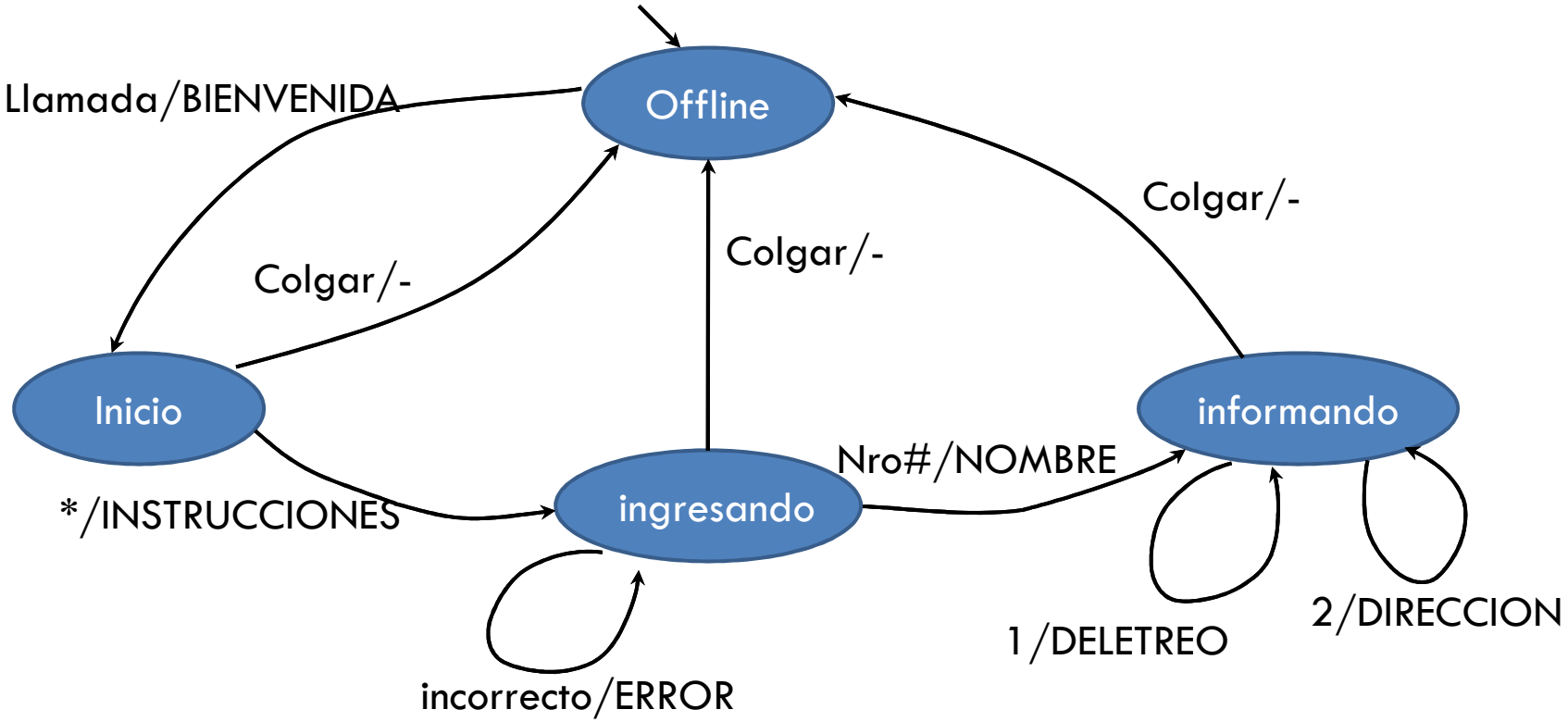
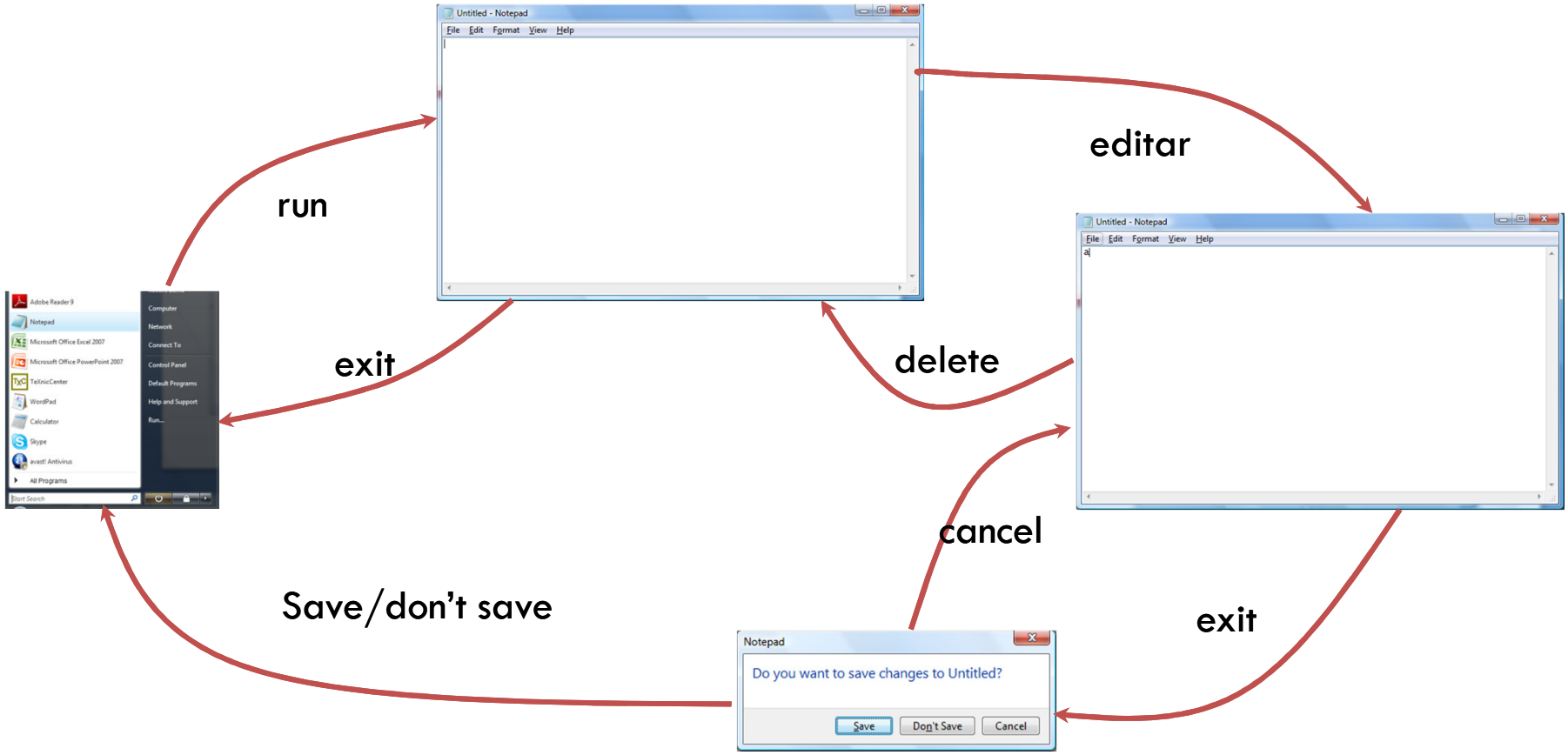
MODELOS DE SISTEMAS REACTIVOS - CONFORMANCE

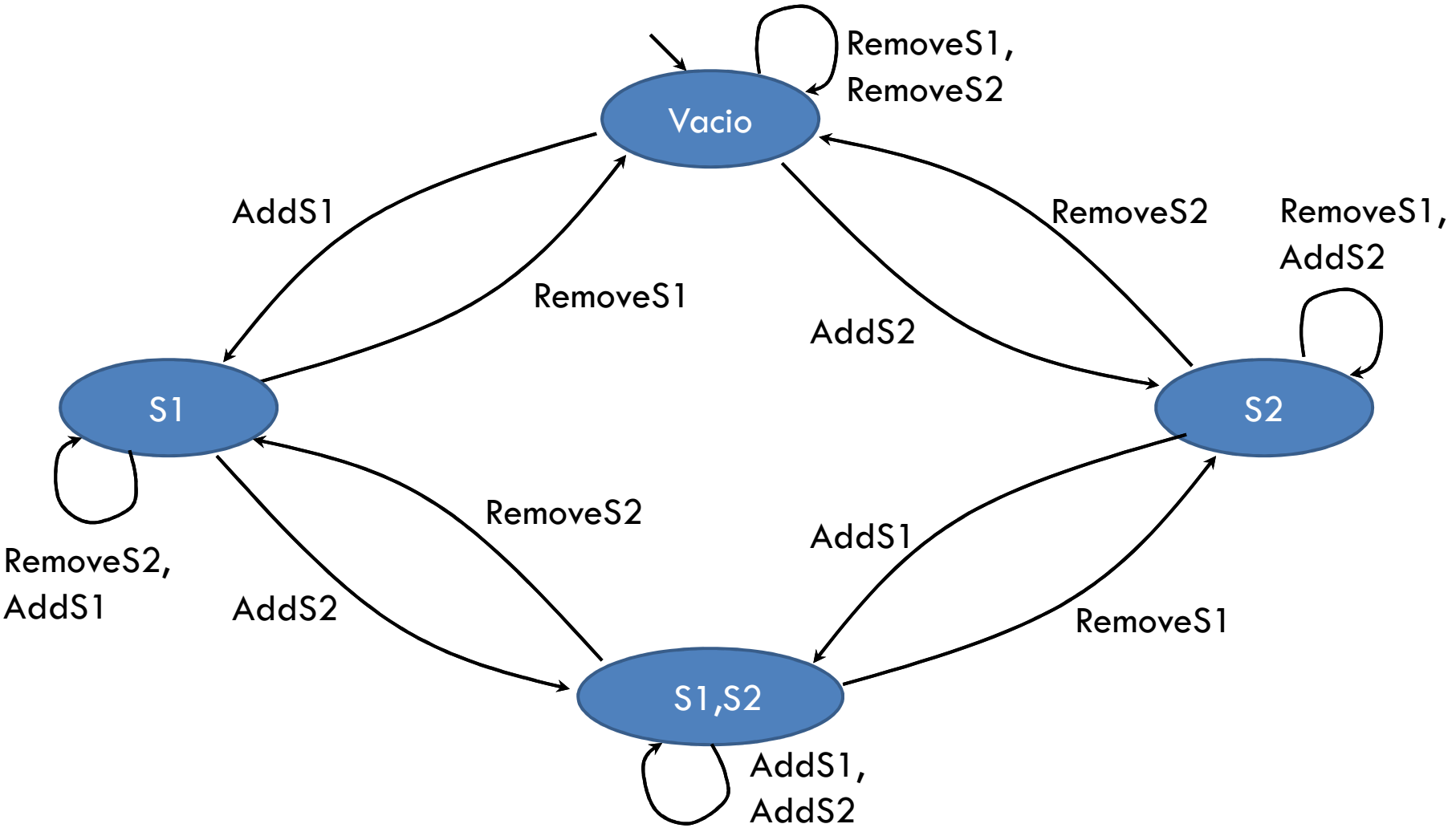
Modelo (MEF)


Offline
Llamada/BIENVENIDA
Colgar/-
Colgar/-
Colgar/-
Inicio
informando
Nro#/NOMBRE
*/INSTRUCCIONES
ingresando
1/DELETREO
2/DIRECCION
incorrecto/ERROR

NotePad


Untitled - Notepad
File Edit Format View Help
run
editar
exit
delete
cancel
Save/don't save
exit
Notepad
Do you want to save changes to Untitled?
Save
Don't Save
Cancel

Modelo: Set<T>

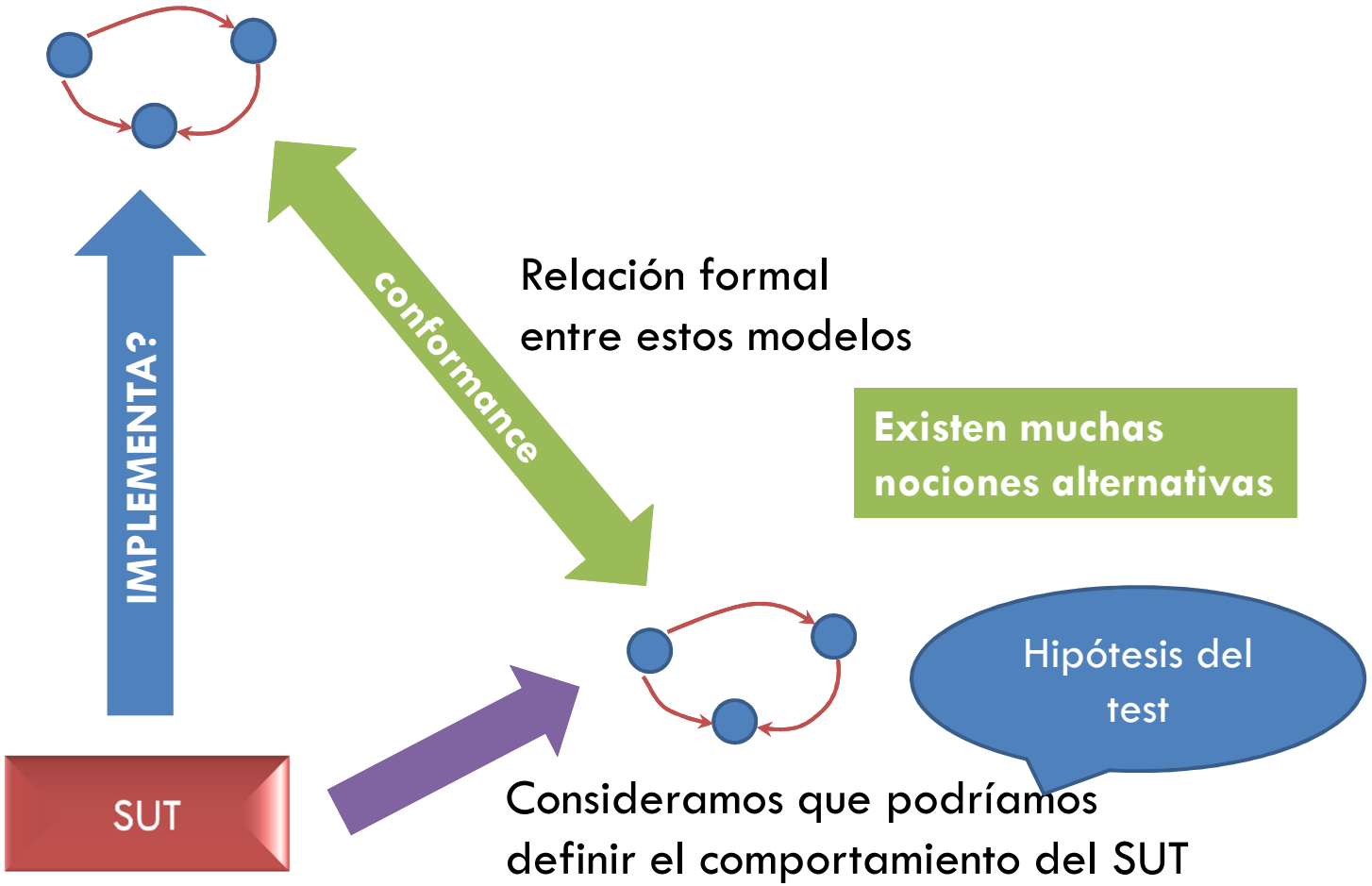

Vacio
RemoveS1,
RemoveS2
AddS1
RemoveS1
RemoveS2
AddS2
RemoveS1,
AddS2
S1
S2
RemoveS2,
AddS1
RemoveS2
AddS2
AddS1
RemoveS1
S1,S2
AddS1,
AddS2

TypeState de Pipe JDK 1.4

Necesidad de revisar nociones básicas

- No pensamos más en el sistema como una transformación de entradas en salidas (función)
- Puedo tener problemas para comprobar el estado
  - Por eso no podemos utilizar la noción tradicional de corrección funcional
    - Hablamos de *conformidad, que* establece formalmente cuál es la relación entre la especificación y la implementación
  - Caso y dato de test tienen que ser revisada
    - Por ejemplo, secuencias de interacciones

Conformance testing



Relación formal
entre estos modelos

Existen muchas nociones alternativas

Consideramos que podríamos definir el comportamiento del SUT con un determinado tipo de modelo

