

UNIVERSIDAD DE BUENOS AIRES

FACULTAD DE CIENCIAS EXACTAS Y NATURALES

LICENCIATURA EN CIENCIAS DE LA COMPUTACIÓN

Teóricas de Ingeniería del Software I

Autor:

Julián SACKMANN

10 de Septiembre de 2012



**Facultad de Ciencias Exactas y
Naturales**
Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

<http://exactas.uba.ar>

Índice

1	¿Qué es la Ingeniería del Software?	5
1.1	De qué se ocupa la Ingeniería del Software?	5
2	Modelos de Desarrollo	6
2.1	Modelo en cascada	6
2.2	Modelo en espiral	6
2.3	Modelo en V	7
2.4	Unified Software Development Process	8
2.5	Twin Peaks	8
3	Modelo de Software	10
3.1	Scope y Span	10
3.2	Lenguaje Formal	10
4	Validación, Verificación y Calidad	12
5	Ingeniería de Requerimientos	13
5.1	Acerca de la Ingeniería de Requerimientos	13
5.1.1	Actividades de la Ingeniería de Requerimientos	13
5.1.2	Ciclo de vida	14
5.1.3	Fenómenos, el mundo y la máquina	15
5.1.4	Aserciones	15
5.2	Modelo de Jackson	15
5.3	Modelo de las 4 variables	16
5.4	Modelo de agentes	16
5.5	Modelo de objetivos	17
5.5.1	Diagrama de objetivos	17
	Y-Refinamientos	17
	O-Refinamiento	17
	Refinamiento por hitos	18
	Refinamiento por casos	18
	Divide & Conquer	18
	Condiciones no monitoreables / no controlables	18
	Análisis de Riesgo	19
5.5.2	Ventajas	19
5.5.3	Objetivos	19
	Clasificación	20
	Elicitación de objetivos	21
	Objetivos vs Operaciones	21
	Contra recíproco y complementario	21
5.6	Modelo de operaciones	22
5.6.1	Diagrama de casos de uso	22
	Relaciones	23
	Super-Actores y Casos de uso auxiliares	24
5.6.2	Relación con los otros modelos	24
5.7	Modelo Conceptual	24
5.7.1	Definiciones	25
5.7.2	Categorías de clases conceptuales	25
5.7.3	Diagrama de clases	25
	Asociaciones entre clases	26

Agregación	26
Composición	26
Asociación calificada	27
Asociaciones <i>n</i> -arias	27
Generalización o Herencia	27
Clases de asociación	28
Otros	29
Diagrama de Objetos	29
5.7.4 Modelo conceptual vs Diseño	29
5.7.5 Relación con los otros modelos	29
5.8 OCL	30
5.8.1 OCL en modelo conceptual	30
Semántica	30
Recorte o Prunning	30
Sintaxis	31
Reglas de subtipificación	32
Queries	32
5.8.2 OCL fuera de modelo conceptual	32
5.9 Modelos de Comportamientos	33
5.9.1 Diagramas de Secuencia	33
HMSCs: High-Level Message Sequence Chart	34
De interaction-based a state-based	35
Escenarios implicados	36
Extensiones	37
5.9.2 Collaboration Diagrams	41
5.9.3 Redes de Petri	41
5.9.4 Diagramas de Actividad	42
Decisión	42
Fork y Join	43
Andariveles	43
5.9.5 Diseño vs Requerimientos	44
6 Máquinas de estado	45
6.1 LTS: Labelled Transition Systems	45
6.1.1 Semántica	46
Ejecución	46
Proyección	46
Trazas	46
Transición o Evolución	47
6.1.2 Concurrencia vs Paralelismo	47
6.1.3 Composición en paralelo	47
Leyes Algebráicas de composición	48
6.1.4 Modelado de Concurrencia	49
6.1.5 LTS no determinístico	49
6.1.6 Propuestas para la igualdad de LTS	49
Propuesta 1: Isomorfismo	49
Propuesta 2: Isomorfismo sin estados no alcanzables	49
Propuesta 3: Igualdad de Trazas	50
¿Qué buscamos?	50
6.1.7 Bisimulación	51
No-bisimilaridad	51
6.1.8 Congruencia	51

6.1.9	Semántica lineal vs semántica arbórea	52
6.1.10	Tau τ	53
6.1.11	Bisimulación considerando τ	53
Propiedades de bisimulación débil	54	
Modelado con <i>LTS</i>	54	
6.2	Autómatas temporizados	54
6.2.1	Semántica	55
6.3	Statecharts	55
6.4	SDL Flowchart	57
6.5	Relación con otros modelos	57
7	Testing	58
7.1	Introducción	58
7.1.1	Ciclo de vida y defectos	58
7.2	Calidad de Software	59
7.3	Definiciones	59
7.4	Testing	60
7.4.1	Limitaciones	60
7.4.2	Test de requerimientos no funcionales	61
7.4.3	Niveles de test	61
7.4.4	Ciclo de vida del testing	61
7.4.5	Testing funcional	62
7.4.6	Testing de conformidad	62
7.5	Generación de casos de testing	62
7.5.1	Notación	62
7.5.2	Técnicas	63
Random testing	63	
Partition testing	63	
Fault-based systems	63	
Criterios de caja negra	64	
Criterios de caja blanca	64	
Category Partition	64	
Técnicas de partición de dominio	64	
7.5.3	Técnicas combinatorias	65
Definición arbórea	65	
Grafo Causa-Efecto	66	
n-wise partition y OATS	66	
Conclusiones	67	
7.5.4	Testing estructural de unidades	67
Cubrimiento de sentencias	69	
Cubrimiento de decisiones o Branch Covering	69	
Cubrimiento de condiciones	70	
Cubrimiento de caminos	70	
MC/DC: Modified Condition/Decision Coverage	70	
Data Flow Testing	70	
7.5.5	Modelo de efectividad en la detección de fallas	71
¿En qué medida la cobertura importa para detectar fallas?	71	
¿Es mejor cubrir más o hacer una test-suite más grande?	71	
7.5.6	Subsunción	72
7.5.7	Properly Covers	72
7.6	Ambiente de testing	73
7.7	Terminación de testing	73

7.8	Documentación de casos de tests	73
7.9	Test de Regresión	74
7.9.1	Casos de regresión	74
8	Testing de sistemas reactivos	75
8.1	Introducción	75
8.2	Testing de Mealy Machines	75
8.2.1	Equivalencia de Mealy Machines	76
8.2.2	Conformidad de Mealy Machines	77
8.2.3	Algoritmo de T.S. Chow	77
Separating sequences	78	
Secuencia distintiva preset x	78	
Secuencia de distinción adaptativa	78	
Caso general	78	
8.2.4	Testing random para una B fija	79
8.3	Testing de LTS	79
8.3.1	Notación	79
8.3.2	Conformance basado en trazas	79
8.3.3	IOLTS	80
8.3.4	IOCONF	82
Input habilitado	82	
8.3.5	IOCO	83
8.4	Generación de casos de test	85
8.4.1	Representando como LTS	86
Interpretación de LTS	86	
8.4.2	Propiedades de una TestSuite	87
8.5	Model based testing	88
8.5.1	Construcción del modelo	89
8.5.2	Ejecutar una testsuite	89
8.5.3	Validar el modelo	89
9	Relación entre los diagramas	90
10	Glosario	91
11	Bibliografía	94
12	Agradecimientos	95

1 ¿Qué es la Ingeniería del Software?

Es una rama interdisciplinaria de la computación que se ocupa de construir un producto de software de **buena calidad**, considerando las restricciones pertinentes al caso (por ejemplo, presupuesto y tiempo disponible, escalabilidad, complejidad, etc). Notar que no sólo se acota al desarrollo del producto, sino de:

- Crear una especificación (con distintos niveles de formalidad) del producto que se desea en base a las restricciones provistas.
- Desarrollarlo.
- Documentarlo.
- Testearlo.
- Mantenerlo.

Formalmente, de acuerdo a la IEEE: *Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.*

1.1 ¿De qué se ocupa la Ingeniería del Software?

- Diseño, adaptación y medición de procesos.
- Gestión de proyectos (riesgos, factibilidad, toma de decisiones, planificación, etc).
- Diseño y operación de infraestructura para el proceso: ej. control de configuraciones.
- Análisis de requerimientos.
- Diseño (*in-the-small* e *in-the-large*).
- Selección y adquisición o desarrollo de componentes para re-uso e infraestructura de desarrollo.
- Verificación, Validación y Testing.

2 Modelos de Desarrollo

Un modelo de desarrollo de software se define como **un marco de trabajo usado para estructurar, planear y controlar el proceso de desarrollo de un sistema de información.**

2.1 Modelo en cascada

En 1970, Royce propuso el **modelo en cascada** del desarrollo de software. Esencialmente, se concibe el proceso de desarrollo como una secuencia ordenada de pasos estrictamente controlados. En este modelo no se permiten revisiones o reformulaciones de cosas ya realizadas. Tradicionalmente los pasos involucrados en este modelo son:

- Requerimientos.
- Diseño.
- Implementación.
- Integración.
- Validación.
- Instalación.

Este modelo tiene la ventaja de ser extremadamente simple y de tener un control muy férreo de las variables y restricciones involucradas. Por ejemplo, se puede decir que se asignan exactamente 2 semanas al Diseño y el modelo garantiza que no se vaya a ocupar más que eso (porque nunca se vuelve sobre lo mismo). Sin embargo, esa misma característica que provee la simpleza y elegancia del modelo es también la que causa su mayor deficiencia: la rigidez. Al mantener una estructura tan dura, no provee la versatilidad que en muchos casos es necesaria para un proceso de desarrollo.

2.2 Modelo en espiral

Para subsanar el problema de la rigidez del modelo en cascada, en el año 1988 Boehm propuso el **modelo en espiral**. Para lograr eso, este modelo concibe el desarrollo de software como una espiral infinita y creciente en donde cíclicamente se alternan períodos de:



En cada una de los períodos, se realizan las siguientes tareas:

- Análisis: se determinan objetivos y se analizan restricciones.
- Evaluación: se evalúan distintas alternativas de diseño.
- Desarrollo: se desarrolla y testea la alternativa elegida en la iteración anterior.

- Planeamiento: se planea la iteración siguiente, considerando los problemas que hubo durante la evaluación y desarrollo de esta.

El problema que presenta este modelo es la dificultad en controlar las restricciones. Si bien en cada iteración se analizan las restricciones, no es trivial determinar cuántas iteraciones serán necesarias (o, mejor dicho, si se puede lograr el producto en una cantidad de iteraciones que no supere las restricciones).

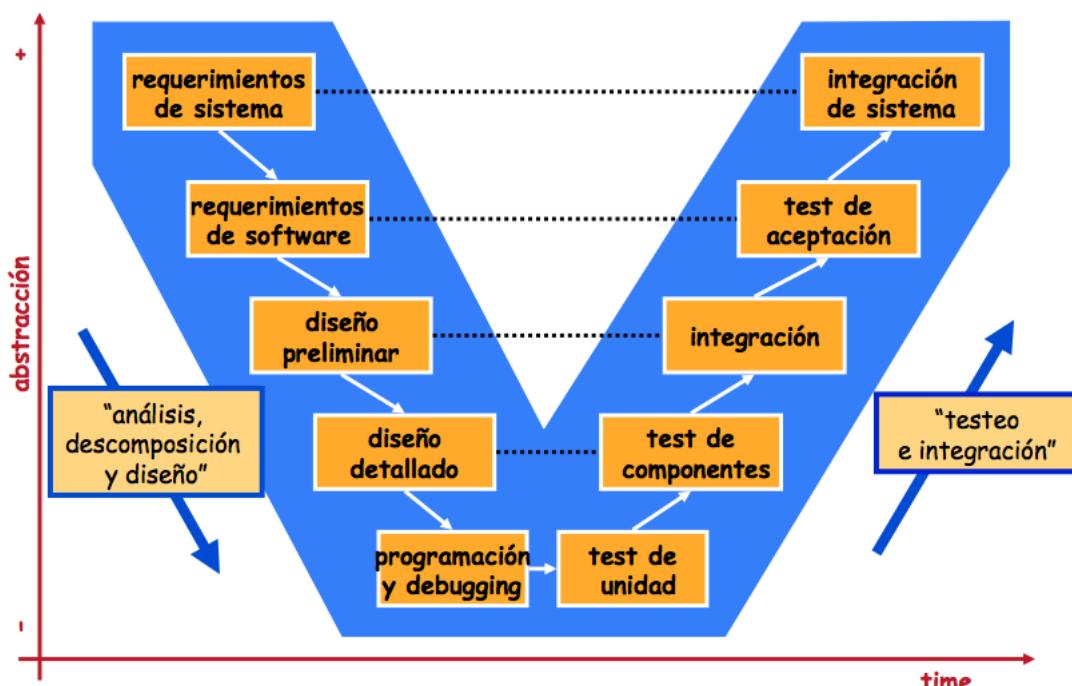
El modelo de ingeniería en requerimientos planteado también por Boehm tiene el mismo ciclo de vida: una espiral infinita y creciente para la captura de requerimiento alternando elicitación, negociación, especificación y validación.

2.3 Modelo en V

El modelo en V, plantea que el desarrollo de software sigue un modelo *top-down* con dos series de fases, similares al modelo en cascada. La primer serie, llamada “*análisis, descomposición y diseño*”, comienza con un alto nivel de abstracción y en cada fase se va bajando el nivel de abstracción, trabajando sobre elementos sucesivamente más concretos.

Por el contrario, en la segunda serie, llamada “*testeo e integración*” el nivel de abstracción va subiendo a medida que se avanza en las distintas fases.

Las fases por las que se atraviesa son:



Como se puede observar, el modelo recibe su nombre del hecho de que si se grafican las etapas en un gráfico de abstracción en función del tiempo, forman una V.

Este modelo presenta una relación simétrica entre los componentes que presentan igual grado de abstracción. En una determinada fase de la serie de *testeo e integración* se realizan los tests correspondientes a la tarea que se haya diseñado en la fase de la serie *análisis, descomposición y diseño* correspondiente al mismo nivel de abstracción.

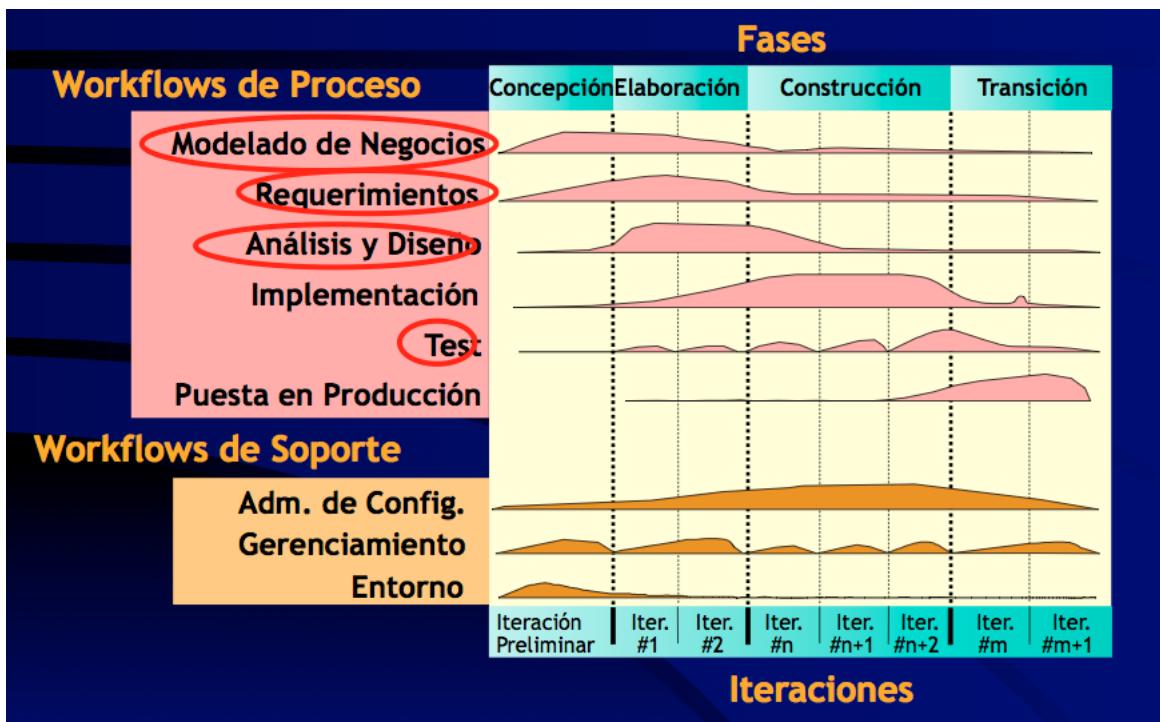
Por ejemplo, en la fase de *diseño preliminar*, se diseñan en abstracto, la interacción de las estructuras de datos a utilizarse dentro del programa, mientras que en *diseño detallado*, el esfuerzo se concentra en el diseño específico de cada una de esas estructuras de datos. Durante la fase de *test de componentes*, se testeaa

la correctitud de cada estructura por separado, mientras que en la de *integración* se testeá la relación entre estructuras.

