemos acceso al IOLTS que *simula* al sistema.

Supondremos que todos los inputs están disponibles en cualquier estado del sistema (***input-enableness***).

Ejemplo **complejo** de IOLTS

Sistema automatizado, altamente robotizado, y de bajo consumo energético, que recibe monedas y proporciona diversas bebidas dependiendo de la moneda recibida.



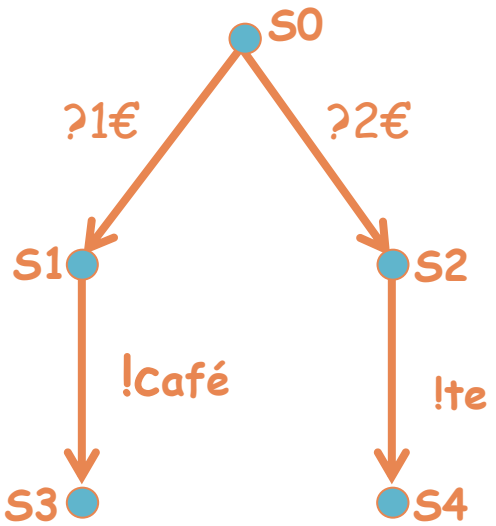
?1€, ?2€

Del usuario a la máquina.
Iniciada por el usuario.
Máquina no puede rechazar.

!café, !te

De la máquina al usuario.
Iniciada por la máquina.
Usuario no puede rechazar

Ejemplo **complejo** de IOLTS

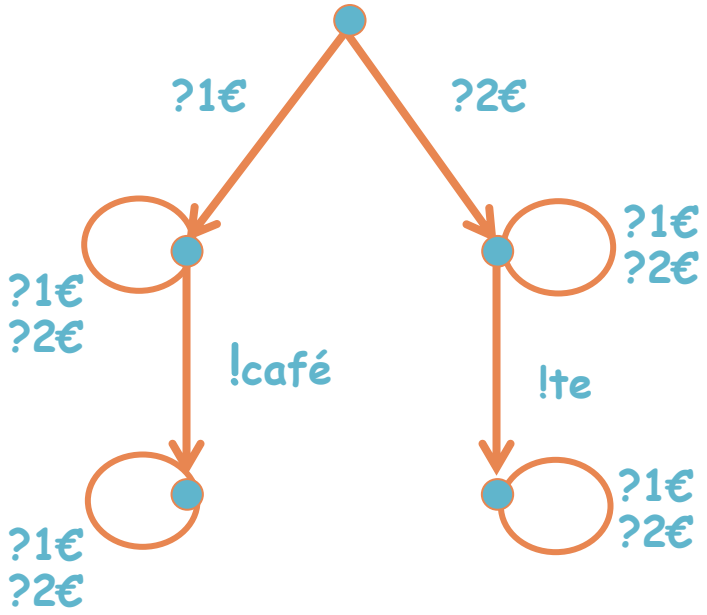


s0 ⟹ s4 (?2€ !te)

Podemos concatenar secuencias de acciones: trazas del sistema

Ejemplo (vale, no era tan complejo!)

Completando el sistema anterior para que sea *input-enabled*.



Estados Quiescentes

Si un sistema se encuentra en un estado q en el que no puede producir ningún output (sólo puede recibir inputs) decimos que está en un estado **quiescente** y añadimos una transición.

$$p \xrightarrow{\delta} p \;=\; \forall\, x \in \mathit{Out} \cup \{\tau\} .\; p \not\xrightarrow{x}$$

ioco: Relación de Implementación

$$I \text{ ioco } S \;=_{def}\; \forall \sigma \in Straces(S) : out(I \text{ after } \sigma) \subseteq out(S \text{ after } \sigma)$$

$$Straces(M) = \{\ \sigma \in (L \cup \{\delta\})^* \mid s_0 \stackrel{\sigma}{\Longrightarrow}\ \}$$

$$p \text{ after } \sigma = \{\ p' \mid p \stackrel{\sigma}{\Longrightarrow} p'\ \}$$

$$out(P) = \{\ !x \in Out \mid p \xrightarrow{!x} \wedge\ p \in P\ \} \cup \{\delta \mid p \xrightarrow{\delta} \wedge\ p \in P\}$$

ioco: Relación de Implementación

$$I \text{ ioco } S \;=_{def}\; \forall \sigma \in Straces(S) : out(I \text{ after } \sigma) \subseteq out(S \text{ after } \sigma)$$