TESTING DE MÁQUINAS DE ESTADO

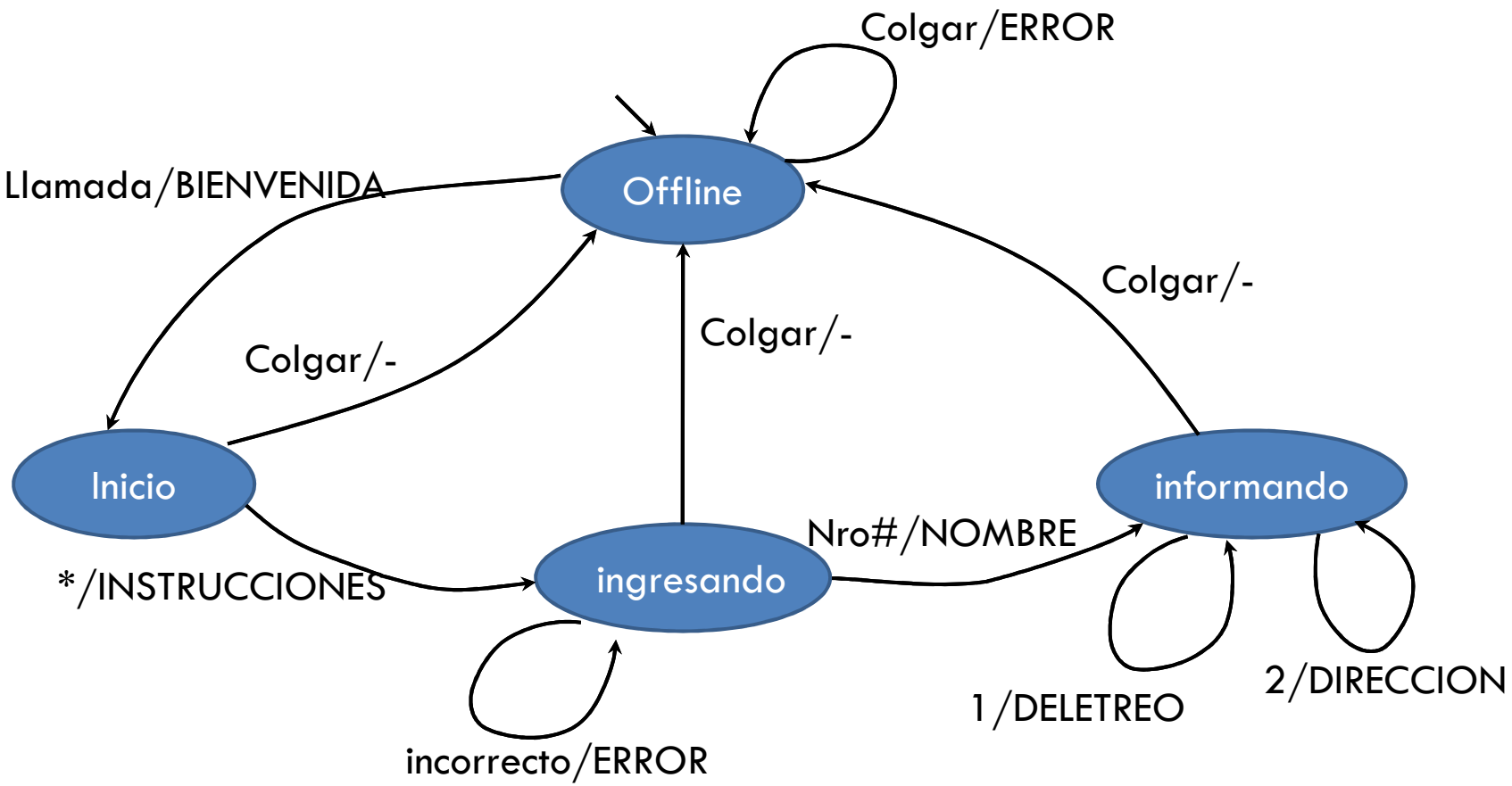
Testing de Mealy Machines

- Un caso particular de modelo de sistema reactivo
- Modelo limitado comparado con los anteriores IOLTS
- Usado para circuitos secuenciales y algunos protocolos de comunicación
- Modelo estudiado desde mediados de los 50´
- Muchos resultados algorítmicos de complejidad
- Introduce conceptos importantes para el testing de sistemas reactivos

Mealy Machine

- Es una tupla $L = \langle I, O, S, \delta, \lambda \rangle$
  - I, O conjuntos de símbolos de input y output
  - S es el conjunto de estados
  - δ: $S \times I \rightarrow S$ función de transición de estados
  - $\lambda : S \times I \rightarrow O$ función de transición de output

Ejemplo: Sistema directorio



Secuencias y equivalencia de estados

- Extendemos las funciones de transición y output a secuencias de input
  - Sea x un input string: $x=a_1,\ldots a_k$, a_i en I, para i=1..k, entonces:
  - S es el conjunto de estados
  - $\delta(s_1,x) = s_{k+1}$ dónde $s_{i+1} = \delta(s_i, a_i)$, para i=1..k
  - $\lambda(s_1,x) = b_1 \ldots b_k$ dónde $b_k = \lambda(s_i, a_i)$, para i=1..k
  - s_i es equivalente a s_j sii para toda x en I^* $\lambda(s_i,x)= \lambda(s_j,x)$

Equivalencia de máquinas

- Una máquina A es equivalente a una máquina B sii para todo estado de A existe uno equivalente y viceversa
- Cada clase de equivalencia tiene una máquina *minimizada* con la cantidad mínima de estados dónde cada estado es no equivalente al resto
- Las minimizadas son únicas (o sea son isomorfas)

El problema de conformidad para máquinas de estado de Mealy

- Tenemos información completa de la máquina **A de especificación**
- Hay una **implementación** que se comporta como una máquina **B** para la cual **sólo podemos observar I/O**
- Objetivo: Determinar si B es equivalente a A aplicando una secuencia de test y observando el output. O sea, encontrar una "checking sequence" de A para cierta familia de máquinas.
  - Una "Checking sequence" para A para una familia de máquinas F es una secuencia x de símbolos de input que distingue a A de cualquier otra máquina no equivalente B de F
  - Se sabe que existe una de estas de tamaño $O(p^2n^4 \log(qn))$
  - Cota inferior $\Omega(pn^3)$

Hipótesis y supuestos usuales para el test de conformidad

- La especificación A es fuertemente conexa
- A está minimizada
- B está fija y tiene el mismo alfabeto de símbolos que A
- B no tiene más estados que A (esa es la familia F)
  - Modelo de fallas: output o transición equivocada
  - Esta restricción puede caer pero hay un costo exponencial sobre el delta sobrante

Estructura típica del algoritmo

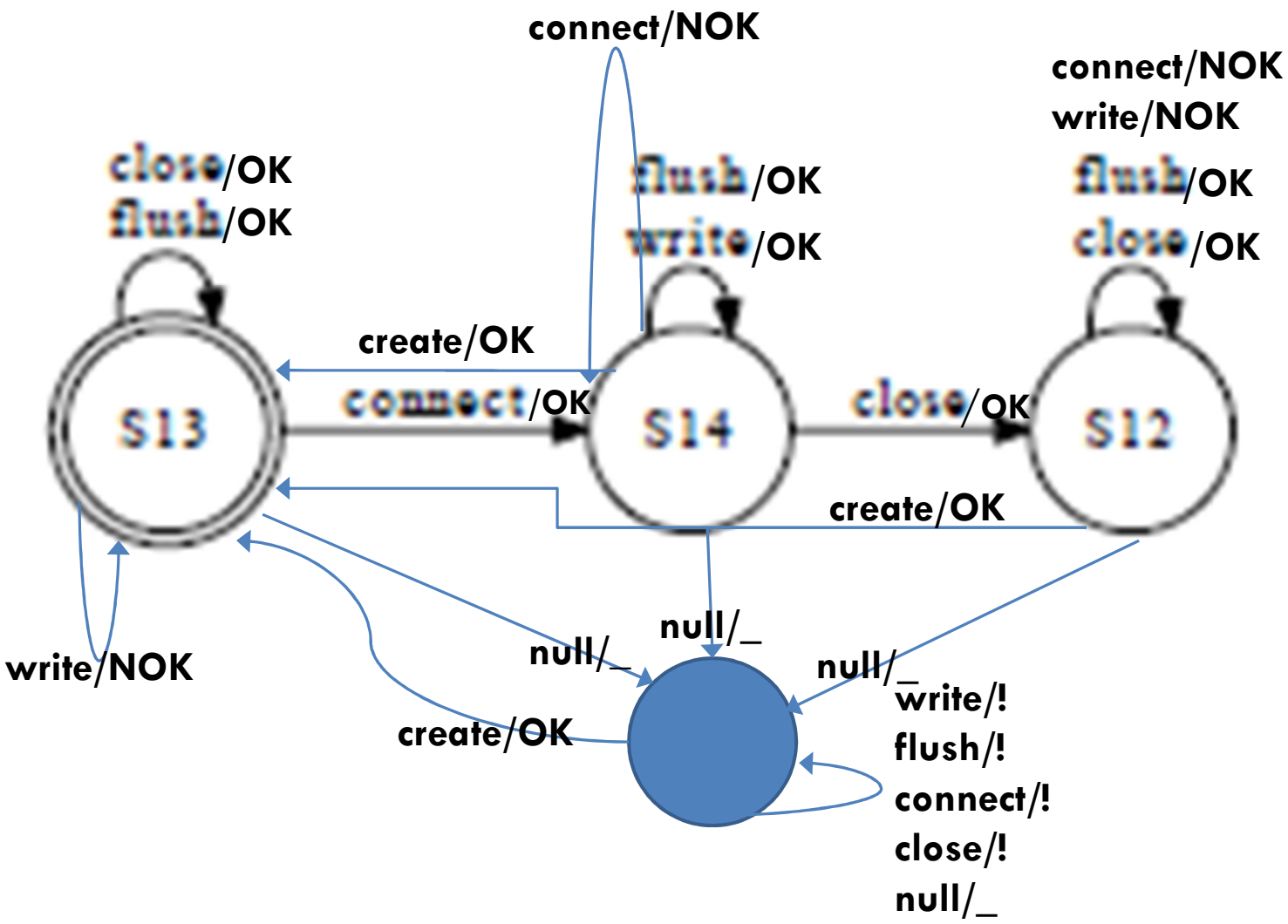
- Inicialización: moverse a un estado conocido s_1
- Verificar la similitud de los estados de B a los de A
- Verificar cada transición $\delta(s_i,a)=s_i$
  - Aplicar una secuencia que mueva a la máquina al estado s_i
  - Aplicar a
  - Verificar que la máquina está en s_i

Algoritmos

- Muchos algoritmos con distintos supuestos de control, identificación de estados y confiabilidad de acciones (ver: "Principles and Methods of Tetsing Finite State Machines –A survey": David Lee and Mihalis Yannakakis, Proccedings of IEEE, 1996)

- Vamos a concentrarnos en algoritmo de T.S.Chow (Chow, T. S. 1978. Testing Software Design Modeled by Finite-State Machines. IEEE Trans. Softw. Eng. 4, 3)
- Correcto para la familia definida asumiendo la hipótesis "reliable reset"
- O (pn^3) dónde p es el número de inputs y n la cantidad de estados

Ejemplo: Type State del Pipe



Separating sequences

- Una "separating family" para A es una colección de n conjuntos de strings de inputs Z_i (uno para cada estado) tal que
  - Para cada par de estados s_i, s_k
  - Existe α tq. $\neq \lambda_A(s_k, \alpha)$ y α es prefijo de un x_i en Z_i y de un x_k en Z_k
- Z_i se lo llama conjunto separador para el estado s_i
- Los elementos de Z_i son secuencias separadoras del estado s_i
- **Propiedad clave:** Dada una máquina B y un estado q_i de B no hay más de un estado s_i de A tal que sometido a todas la secuencias separadoras de s_i coincide el output con el de s_i

Cómo conseguir Familias

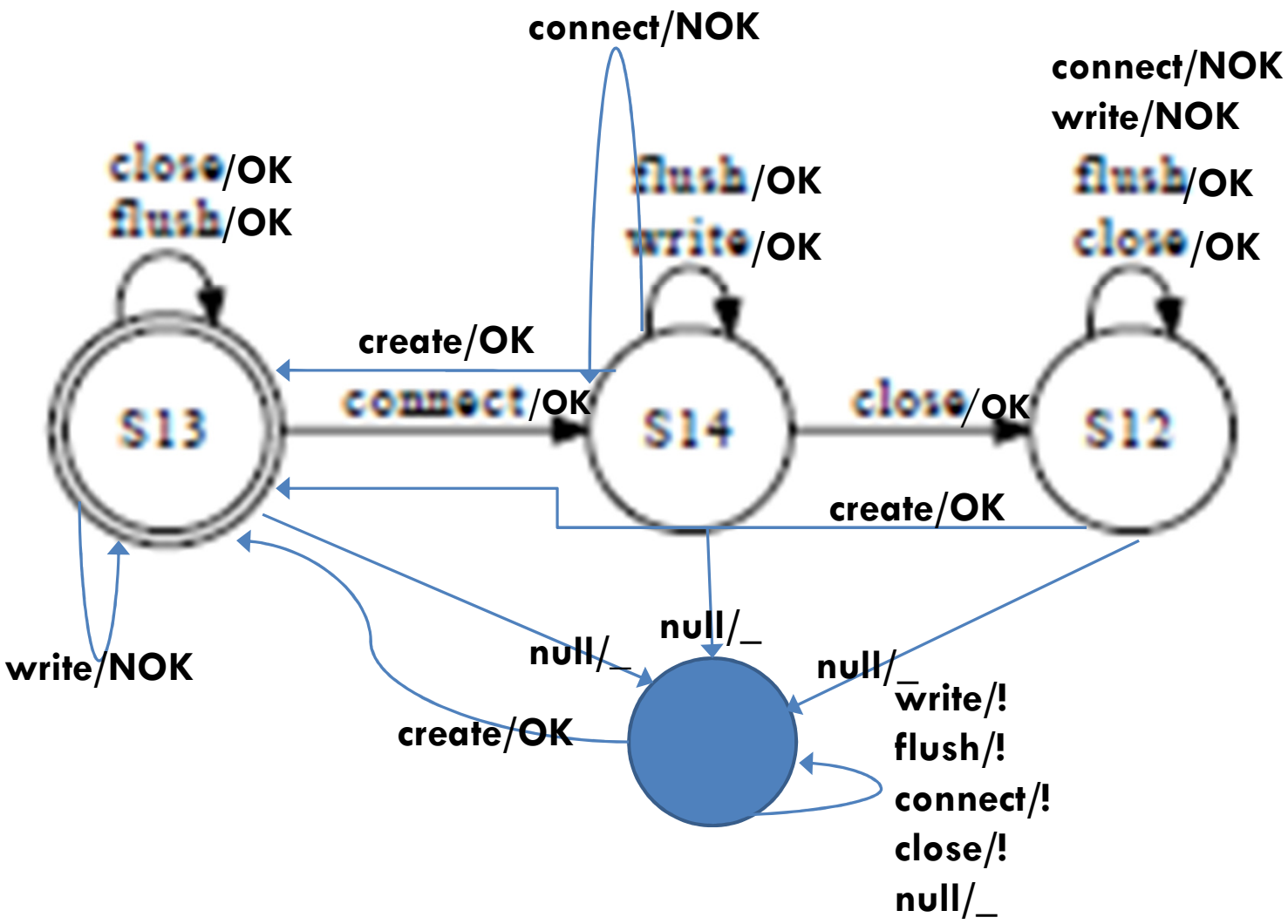
- Si A tiene una secuencia distintiva preset x, entonces los $Z_i = \{x\}$
- Si A tiene una secuencia de distinción adaptativa entonces podemos definir $Z_i = \{x_i\}$ donde cada x_i camino de ese árbol de decisión que termina en el veredicto s_i
- No siempre existen secuencias distintivas
- A continuación un algoritmo para encontrar familias para todo caso

Algoritmos general de construcción de familias para FSMs minimizadas

- Sabemos que al estar reducida A para todo s_i, s_J existe x que los distingue
- Partimos los estados de acuerdo al resultado de $\lambda(s_k,x)$ y ponemos en cada Z_k a x
- Repetir para cada bloque hasta que todos los bloques sean singletons

- Al final cada par de estados tiene una secuencia que los distingue con un prefijo común
- Z_i contiene <n-1 secuencias de long. menor o igual a n