El ciclo de vida del testing dinámico tiene el mismo ciclo de vida en “V” y la misma relación de simetría entre cada etapa y su test (requerimientos con testing de sistema, diseño preliminar con testing de integración y diseño detallado con testing de unidad), con la diferencia de que al final de todo debe correrse el test de aceptación.

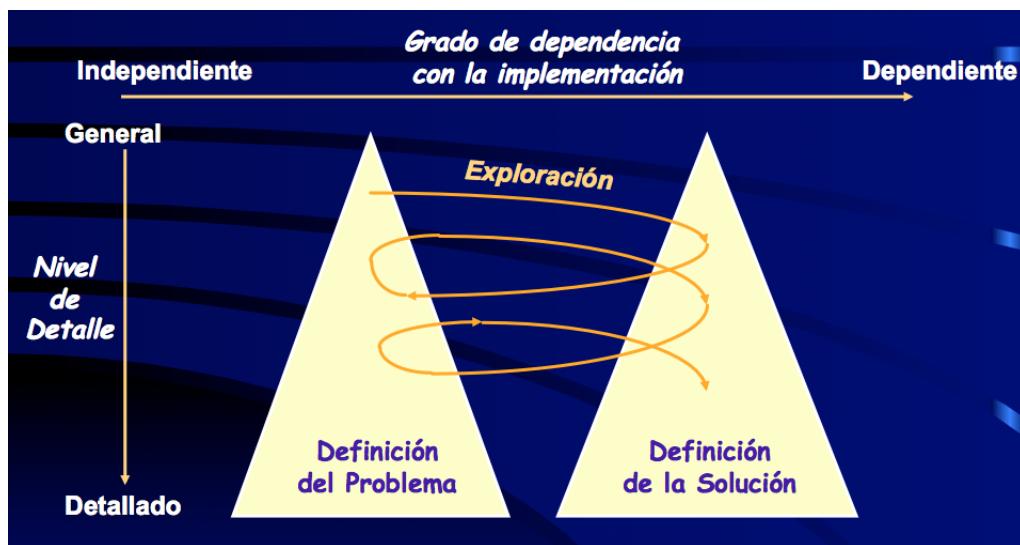
2.4 Unified Software Development Process

Propuesto por Jacobson en 1999.



2.5 Twin Peaks

El modelo *Twin Peaks*, propone un modelo de exploración alternada entre la definición del problema y la definición de la solución, incrementando en cada salto, el nivel de detalle de cada una de las definiciones.



3 Modelo de Software

En la Ingeniería del Software cobran un rol fundamental los **modelos de software**. Estos modelos son asiduamente utilizados porque permiten analizar un aspecto particular (o no) del problema. A su vez, permiten comunicar en forma precisa y ordenada aspectos relevantes tanto del problema como de la solución propuesta a otras personas.

Al trabajar con modelos se abaratan costos (construir una solución completa es en general notablemente más caro que un modelo) y se pueden detectar errores tempranamente en el proceso de desarrollo de una solución o producto.

Los modelos tienen 3 funciones claras:

- **AbstRAE:** permite al modelo enfocarse en un aspecto particular del problema, dejando de lado detalles irrelevantes al análisis (o de posterior solución).
- **Estructura:** simplifica y ordena el análisis.
- **Denota:** permite establecer la relación entre el mundo del documento y el mundo real.

Para la construcción de un buen modelo es necesario conocer el **propósito** del análisis.

En la ingeniería del software, a diferencia de otras disciplinas, el producto final (una pieza de software) es **intangible**. Esto implica que los modelos de software que se generan también son intangibles. Por esto, los modelos de software son **lenguajes formales** con una **denotación** precisa.

3.1 Scope y Span

Se define el **Scope** de un modelo como el tipo de fenómeno que se capta. Se relaciona con buscar seleccionar qué información se desea modelar y cuál ignorar. Básicamente, define el aspecto y alcance del modelo.

El **Span** de un modelo define el conjunto de individuos a los que describe. Influye fuertemente sobre cuánto debe alcanzar el modelo.

3.2 Lenguaje Formal

Un **Lenguaje Formal** se define como un lenguaje (sistema de comunicación estructurado) con símbolos y reglas para su combinación formalmente especificados. Tiene dos aspectos fundamentales

- **Sintaxis:** define el conjunto de símbolos de representación utilizados.
- **Semántica:** define el subconjunto de reglas con las que se evitan accidentes sintácticos (por ejemplo, ambigüedades). En general existe una relación muy cercana entre la *semántica* y la *denotación* del modelo. Para definir la semántica, se pueden utilizar técnicas como:
 - Definir una serie de reglas de traducción a otro lenguaje formal cuya sintáctica ya esté establecido.
 - Definir un conjunto de relaciones (de equivalencia u orden) entre los distintos elementos sintácticos del lenguaje.

La semántica de un lenguaje define una función que, dado un elemento de la sintaxis, devuelve un elemento (o subconjunto) del **dominio semántico** (conjunto de lexemas que pertenecen a un mismo campo de significación). Por ejemplo, en español, el dominio semántico de la palabra *asiento* puede ser: *silla*, *sillón*, *sofá*, etc.

Las reglas de transformación sintáctica deben mantener coherencia tanto con función de asignación al dominio semántico, como con la denotación y su correlación modelo-realidad. Una regla de transformación sintáctica es correcta si lleva dos elementos iguales en el modelo al mismo dominio semántico y a lo mismo en el mundo real.

Debido a que los problemas que se plantean en la computación suelen tener diversos aspectos con distintos objetivos y niveles de complejidad, y que cada modelo se enfoca en sólo un aspecto, suele ser necesario plantear no uno sino diversos modelos a la hora de modelar un sistema. Esto plantea el problema de la relación entre distintos modelos. Al ser modelos basados en lenguajes formales, idealmente, establecer una relación también debería tener un elevado grado de formalismo. Sin embargo, en la práctica es imposible vincular distintos modelos manteniéndolos analizables y entendibles. Es por esto que generalmente la relación entre modelos se da en lenguaje natural.

4 Validación, Verificación y Calidad

La **validación** es un proceso cuyo objetivo es aumentar la confianza en que la *denotación* del modelo es correcta. O sea que su descripción formal se corresponda con la realidad. Dado que involucra la realidad, necesariamente tiene que lidiar con informalidad. Más aún, debe establecer una noción de *correctitud* entre un modelo formal y algo informal.

La **verificación** es un proceso cuyo objetivo es garantizar que una descripción formal es *correcta* con respecto a otra. A diferencia de la validación, sólo involucra modelos formales. Sin embargo, la vinculación de los modelos no es trivial de formalizar. Si bien una vez formalizado se podrían generar pruebas automáticas, la complejidad (tanto espacial como temporal) de estas puede ser excesiva.

La **calidad** se define como el grado en el cual un software cumple con su propósito.

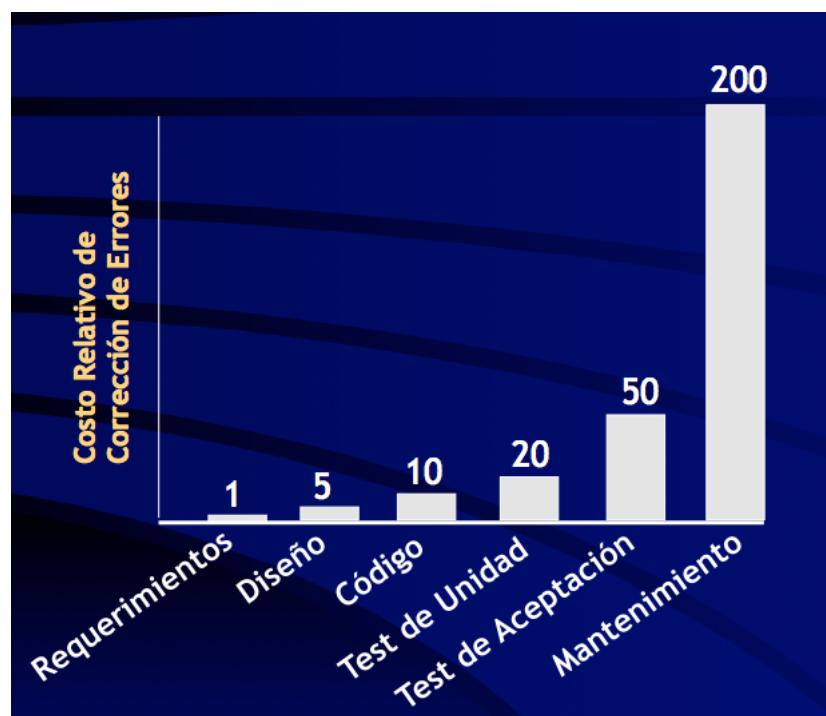
5 Ingeniería de Requerimientos

Toda pieza de software se desarrolla con un **propósito** dentro de un **sistema**. El software es, en mayor o menor medida, un componente más del sistema. La **ingeniería de requerimientos** se ocupa, parcialmente, de la identificación y comprensión de dicho propósito. La ingeniería de requerimientos suele ser eficiente en sistemas *intensivos* en software, es decir, sistemas en los que el componente de software del sistema cobra mucha relevancia.

Formalmente, se define a la ingeniería de requerimientos como: *Un conjunto de actividades cuyo objetivo es identificar y comunicar el propósito de un sistema intensivo en software y los contextos en los que se utiliza. De esta forma, la ingeniería de requerimientos actúa como un puente entre las necesidades de los usuarios, clientes y otros entes involucrados en el uso del sistema y las capacidades y funciones que son factibles dadas las restricciones.*

Uno de los usos de la ingeniería de requerimientos es el de encontrar errores tempranamente en el desarrollo del software. Esto es extremadamente importante porque la detección de errores en forma temprana redonda en costos muy inferiores para su arreglo. Un error detectado, por ejemplo, durante el período de mantenimiento puede tener un costo 200 veces más elevado para su reparación que detectado en la etapa de requerimientos.

Otro de los usos de la ingeniería de requerimiento es la estructuración en modelos para facilitar la validación del producto a desarrollar (por ejemplo: la estructuración de objetivos para validar el propósito del producto en el modelo de objetivos, la estructuración en operaciones para validar que cada operación incluya requerimientos que inducirán a propósitos en el modelo de operaciones, etc).



5.1 Acerca de la Ingeniería de Requerimientos

5.1.1 Actividades de la Ingeniería de Requerimientos

- **Elicitación:** es el proceso de investigación durante el cual se entrevista a clientes, usuarios y otros *stakeholders* con el objeto de obtener información sobre el sistema que quieren. A su vez se investiga

el sistema y documentación existente (si hubiera). No es un proceso trivial dado que no siempre los *stakeholders* tienen claro lo que quieren y muchas veces, aún cuando lo tienen claro no saben expresarlo.

- **Modelado:** es el proceso de documentación, de forma lo más rigurosa posible, de la información obtenida en la fase de elicitudación. Tiene como objetivo abstraer los conceptos importantes y estructurarlos de forma conveniente.
- **Análisis:** es el proceso cuyo objetivo es verificar el modelo creado en la fase anterior. Se verifica tanto la coherencia interna como la coherencia entre modelos del mismo sistema.
- **Validación:** es el proceso en el cual se valida el modelo de la segunda fase (y el análisis de la tercera) contra el entendimiento que tenemos de la realidad.
- **Priorización:** es la fase en la que se comparan las distintas estrategias de alcance de objetivos, criterios de evaluación, criterios de los *stakeholders*, etc.
- **Negociación:** es la fase en la que se discute con los *stakeholders* sobre qué decisión es la apropiada, por qué motivos y cómo se unifican los criterios e intereses de cada una de las partes.
- **Especificación:** es el proceso de formalización del modelado. Hay que crear una documentación completa y detallada. Depende lo que se necesite, esta documentación puede ser para lectura, para un contrato, etc.

5.1.2 Ciclo de vida

Según Boehm la ingeniería de requerimientos presenta un ciclo de vida creciente en espiral similar al modelo de desarrollo:



5.1.3 Fenómenos, el mundo y la máquina

La ingeniería de requerimientos trata los **fenómenos** que ocurren en el **mundo** y no en la **máquina**.

- **Fenómeno:** es un hecho, situación o evento cuya existencia puede observarse.
- **Máquina:** es una porción del sistema a desarrollar o modificar.
- **Mundo:** es una porción del mundo, afectado por la máquina.
- **Interfaz:** es el punto de contacto entre la máquina y el mundo.

5.1.4 Aserciones

Una aserción se define como *una afirmación, aseveración, expresión en que se da por cierta*. En la ingeniería de software se las categoriza en:

- Aserciones **descriptivas**: definen cosas que son o presumimos ciertas en el mundo sobre el que trabajamos. En general son del tipo de restricciones de dominio.
Taxonomía de aserciones descriptivas: si son propiedades físicas son *Propiedades del Dominio*; si son características aunque sujetas a cambios (por ejemplo: “la moneda argentina es el peso”) se denominan *Hipótesis del Dominio*.
- Aserciones **prescriptivas**: definen cosas que esperamos que vayan a ser ciertas una vez que sea afectado por nuestro sistema. Pueden ser de dos tipos:
 - **Objetivos**: cosas que esperamos que vayan a ser ciertas **en el mundo** una vez que sea afectado por nuestro sistema (objetivos multiagente).
 - **Requerimientos**: cosas que debemos hacer que sean ciertas **en la interfaz** (objetivos uniajentes).
Taxonomía de requerimientos: *expectativa* (si el requerimiento es una asignación de responsabilidades a un agente externo) o *requerimientos* (nuestro sistema deberá encargarse de cumplirlos).

Para entender bien la diferencia entre Objetivo y Requerimiento: mientras que los objetivos son fenómenos globales (por ejemplo: “prender el auto”), los requerimientos son objetivos uniajentes concretos (ejemplo: “la bujía prende con una chispa el motor”). Un objetivo podría considerarse como muchos requerimientos haciendo algo para un fin mayor (el objetivo.)