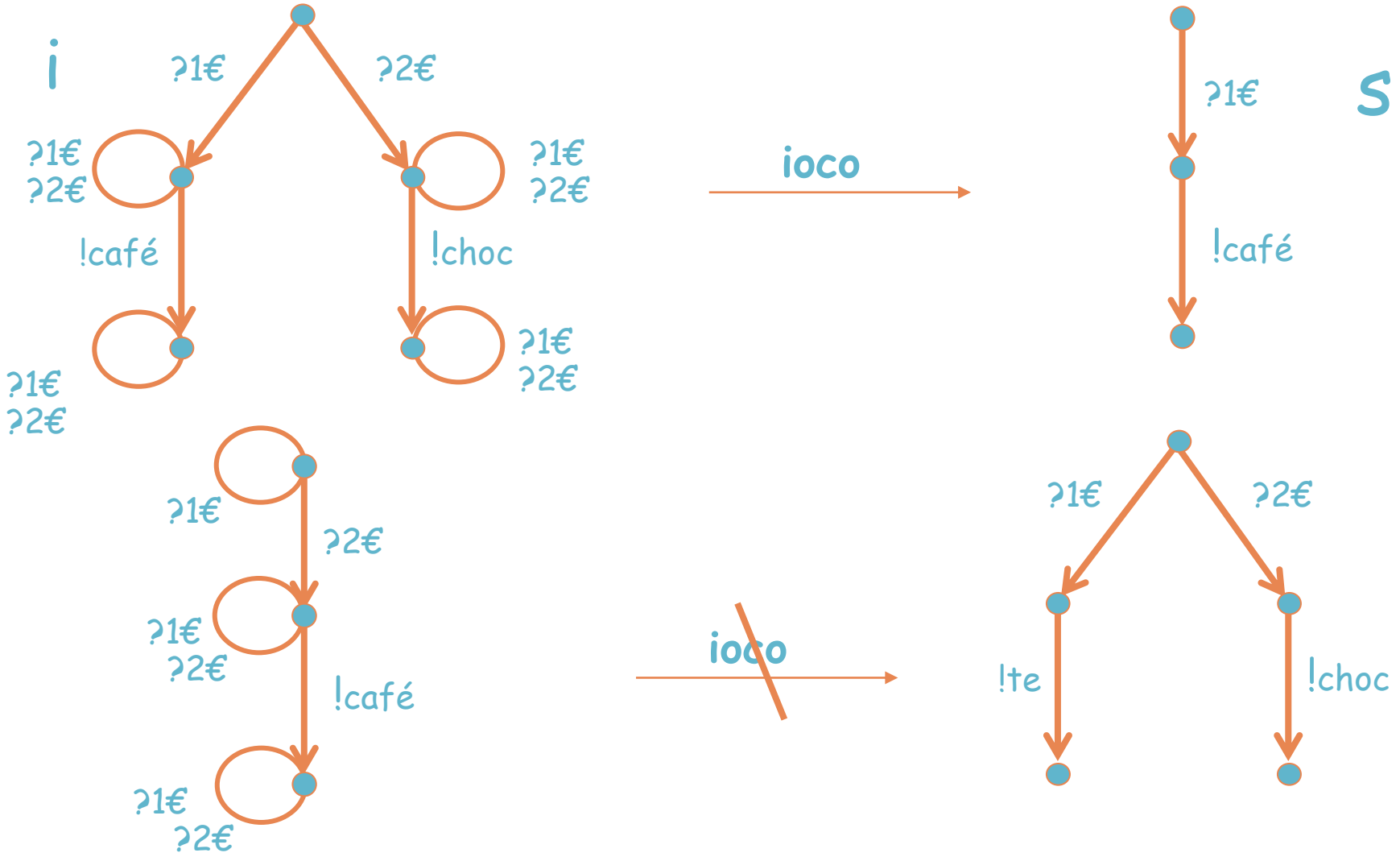
Intuición:

I es **ioco**-conforme a *S* sii

Si *I produce un output x después de una traza de* **σ** de *S* entonces *S* también puede generar x después de **σ.**

NO nos preocupamos de comportamiento de *I* para trazas que *S* no puede hacer.

2 ejemplos sencillos de conformidad



Tests

Un test es la especificación del comportamiento de un testeador que realiza un experimento.