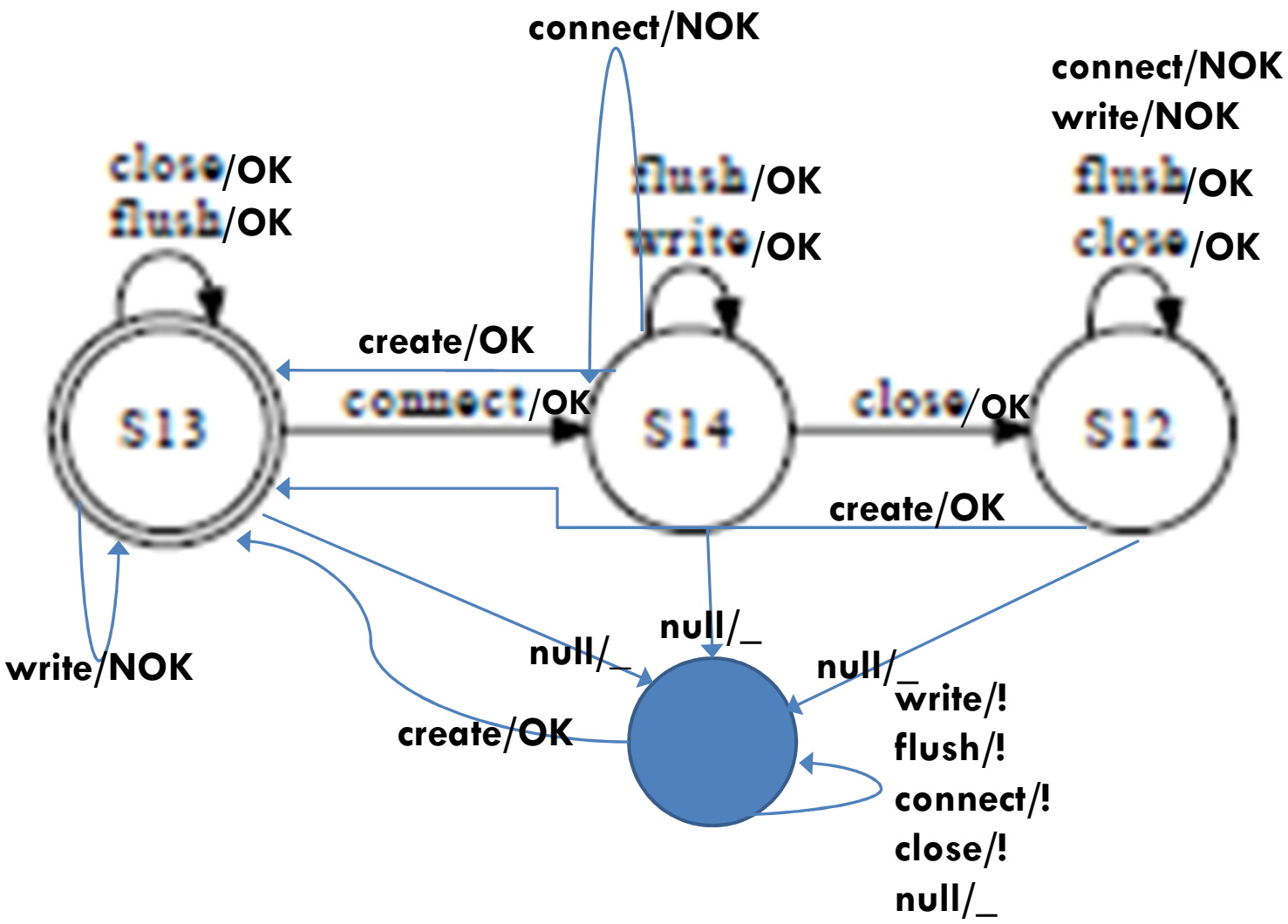
Ejemplo: Type State del Pipe



La hipótesis del Reset Confiable

- Una máquina tiene capacidad de reset si un input especial r lleva la máquina desde cualquier estado a uno inicial s_1
- Decimos que vale el supuesto de reset confiable cuando tenemos como hipótesis que funciona correctamente en la máquina de implementación B
- El algoritmo que vamos a ver supone reset confiable
- Hay algoritmos que no necesitan este supuesto

Ejemplo: Type State del Pipe

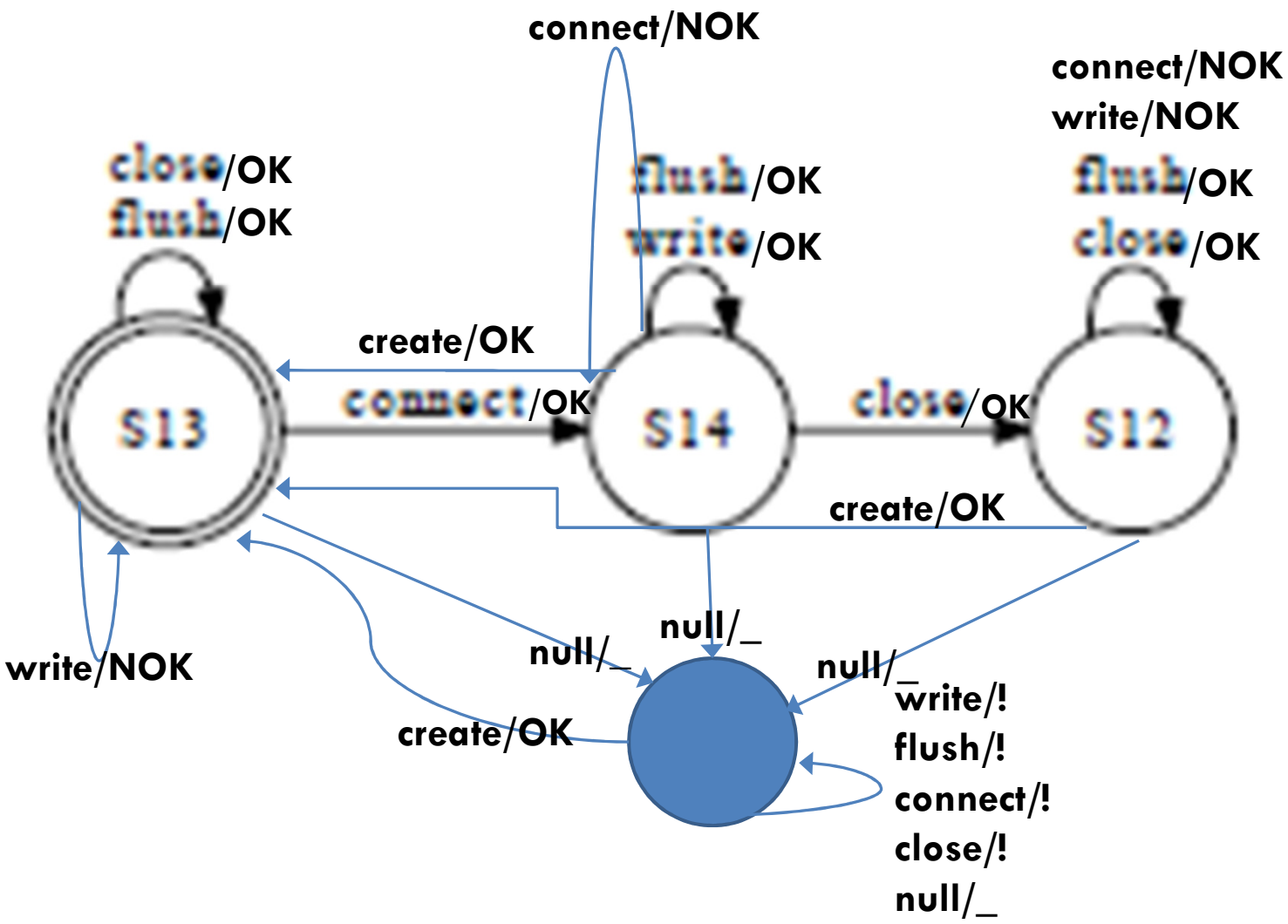


Test de Conformidad con Reset Confiable

- Sea Z_i una familia de conjuntos separadores
- Construir un árbol generador A con raíz en s_1
- Para cada s_i de S_A
  - Para cada x de Z_i
    - Resetear B al estado que debería ser similar a s_1
    - Moverse usando el camino propuesto por el árbol de s_1 al supuesto estado s_i
    - Aplicar x
- Para cada transición no cubierta por el árbol generador hacer algo similar

- O (pn^3) dónde p es el número de inputs y n la cantidad de estados

Ejemplo: Type State del Pipe



Más en general (pero con la restricción de cantidad de estados de B)

- Si no valen supuestos de existencia de secuencias o transiciones especiales,
  - Hay métodos exponenciales
  - No se conocen algoritmos determinísticos polinomiales para generar una secuencia de chequeo y sabemos que hay de longitud polinomial
- Hay algoritmos randomizados polinomiales

Testing Random para una B fija

- Para I = 1,…, k:
  - Elegir una transición al azar, ej. : $\delta(s_i,a)=s_j$
  - Aplicar una secuencia de inputs que transporta en A desde el estado corriente a s_i
  - Aplicar input a
  - Elegir uniformemente una secuencia de Z_j y aplicarla
- Sea B una máquina fallada con a lo sumo n estados
- Vale que para $\varepsilon > 0$, con una secuencia de longitud máxima $2pn^2z\log(1/\varepsilon)$ que es generada después de $k=pnz\log(1/\varepsilon)$ iteraciones detecta que B está fallada con probabilidad $>= 1-\varepsilon$

Más estados

- El problema es mucho más duro
- Problema del Universal traversal de grafos
- B tiene $n+\Delta$ estados , la cota inferior de una secuencia de chequeo es $\Omega\ (p^{\Delta+1}\ n^3)$

- Una máuina defectusa B con a lo sumo $n+\Delta$ estados falla un test de longitud $2\ p^{\Delta+1}\ n^2 z \log(1/\varepsilon)$ con probabilidad al menos $>= 1-\varepsilon$

No determinismo

- Útil: abstracción, asincronismo, eventos no controlables, etc.
- EXP-time complete el problema de distinción de estados
- Se usan otros modelos con teoría sólida de conformidad pero con menos resultados teóricos sobre la corrección de la relación establecida

Heurísticas y Optimizaciones

- UIO junto con Rural Postman Tour

- Random Walk

TESTING DE LTS

Re[asemos: Sistema de transición etiquetadas (LTS)

- Es una tupla $L = \langle S, s_0, \Sigma, \rightarrow \rangle$
  - S es el conjunto de estados
  - $s_0 \in S$ es el estado inicial
  - Σ es el conjunto de acciones observables
  - $\rightarrow \in S \times \Sigma \cup \{\tau\} \times S$ es la relación de transición
    - $s \xrightarrow{\mu} s'$ denota $(s,\mu,s') \in \rightarrow$
    - τ denota una acción interna

Composición de transiciones

- $s \xrightarrow{\mu} s'$: cuando el sistema está en el estado s puede ejecutar la acción μ e transitar al estado s'
- Las transiciones se pueden componer

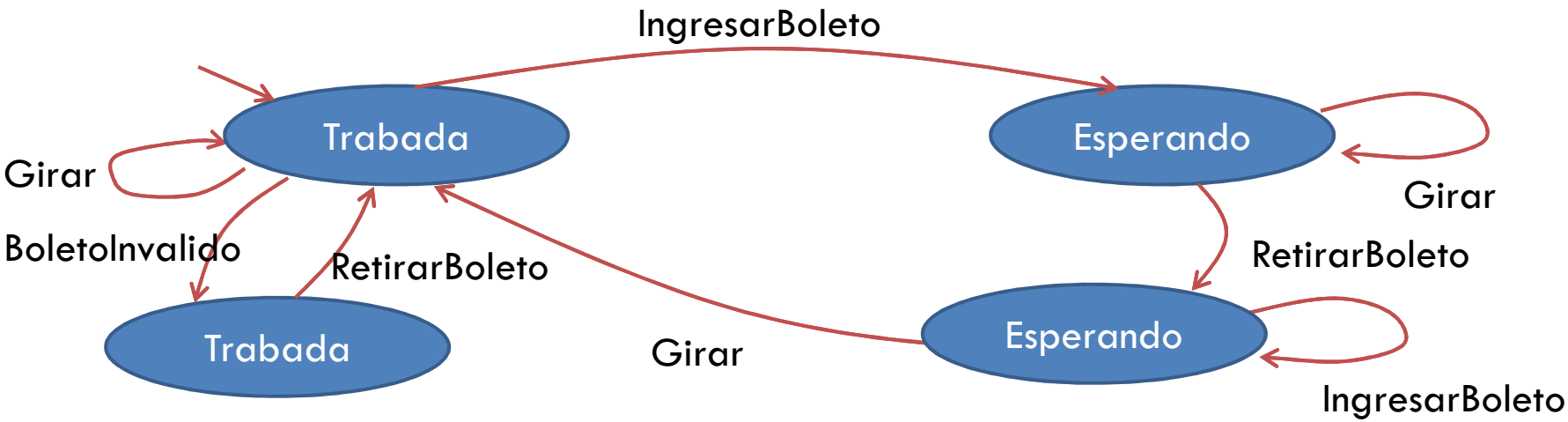
  $$s \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} s'$$

  - Abreviamos $s \xrightarrow{\mu_1\mu_2} s'$
- Para $\alpha=\mu_1\mu_2 \ldots\mu_n \in \Sigma^*$
  - $s \xrightarrow{\alpha} s'$ denota $s \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} s'$
  - $\alpha = \in$ cuando n=0. En este caso s'=s

Nociones básicas

- Trazas
  - $Tr(s) = \{\alpha \mid \alpha \in \Sigma^* \text{ y } s \xrightarrow{\alpha} s'\}$
  - $Tr(L) = Tr(s_0)$
- Alcanzables
  - $Alc(s, \alpha) = \{s' \mid s \xrightarrow{\alpha} s'\}$

Trazas y Alcanzables: Ejemplo

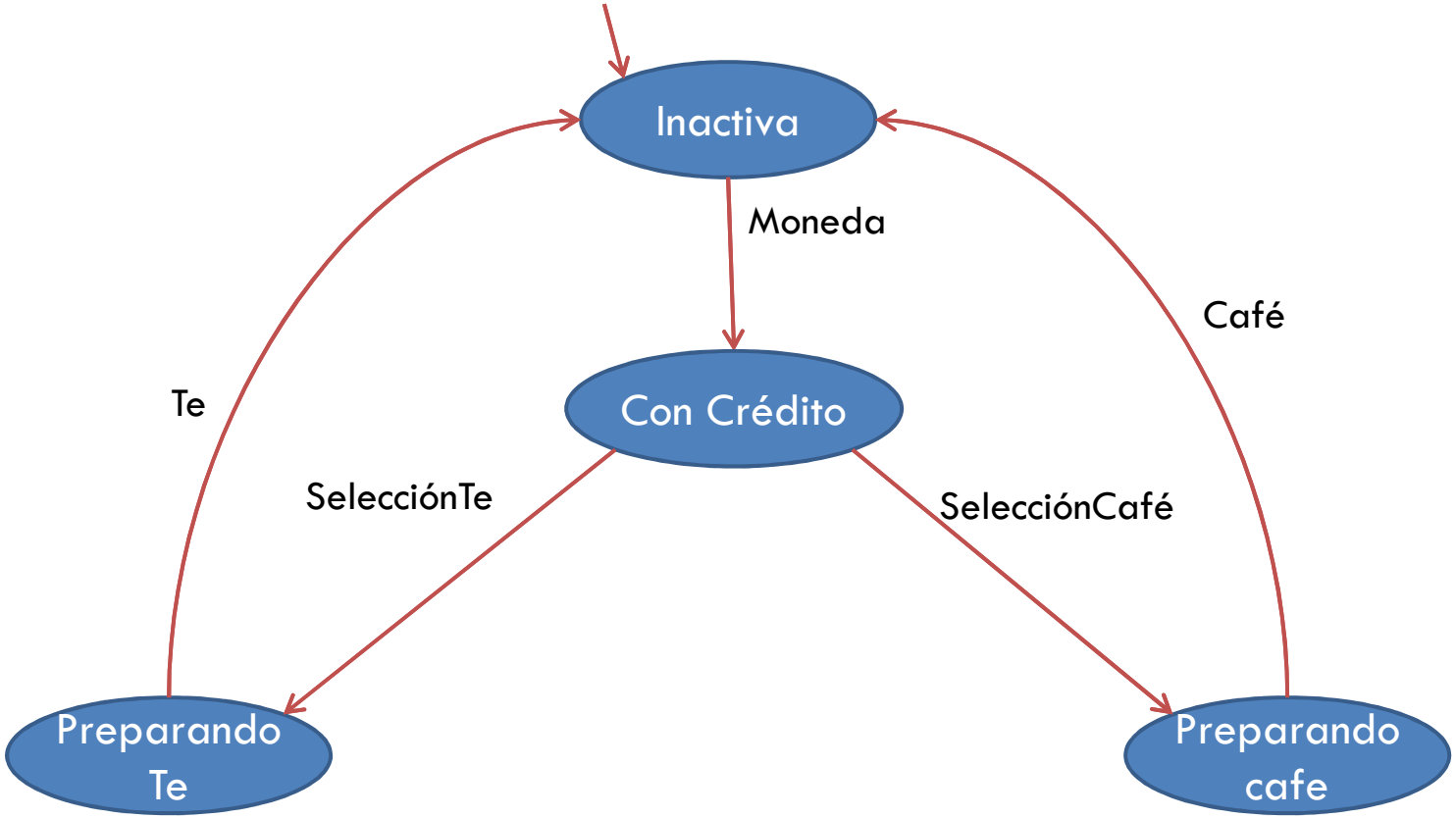


Tr(Trabada) = {∈,Ib, Ib.G, Ib.G.Rb, I.Rb.G,… }
Alc(Trabada, Ib.Rb.IB) = {Esperando}
Alc(Trabada, Ib.Rb.G) = {Trabada}