5.2 Modelo de Jackson

El modelo de Jackson es una forma de operar con ingeniería de Requerimientos propuesta por Michael Jackson. Tiene una estructura que permite formular criterios de verificación:

- Dadas las suposiciones de dominio (**D**), ¿los requerimientos (**R**) satisfacen los objetivos (**G**)? (**R,D⇒G**)
- ¿El programa (**P**) corriendo sobre el hardware (**C**) satisface los requerimientos (**R**)? (**P,C⇒R**)

Y numerosos de validación:

- ¿Tenemos todos los objetivos? ¿Son objetivos válidos?
- ¿Todas las presunciones de dominio son verdaderas? ¿Existen presunciones de dominio relevantes que no hayamos considerado?

Se definen:

- **Completitud** de un conjunto de requerimientos: están todos los requerimientos necesarios para lograr los objetivos.

- **Pertinencia** de un conjunto de requerimientos: no existe requerimientos que no ayuden a lograr los objetivos.

Este modelo garantiza la **completitud** de los requerimientos si:

- Dado **D, R** garantiza **G**.
- **G** captura adecuadamente las necesidades de los stakeholders.
- **D** representa presunciones válidas acerca del mundo.

Es digno observar que evaluar el conjunto de presunciones del dominio no es un detalle menor. Si uno supone presunciones demasiado fuertes, si bien la construcción del sistema se economiza mucho, es demasiado propenso a fallas en el caso de que esas suposiciones no sean verdaderas. Por el contrario, presunciones demasiado débiles redundarán en un software robusto, pero potencialmente demasiado encarecido.

Jackson plantea no validar los requerimientos considerados contra el cliente, sino sólo los objetivos. Los requerimientos se deducen intrínsecamente de los objetivos. Para esta validación de objetivos el modelo extiende la definición de **completitud** y **pertinencia** a objetivos e intenta validar eso contra los stakeholders.

5.3 Modelo de las 4 variables

El modelo de las 4 variables fue propuesto por Parnas y Madey en 1995. Su característica predominante es que elabora la relación entre objetivos y requerimientos. Originalmente estaba orientado sólo a software de control.

Se introducen los conceptos de **Variables monitoreadas** (variables que el sistema sólo puede medir), **Variables controladas** (variables sobre las que el sistema tiene influencia), **Datos de salida** y **Datos de entrada**. A su vez, se dota al sistema de **sensores** (dispositivos encargados de sensar el entorno) y **actuadores** (dispositivos encargados de modificar el entorno).

Los requerimientos definen relaciones entre los datos de entrada y salida. Los objetivos relacionan variables monitoreadas y controladas, y se logran en la medida que los sensores y actuadores “traducen” correctamente datos y variables.

5.4 Modelo de agentes

Vimos el “Diagrama de Contexto”.

El modelo de agentes intenta estructurar el mundo para poder manejar su enorme complejidad. Se define un **agente** como una entidad activa cumpliendo un rol determinado con capacidad de controlar o monitorear algún fenómeno del mundo (determinado por la interfaz). Puede ser un humano, un software, una máquina, etc, aunque no necesariamente hay una correlación 1 a 1: puede ser que un usuario determinado cumpla el rol de varios agentes (por ejemplo un humano que accede como usuario o como administrador).

Este modelo permite una representación gráfica conocida como *diagramas de contexto*. En estos, se grafican a los distintos agentes involucrados en el sistema a modelar y los distintos fenómenos en la siguiente forma:



5.5 Modelo de objetivos

Vimos el “Árbol de Objetivos”.

En el modelo de objetivos se define un **objetivo** como una aserción prescriptiva que el sistema deberá satisfacer a través de la cooperación de sus agentes. Esta aserción deberá estar dada en función de fenómenos en la **interfaz de agentes** (no necesariamente de la máquina). Los agentes deberán ajustar su comportamiento local de manera adecuada para garantizar el comportamiento global. Sólo podrá hacerlo dando garantías sobre la ocurrencia de fenómenos controlados por ellos mismos. La aserción deberá ser **declarativa** y no operacional. Es decir describir el objetivo a lograr y no cómo lograrlo.

5.5.1 Diagrama de objetivos

La forma de representar este modelo es mediante un **diagrama de objetivos**. Estos diagramas son un *DAG*, donde casi todos los nodos son distintos tipos de aserciones. Un nodo puede ser un objetivo, una propiedad de dominio o un agente. Los ejes entre objetivos denotan contribuciones para satisfacción. Los ejes entre objetivos y agentes denotan asignación de responsabilidad.

Los objetivos se grafican con un paralelogramo cuyo contenido (supongamos Q) denota el objetivo. Debe leerse “*Lograr el objetivo Q* ”. La relación entre dos objetivos debe entenderse como “*El objetivo Q contribuye a lograr el objetivo P* ”.

Los objetivos pueden ser categorizados en *alto* o *bajo* nivel en función de su grado de abstracción. Un objetivo de *alto* nivel es más estratégico, de negocios. Por el contrario, los objetivos de *bajo* nivel suelen ser más técnicos, ser más cercanos a las decisiones de diseño e involucrar menos agentes para su realización. En el caso extremo, en el que un objetivo tenga asociado un sólo agente **externo** se lo llama objetivo **uni-agente** o **expectativa**. Un **requerimiento** es un objetivo uni-agente que tiene que cumplir el sistema.

Se dice que un objetivo es **realizable** por un agente si:

- El agente puede **monitorear** los fenómenos necesarios para satisfacer el objetivo.
- El agente puede **controlar** los fenómenos cuya ocurrencia necesita ser restringida para satisfacer el objetivo.
- No es necesario que el agente pueda conocer el futuro para garantizar el objetivo en el presente.

Y-Refinamientos

Para lograr un objetivo dado, a menudo es necesario haber realizado más de un objetivo previo; o sea, suele ser necesario que un objetivo sea *refinado* por dos o más objetivos. Por eso, en el diagrama de objetivos se introduce la noción de **y-refinamiento**: una relación entre un objetivo y un conjunto de objetivos, en la cual todos los objetivos del conjunto *contribuyen* a lograr el otro objetivo. Un y-refinamiento de G en los nodos G_1, G_2, \dots, G_n debe ser:

- **Completo**: si la conjunción de todos los nodos G_1, \dots, G_n y las presunciones de dominio P lo implican. O sea, $G_1 \wedge G_2 \wedge \dots \wedge G_n \wedge P \Rightarrow G$. Informalmente, si están satisfechos todos los objetivos G_1, \dots, G_n y las presunciones de dominio son válidas, entonces G es válido.
- **Mínimo (o minimal)**: si elimino cualquier objetivos G_i de G_1, \dots, G_n , entonces dejo de poder inferir G .
- **Consistente**: si la conjunción de todos los objetivos G_1, \dots, G_n y las presunciones de dominio P no se contradicen. O sea $G_1 \wedge G_2 \wedge \dots \wedge G_n \wedge P \neq False$.

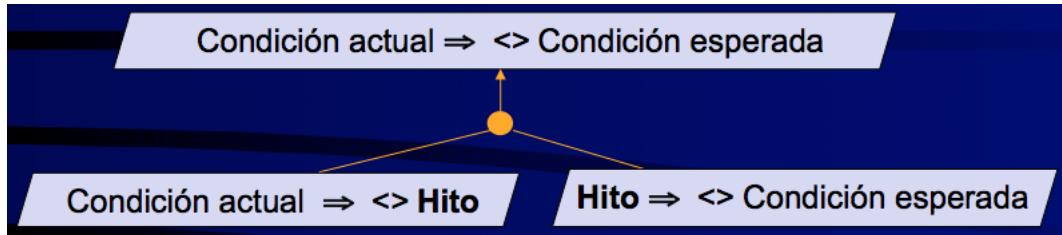
O-Refinamiento

Los **o-refinamientos** permiten vincular un objetivo con un conjunto de y-refinamientos, proveyendo “caminos” alternativos para contribuir a lograr un objetivo dado.

También relacionado con la provisión de alternativas, existe la **O-Asignación de Responsabilidades**, que modela la asignación de responsabilidades alternativas a hojas del grafo a agentes.

Refinamiento por hitos

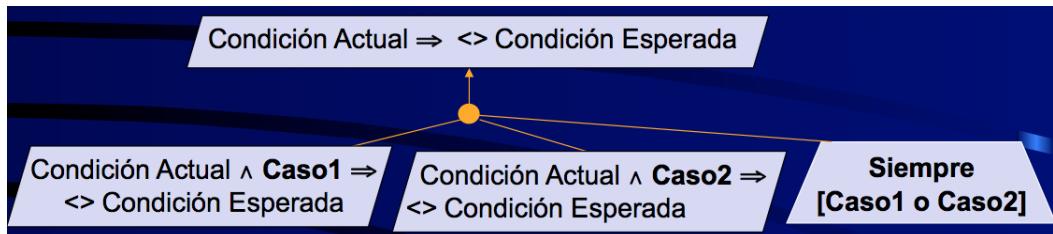
Los **refinamientos por hitos** (también llamado “refinamiento por milestones”) son un caso particular de y-refinamiento, en el cual para lograr objetivo de la forma: $Condición\ actual \Rightarrow Condición\ esperada$ se lo refina en dos objetivos de más bajo nivel:



De esta forma, se especifica, el “paso intermedio” por el que debe pasarse para pasar de la *condición actual* a la *esperada*. Observar que esta técnica es aplicable indistintamente del tipo de objetivo (*lograr*, *mantener*, *evitar*, etc.). Además, puede ser generalizada empleando n hitos intermedios.

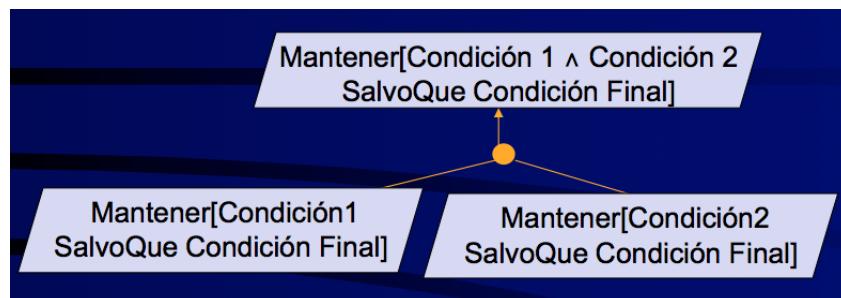
Refinamiento por casos

La técnica de **refinamiento por casos** es otro caso particular de y-refinamiento, que permite introducir objetivos complementarios. Nuevamente, se aplica en objetivos de la forma $Condición\ actual \Rightarrow Condición\ esperada$ (tanto *lograr* como *mantener*) y en los cuales haya una partición completa (aunque no necesariamente disjunta) en casos. Entonces, se refina el objetivo en dos objetivos de más bajo nivel:



Entonces, como cuando vale en el Caso1 y vale cuando vale en el Caso2 (y la partición es completa) la conjunción de eso (pues es y-refinamiento) implica que vale en todos los casos. Al igual que el refinamiento por hitos, se puede generalizar a n casos.

Divide & Conquer



Condiciones no monitoreables / no controlables

En el caso de tener objetivos *no realizables* por que involucran condiciones *no monitoreables* o *no controlables*, se puede aplicar una técnica para lograrlo. Sin embargo, para realizar esta técnica es necesario agregar una presunción de dominio específica, cosa que no siempre se puede hacer.



Análisis de Riesgo

En un nodo de presunción de dominio P , puede aplicarse la técnica de **análisis de riesgo**. Esta técnica consiste en agregar un nodo con la negación de la presunción ($\neg P$) colgando de P y refinar las condiciones que tienen que darse para que pase $\neg P$. A su vez se puede agregar (si se tuvieran) las probabilidades de que se de cada una de esas condiciones, pudiendo estimar fácilmente la probabilidad final de que ocurra el evento $\neg P$.

Otra mejora que se puede realizar a este modelo es agregar el costo de falla de la presunción, lo que nos permite directamente calcular la probabilidad de tener una pérdida si se diera $\neg P$ (nuevamente, tiene el problema de que esa información no siempre es accesible).

5.5.2 Ventajas

El modelo de objetivos presenta numerosas ventajas:

- La técnica de refinamiento permite estructurar especificaciones.
- La estructura provee una justificación clara para los objetivos. Se establece un orden topológico en los objetivos.
- Simplifica mucho la tarea de detectar objetivos faltantes (**completitud**) y sobrantes (**pertinencia**).
- Permite plantear numerosas alternativas, documentando los pros y contras de cada una.
- Permite generar *argumentos de satisfacción* (o sea, este objetivo lo podemos cumplir porque ya cumplimos todos sus hijos).
- Permite realizar análisis de riesgos, así como determinar el alcance del sistema.

5.5.3 Objetivos

La noción de objetivo puede ser extendida para aumentar la riqueza del diagrama y permitir que se grafe más información. Para esto, a los objetivos se los construye, no como hasta ahora, simplemente con una aserción prescriptiva, sino con:

- **Tipo de objetivo:** describe el tipo del objetivo. Puede ser:

- *Lograr.*
- *Evitar.*

- *Mantener.*
- *Objetivo Blando.*

- **Nombre.**

- **Categoría** (opcional): la partición en categorías de los objetivos puede emplearse como una heurística para emitir requerimientos o analizar y reutilizar modelos. Permite establecer patrones de objetivos frecuentes.
- **Definición natural:** es la aserción prescriptiva, expresada en lenguaje natural.
- **Definición formal** (opcional): es la aserción prescriptiva, expresada en algún lenguaje formal (por ejemplo, lógica de primer orden, lógica proposicional, etc). Notar que no siempre es posible (o sencillo) expresar objetivos formalmente.

A su vez, también se pueden agregar otros atributos como *referencia*, *stakeholders afectados*, *prioridad*, *estabilidad*, *costo*, etc.

Clasificación

Un objetivo se clasifica de **funcional** si es una función o servicio a ser provisto por el sistema. Deriva en una o más operaciones concretas en la interfaz.

A diferencia de esto, un objetivo **no funcional** es aquel que describe restricciones adicionales y no describe comportamiento específico del software. No dice *qué* debe hacer el software sino por ejemplo *cuán bien* o *cuán rápido* lo debe hacer. En general no afectan a una operación concreta sino a grandes porciones de la funcionalidad (*cross-cutting*).

Ortogonalmente a esto, un objetivo se puede categorizar como **de comportamiento** o **blandos**. Un objetivo se dice de comportamiento cuando recorta el espacio de comportamiento permitido del software. La comprobación del objetivo es una función binaria que toma una traza o ejecución y devuelve si el objetivo se satisfizo. Para un sistema debo prohibir cualquier ejecución del mundo que no satisfaga alguno de mis objetivos. Los objetivos de comportamiento tienen un correlato con modelos operacionales de comportamiento (FSM y diagrama de secuencia). Son los de tipo *lograr*, *mantener* y *evitar*.

Los objetivos blandos, en cambio son aquellos que denotan preferencia entre comportamientos. Permiten expresar ventajas y desventajas en cuanto a aspectos puntuales en o-refinamientos. Su satisfacción no puede establecerse mirando un sistema o una traza, deben compararse al menos dos. Es muy difícil vincular con modelos de comportamiento. Son los objetivos de tipo *objetivo blando*. Pueden categorizarse en cosas del tipo *maximizar*, *incrementar*, *mejorar*, etc. Existe una dualidad entre los objetivos blandos y las alternativas: los objetivos blandos pueden introducir alternativas nuevas, así como las ventajas/desventajas de una alternativa existente pueden introducir nuevos objetivos blandos.

Este tipo de objetivos tiene dos representaciones gráficas posibles:

- Gráfica: nodos en el grafo del diagrama de objetivos. Cada nodo está asociado a al menos dos ejes de un o-refinamiento, especificando con ‘+’ o ‘-’ el orden relativo de las alternativas.
- Tabular: se grafican todos los objetivos blandos en una sola tabla. Las columnas representan los objetivos blandos mientras que las filas son las distintas alternativas. En la intersección entre una alternativa y un objetivo blando, se representa su “medición” (con ‘+’ o ‘-’).