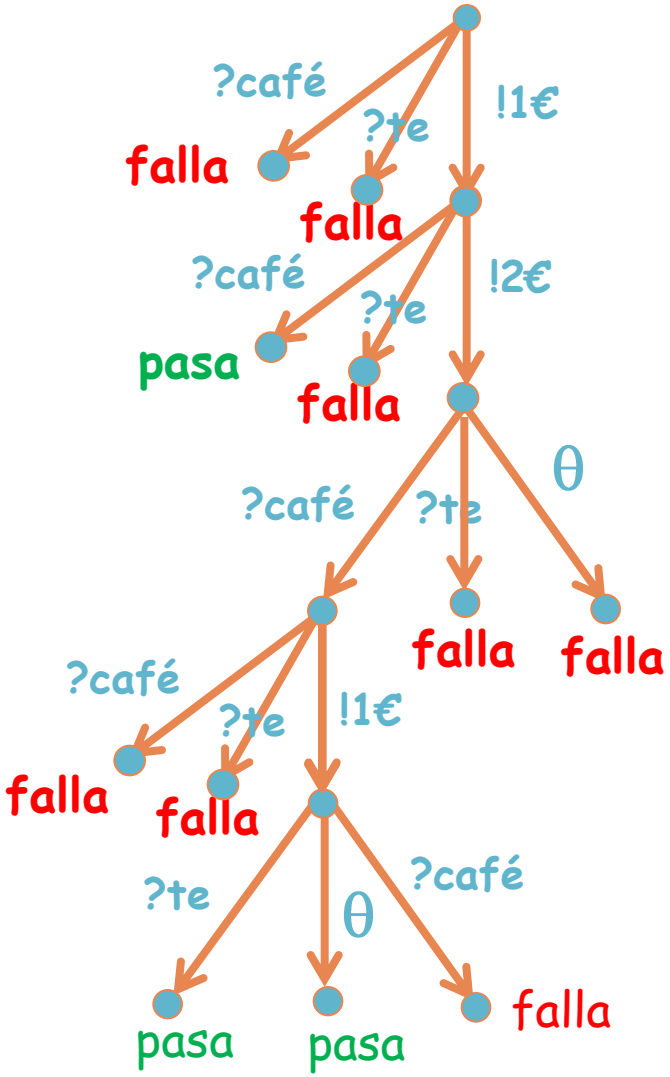
Modelaremos los tests mediante IOLTS pero intercambiando el papel de inputs y outputs.

Añadimos un símbolo especial para *observar* quiescencia: Si observamos θ *sabemos que la implementación no puede producir outputs en su estado actual.*

Tests

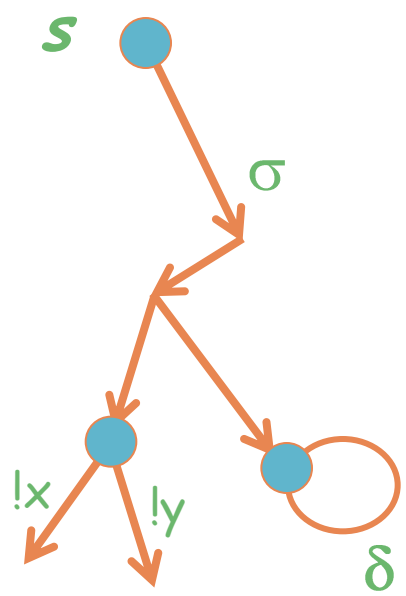
Test = sistema de transiciones

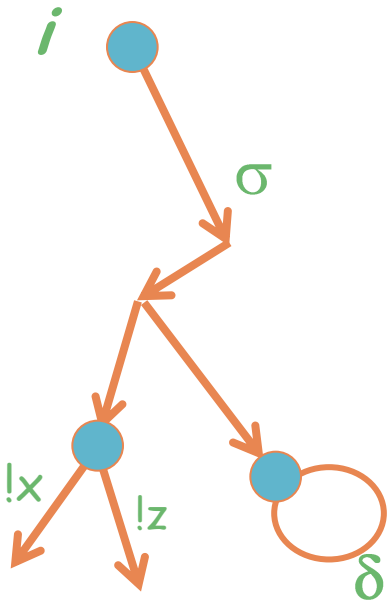
- Etiquetas pertenecen a $L \cup \{ \theta \}$
  - θ es la etiqueta de quiescencia
- Forma de árbol
- Finito y determinista
- Estado finales: **pasa** y **falla**
- Desde cada estado
  - Un input !a y todos los outputs ?x
  - O, todos los outputs ?x y θ.