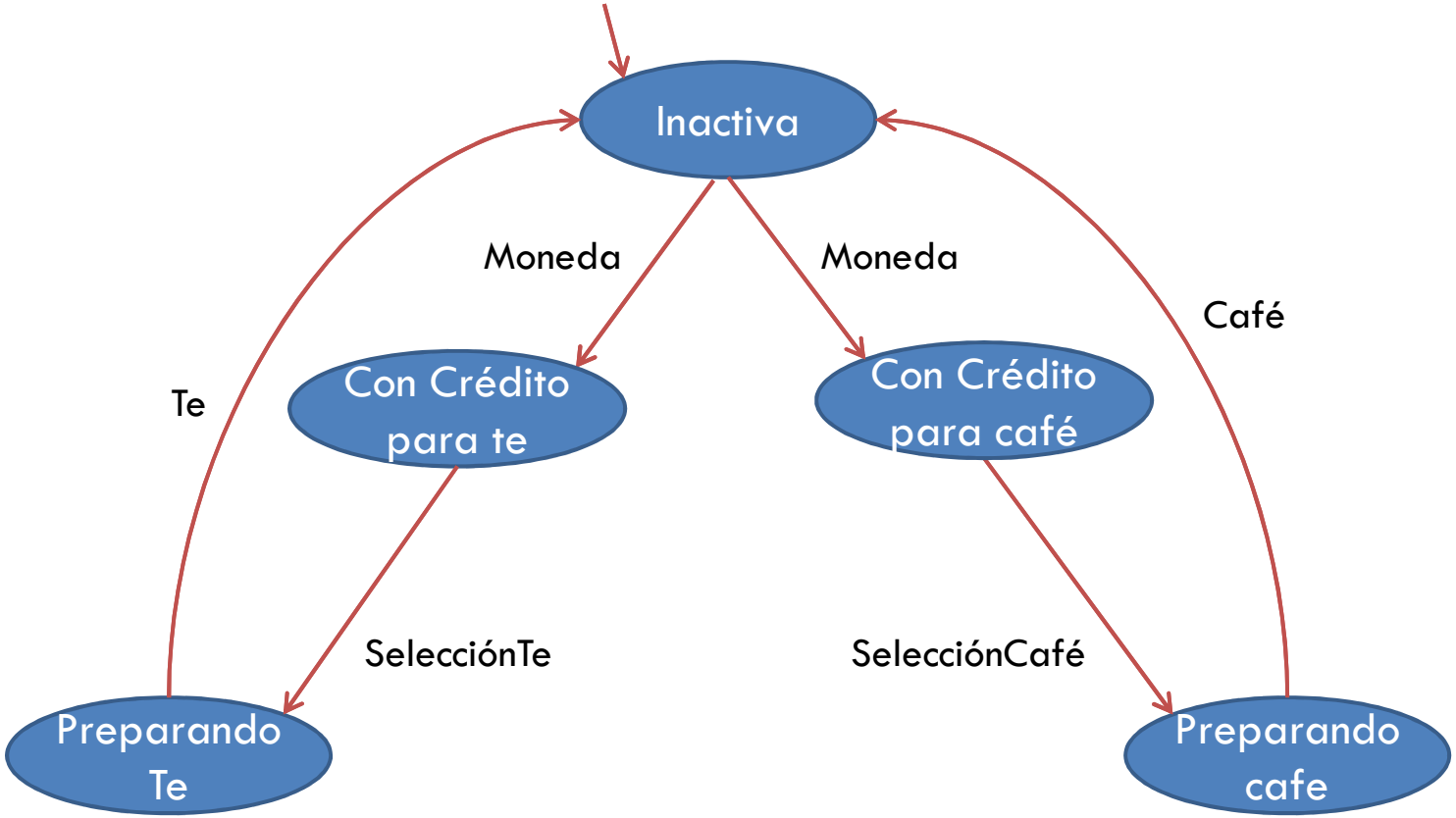
Máquina expendedora rudimentaria

- Expende café o té
- Acepta sólo monedas de un peso
- Todo cuesta un peso

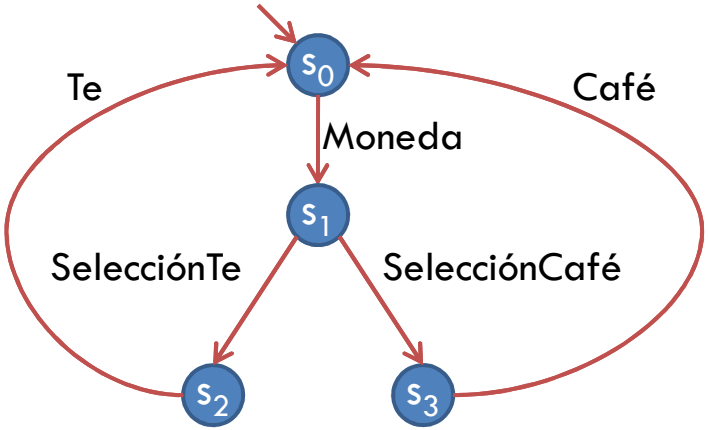
Máquina expendedora rudimentaria (I)

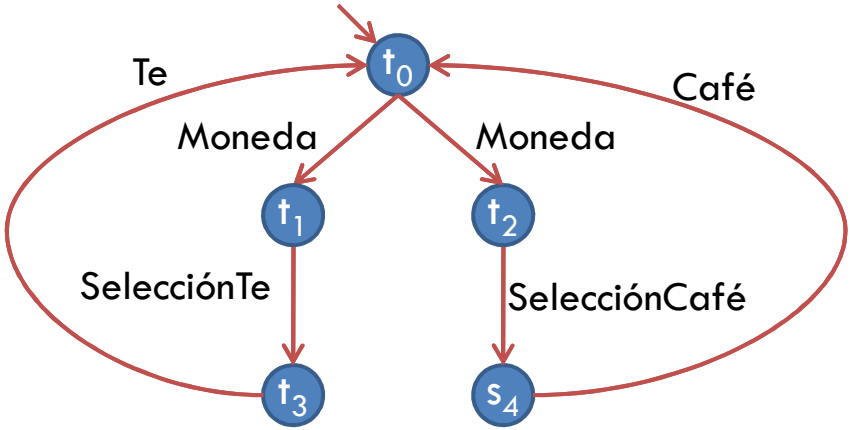


Máquina expendedora rudimentaria (II)



Trazas





$Tr(s_0) = Tr(t_1) = \{\in, M, S.St, M.Sc, M.St.t, M.Sc.C, \ldots.\}$

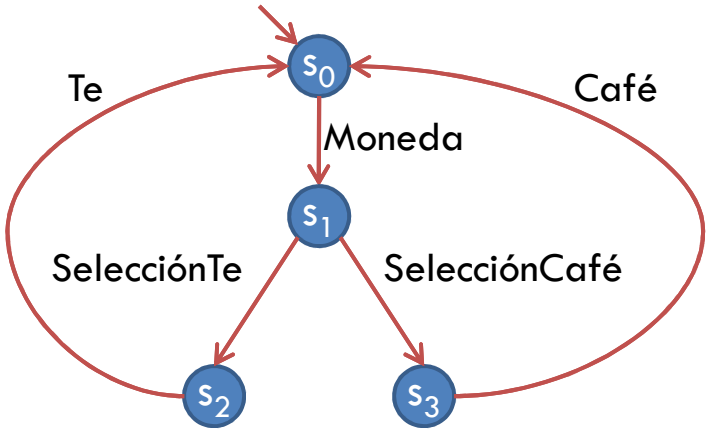
$Alc(s_0, M) = \{s_1\}$

$Alc(s_0, M) = \{t_1, t_2\}$

Nociones básicas

- L *tiene comportamiento finito* si existe un número natural n tal que para toda traza $\sigma \in Tr(L)$ se cumple que σ tiene longitud menor a n
- L es *determinístico* si para todo $\sigma \in \Sigma$, $\#Alc(s_0,\sigma) \leq 1$
  - Es no determinístico si no es determinístico

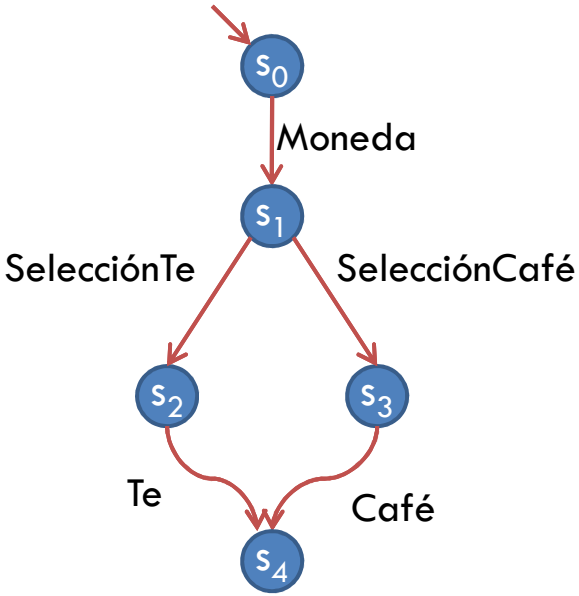
Ejemplos (1)



No Tiene comportamiento finito

Es determinístico

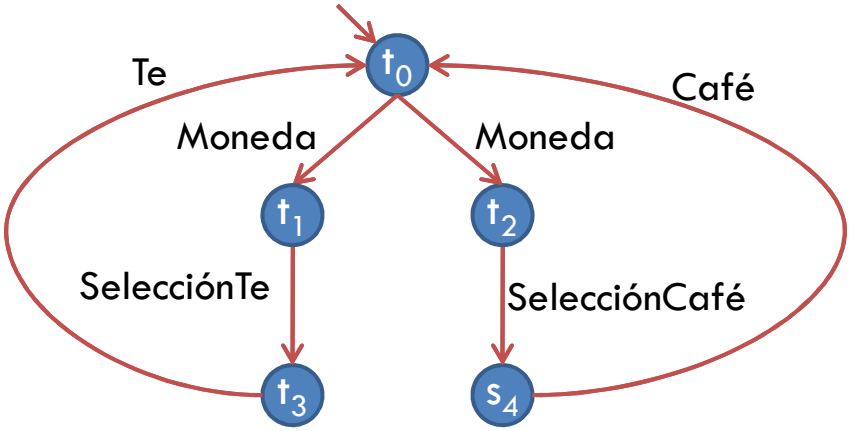
Ejemplos (2)



Tiene comportamiento finito (n = 3)

Es determinístico

Ejemplos (3)



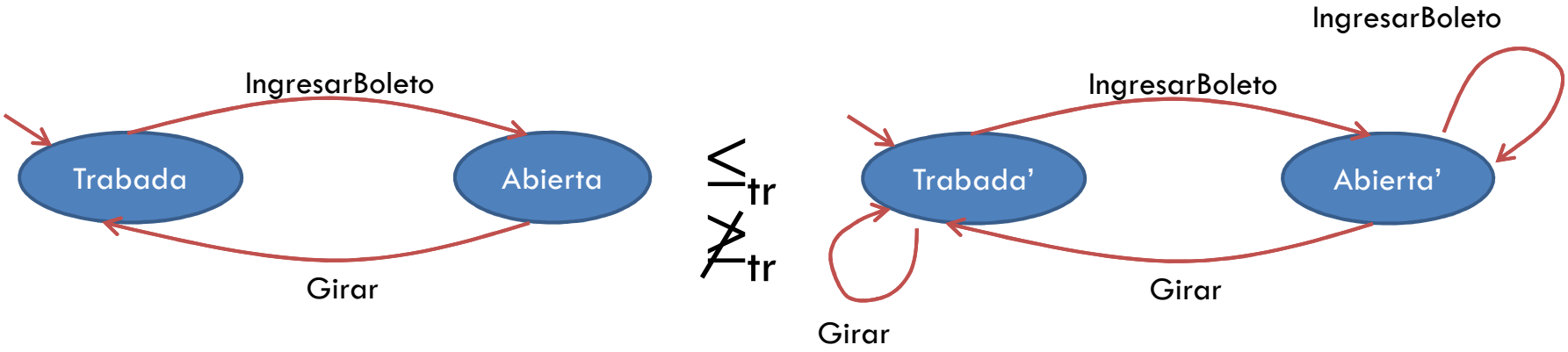
No tiene comportamiento finito

Es no determinístico

Conformance basada en trazas (1)

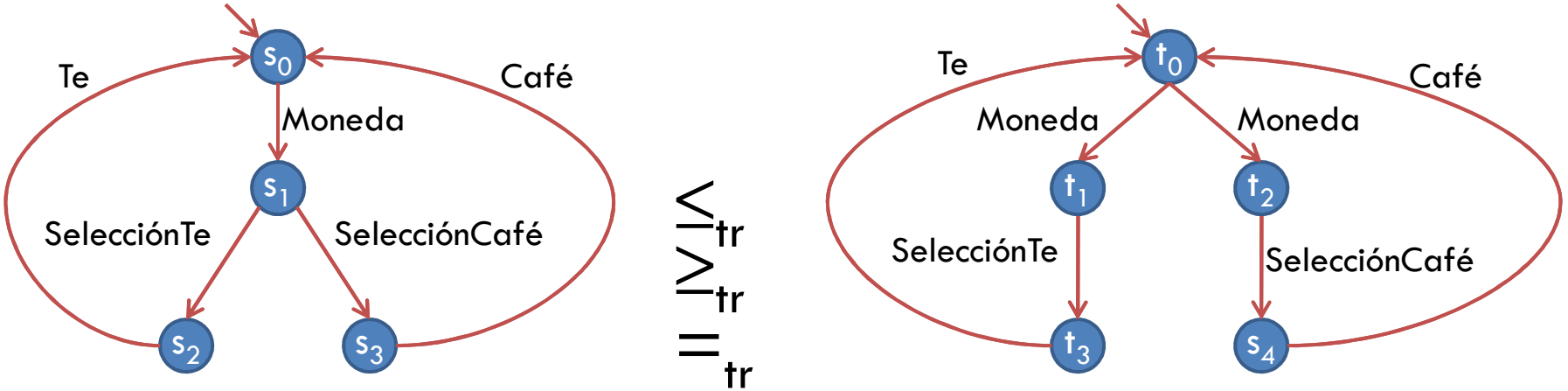
- Preorden de trazas:
  - $L \leq_{tr} L'$ sii $Tr(L) \subseteq Tr(L')$
- Equivalencia de trazas:
  - $L =_{tr} L'$ sii $Tr(L) = Tr(L')$
    - $Tr(L) \subseteq Tr(L')$ y $Tr(L') \subseteq Tr(L)$

Ejemplos (1)



Tr(Trabada) = {∈ , Ib, Ib.G, Ib.G.Ib,…} $\subseteq$ $\not\supseteq$ Tr(Trabada') = {∈ ,G, Ib, Ib.Ib, Ib.G, Ib.Ib.G, Ib.G.Ib, Ib.G.G, …}

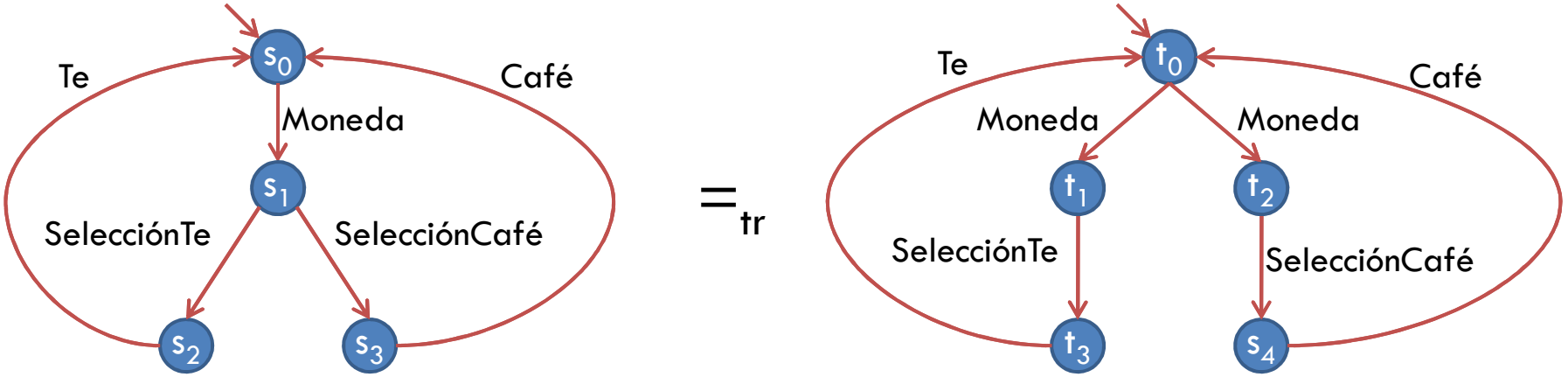
Ejemplos (2)


Te
Café
Moneda
SelecciónTe
SelecciónCafé
s_0
s_1
s_2
s_3
$\not\simeq_{tr}$
$\not\simeq_{tr}$
$\not\simeq_{tr}$
Te
Café
Moneda
Moneda
SelecciónTe
SelecciónCafé
t_0
t_1
t_2
t_3
s_4
$Tr(s_0)$
$\not\subseteq$
$\not\subseteq$
$Tr(t_0)$

Conformance basada en trazas (1)

- imp *implementa* espec $\Leftrightarrow$ imp $\leq_{tr}$ espec
  - Todo lo que hace la implementación es parte del comportamiento especificado
- imp *implementa* espec $\Leftrightarrow$ imp $\geq_{tr}$ espec
  - El comportamiento de la implementación cubre todo el comportamiento especificado
- imp *implementa* espec $\Leftrightarrow$ imp $=_{tr}$ espec
  - La implementación y la especificación tienen las mismas trazas