Un objetivo se clasifica de **medible** si se puede medir. Idealmente, todos los objetivos deberían ser medibles. Los objetivos de comportamiento son trivialmente medibles (pues tienen la función de comprobación). Sin embargo, para los objetivos blandos, es necesario que haya un *criterio de comparación* y para los de comportamiento debe haber un *criterio de aceptación*.

Estas elecciones de criterios no siempre son triviales: puede haber objetivos para los cuales el concepto de medición sea difuso, que los *stakeholders* no sean específicos, etc. Más aún, existe el problema de que generalmente, aún los objetivos que son medibles no los puede medir uno mismo; sólo los pueden medir los *stakeholders*.

Elicitación de objetivos

El proceso de elicitation de objetivos puede realizarse **temprana-** o **tardía-** mente. En el primer caso, se indaga sobre los problemas y deficiencias del sistema actual (suponiendo que haya uno) y los objetivos de mejora y estrategia del cliente a futuro. Esto se puede realizar mediante distintas técnicas de elicitation, tales como *basadas en stakeholders* (entrevistar stakeholders, tanto individual- como grupal- mente) o *basadas en documentación* (revisar documentación del sistema existente). Es frecuente utilizar *checklists de categorías* de objetivos.

En el caso de elicitation tardía (llamada elicitation **posterior**) se puede realizar por tres técnicas:

- **Por abstracción** (bottom-up): se pregunta “*¿por qué?*” sobre distintos elementos (objetivos de bajo nivel, escenarios, descripciones operaciones, FSMs, manuales de procedimientos, etc). Por ejemplo, se indaga por algún objetivo puntual y conocido de bajo nivel (o conjunto de) y se pregunta “*¿Por qué se hace esto? ¿con qué fin?*”. La respuesta a esto induce a descubrir uno o más objetivos a los que este objetivo contribuye, de nivel un poco más alto. Iterando este proceso, se puede ir planteando “desde abajo hacia arriba” un posible diagrama de objetivos. El alcance acota los objetivos. Se intenta relevancia de todos los fenómenos y propiedades en otros modelos. Se evita el riesgo de regresión de objetivos.
- **Por refinamiento** (top-down): se pregunta “*¿cómo?*” sobre objetivos disponibles. Similarmente al caso anterior, se indaga por algún objetivo pero de alto nivel y se pregunta: “*¿y cómo se logra esto? ¿hay otra opción?*”. La respuesta permite plantear objetivos de más bajo nivel que contribuyan a lograrlo. Iterando, se puede plantear “desde arriba hacia abajo” un posible diagrama de objetivos. En general uno detiene el proceso cuando logra objetivos uni-agente (expectativas) o propiedades del entorno.
- **Por resolución de conflictos y obstáculos**: se plantea el diagrama a medida que van surgiendo conflictos, basándose en las soluciones planteadas.

Objetivos vs Operaciones

Es importante la distinción entre **objetivos** y **operaciones / casos de uso**. Las operaciones inducen objetivos mediante sus poscondiciones. Las operaciones podrían ser más generales. Pueden plantearse relaciones muchos a uno.

Contra recíproco y complementario

Frecuentemente, los objetivos del tipo *lograr* tienen asociada una *precondición* y una *condición esperada*. O sea, son de la forma:

“Si *precondición* entonces en el futuro *condición esperada*”

De los objetivos de esta forma suelen inducirse **objetivos de seguridad relevantes** (no siempre ocurre que se puedan inducir y, cuando se puede, no siempre son relevantes). Estos objetivos son del tipo *mantener*, se los llama **contra recíproco** y son de la forma:

“Siempre que *condición esperada* entonces *precondición*”

De los objetivos lograr de la antedicha forma también suelen inducirse objetivos relevantes llamados **complemento**, de la forma:

“Si **no** *precondición* entonces ...”

5.6 Modelo de operaciones

Vimos el “Diagrama de Casos de Uso”.

El **modelo de operaciones** es un modelo que permite **estructurar las operaciones**. Determina las operaciones o servicios que debe proveer la máquina y las transformaciones que deben ocurrir en el mundo como consecuencia de las antedichas operaciones.

Una **operación** se puede ver como una función que toma un estado del mundo y devuelve otro estado (una modificación de la entrada). Observar que **sólo pueden haber cambiado las variables controladas por la máquina**. Una operación es introducida por un objetivo uni-agente del modelo de objetivos (si son requerimientos introducen operaciones de software, si son expectativas inducen operaciones de agentes. No necesariamente uno a uno). Consta de:

- **Operación:** nombre de la operación.
- **Responsable:** actor responsable de la operación.
- **Usuarios.**
- **Definición:** explicación de la operación.
- **Entrada:** la entrada que tomará la máquina para realizar la operación.
- **Salida:** la salida que devolverá la máquina después de realizar la operación sobre la entrada.
- **Pre-condición:** condiciones necesarias para que la operación pueda realizarse.
- **Post-condición:** condiciones cuya validez está garantizada después de la operación.
- **Trigger:** condición que si se da, la operación necesariamente ocurre.

5.6.1 Diagrama de casos de uso

Es una posible notación gráfica para el modelo de operaciones. Surge de la necesidad de estructurar las operaciones para facilitar su validación (caso contrario, la tarea se dificulta muchísimo por el enorme volumen de operaciones). El **diagrama de casos de uso** estructura el conjunto de operaciones atendiendo a la categoría de usuarios que participan en el mismo. Describen en forma de acciones y reacciones las operaciones provistas por la máquina desde el punto de vista del usuario (lo que facilita la validación por parte de los usuarios). Sólo se enfoca en la **funcionalidad** provista por la máquina a construir y la interacción máquina-agente, no busca dar una descripción detallada de funcionalidad.

Los diagramas de casos de uso sirven para definir claramente el alcance de la máquina a construir (lo que esté adentro de la caja) y facilitar la validación de funcionalidad con los *stakeholders*.

Los diagramas de casos de uso constan de los siguientes componentes:

- **Actor:** se define como un conjunto de entidades concretas clasificados de acuerdo a una característica común a ellos. Representa a un tipo de usuario, abstrayendo al usuario real (que puede ser personas, sistemas, dispositivos, softwares, máquinas, etc). El nombre del actor describe su rol desempeñado. Si bien no es uno a uno, existe una correlación entre los agentes y los actores del modelo de objetivos (un agente puede estar modelado por varios actores, sea simultáneamente, alternadamente o algo así).
- **Máquina:** se representa con un rectángulo. Dentro de ella viven los casos de uso que se llevan a cabo dentro de ella.
- **Caso de uso:** especifica una o más secuencias de acciones que el sistema puede llevar a cabo interactuando con sus actores. Describe un conjunto de escenarios. Su nombre suele expresarse en gerundio. Consideremos una ejecución como un conjunto de tuplas (a, o, a') , que debe interpretarse como que el actor a ejecuta la operación o sobre el actor a' . Entonces un caso de uso puede verse como un conjunto de estas secuencias que verifica que antes de la primer tupla vale la precondición y después de la última vale la postcondición.

En muchas ocasiones, un nombre en gerundio no alcanza para especificar un caso de uso (por ejemplo, para explicar una interacción crítica, o clasificar aspectos de la funcionalidad o roles de los actores). Existen numerosas formas de subsanar esta carencia, siempre generando una **descripción detallada de casos de uso**. Esta descripción, puede realizarse en lenguaje formal, informal, secuencia de operaciones, tablas, diagramas de secuencia, máquinas de estado, etc. Es decir, es extremadamente versátil la forma de documentar detalladamente un caso de uso, aunque suele ser buena práctica agregar pre y post condiciones.

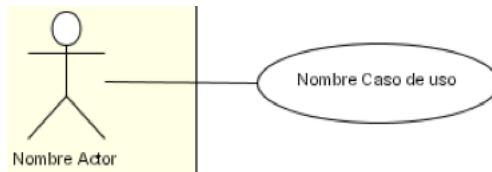
Sea cual sea la forma de detallar los casos de uso que se elija, es habitual que durante la ejecución de un caso de uso aparezcan **errores o excepciones** en los cuales se suele desviar el curso normal del programa. A esta desviación se la llama **alternativa**. Es importante que el detalle de caso de uso explice estas alternativas.

En estos detalles de casos de uso, frecuentemente se explicitan las circunstancias en las que un caso de uso “usa” a otro. Estas “llamadas” denominan **punto de uso**.

Relaciones

Dado que el diagrama de casos de uso se usa muy informalmente, se introducen las relaciones entre casos de uso para proveer un poco de estructura y establecer relaciones entre los distintos elementos del diagrama. Algunas de estas relaciones son:

- **Participa en:** un actor A *participa en* un caso de uso U si y sólo si la descripción detallada de U hace referencia explícita al actor A como participante de la interacción. Es una notación sintáctica. Observar que que A participe en U no garantiza que en todo escenario $s \subseteq U$, A interactúe con el sistema en s (por ejemplo, podría no ocurrir en una alternativa).

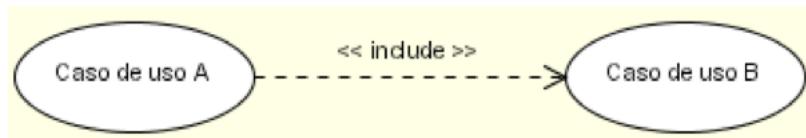


- **Herencia:** semánticamente, X hereda de Y si y sólo si $X \subseteq Y$. Se utiliza para estructurar actores según la relación “es un tipo especial de”. Si un actor a' hereda de un actor a (se nota con $a' \rightarrow a$) entonces todos los casos de uso que son relevantes para a , lo son para a' , pero no vale la vuelta.

Si Y tiene especializaciones x_1, \dots, x_n y vale que $\forall y \in Y, y \in x_i$ para algún $1 \leq i \leq n$, entonces Y se denomina **abstracto**. O sea una entidad es abstracta sólo si todos sus elementos pertenecen a alguna de sus especializaciones. Se utilizan para estructurar más cómodamente e introducir conceptos relevantes.



- **Inclusión:** un caso de uso A incluye a un caso de uso B si y sólo si cuando un escenario s es descripto por A , entonces existe una porción de s que contiene un escenario descripto por B .



- **Extensión:** un caso de uso *A* extiende a un caso de uso *B* si y sólo si existe un escenario *s* denotado por *A*, que contiene un escenario denotado por *B* (pueden haber escenarios denotados por *B* que no aparezcan en escenarios denotados por *A*). Representan una parte de la funcionalidad del caso que no siempre ocurre. Suelen utilizarse para errores, excepciones y casos alternativos.



Super-Actores y Casos de uso auxiliares

Un **super-actor** es un actor abstracto, agregado con el sólo propósito de tipar un caso de uso incluido en muchos otros. Para que no ensucie la notación, es importante que sea realmente relevante desde el punto de vista del problema.

Es frecuente en lenguajes formales sea necesario introducir elementos poco intuitivos para saltar las restricciones sintácticas del lenguaje. Si bien en ocasiones es imposible eludir el uso de este tipo de construcciones, es buena práctica intentar evitarlas. Para esto, los diagramas de actividad permiten agregar casos de uso **auxiliares**. Estos casos de uso no agregan expresividad al lenguaje, pero sí permiten ordenar y aclarar, resumiendo cosas ya dichas. Sirven para evitar introducir **elementos foráneos** en el diagrama de casos de uso.

Así, un caso de uso *U* es auxiliar si:

- En el diagrama, no hay actores que participen en él (prohibido en la cátedra de IngSoft 1).
- Las referencias a actores en la descripción detallada de *U* son de un tipo de actor abstracto *A* definido implícitamente.
- El actor abstracto *A* es super-actor de todos los actores que participan de casos de uso que usan a *U* y todos los casos de uso que son extendidos por *U*.

5.6.2 Relación con los otros modelos

- **Modelo de Jackson:** fenómenos en la interfaz vs operaciones.
- **Modelo de agentes:** consistencia agentes y actores: interfaz, visibilidad de fenómenos.
- **Modelo de las 4 variables:** monitorabilidad / controlabilidad vs. Agente responsable / entrada / salida.
- **Modelo de Objetivos:** relación muchos a muchos entre requerimientos y expectativas vs. operaciones

5.7 Modelo Conceptual

Vimos el “Diagrama de Clases”.

También conocido como “modelo de dominio”, es el modelo más clásico de la ingeniería de requerimientos. Intenta explicar y estructurar la definición de los **conceptos** relevantes (en la definición del problema) de tal forma que sea validable. Los conceptos son los sustantivos asociados al dominio del problema.

Para su representación existen diversas técnicas, con distintos niveles de complejidad:

- **Diccionario / Glosario:** es una lista de clases con sus atributos y relaciones. Tiene poca estructura y es complicado de analizar/validar. Se usa lenguaje natural para explicar la denotación en detalle.
- **Diagrama de Entidad-Relación (DER):** usa un lenguaje gráfico que provee estructura. Muy usado para bases de datos.
- **Diagrama de clases:** extiende DER con varias características, como *herencia, modificadores*, etc.

5.7.1 Definiciones

- **Objeto conceptual:** denota una entidad o concepto del dominio del problema. Puede ser un *objeto pasivo, objeto activo, personas, estructuras, etc.*
- **Clase conceptual:** denota un conjunto de objetos conceptuales que comparten características comunes. Estas características pueden ser:
 - **Atributo:** es una característica intrínseca al objeto, completamente independiente de otros objetos. Consta de un nombre y, posiblemente, un rango.
 - **Relación:** es una característica que vincula conceptualmente un objeto a otros. Cada objeto juega un rol conceptual en ese vínculo.

Para identificar clases conceptuales se utilizan estrategias tales como: *identificar frases nominales¹ en la descripción de dominio o utilizar listas de categorías de clases conceptuales.*

Expresan “tipos” de entidades conceptuales del mundo real.

5.7.2 Categorías de clases conceptuales

- Objetos tangibles o físicos.
- Especificaciones, diseños o descripciones.
- Lugares.
- Transacciones.
- Líneas de la transacción.
- Roles de gente.
- Contenedores de otras cosas.
- Contenidos.
- Otros sistemas informáticos o electromecánicos externos al sistema.
- Conceptos abstractos.
- Organizaciones.
- Hechos.
- Procesos (no frecuente).
- Reglas y políticas.
- Catálogos.
- Registros de finanzas, trabajos, contratos, etc.
- Instrumentos y servicios financieros.
- Manuales, documentos, artículos de referencia, libros.
- Relaciones.

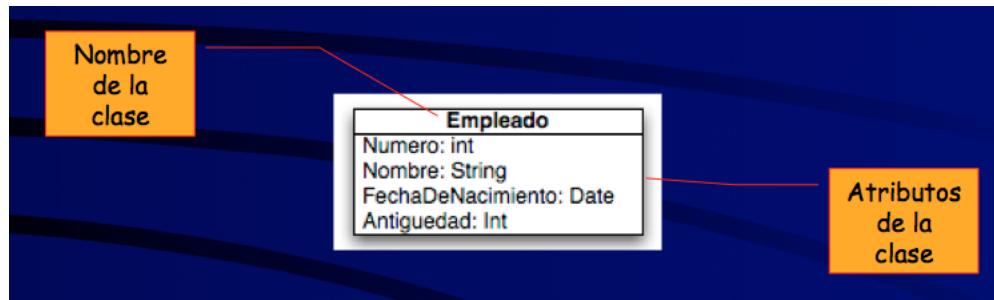
5.7.3 Diagrama de clases

En los diagramas de clases se emplea una notación gráfica particular para denotar las clases conceptuales, sus atributos y sus relaciones.