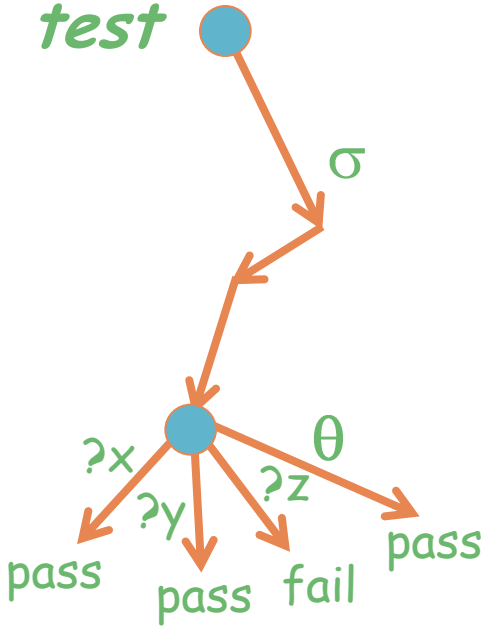


Generación de tests

i ioco s $=_{def}$ $\forall \sigma \in Straces(s) : out(i \text{ after } \sigma) \subseteq out(s \text{ after } \sigma)$







out (s **after** σ)
= { !x, !y, δ }

out (i **after** σ)
= { !x, !z, δ }

out (*test* **after** σ)
= $L_U \cup \{ \theta \}$

Algoritmo para Generación de tests

Algoritmo

Genera un test t(S) a partir de una especificación S.

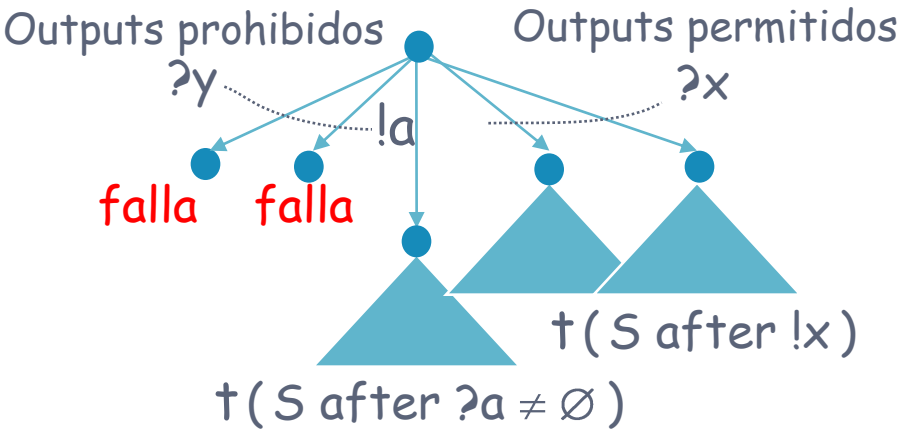
Inicializar conjunto de estados del test con S0 after ε)