Trazas y no determinismo



Después de colocar una moneda (s_1), siempre es posible realizar SelecciónTe o SelecciónCafé

Después de colocar una moneda (t_1 o t_2), sólo es posible realizar una acción SelecciónTe o SelecciónCafé

Un observador más potente podría distinguir estas dos máquinas

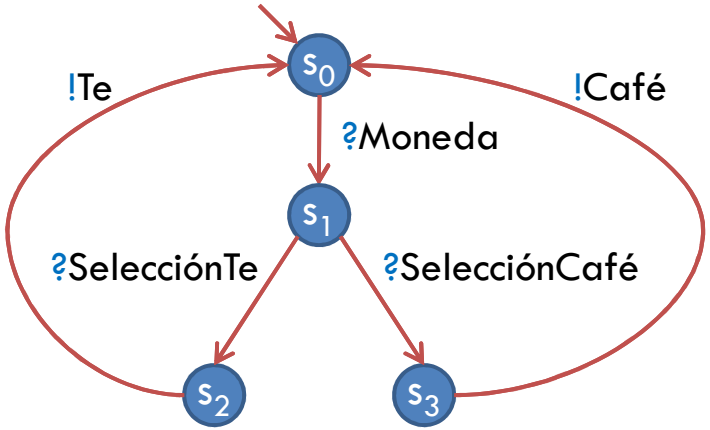
Controlabilidad vs Observabilidad

- Una acción es controlable, si puede ser usada para estimular al sistema
- Una acción es observable, si puede ser usada para comprobar la respuesta del sistema frente a un estímulo

Sistema de transición etiquetadas Input/output (IOLTS)

- Es una tupla $L = \langle S, s_0, \Sigma_I, \Sigma_O, \rightarrow \rangle$
  - Σ_I Es el conjunto de entradas
  - Σ_O Es el conjunto de salidas
  - $\Sigma_I \cap \Sigma_O = \varnothing$
  - $\langle S, s_0, \Sigma_I \cup \Sigma_O, \rightarrow \rangle$ es un LTS

Ejemplo



Σ_I = {Moneda, SelecciónTe, SelecciónCafé}

Σ_O = {Te, Café}

Nociones

- s es un *estado quiescente*, escrito $\delta(s)$, sii

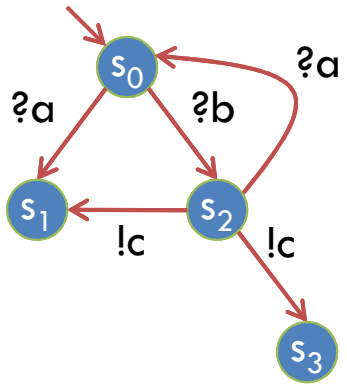
$$\forall \mu \in \Sigma_O \cup \{\tau\} : s \xrightarrow{\mu} \not$$

- σ es una *traza quiescente* de s sii

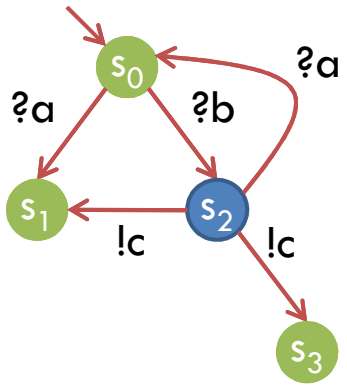
$$\exists s' \in (s \textbf{ after } \sigma): \delta(s')$$

- $QTr(s) = \{\sigma \mid \sigma$ es una *traza quiescente* de s$\}$
- $out(s) = \{\mu \mid \mu \in \Sigma_O \text{ y } s \xrightarrow{\mu}\} \cup \{\delta \mid \delta(s)\}$
- $out(S) = \cup\{out(s) \mid s \in S\}$

Ejemplos

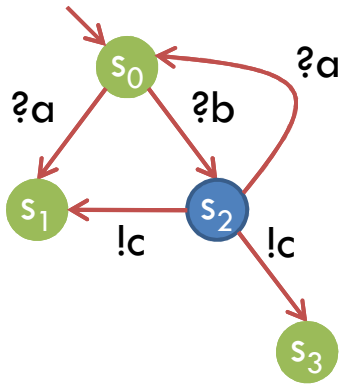


Ejemplos



$\delta(s_0)$, $\delta(s_1)$, $\delta(s_3)$

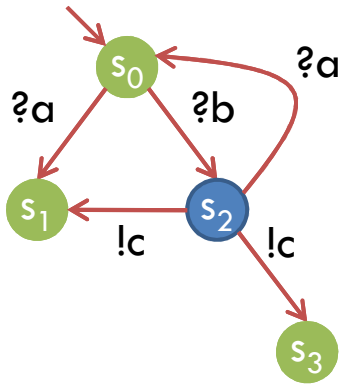
Ejemplos



$\delta(s_0)$, $\delta(s_1)$, $\delta(s_3)$

$QTr(s_0)=$

Ejemplos

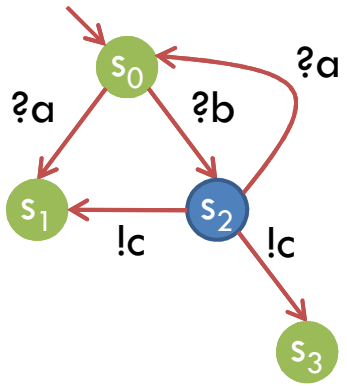


$\delta(s_0)$, $\delta(s_1)$, $\delta(s_3)$

$QTr(s_0)=\{\in, a, bc, ba, baa, babc, baba, babaa, \ldots\}$

$QTr(s_1)=$

Ejemplos



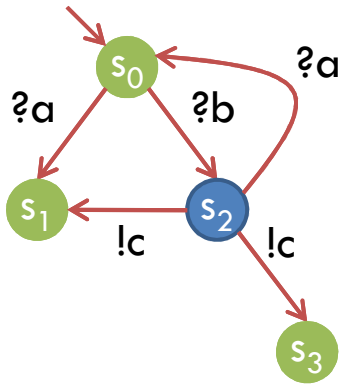
$\delta(s_0)$, $\delta(s_1)$, $\delta(s_3)$

$QTr(s_0)=\{\in, a, bc, ba, baa, babc, baba, babaa, \ldots\}$

$QTr(s_1)= QTr(s_3)=\{\in\}$

$QTr(s_2)=$

Ejemplos



$\delta(s_0)$, $\delta(s_1)$, $\delta(s_3)$

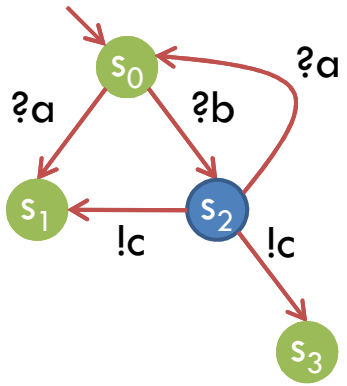
$QTr(s_0)=\{\in, a, bc, ba, baa, babc, baba, babaa, \ldots\}$

$QTr(s_1)= QTr(s_3)=\{\in\}$

$QTr(s_2)=\{c,a,aa,abc,aba,abaa,ababc,\ldots\}$

$out(s_0)=$

Ejemplos



$\delta(s_0)$, $\delta(s_1)$, $\delta(s_3)$

$QTr(s_0)=\{\in, a, bc, ba, baa, babc, baba, babaa, \ldots\}$

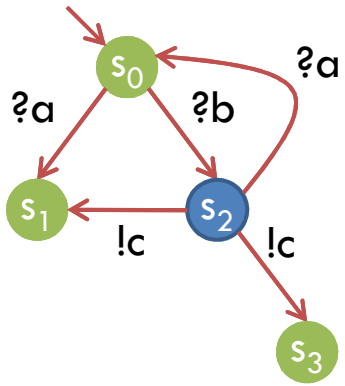
$QTr(s_1)= QTr(s_3)=\{\in\}$

$QTr(s_2)=\{c,a,aa,abc,aba,abaa,ababc,\ldots\}$

$out(s_0)= out(s_1)= out(s_3)= \{\delta\}$

$out(s_2)=$

Ejemplos



$\delta(s_0)$, $\delta(s_1)$, $\delta(s_3)$

$QTr(s_0)=\{\in, a, bc, ba, baa, babc, baba, babaa, \ldots\}$

$QTr(s_1)= QTr(s_3)=\{\in\}$

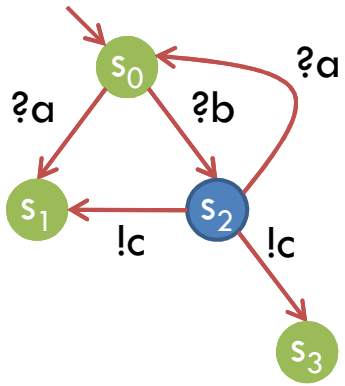
$QTr(s_2)=\{c,a,aa,abc,aba,abaa,ababc,\ldots\}$

$out(s_0)= out(s_1)= out(s_3)= \{\delta\}$

$out(s_2)= \{c\}$

$out(\{s_0,s_1,s_2)=$

Ejemplos



$\delta(s_0)$, $\delta(s_1)$, $\delta(s_3)$

$QTr(s_0)=\{\in, a, bc, ba, baa, babc, baba, babaa, \ldots\}$

$QTr(s_1)= QTr(s_3)=\{\in\}$

$QTr(s_2)=\{c,a,aa,abc,aba,abaa,ababc,\ldots\}$

$out(s_0)= out(s_1)= out(s_3)= \{\delta\}$

$out(s_2)= \{c\}$

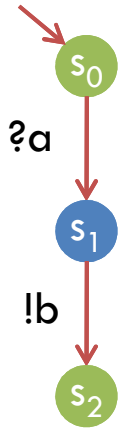
$out(\{s_0,s_1,s_2)= \{\delta, c\}$

Conformance en término de trazas

- $L \leq_{iotr} L'$ sii $Tr(L) \subseteq Tr(L')$ y $QTr(L) \subseteq QTr(L')$
- Alternativamente

  $L \leq_{iotr} L'$ sii $\forall \sigma$: $out(Alc(L,\sigma)) \subseteq out(Alc(L',\sigma))$