Las clases conceptuales se denotan con un rectángulo cuyo título es el nombre de la clase y, abajo, se listan sus atributos. Las relaciones entre clases conceptuales se notan con líneas uniendo los distintos rectángulos, sobre las que se escriben las características de la relación (nombre y cantidad).

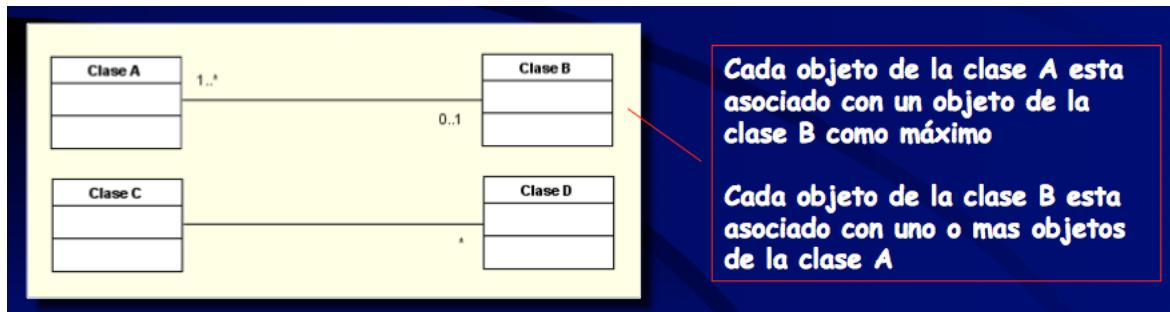
Junto con los diagramas de objetos, los diagramas de clases se usan para modelar el dominio del problema y el de la solución.

¹Frase nominal: sustantivo o conjunto de palabras que actúan como tal.



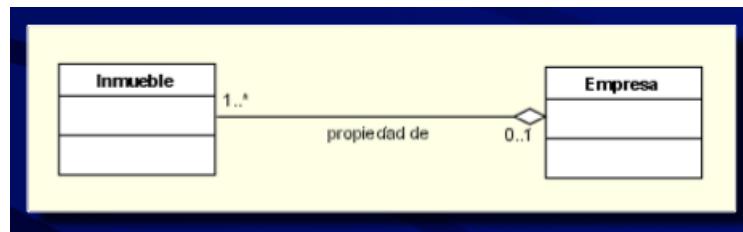
Asociaciones entre clases

Una **asociación** entre dos clases expresa una conexión bidireccional entre objetos. Es una abstracción de la relación existente en los enlaces entre objetos. Para agregar restricciones sobre las asociaciones y enlaces se agregan elementos de **multiplicidad**: en la asociación, se especifican cotas inferiores y superiores para la cantidad de instancias de las clases que se relacionan por esa asociación. A su vez, para simplificar la lectura de los vínculos se emplean **roles**: se asigna un nombre al final de la asociación, que explica la relación entre los conceptos en un sentido particular. Cada asociación tiene dos roles que permiten navegar la asociación en ambos sentidos.



Agregación

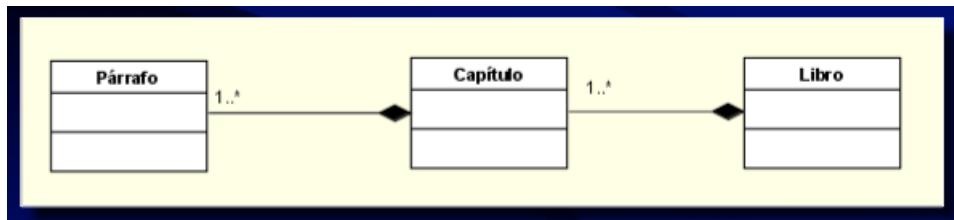
Es un tipo de asociación especial, del tipo “parte de / todo” dentro de la cual una o más clases son parte de un conjunto. Se indica con un rombo. Indica cuándo un objeto de una clase (que es un todo, un objeto indivisible) se piensa como parte de otro objeto de otra clase.



Composición

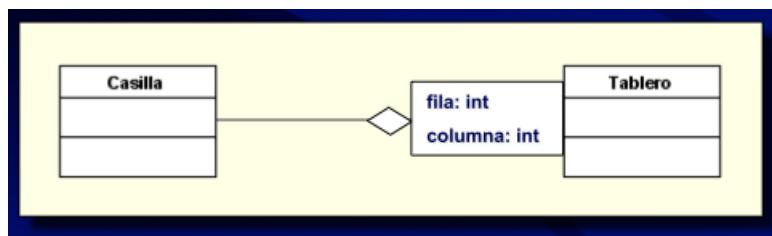
Es una forma fuerte de agregación que denota que un objeto de una clase es parte de un objeto de otra. Si no forma parte de tal objeto, entonces carece de sentido. Se diferencia de la agregación normal en que:

- En la composición tanto el todo como las partes tienen el mismo ciclo de vida.
- Un objeto puede pertenecer solamente a una composición.



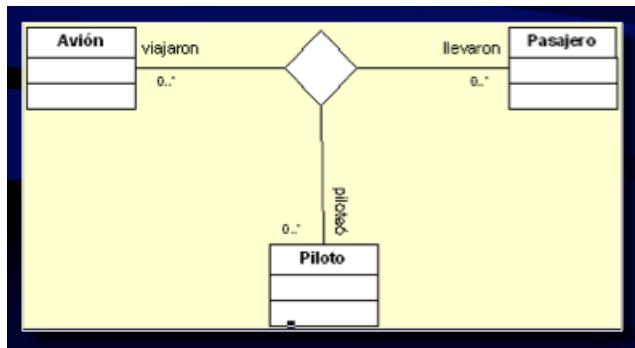
Asociación calificada

Es un azúcar sintáctico que permite representar muchas asociaciones en forma compacta. Se utiliza un **calificador**: un atributo (o tupla de atributos) de la asociación cuyos valores sirven para partitionar el conjunto de objetos enlazados a otro (observar que el rectángulo calificador es parte de la asociación y no de la clase).



Asociaciones *n*-arias

Son asociaciones que se establecen entre más de dos clases. Una clase puede aparecer varias veces desempeñando distintos roles. Semánticamente, son una forma de aplicar restricciones y explicitar relaciones entre *n* clases de equivalencia. Pueden ser usadas para refinar un modelo. La multiplicidad en las clases *n*-arias suele ser difícil de definir. Generalmente se refieren a una clase contra todas las demás (o se aclara en lenguaje natural a cuál se refiere).



Generalización o Herencia

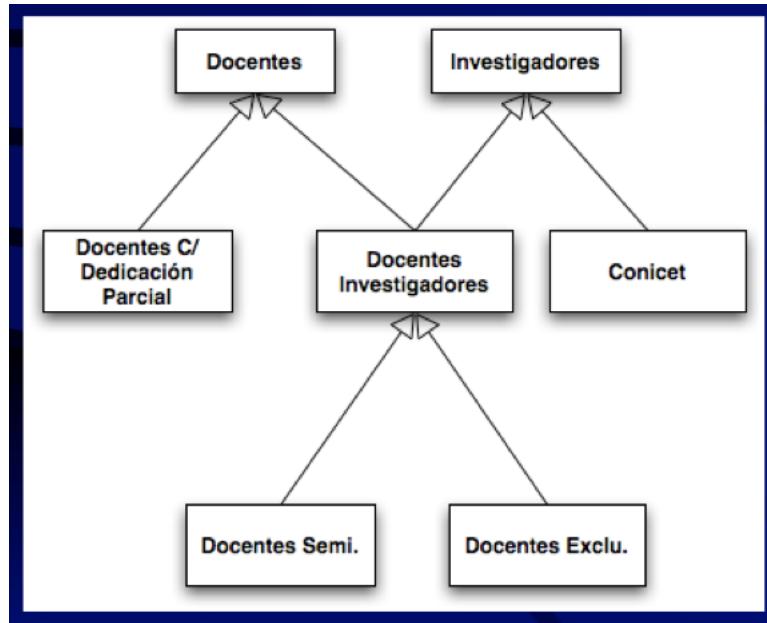
Es una relación entre una clase general (superclase o padre) y una versión más específica de dicha clase (subclase o hija). El hijo hereda todos las características y atributos del padre (aunque pueden incluir atributos o relaciones que el padre no tiene). A la hora de realizar una interpretación, la herencia denota **inclusión de instancias**: todo objeto de la clase hija es a su vez un objeto de la clase padre.

Se extiende la herencia con los siguientes modificadores:

- Overlapping/Disjoint.
- Complete/Incomplete.

Observar que una clase puede generalizar a dos conjuntos de hijos distintos. Se utiliza una anotación (llamada **discriminador**) para documentar la distinta intención que cada clasificación tiene.

También puede extenderse con **herencia múltiple** en el cual la clase hijo es la intersección de dos clases padre.

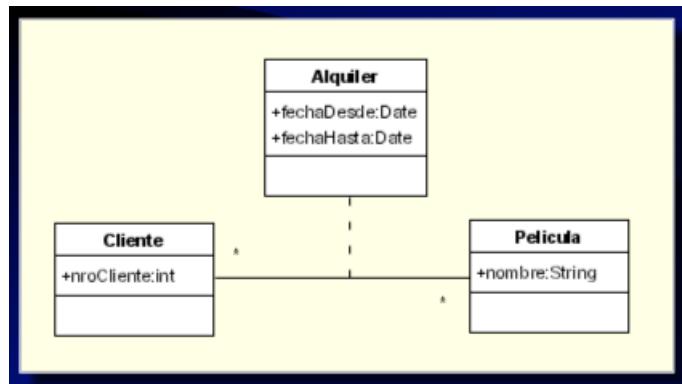


Clases de asociación

Las clases de asociación modelan características de una asociación que son independientes de las clases que asocian. Informalmente, es una clase que no tiene sentido por sí sola. Si algo cambia en la clase es porque alguna de las clases que involucra cambió.

Formalmente, si C es una clase de asociación para R en $A \times B$, entonces introduce una función $f : R \rightarrow C$. Se garantiza que

- Para un mismo par (a, b) no existe más de un c .
- Para todo c en C existe r tal que $f(r) = c$.
- Si $c = f(r)$, c no puede cambiar por sí sola (si cambia es porque la naturaleza de la asociación cambió).



Otros

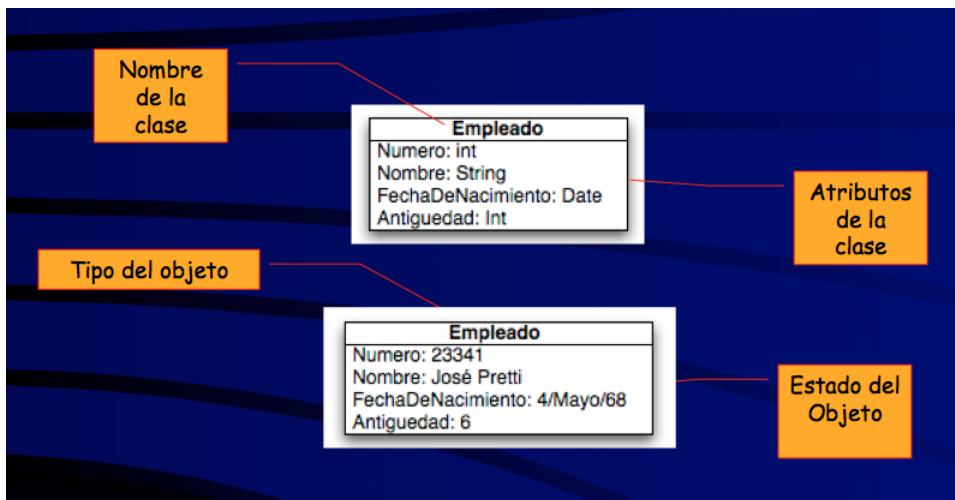
Existen muchos otros modificadores y elementos sintácticos que pueden ser usados en los diagramas de clases. Algunos muy usados son:

- **Clases enumeradas:** conjuntos de valores prefijados.
- **OR / XOR de relaciones.**
- **Subset.**
- **Atributos derivados:** explicitan invariantes.

Diagrama de Objetos

Informalmente, se puede ver como una instanciación a un momento determinado de un diagrama de clases: define el mundo en un instante dado. La relación entre dos objetos se corresponde con la de sus clases (toda instancia de una clase *X* tiene enlaces con instancias de *Y* de acuerdo al tipo de relación, atributos y modificadores que *X* tiene con *Y* en el modelo de clases).

En este contexto, un **objeto** denota una entidad conceptual del mundo real.



5.7.4 Modelo conceptual vs Diseño

A diferencia de esto, en el modelo de diseño las clases y objetos discriminan cómo se agrupará el *código* (no conceptos) y los datos al momento de programar el software y al momento de ejecutarlo. A su vez, las operaciones son frecuentemente asignadas a objetos y clases (en el modelo conceptual esto sólo tiene sentido para **entidades activas**).

5.7.5 Relación con los otros modelos

- **Modelo de objetivos:** describe la estructura estática del mundo sobre la que los objetivos predicen. Es la base para formalizar objetivos.
- **Modelo de agentes:** no existe una correspondencia uno a uno entre agentes y clases. Los diagramas de contexto describen el estado interno de los agentes y las operaciones que provee.

5.8 OCL

El **OCL** (*Object Constraint Language*) es un lenguaje formal que permite definir restricciones sobre objetos. Observar que no es un modelo en sí mismo, sino una herramienta para ser utilizada con otros modelos:

- **Modelo conceptual.**
- **Modelo de operaciones:** se utiliza para definir formalmente las condiciones (*pre*, *post* y *trigger*).
- **Modelo de objetivos:** si bien es necesario dotarlo de operadores temporales, se puede utilizar para generar una definición formal de objetivos.
- **Diseño:** es utilizado para formalizar contratos y aumentar la expresividad de los diagramas de clases.
- **Navegación de modelos estructurales.**

5.8.1 OCL en modelo conceptual

Se utiliza para agregar restricciones adicionales a los diagramas de clases. Estas restricciones en general tienen un carácter complejo que redunda en que no puedan ser expresadas por la acotada e inexpresiva sintaxis de los antedichos diagramas. Las restricciones ayudan a definir qué es un estado válido en el contexto del modelo.

El vocabulario de OCL es dependiente del modelo conceptual sobre el que se esté aplicando (o sea, el modelo conceptual introduce una signatura para las aserciones OCL).

Las aserciones en OCL usado en modelo conceptual siguen la siguiente estructura:

Context *contexto* **inv:** *invariante*

donde

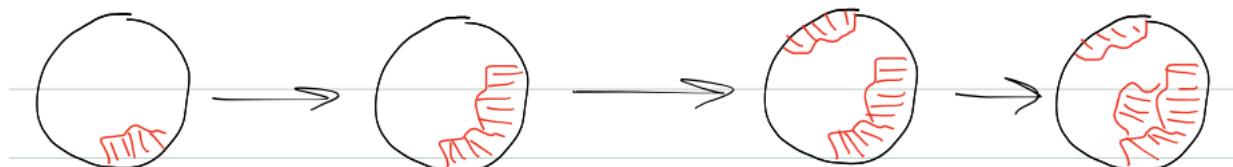
- *contexto:* define *en dónde* se espera que la aserción sea verdadera.
- *invariante:* define la aserción misma.

Semántica

Semánticamente, una descripción de modelo conceptual usando OCL está dada en los mismos términos que una descripción de modelo conceptual: conjuntos de objetos con atributos y las relaciones entre sí. La diferencia es que el OCL es notablemente más expresivo, con lo que podemos **recortar** modelos inválidos con más precisión.