Applicar los siguientes pasos, recursivamente, de forma **no-determinista**

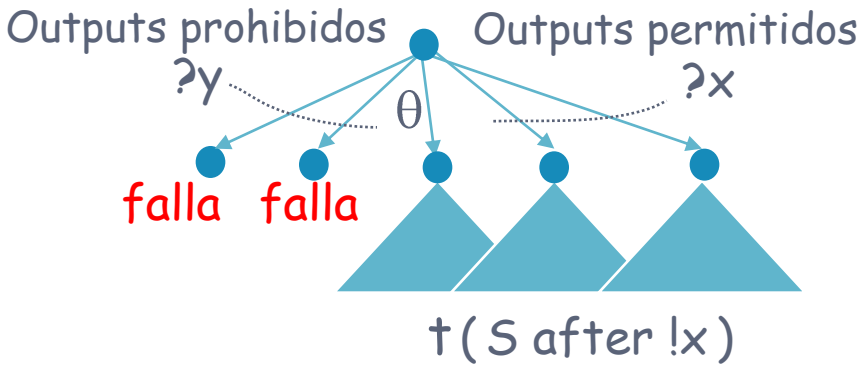
1 finalizar el test

- pass

2 Aplicar un input !a



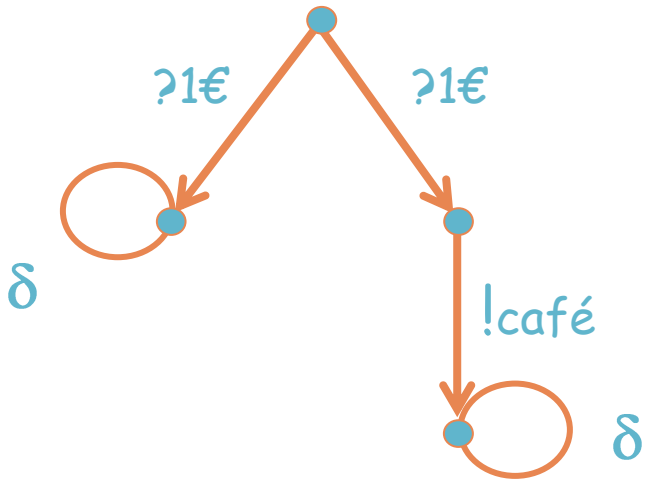
3 Observar outputs

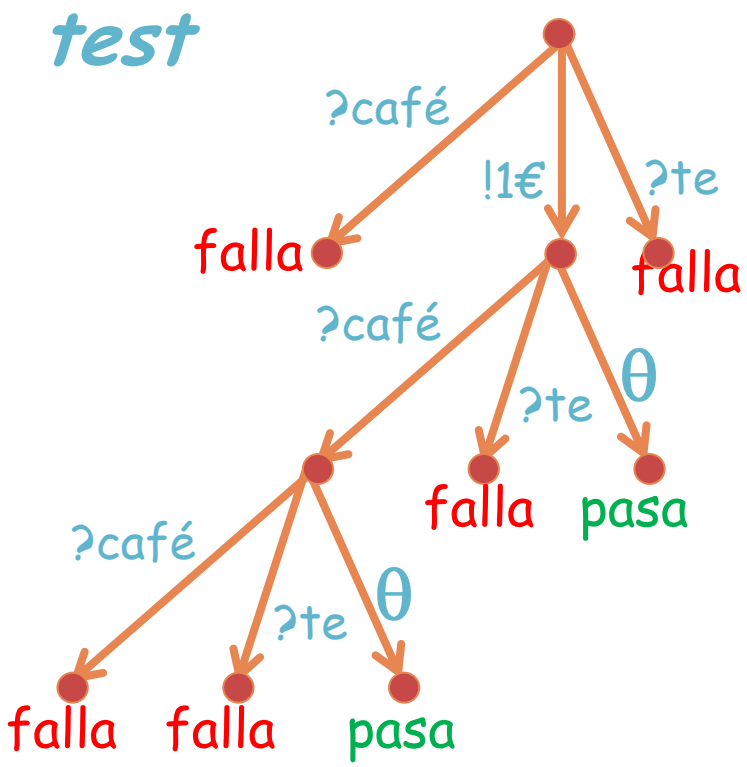


outputs (o δ) permitidos: $!x \in out(S)$

outputs (o δ) prohibidos: $!y \notin out(S)$

Ejemplo de generación de un test





Aplicación de un test

Conjunto de las posibles ejecuciones del test t a la implementación i que alcanzan un estado pasa o falla.

$$t \,\|\, i \stackrel{\sigma}{\Longrightarrow} pasa \,\|\, i' \quad o \quad t \,\|\, i \stackrel{\sigma}{\Longrightarrow} falla \,\|\, i'$$

$$\frac{t \xrightarrow{a} t', \quad i \xrightarrow{a} i'}{t \,\|\, i \xrightarrow{a} t' \,\|\, i'} \qquad \frac{i \xrightarrow{\tau} i'}{t \,\|\, i \xrightarrow{\tau} t \,\|\, i'}$$

$$\frac{t \xrightarrow{\theta} t', \quad i \xrightarrow{\delta} i'}{t \,\|\, i \xrightarrow{\theta} t' \,\|\, i'}$$

Observaciones

- Las implementaciones pueden ser no-deterministas: por tanto, distintas ejecuciones pueden dar lugar a distintos resultados.
- Una implementación pasa un test sii **todas** las posibles ejecuciones alcanzan un estado de **éxito**.
- Asumimos que si ejecutamos un test un número suficiente de veces entonces observaremos todos los comportamientos posibles para ese test (*fairness*).

Validez de la Generación de Tests

Para cada test t generado por el algoritmo tenemos:

- **Corrección:** t no fallará ante una implementación correcta

  i **ioco** *s* *implica* *i pasa t*

- **Completitud:** cada implementación incorrecta se puede detectar con un test t

  i **ioco** *s* *implica* $\exists t$: *i falla t*

Validez de la Generación de Tests

Aunque el proceso anterior es correcto con respecto a ioco, no podemos hablar, propiamente, de completitud

¡El conjunto de tests puede ser infinito!

Podemos hablar de completitud en el límite: dado *n*, podemos asegurar que la implementación es correcta para todas las trazas de longitud *n* generando un conjunto finito de tests.

Cuando hacemos tender *n* a infinito, obtenemos completitud (en el límite).

Comentarios finales

Utilizar un proceso formal para realizar testing permite, entre otras cosas, la (semi-)automatización del proceso.

Las técnicas formales están ganando peso, incluso en la industria, pero todavía queda mucho camino.

En mi opinión, es importante formalizar lo máximo posible porque ayudará a incrementar la confianza en la corrección de los sistemas producidos.

¡Gracias por la atención!

¿Preguntas?