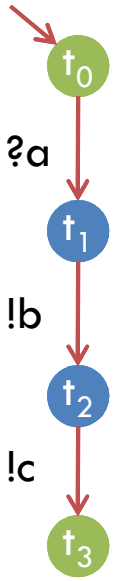
Ejemplo



$\leq_{tr}$

$\not\geq_{tr}$

$\not\leq_{iotr}$



$Tr(s_0)=\{\in, a, ab\}$

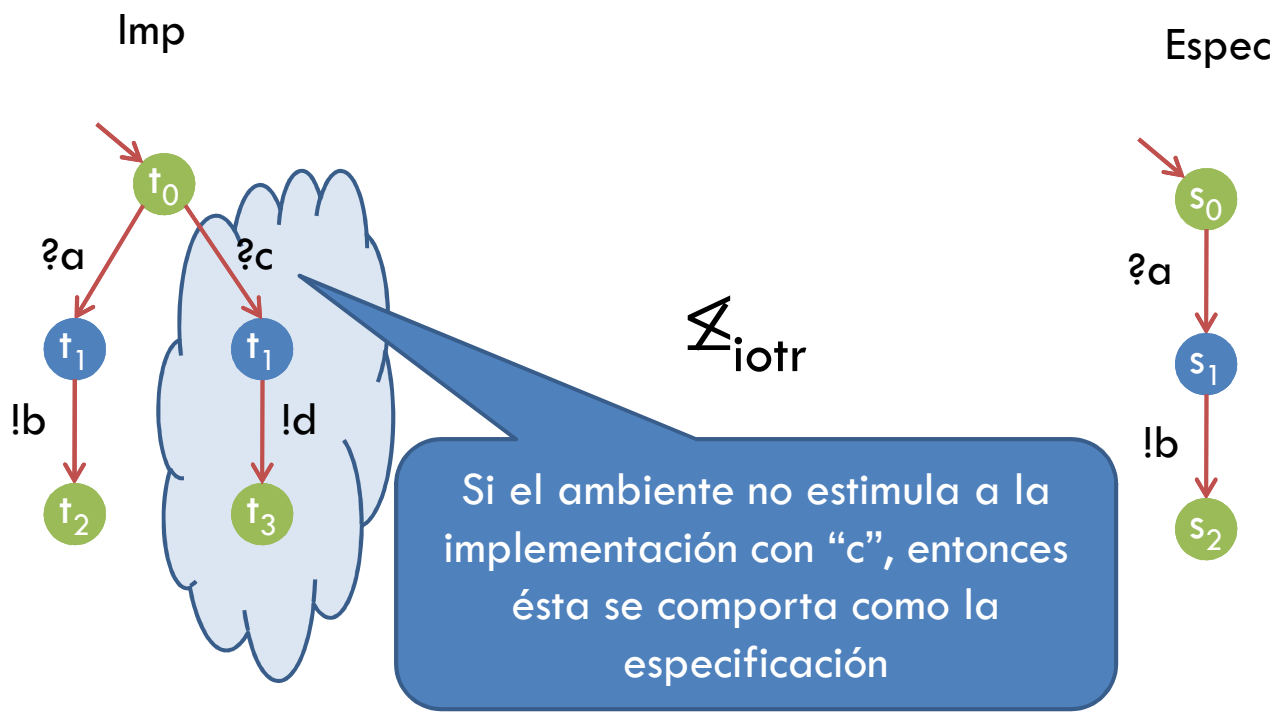
$QTr(s_0)=\{\in, ab\}$

$Tr(t_0)=\{\in, a, ab, abc\}$

$QTr(t_0)=\{\in, abc\}$

Limitaciones de $\leq_{iotr}$

- Una implementación podría proveer más operaciones que su especificación

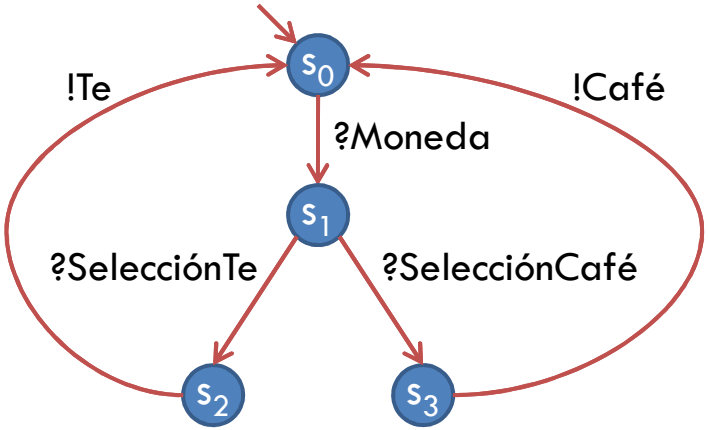


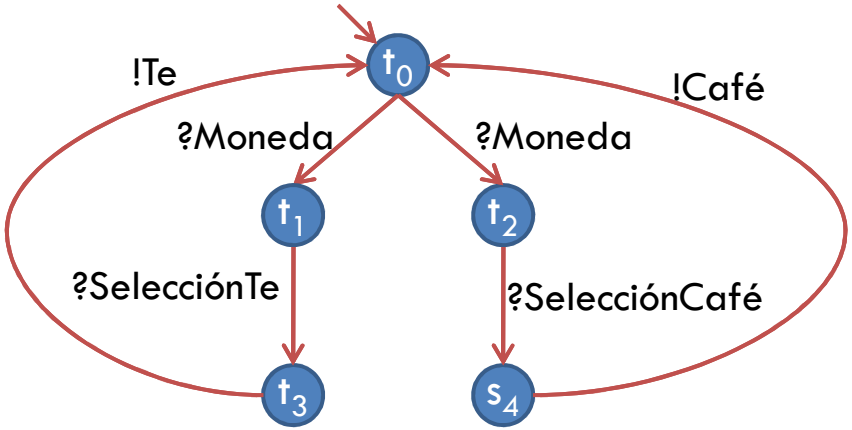
Conformance en término de trazas

- L **ioconf** L' sii

  $\forall \sigma \in Tr(L')$: $out(Alc(L,\sigma)) \subseteq out(Alc(L',\sigma))$

Limitaciones de ioconf





$$\forall \sigma : out(Alc(s_0, \sigma)) = out(Alc(t_0, \sigma))$$

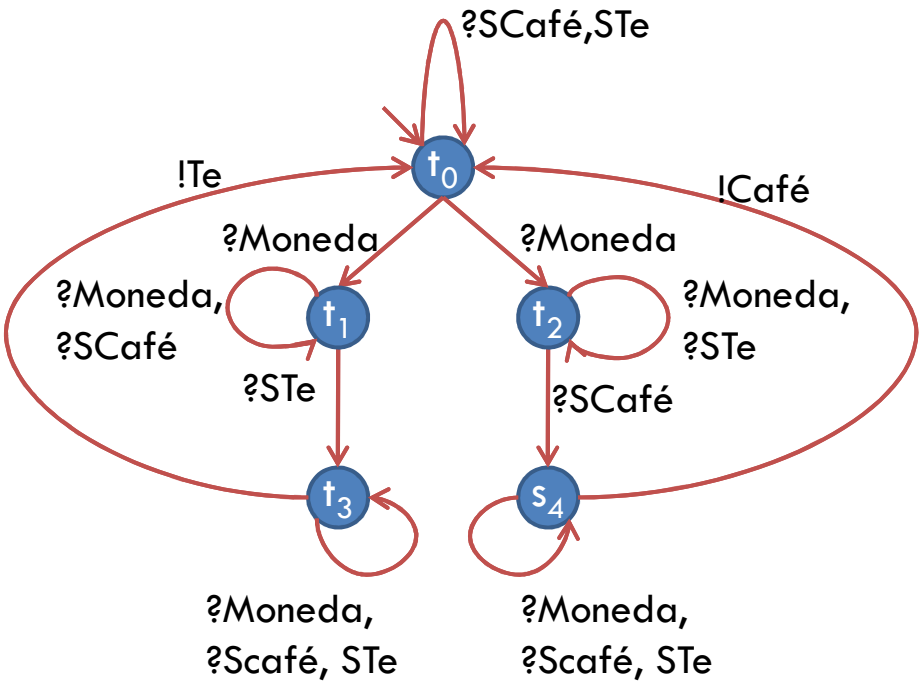
s_0 ioconf t_0 y t_0 ioconf s_0

Hipótesis de testing más fuerte

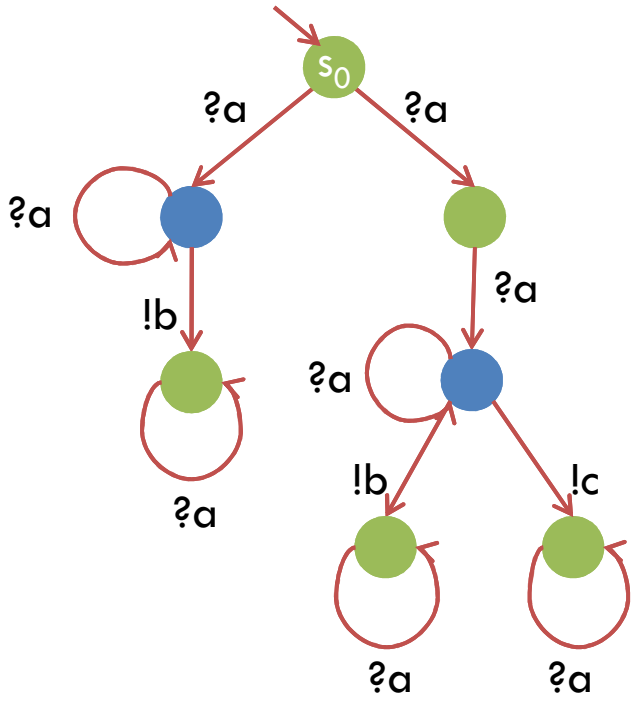
- La implementación puede modelarse como un IOLTS con input siempre habilitadas
- L es un IOLTS con input siempre habilitadas sii

$$\forall s : s_0 \xrightarrow{\sigma} s, \forall \mu \in \Sigma_I : Alc(s, \mu) \neq \emptyset$$

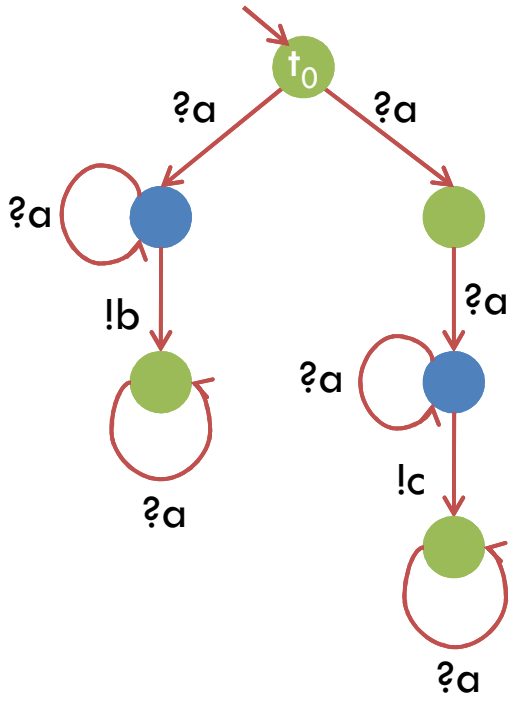
Ejemplo


?SCafé,STe
!Te
!Café
?Moneda
?Moneda
?Moneda,
?SCafé
?Moneda,
?STe
?STe
?SCafé
t0
t1
t2
t3
s4
?Moneda,
?Scafé, STe
?Moneda,
?Scafé, STe

Todavía hay un problema…



s_0 ioconf t_0



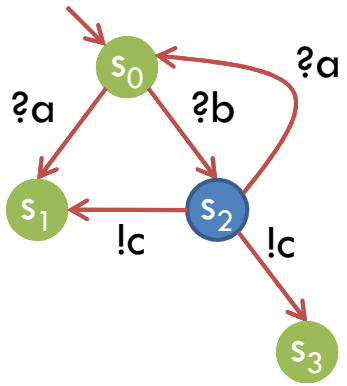
Podría distinguirlos

1. Estimulo con a
2. Espero ver que no hay respuesta.
3. Estimulo con a
4. Puedo observar b o c

1. Estimulo con a
2. Espero ver que no hay respuesta.
3. Estimulo con a
4. Puedo observar sólo c

Trazas con suspensión

- Las trazas incluyen δ para marcar explícitamente cuando alcanzan un estado quiescente



$Tr(s_0)=\{\in, a, b, bc, ba, baa, bab, bab\ldots\}$

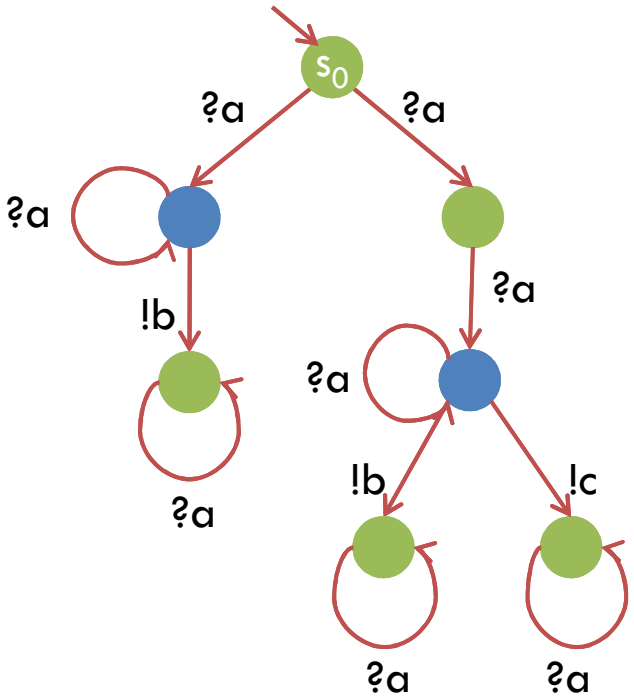
$STr(s_0)=\{\in, \delta, a, a\delta, b, bc, bc\delta, ba, ba\delta, b\delta a\delta, \ldots\}$

ioco

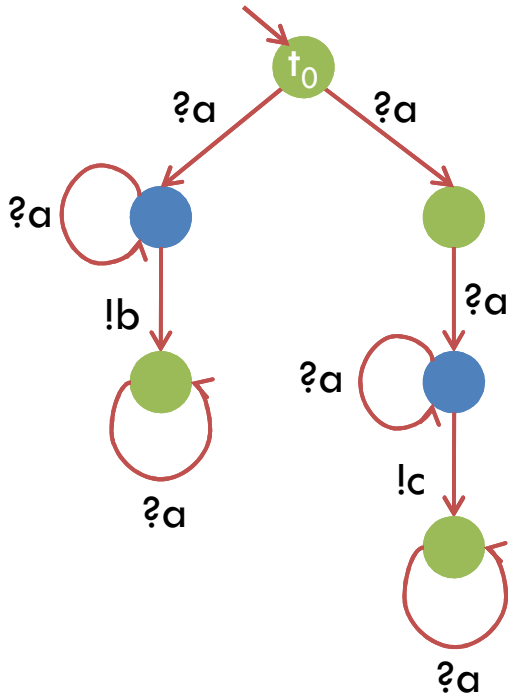
- L **ioco** L' sii

  $\forall \sigma \in STr(L')$: $out(Alc(L,\sigma)) \subseteq out(Alc(L',\sigma))$

Ejemplo



s_0 $io \not\leq o$ t_0

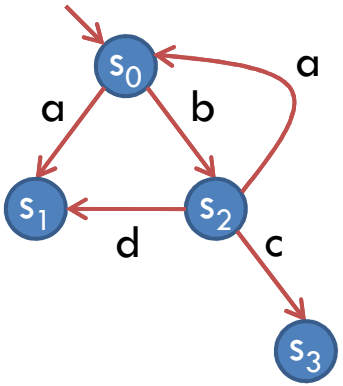


out(Alc(s0, aδa)) = {b,c}

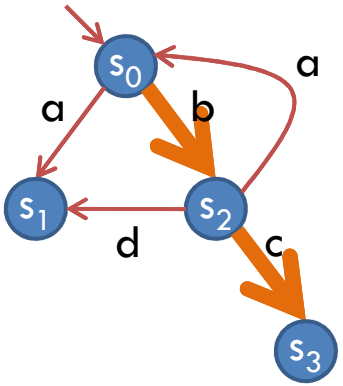
out(Alc(s0, aδa)) = {c}

GENERACIÓN DE CASOS DE TEST

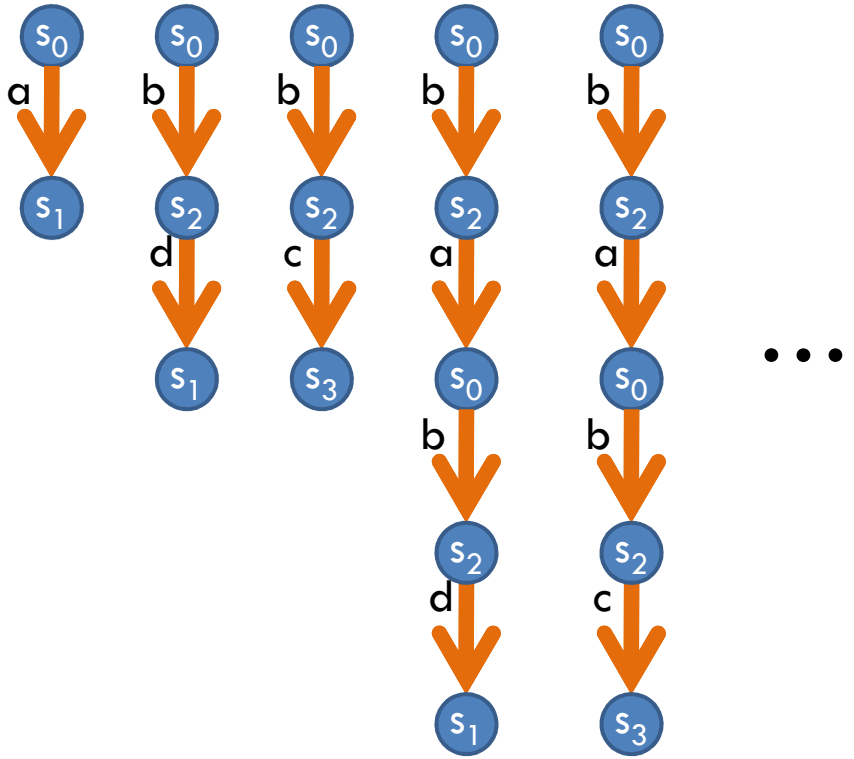
¿Cuál es la noción de caso?



¿Cuál es la noción de caso?



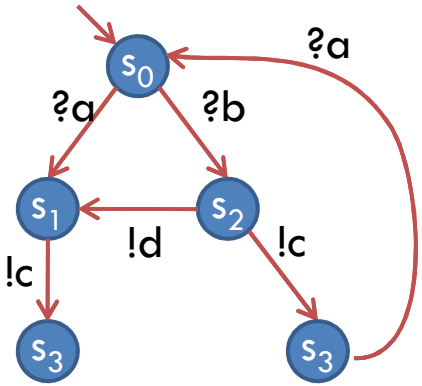
La descripción de una traza

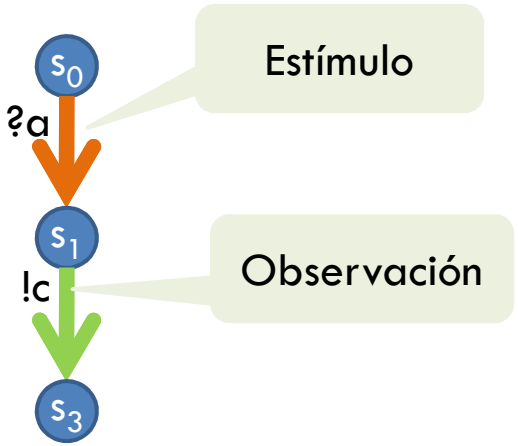


¿Cuál es la noción de caso?

Sistemas I/O

No deterministicos

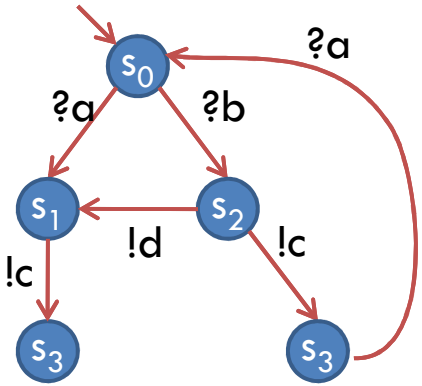


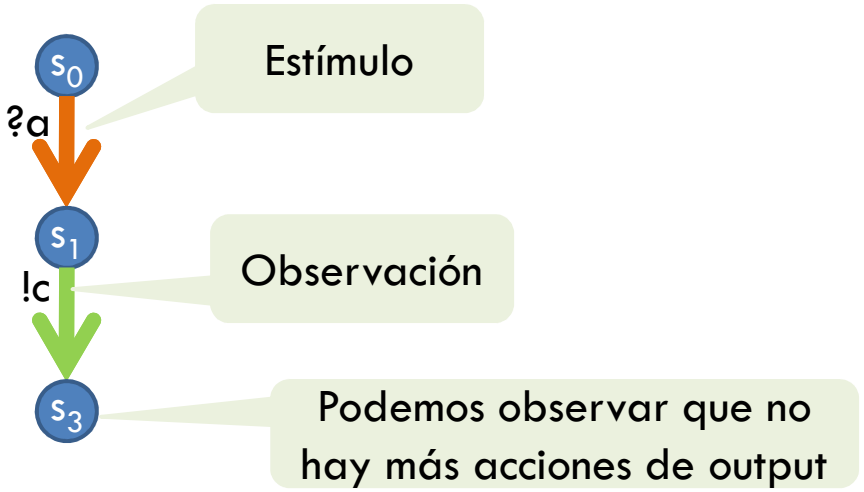


¿Cuál es la noción de caso?

Sistemas I/O

No deterministicos

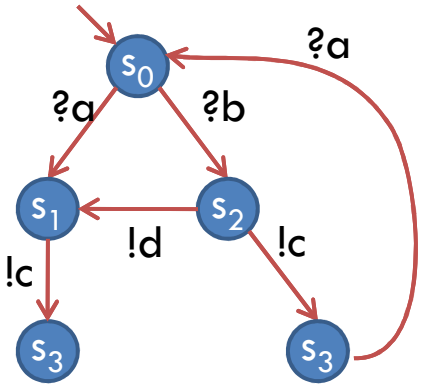


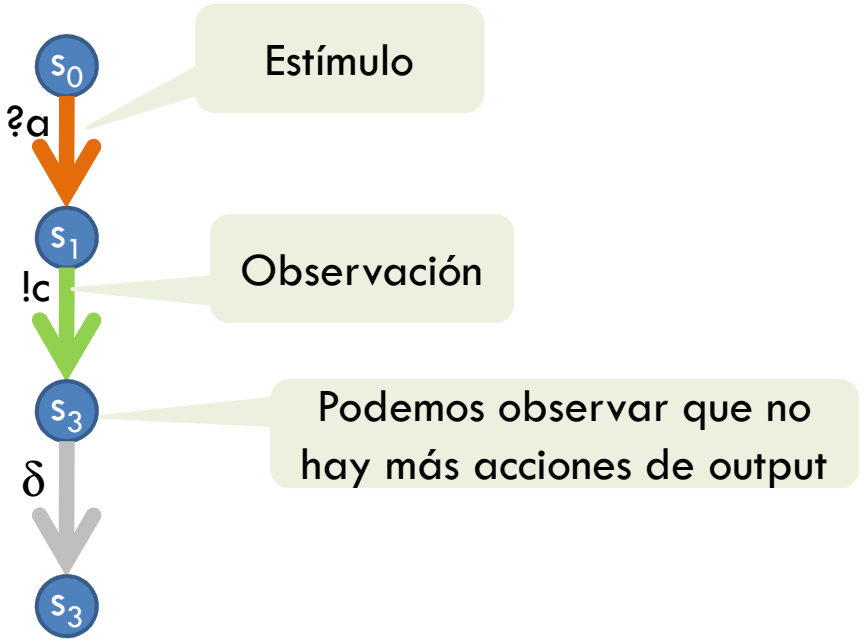


¿Cuál es la noción de caso?