Recorte o Prunning

La semántica de **recorte** o **prunning** define que a medida que agregamos restricciones y especificaciones el espacio de interpretaciones válidas se va achicando (justamente, lo estamos recortando). Tanto el modelo de operaciones (con pre- y post-condiciones) como el de objetivos (mediante objetivos de comportamiento que deban cumplirse) utilizan semántica de pruning.



Sintaxis

La sintaxis de OCL se define recursivamente a partir del modelo conceptual sobre el que se aplica. Cada clase conceptual introduce un **tipo** en el lenguaje. **Self** indica el objeto actual.

Las expresiones se construyen a partir de:

- **Constantes:** Ej. “5”
- **Variables tipadas:** Ej. “Estudiante”
- **Queries:** funciones sin efectos colaterales para especificaciones más concisas.
- **Atributos:** Ej. “Estudiante.nombre”
- **Roles:** Ej. “Estudiante.materias_aprobadas”
- **Tipos predefinidos y sus operaciones:** Ej. *tipos básicos, comparadores <, >, =, bag, set*, etc.

El lenguaje de OCL viene con tipos **paramétricos** predefinidos para modelizar **colecciones** de objetos. Algunos de estos son:

- **Collection(T):** colección genérica de elementos de tipo T .
- **Set(T):** colección *no ordenada y sin repetidos* de elementos de tipo T .
- **OrderedSet(T):** colección *ordenada y sin repetidos* de elementos de tipo T .
- **Bag(T):** colección *no ordenada y con repetidos* de elementos de tipo T .
- **Sequence(T):** colección *ordenada y con repetidos* de elementos de tipo T .

Estos tipos a su vez vienen con operaciones asociadas. Por ejemplo, supongamos que X e Y son dos instancias del tipo $collection(T)$, t de tipo T y $P(t)$ un predicado $P(t) : T \rightarrow Bool$. En ese caso, son válidas las operaciones:

- $X \rightarrow size()$
- $X \rightarrow intersection(Y)$, $X \rightarrow union(Y)$
- $X \rightarrow isEmpty()$, $X \rightarrow notEmpty()$
- $X \rightarrow includes(t)$, $X \rightarrow excludes(t)$
- $X \rightarrow includesAll(Y)$, $X \rightarrow excludesAll(Y)$
- $X \rightarrow collect(P(t))$, $X \rightarrow select(P(t))$, $X \rightarrow reject(P(t))$, $X \rightarrow forAll(P(t))$

Set, orderedSet, bag y *sequence* son subtipos de *collection* (o sea, heredan todas son operaciones), pero agregan algunas propias de cada uno. Por ejemplo,

- *Bag* agrega $X \rightarrow count(t)$
- *OrderedSet* y *sequence* agregan $X \rightarrow first()$ y $X \rightarrow last()$

La navegación por roles permite recorrer el diagrama de clases a través de las asociaciones de una determinada clase. Eso es extremadamente útil para imponer restricciones: en el lenguaje OCL $<\text{clase}>.<\text{rol}>$ retorna la **colección** de objetos que están del otro lado de la asociación. Si tiene multiplicidad 1, retorna sólo el objeto (y no la colección).

Esto induce un problema de tipos. No siempre es posible inducir sólo con mirar el diagrama de clases el tipo de expresión. $<\text{Clase}>.<\text{rol}>$ puede ser una colección o un objeto en sí mismo. Para solucionar esto, se introducen dos conceptos: el **tipo aparente** de una expresión es el que se puede deducir **estáticamente** de la firma del diagrama de clases. A diferencia de esto, el **tipo real** debe deducirse **dinámicamente** del objeto mismo. Para esto, el lenguaje OCL incluye funciones que permiten realizar esta deducción de tipos dinámica.

- $x.\text{oclIsKindOf}(t)$ devuelve *true* si el tipo real de x es subtipo de t .
- $x.\text{oclIsTypeOf}(t)$ devuelve *true* si el tipo real de x es t .
- $x.\text{oclAsType}(t)$ devuelve una referencia denotando lo mismo que x , pero casteado² con tipo aparente t .

Reglas de subtipificación

Definimos la relación $<$ de la siguiente manera: *sean T_1 y T_2 tipos correspondientes a clases T_1 y T_2 . Vale que $T_1 < T_2$ si y sólo si T_1 es subclase de T_2* . Entonces, para toda expresión de tipo T , vale que:

- $\text{set}(T) < \text{collection}(T)$
- $\text{sequence}(T) < \text{collection}(T)$
- $\text{Bag}(T) < \text{collection}(T)$
- $T < \text{OclAny}$
- Si $T_1 < T_2$ y C es *collection*, *set*, *bag* o *sequence*, entonces $C(T_1) < C(T_2)$.

Queries

Las **queries** son funciones sin efectos colaterales, usadas en el modelo conceptual para especificaciones muy concisas. Son definiciones auxiliares que no van en el diagrama de clases. Las clases conceptuales no proveen queries.

5.8.2 OCL fuera de modelo conceptual

OCL no es muy utilizado fuera del modelo conceptual (especialmente en esta materia). Sin embargo, existen otros usos, tales como:

- **Modelo de operaciones:** puede utilizarse para definir pre- y post- condiciones de operaciones o casos de uso. Se utiliza el operador **@pre**. Puede ser el lenguaje utilizado para formalizar operaciones provistas por la máquina y otros agentes.
- **Diseño:** puede utilizarse para definir pre- y post- condiciones de operaciones o casos de uso. Se utiliza el operador **@pre**.
- **Modelo de objetivos:** puede utilizarse para formalizar objetivos.

²Casting: proceso de cambiarle el tipo aparente a una variable en OCL.

5.9 Modelos de Comportamientos

Vimos el “Diagrama de Actividad” y FSMs.

En los **modelos de comportamientos** se estudian las reacciones y respuestas de un agente cuando es estimulado. Llamamos **modelos de comportamiento** a una familia de notaciones que describen comportamiento.

Formalmente, se define **comportamiento** como *el conjunto de respuestas o reacciones o movimientos hechos por un organismo en cualquier situación*. Para representar eso, se emplean cadenas de estímulos y respuestas con una determinada sintaxis y semántica. Estas cadenas se realizan una para cada agente. Para verificar si el sistema logra sus objetivos, se debe verificar la **composición** de toda cadena de estímulos y respuestas del conjunto de agentes.

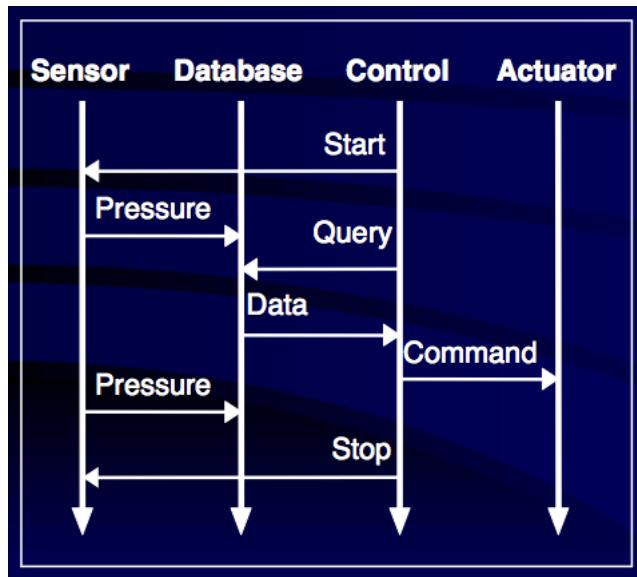
Los modelos de comportamiento tienen una semántica **generativa** (induce una forma de generar lo que se concibe) o **ejecutable**. Contrariamente al *prunning*, al agregar elementos se expande el conjunto de ejecuciones válidas del sistema.

Se categorizan en dos:

- **Basados en interacciones:** es una visión de traza global (visión omnisciente) en la que selecciono un conjunto de agentes y describo todas las posibles interacciones entre sí. No “se para” en ningún agente en particular. Su foco está en describir cómo es la interacción entre varias entidades.
- **Basados en estados:** es una visión centrada en un agente en particular, y describe su comportamiento completo. Es una visión local, sesgada a un único agente.

5.9.1 Diagramas de Secuencia

Son una de las posibles herramientas **basada en interacciones** para representar gráficamente y trabajar con modelos basados en interacciones. Provienen de las Telco. Su scope permite describir escenarios de intercambios de mensajes estableciendo posibles ordenamientos para los eventos. Poseen una semántica de órdenes parciales. Se componen por intermedio de un grafo.

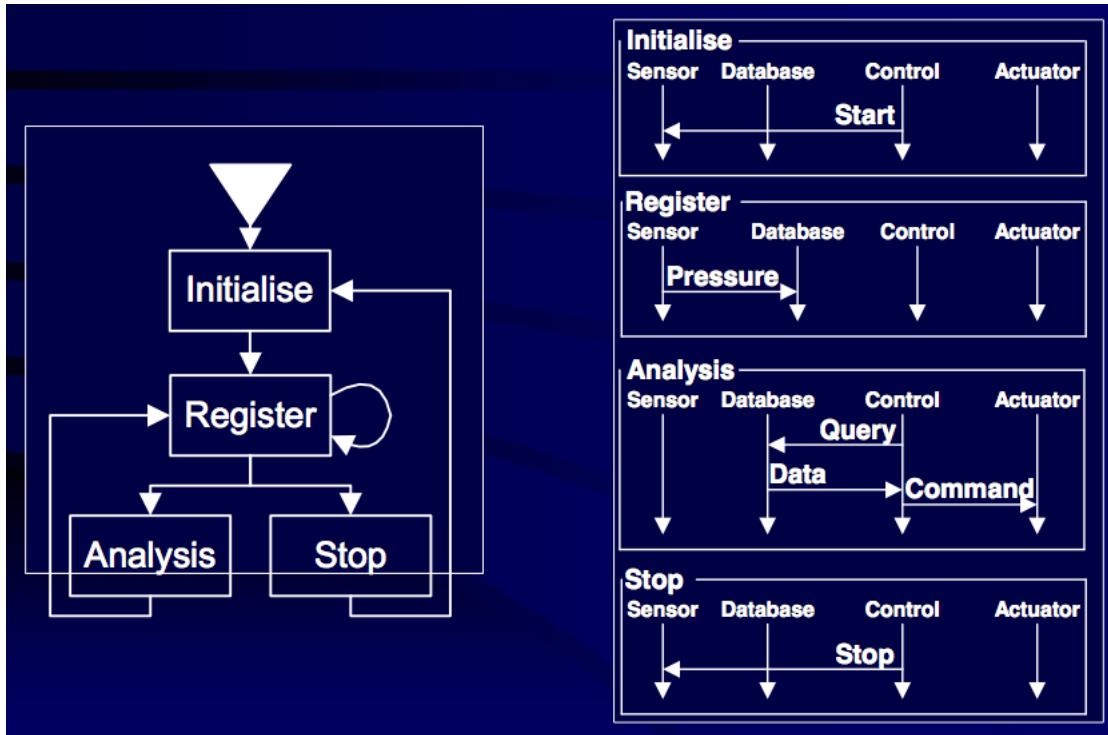


Las líneas verticales pueden representar agentes (dependiendo de para qué se esté usando el diagrama) y las horizontales representan **comunicación sincrónica** entre ellos. También se puede representar comunicación **asincrónica**, en cuyo caso el orden parcial es de eventos de envío y recepción.

HMSCs: High-Level Message Sequence Chart

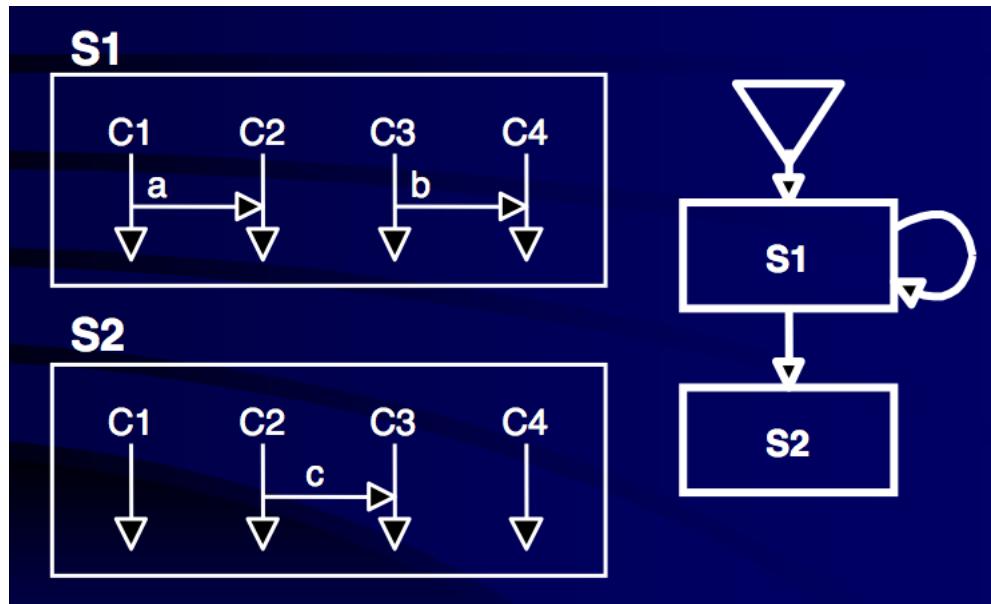
Un HMSC es una descripción gráfica de la relación entre MSCs. Establece un orden topológico entre las MSCs, permitiendo cosas como loops condicionales, etc. La forma de reconstruir el comportamiento de la máquina es realizando una **composición secuencial débil**. Pueden usarse con semántica de comunicación asincrónica.

Sincrónico: dícese de un fenómeno que ocurre en perfecta correspondencia temporal con otro evento o proceso.



El nivel de expresividad es relativamente limitado, porque no pueden expresar a todos los lenguajes regulares (por ejemplo, hay protocolos de estados finitos que no se pueden describir mediante HMSCs). Sin embargo, existen algunos lenguajes no-regulares que sí pueden ser expresados. *Temporal Model Checking* no es decidible.

Un ejemplo de una *HSMC* de lenguaje no regular (aún considerando comunicación sincrónica) es:

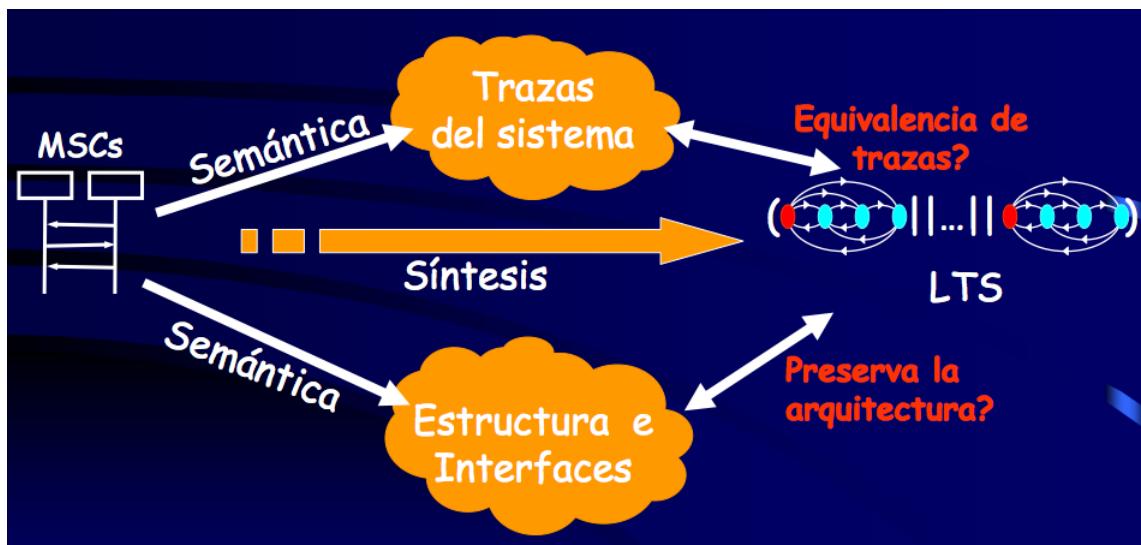


No todo puede ser representado mediante una *HMSC*. Por ejemplo, los **protocolos de estados finitos**.

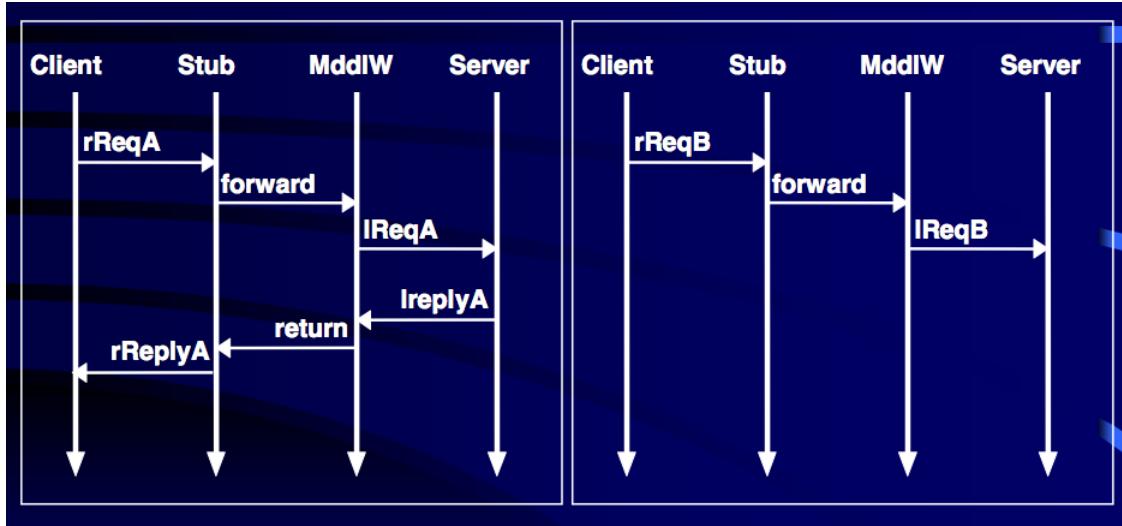
Aún si suponemos comunicación sincrónica y una *HMSC* que genera un lenguaje regular **no necesariamente** podemos armar un sistema que implementa exactamente el comportamiento especificado. Esto se puede ver en que puede generarse un **escenario implicado**.

De interaction-based a state-based

Un *MSC* (*Message Sequence Chart*) dado tiene dos interpretaciones semánticas: se pueden extraer tanto las **trazas del sistema** como las **estructuras e interfaces** de los que el sistema se vale. Pasado a un plano de **state-based modelling**, puede construirse una *FSM* (*Finite State Machine*) basándose en cada una de las *MSCs*. A este proceso se lo llama **síntesis**. Sin embargo, vale preguntarse si esta *FSM* resultante preserva la arquitectura (o sea, las estructuras e interfaces se mantienen coherentes con los *MSCs* del modelo basado en interacción) y si respeta las trazas del sistema (o sea, si existe una equivalencia entre las trazas del sistema extraídas de las *MSCs* y las de la *FSM*).



El llamado problema de la síntesis plantea que no siempre es posible observando las trazas del sistema sintetizar los componentes individuales. Por ejemplo, supongamos el siguiente caso:



Suponiendo que queremos que sólo esas trazas sean aceptables en nuestro sistema, surge el siguiente problema: cuando el *middleware* recibe un *forward*, ¿qué debe mandar? *IReqA* o *IReqB*? Dado que la respuesta depende de la interacción entre otro par de agentes, a menos que posea una visión global del sistema, el *middleware* no tiene forma de saberlo.

Es evidente que la *FSM* sintetizada puede tener trazas adicionales que no son deseadas en el sistema. O sea, vale que:

$$\text{Trazas del sistema} \subseteq \text{Trazas de la FSM}$$

Pero:

$$\text{Trazas del sistema} \not\subseteq \text{Trazas de la FSM}$$

Escenarios implicados

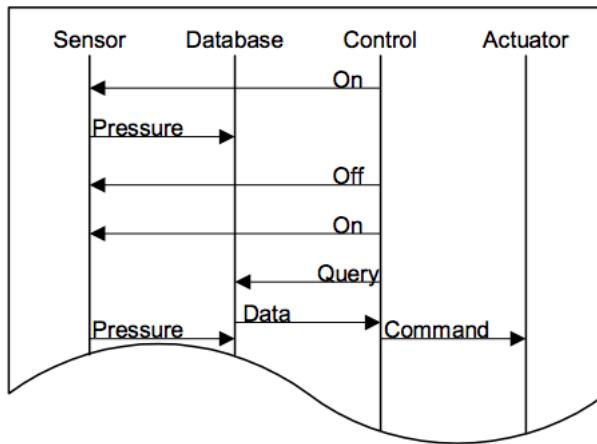
Formalmente, se define un **escenario** como un conjunto de trazas tal que, combinada con todos los otros escenarios, proveen una descripción completa del sistema. Uno en principio esperaría que dada uno un proceso correcto de síntesis, la *FSM* tenga las mismas trazas (se comporten de igual manera). Sin embargo, dado que las componentes sólo poseen una visión local del sistema puede suceder que se comporten incorrectamente en términos de trazas del sistema. Es posible que algunos escenarios se combinen en formas no esperadas y que induzcan que ciertos comportamientos no presentes en la especificación del escenario aparezcan en todas las posibles implementaciones del sistema. A estos comportamientos se los llama **escenarios implicados**.

La existencia de escenarios implicados es un indicador de comportamiento no deseado en el sistema. Puede significar que:

- Un escenario aceptable no fue tomado en cuenta (o sea, la especificación del escenario está incompleta).
- Un escenario inaceptable fue tomado en cuenta (o sea, la especificación del escenario es incorrecta). Esto denota una arquitectura incorrecta o muy abstracta.

Existen métodos de detección de escenarios implicados, que se basan en construir modelos locales, componerlos y detectar trazas que no están incluidas en un modelo global de la especificación.

Ejemplo:



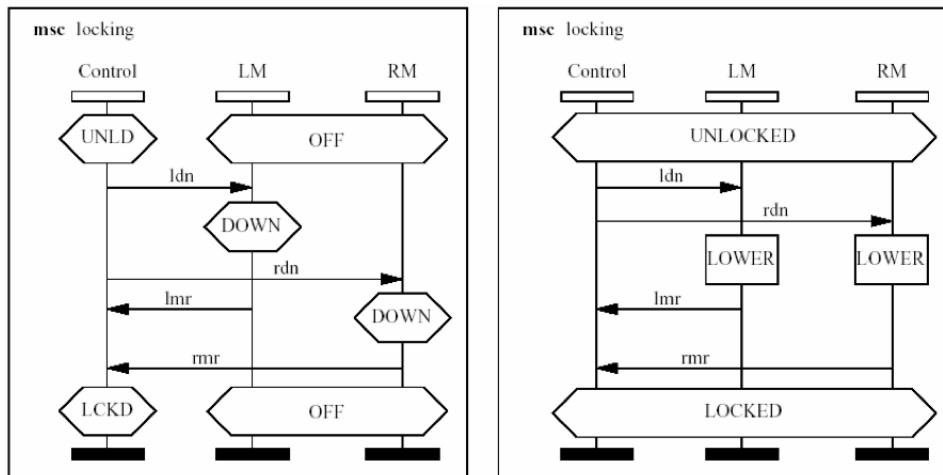
La figura muestra una ejecución posible (aunque incorrecta) de la máquina. Sin embargo, cada componente se está comportando correctamente acorde con su especificación: El *Sensor*, *Control* y *Actuator* están siguiendo los escenarios *Initialise*, *Register*, *Terminate*, *Initialise*, *Analysis* y *Register*. Sin embargo, *Database* está siguiendo *Initialise*, *Register*, *Analysis* y *Register*.

Esto se debe a que *Control* no puede ver cuando *Sensor* registró cosas en *Database*, entonces si tiene que pedir información a *Database* **después** de que haya sido submiteada por el *Sensor*, debe esperar que *Database* active y desactive los pedidos apropiadamente. Sin embargo, como *Database* no puede saber cuando el sensor está o no activado, no puede hacerlo en los momentos correctos.

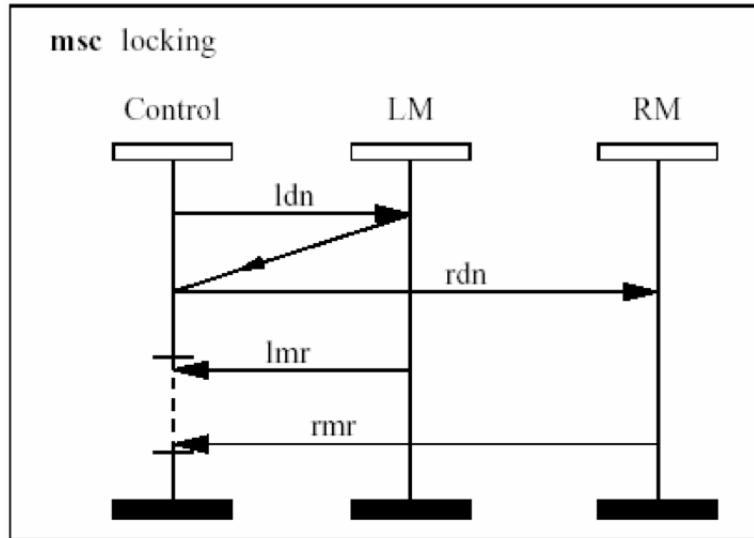
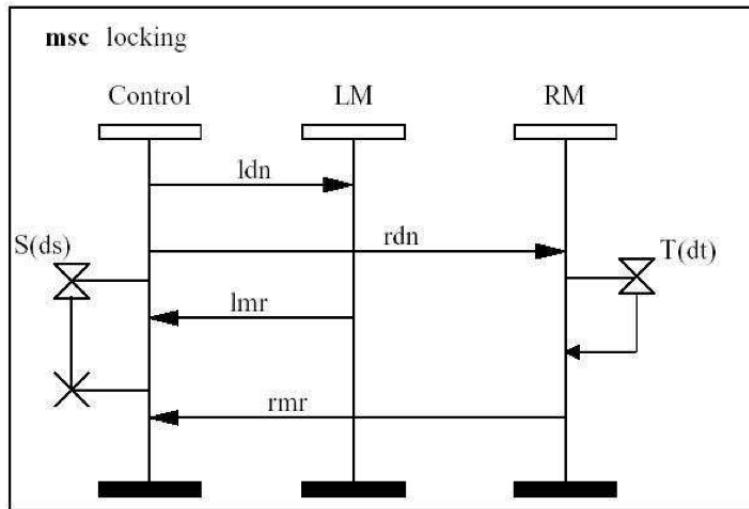
Extensiones

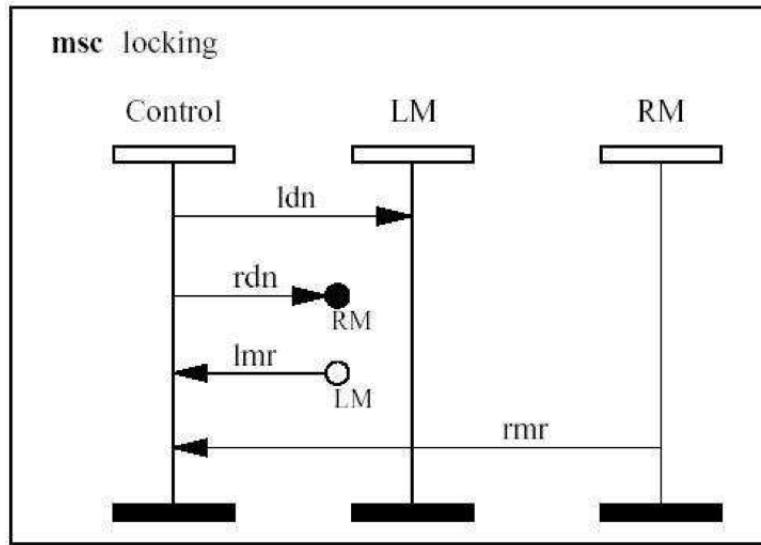
A continuación se presentan algunas extensiones en diagramas de secuencia:

Condiciones y estados

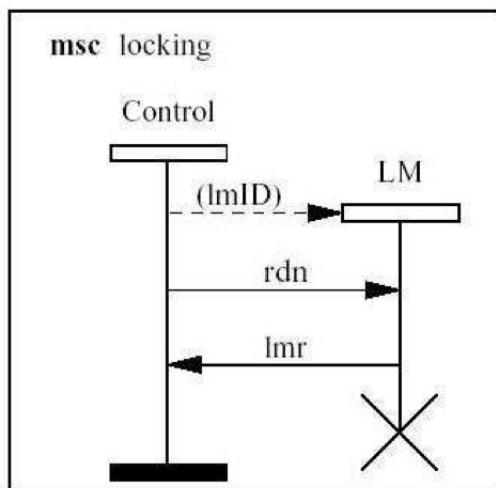


Co-Regiones y Ordenamientos generales

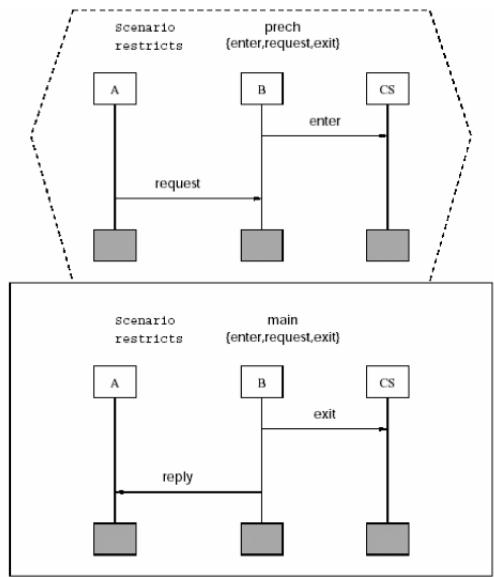
**Timers y TimeOuts****Mensajes Perdidos y Encontrados**



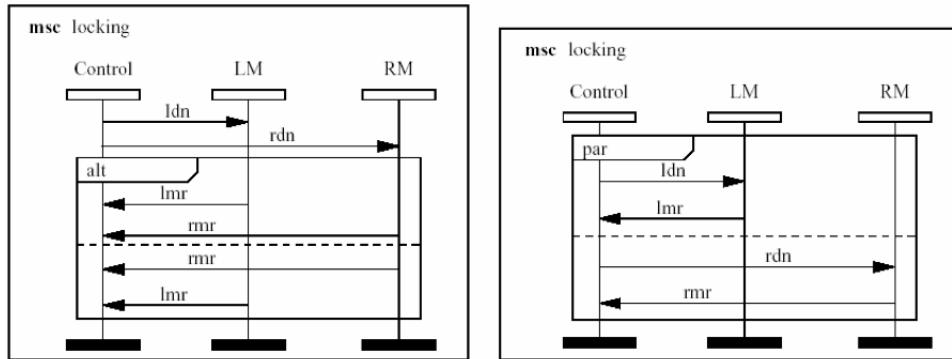
Dynamic Creation and Destruction



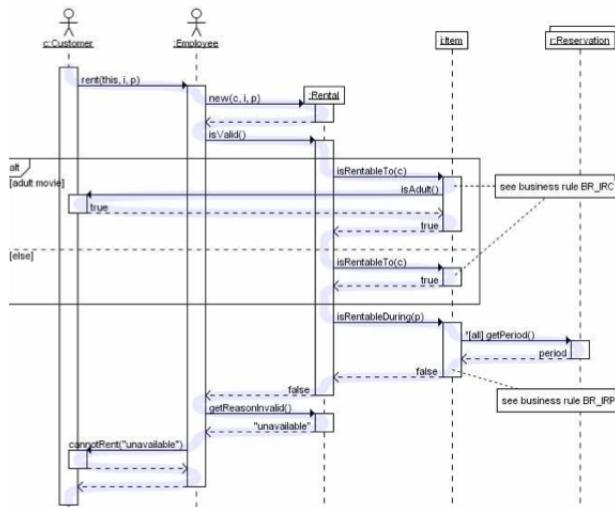
Gatillo o Trigger



Alternativas, Ciclos y Paralelismo



Barras de Activación y Returns



5.9.2 Collaboration Diagrams

Es una herramienta **basada en interacción** para expresar gráficamente comportamiento.