Sistemas I/O

No deterministicos

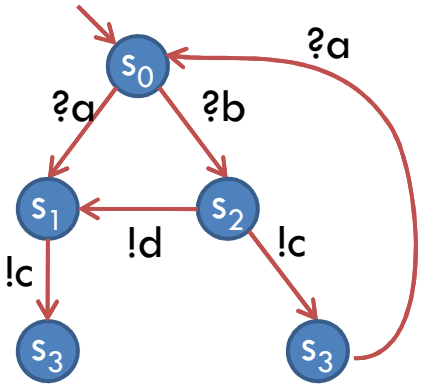


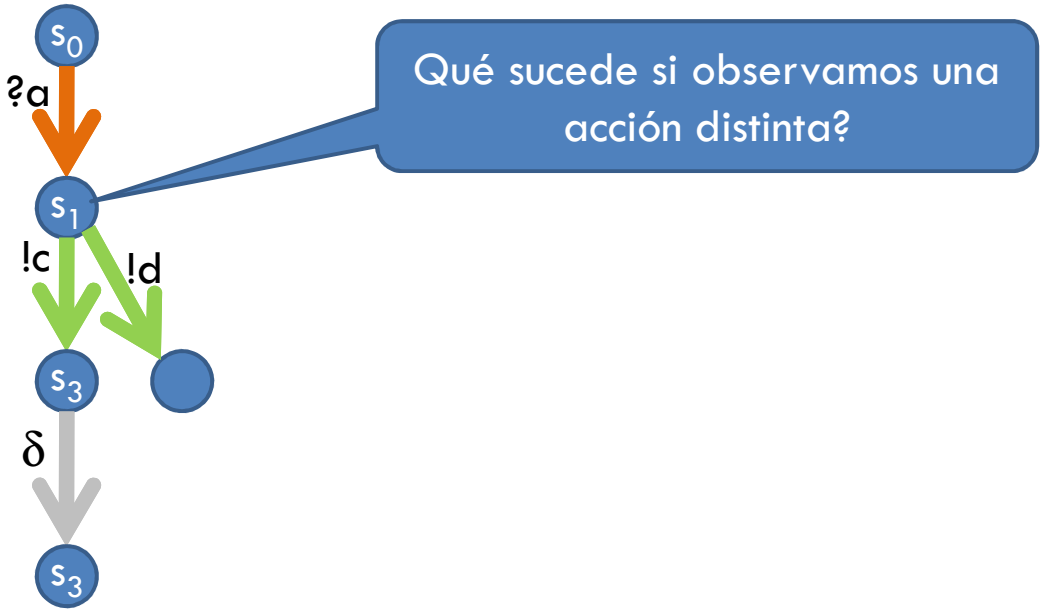


¿Cuál es la noción de caso?

Sistemas I/O

No deterministicos

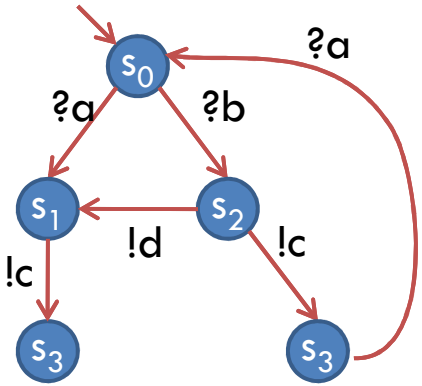




¿Cuál es la noción de caso?

Sistemas I/O

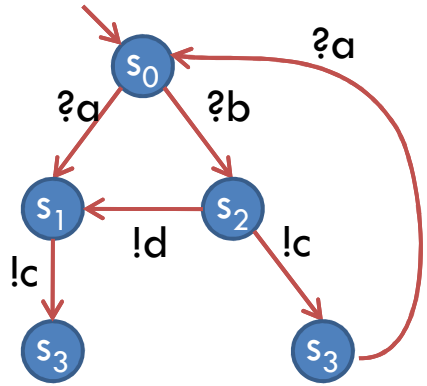
No determinísticos

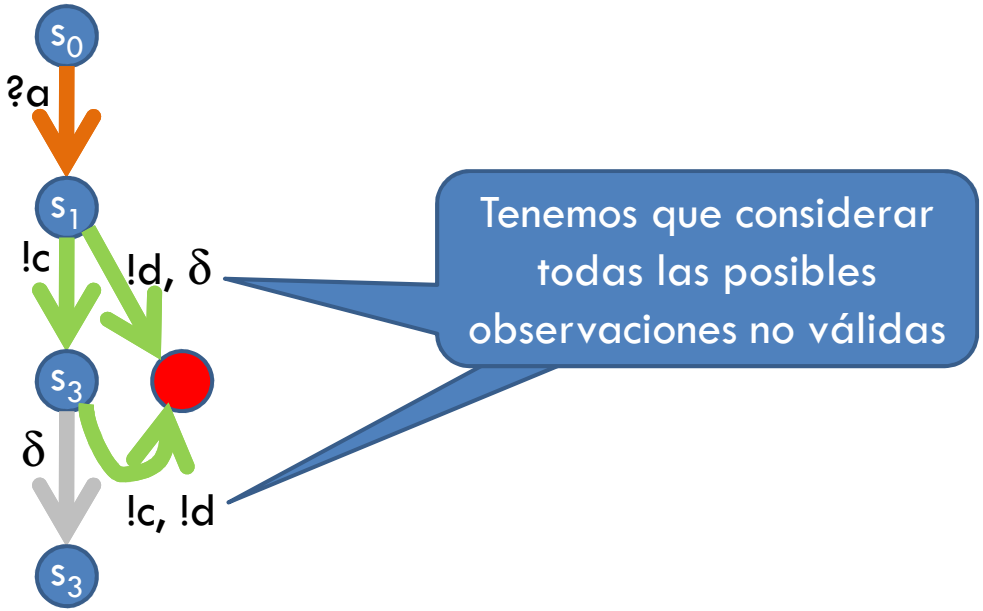


¿Cuál es la noción de caso?

Sistemas I/O

No deterministicos

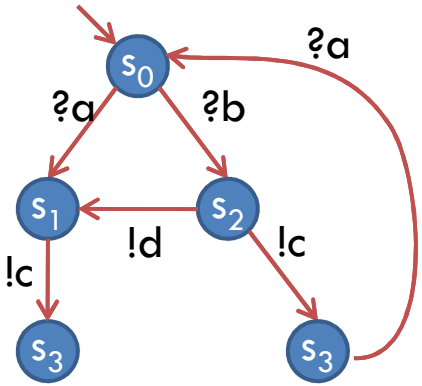


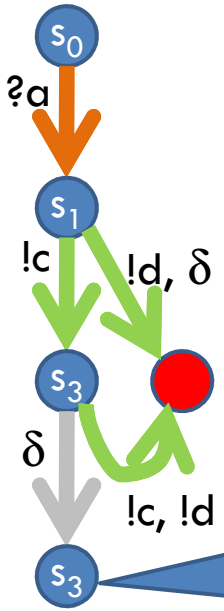


¿Cuál es la noción de caso?

Sistemas I/O

No deterministicos



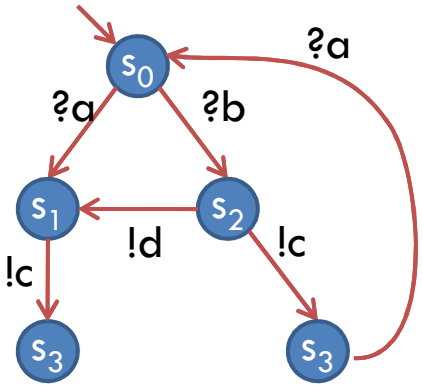


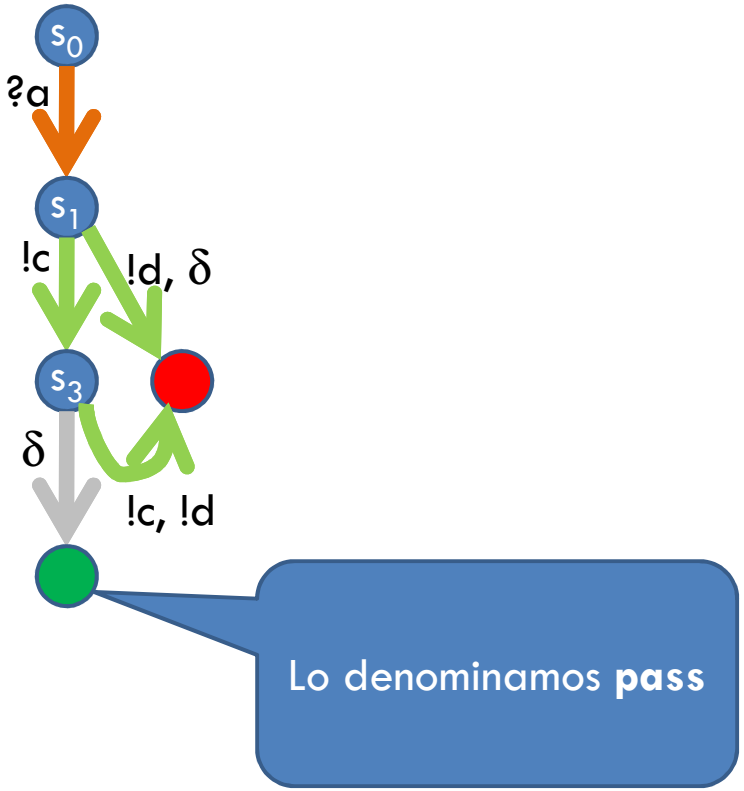
Este estado representa la finalización de la ejecución del caso de test donde se observa el resultado esperado

¿Cuál es la noción de caso?

Sistemas I/O

No deterministicos

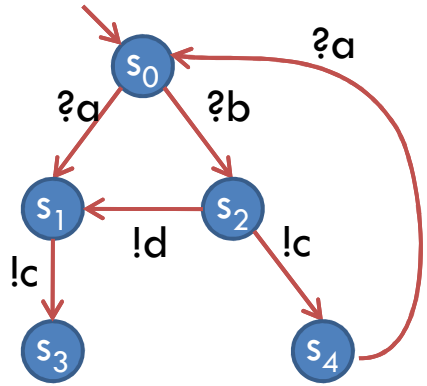


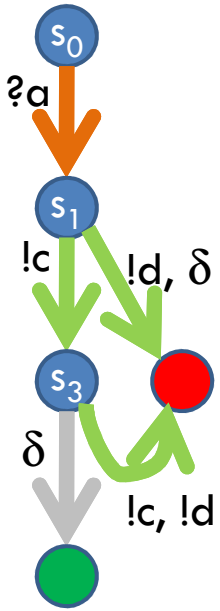


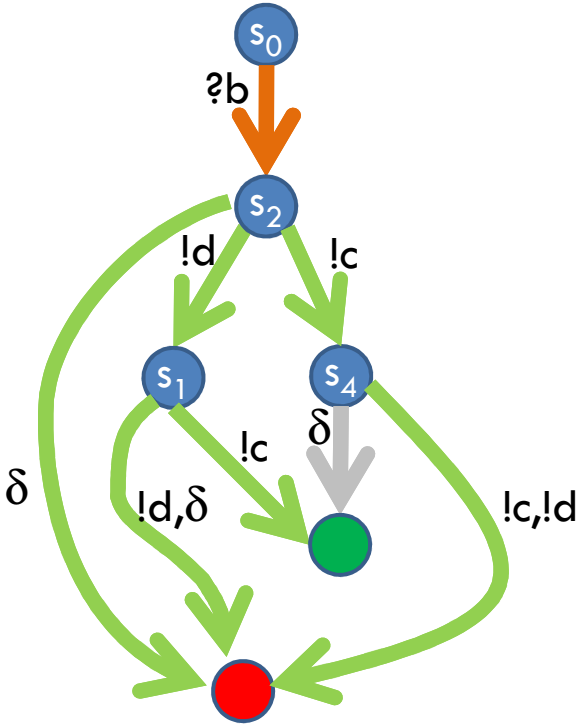
¿Cuál es la noción de caso?

Sistemas I/O

No deterministicos



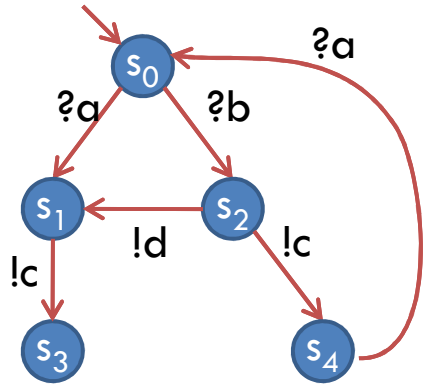


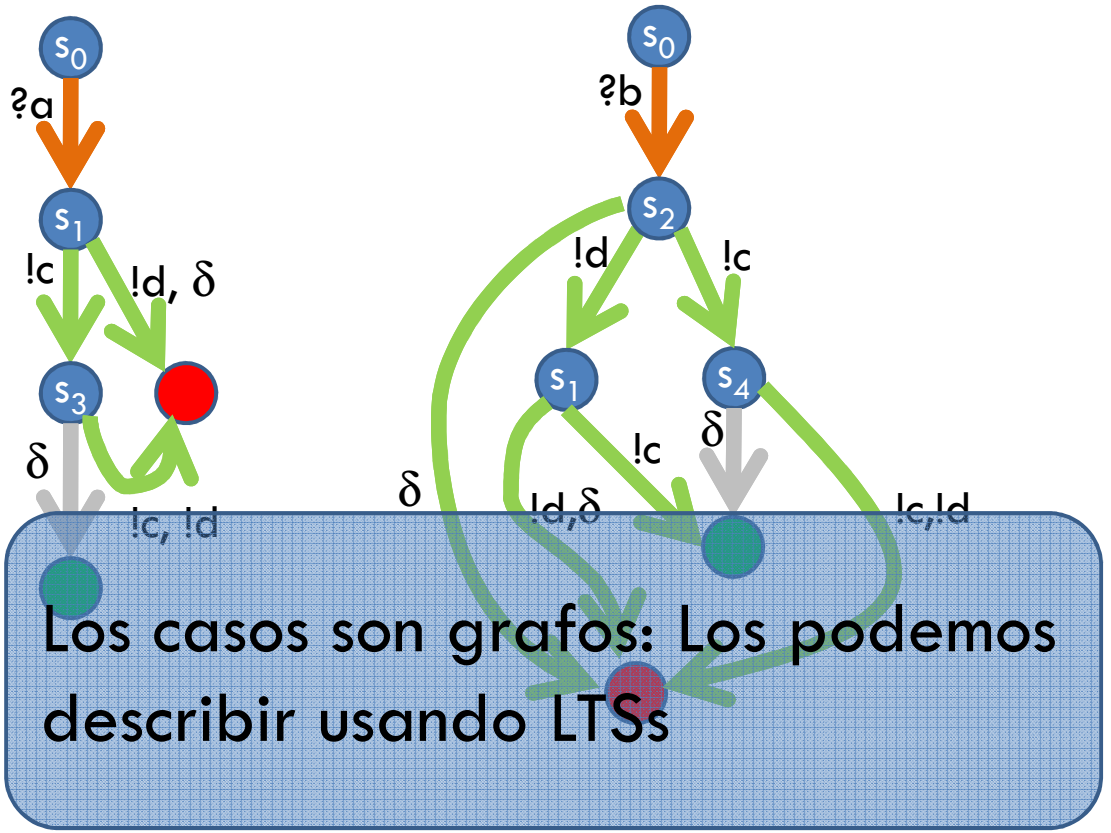


¿Cuál es la noción de caso?

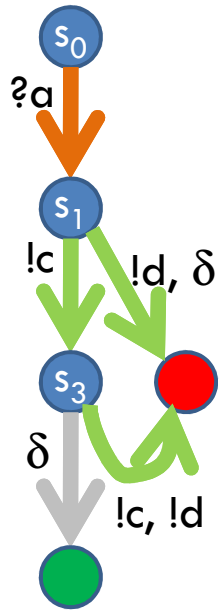
Sistemas I/O

No deterministicos



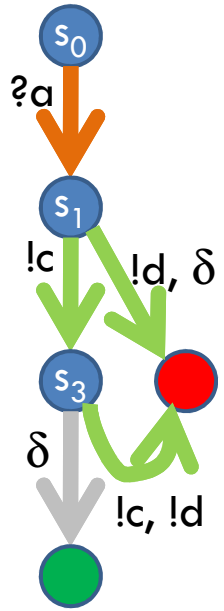


¿Cuál es la noción de caso?



- Un caso de test es un LTS
  $t = \langle S, \Sigma_I \cup \Sigma_O \cup \{\delta\}, T, s_0 \rangle$ tal que
  - t es finito
  - $\{$**pass, fail**$\} \subseteq S$ y init(**pass**)=init(**fail**)= $\varnothing$
  - Para cada $s \in S$ $\{$**pass, fail**$\}$ se cumple:
    - $init(s) = \{a\}$ con $a \in \Sigma_I$; o
    - $init(s) = \Sigma_O \cup \{\delta\}$

Interpretación del caso de test



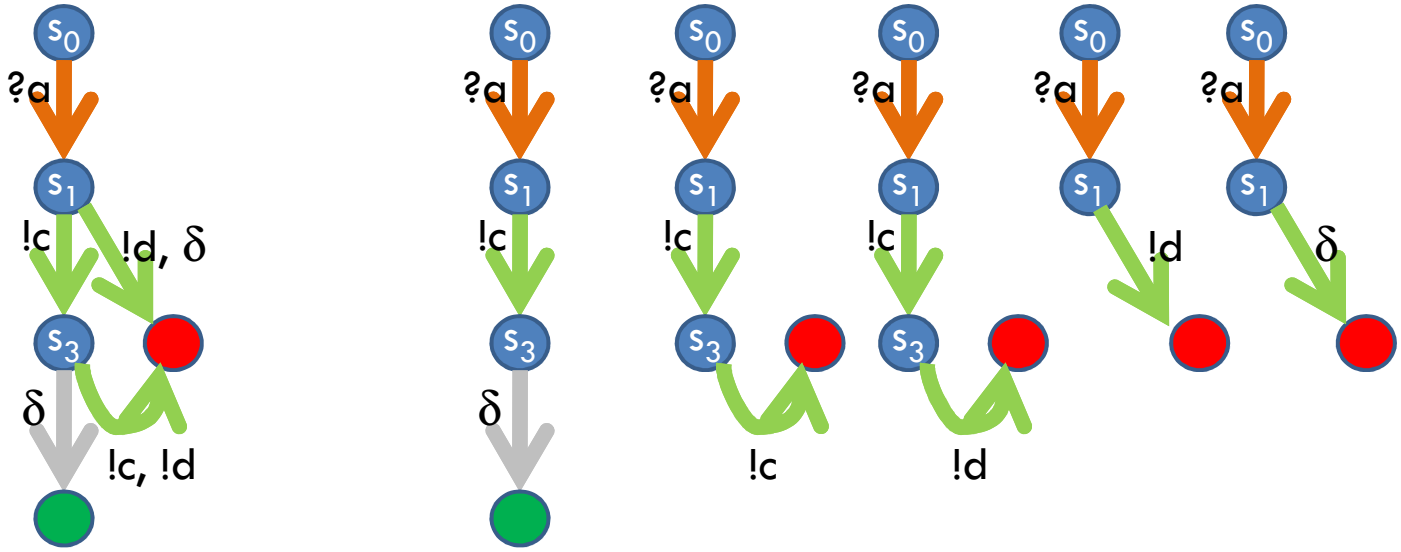
- Estimular al SUT con **a**
  - Invocar el método **a**, enviar el mensaje **a**
- Esperar respuesta
  - el valor de retorno, recibir un mensaje
- Si el valor de retorno no es c (o no se recibe respuesta), la ejecución del caso **falla**
- Si el valor es c, observar que no existe otra salida del SUT
  - Útil cuando hay comunicación asincrónica
- Si no hay salidas adicionales, entonces la ejecución **pasa**, en caso contrario **falla**

¿Cuándo un test pasa o falla?

- La *ejecución de un caso* de test contra una implementación:
  - **Pasa** cuando la ejecución conduce al estado "pass" del caso de test
  - **Falla** cuando la ejecución alcanza el estado "fail" del caso de test
- Un caso de test:
  - **Pasa** si toda ejecución pasa
  - **Falla** si alguna de sus ejecuciones falla

¿Cuántas ejecuciones de un caso son necesarias?

- En general varias



- Basta una ejecución que falla para afirmar que el caso falla
- En teoría, necesitamos infinitas ejecuciones para asegurar que el caso pasa
  - En la práctica nos conformamos con una o algunas.

Propiedades de una test suite T

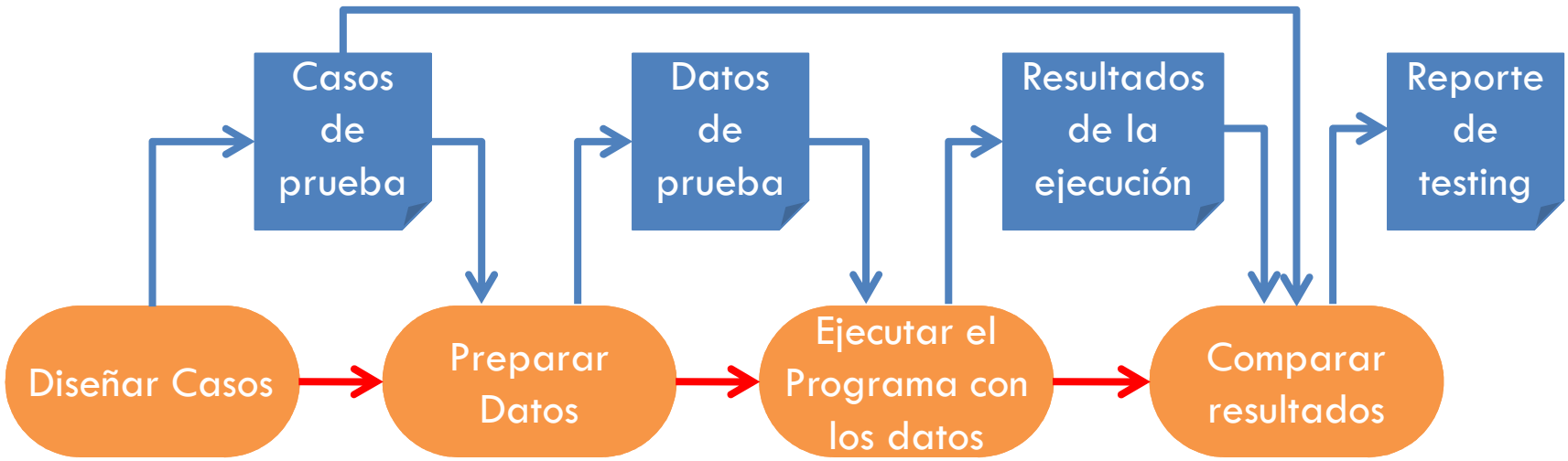
- T es consistente respecto de e cuando
  $\forall i$: i ioco e $\Rightarrow$ i pasa T
- T es exhaustiva respecto de e cuando
  $\forall i$: i pasa T $\Rightarrow$ i ioco e
- T es completa cuando es consistente y exhaustiva
- En la práctica esperamos construir test suite consistente
  - Hacerlas exhaustiva podría requerir infinitos casos

Generación de casos

- Randon-walks
- Chinese postman

MODEL BASED TESTING

Proceso de testing



Objetivo

- Automatizar el diseño del test

Objetivo

- Automatizar el diseño del test

Ejemplo: Qui-donc

- Servicio telefónico que dado un número de teléfono responde el nombre y la dirección de la persona
- Caso de Uso típico

Usuario	Sistema
Discar número de servicio 0800…	Responde “Hola. Marque ‘*’ para comenzar
Marcar ‘*’	Responder “Ingrese el número seguido de ‘#’”
Ingresar “011 ….”	Responder “Hernán …. Marque 1 para la dirección /2 para …..”
Marcar ‘1’	Responder “Agustin Delgado …”
Cortar	

¿Por qué modelos?

- Reducir costo de diseño de test:
  - Tiempo de modelado < Tiempo de diseño manual
  - Definición automática de oráculos
- Sistematizar el testing
  - Disminuye la subjetividad
  - Controlar la cobertura del modelo y el número de test
- Detección tempranas de problemas en la especificación
- Mejor desempeño cuando los requerimientos cambian
  - Cambiar el modelo y regenerar test
- Mejora la trazabilidad Requerimientos/Casos

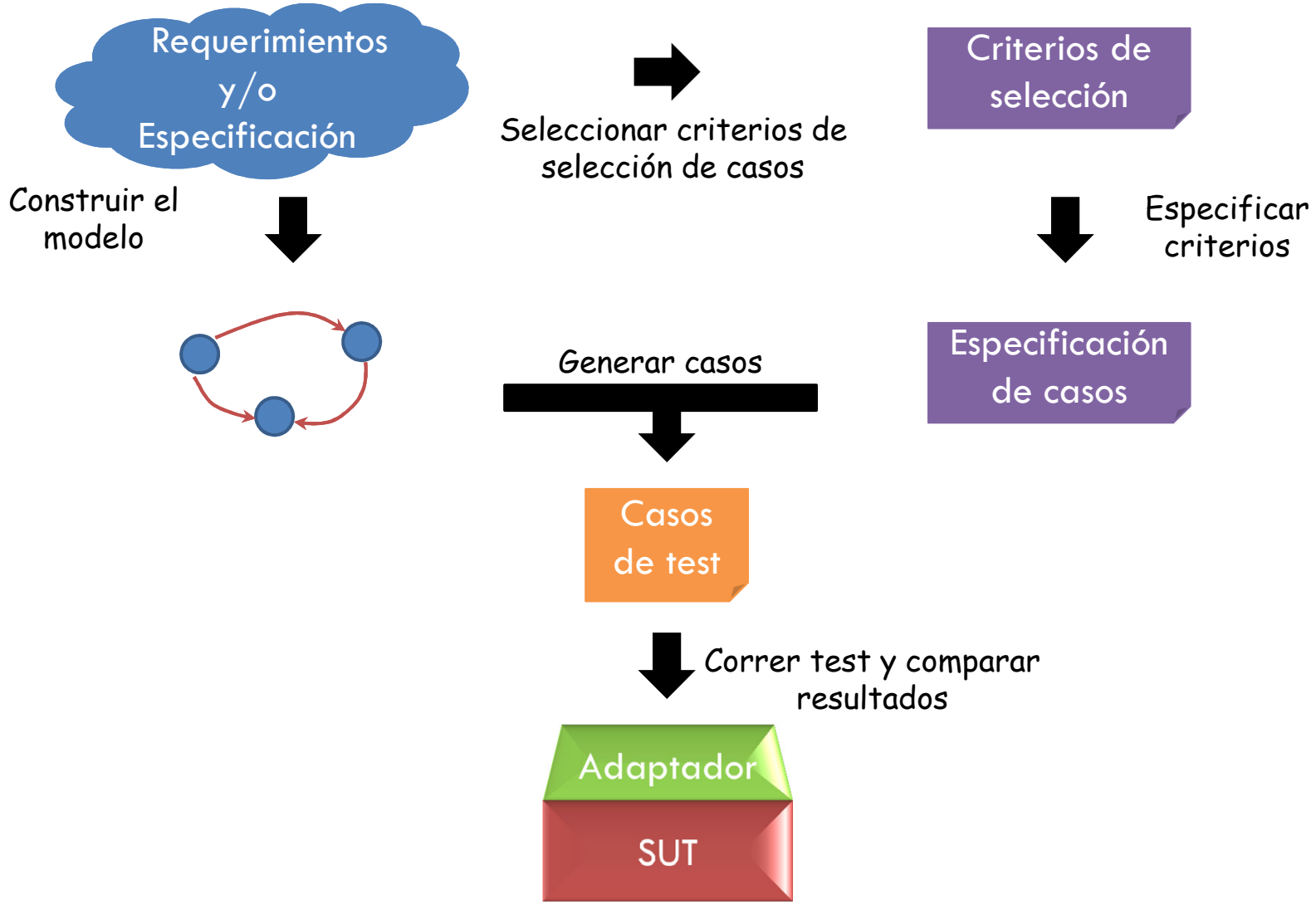
Model-based testing

- La derivación automatizable de casos de test concretos a partir de modelos formales abstractos

Model-based testing

- Es una variante del testing que se vale de modelos que codifican el comportamiento esperado del sistema bajo test (SUT) y/o del ambiente para derivar casos de test [Utting, Pretschner, Legeard]
- Pares de entradas y salidas del modelo de la implementación se interpretan como casos de test para la implementación
  - Las salidas del modelo son los resultados esperados (el oráculo) del SUT

Actividades



Construcción del modelo

- El modelo del SUT se construye a partir de los requerimientos y/o la especificación existente
- Debe codificar el comportamiento esperado
- Puede abstraer
  - Excepciones o fallas
  - Funcionalidades
  - Requerimientos no funcionales
    - Restricciones temporales
    - seguridad

Abstracción en el modelo (1)

- Funcionalidad: omitir funcionalidades del SUT
  - Omitir funcionalidades no críticas
- Datos:
  - Entradas: el modelo omite o simplifica algunas entradas de una operación del SUT
    - Nombre de usuario {existente, inexistente}
    - Omite un parámetro que no afecta al comportamiento del SUT
  - Salida: abstrae detalles de la salida del SUT
    - Puede afectar el poder del oráculo

Abstracción en el modelo (2)

- Comunicación: secuencias de estímulos pueden ser descriptas en el modelo cómo un único estímulo
  - Se usa generalmente para testear protocolos
    - Abstracción de un handshaking
  - El modelo puede ignorar algunas señales intercambiadas con el SUT
- Calidad de Servicio:
  - El modelo puede ignorar cuestiones tales como restricciones temporales, utilización de memoria, seguridad, etc.

Ejecución de la test suite

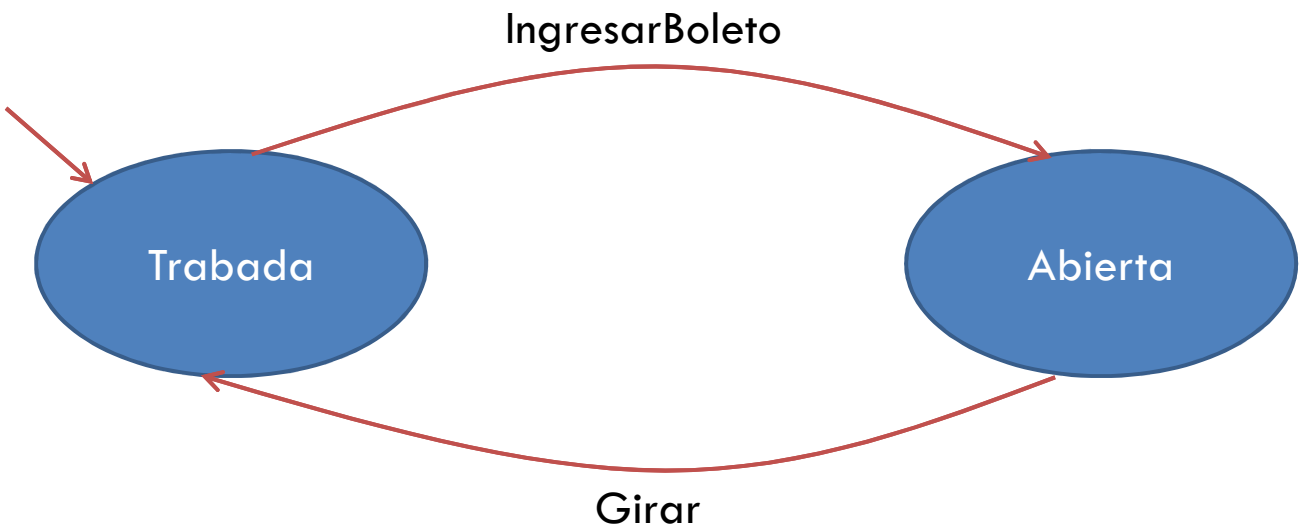
- Ejecutar un caso de test require:
  - Aplicar al SUT una instancia concreta del input descripto por el caso
  - Registrar la salida del SUT
- En general existe una brecha entre el modelo y el SUT (el modelo es una abstracción)
  - El adaptador es responsable de cubrir esta brecha
    - Traduce las entradas descriptas en el modelo en entradas del SUT
    - Abstrae las salidas del SUT

Validación del modelo

- El modelo debe ser validado : ¿Es el modelo una descripción cabal del sistema?
- Esta es una actividad ortogonal que requiere revisar los requerimientos por consistencia y completitud
- Esto implica que el modelo debe ser mas simple del SUT, o al menos, más simple de chequear, modificar y mantener

La maquinita del subte

La maquinita del subte



La maquinita del subte

Config.cord

```
config Main
{
    action abstract static void Implementacion.IngresarBoleto();
    action abstract static void Implementacion.Girar();
}


machine Program() : Main
{
construct model program from Main
}
```

La maquinita del subte

Model

```
enum Estados {Trabada, Abierta}

    static class ModelProgram
    {
        static Estados estado = Estados.Trabada;

        [Action]
        static void IngresarBoleto()
        {
            Contracts.Requires(estado == Estados.Trabada);
            estado = Estados.Abierta;
        }

        [Action]
        static void Girar()
        {
            Contracts.Requires(estado == Estados.Abierta);
            estado = Estados.Trabada;
        }
    }
```

Bibliografía

- M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach,* Morgan-Kaufmann, 2007.
- M Utting, A Pretschner, B Legeard. A taxonomy of model-based testing,Tech. Rep, 2006
- M.Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, L.Nachmanson: Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. Formal Methods and Testing 2008: 39-76
  Mas referencias en http://www.geocities.com/model_based_testing/.