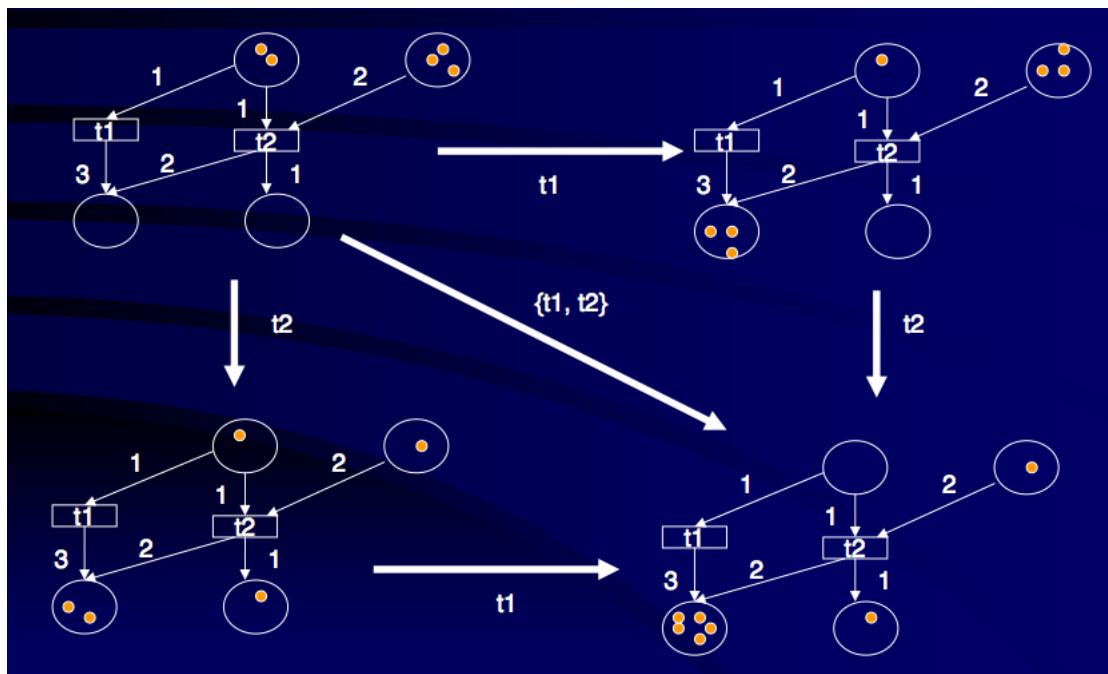
Sus características principales son:

- Se enfoca en la estructura del sistema.
- Se numeran las interacciones, lo que permite establecer órdenes totales.
- No es apto para trazas complejas.

5.9.3 Redes de Petri

Las redes de Petri son un modelo de comportamiento que poseen características de modelo basado en interacciones y basado en estados. Estructuralmente, son un digrafo bipartito con pesos asociados a los aristas. Los dos conjuntos de la bipartición del grafo se denominan *places* y *transiciones*. Los *places* que alimentan a una transición se denominan *preset*, mientras que los que son alimentados por una transición se denominan *poset*. Los nodos del tipo *transiciones* están etiquetados.

Cada nodo de tipo *place* puede contener 0 o más *tokens*. Una asignación de *tokens* a *places* se denomina *marking*. Existe un determinado *marking inicial*. Una *transición* t puede dispararse sólo cuando la cantidad de *tokens* en su *preset* supera el peso de la arista que lo une con t . Una vez que se realiza esa *transición* t , a cada uno de los *posets* de esa transición se le suma una cantidad de *tokens* igual al peso de la arista que une t con el correspondiente *poset*.



Se dice que un *marking* es **alcanzable** si es el resultado de una secuencia de transiciones que resulten en ese *marking*, partiendo desde el *marking inicial*.

A su vez, se dice que una red de Petri es ***n-safe*** si no es alcanzable ningún *marking* que contenga más de *n tokens* en un *place*.

Existen herramientas automáticas que permiten analizar propiedades e invariantes de las redes de Petri, tales como existencia de *deadlocks*, alcanzabilidad, etc. Algunas de estas herramientas se basan en propiedades del álgebra lineal.

Características

- Formal, operacional, gráfico, refutable.
- Scope: concurrencia, control (uso similares a FSM).
- Semántica: sistema de transición u ordenes parciales.
- Mecanismo de descomposición: no standard.
- Al igual que las FSM, muchos tipos de análisis posibles.
- Relación con FSM: si es *n-safe*, hay una relación de conversión en ambos sentidos.
- Muchas variantes: tiempo, probabilidades, datos, etc.

5.9.4 Diagramas de Actividad

Los diagramas de actividad son una versión *user-friendly* de las redes de Petri. En ellos, cada estado tiene una semántica particular. El conjunto de todos los estados (con su respectiva semántica) presenta un **flujo de actividades**. Los estados están unidos por medio de flechas que representan un **flujo de control**. Poseen ciertas características destacables:

- Tienen soporte para asignación de responsabilidades, así como para concurrencia de actividades, entre otras.
- Son una visión global y temprana de *proceso*.
- Proveen una descripción de paralelismo y coordinación.
- Aplicables a diseño y requerimientos.

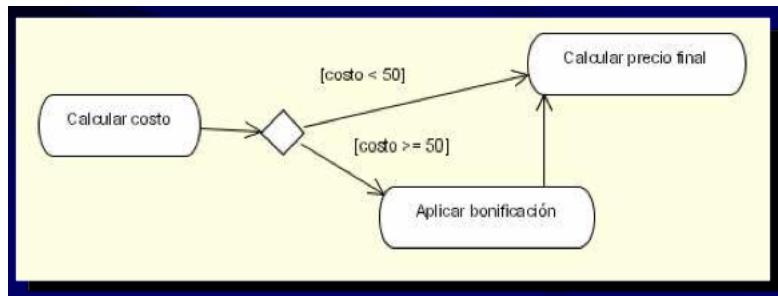
Permite vincular casos de uso:

- Mostrando la secuencia de un caso de uso.
- Reflejando las interacciones entre varios casos.
- Modelando el comportamiento interno del sistema.

Decisión

Un diagrama de actividad expresa una decisión cuando una condición es usada para indicar diferentes transiciones posibles que dependen de un valor booleano. El ícono provisto para una decisión es un diamante, con una o más flechas entrantes y con dos o más flechas salientes, cada una etiquetada por una condición diferente y sin evento que la dispare.

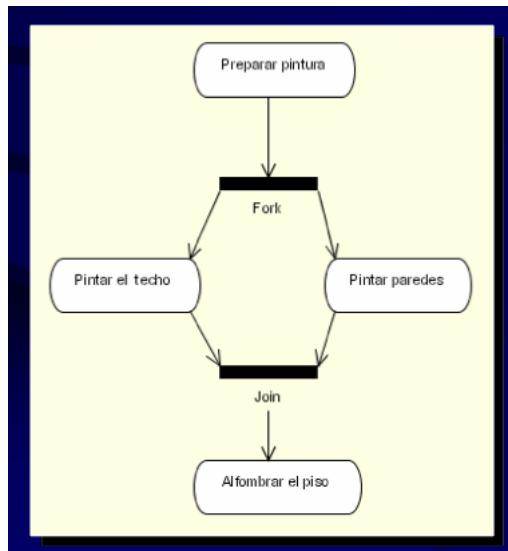
Todos los posibles valores para la condición deben aparecer en las transiciones salientes: debe ser completo y determinístico. Si no se especifican las condiciones en las flechas salientes, puede interpretarse como una elección aleatoria equiprobable en la cantidad de flechas salientes.



Fork y Join

Permiten establecer concurrencia entre acciones. El operador de *Fork* separa la ejecución en dos o más ramas. Obligatoriamente todas las ramas se siguen y no es posible hacer ningún tipo de asunción sobre el orden en el que se van a realizar, puesto que no está especificado.

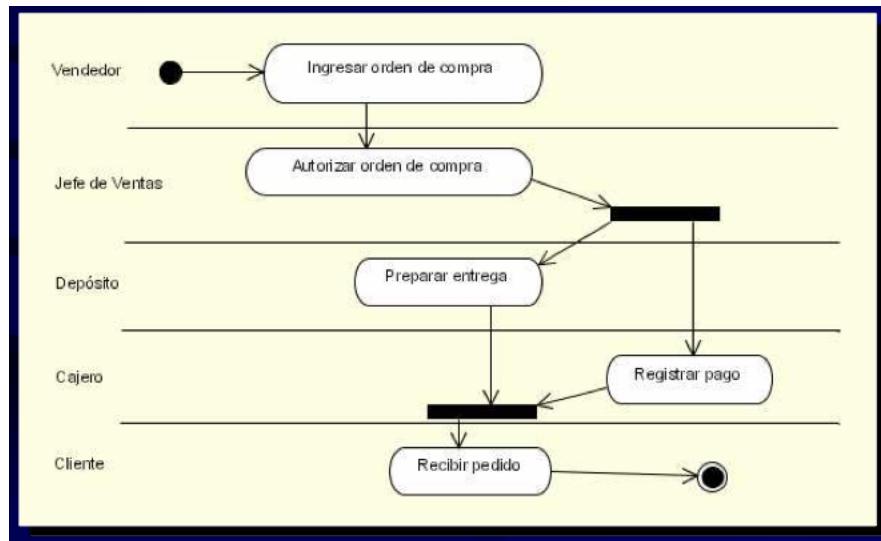
Después de un operador *fork*, el hijo de ejecución se vuelve a unir con uno o más operadores *join*. Estos operadores toman dos hilos de ejecución distintos y lo convierten en uno sólo. Para que se continúe la ejecución en un *join* es necesario que **todos** los hilos de ejecución que desembocan en ese *join* hayan llegado a él.



Andariveles

Los andariveles permiten la asignación de responsabilidad en los diagramas de actividad. Las acciones se organizan en andariveles, que corresponden uno a cada “unidad organizacional” (en general existe una relación entre estos y los actores de los casos de uso). Si una actividad cae dentro del andarivel de un actor t , entonces es ese actor t quien tiene la responsabilidad por la susodicha acción.

También puede verse una analogía entre cada estado del diagrama de actividad con un caso de uso del modelo de operaciones.



5.9.5 Diseño vs Requerimientos

Los modelos de comportamiento pueden usarse para describir el diseño:

- Los eventos pueden ser llamadas a métodos, procedimientos, mensajes, interrupciones, etc.
- Las entidades pueden ser procesos, módulos, objetos, threads, etc.

A su vez, también pueden ser usados para describir el dominio.

6 Máquinas de estado

Según las diapositivas, es un área con un “despelote terminológico”. Una **máquina de estado** es un término genérico que denota una familia de notaciones que tienen **estados** (discretización del estado), **transiciones** (relación “sigue a”) y **etiquetas** (enriquecen la noción de estado y transición).

Ejemplos:

- Statechart.
- *FSM* (Finite State Machines).
- *LTS* (Labelled Transition System).
- Automata.
- Timed Automata.

Las máquinas de estados estructuran el mundo de acuerdo con la relación “*sigue a*”, permitiendo representar secuencias de estado de manera compacta. Esto genera que tengan numerosos usos, tales como:

- Descripción del problema y la solución.
- En ingeniería de requerimientos, se usan para graficar el comportamiento de agentes y estados de entidades pasivas.
- En diseño, se usan para protocolos, arquitecturas de software, estados de objetos y clases, etc.

Diferencias entre Diagrama de Actividad y FSM: la manera de cómo describir el comportamiento. En DA se ve fácil porque la semántica importante está en los estados. En FSM, en cambio, el foco está en las transiciones. Por esta razón puedo escribir para una máquina otra que tenga las mismas trazas pero con diferente cantidad de estados. Además, en DA se podía interpretar paralelismo o simultaneidad con un Fork; en realidad, daba una idea de independencia de acciones y no conocemos su orden. En FSM se podrá decir esto explícitamente con composición en paralelo: se ejecutan etiquetas compartidas si hay sincronización, todas las máquinas con esa etiqueta deben estar listas, por eso hay simultaneidad.

6.1 LTS: Labelled Transition Systems

Informalmente, un *LTS* es un tipo de máquina de estados. Se caracteriza por tener transiciones etiquetadas (eventos instantáneos) y que los estados sean semánticamente nulos, excepto uno que indica el estado inicial de la máquina (pueden etiquetarse por comodidad de lectura o referencia, pero no es relevante a los efectos de la semántica). Se utiliza para **describir comportamiento de entidades y razonar sobre el comportamiento emergente resultante de su ejecución concurrente**.

Formalmente, se define una LTS de la siguiente manera:

Sea $Estados$ el universo de estados posibles y Act el universo de acciones observables. Además, definimos $Act_{\tau,\in} = Act \cup \{\tau\} \cup \{\in\}$, siendo τ la acción *no observable* o *silenciosa* y \in la acción *nula*.

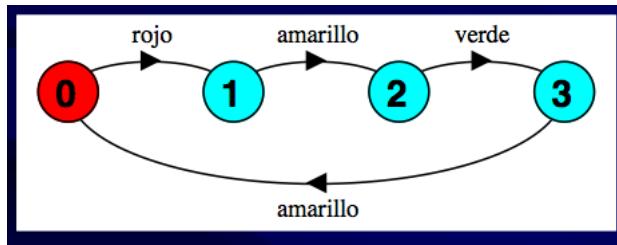
Un **LTS** es una tupla $P = < S, L, \Delta, s_0 >$, donde

- $S \subseteq Estados$ es un conjunto de estados finito.
- $L \subseteq Act_{\tau,\in}$ es un conjunto de etiquetas.
- $\Delta \subseteq (S \times L \times S)$ es un conjunto de transiciones etiquetadas.
- $s_0 \in S$ es el estado inicial.

Definimos el **alfabeto de comunicación** de P como $\alpha(P) = L \setminus \{\tau, \in\}$

Observación: L también es llamado \sum o \sum^* si incluye la transición \in .

Ejemplo:



Observación: El conjunto de etiquetas de un *LTS* no tiene por qué ser minimal. Pueden existir etiquetas que no se usen y sin embargo sí siguen categorizándose como etiquetas compartidas y, como tales, vale la regla de paralelización de etiquetas compartidas.

6.1.1 Semántica

Ejecución

Sea $P = (S, L, \Delta, q)$ una *LTS*. Se define una **ejecución** de P como una secuencia $w = q_0, l_0, q_1, l_1, \dots$ de estados $q_i \in S$ y transiciones $l_i \in L$ tal que

- $q_0 = q$
- $\forall 0 \leq i \leq \left\lfloor \frac{w}{2} \right\rfloor : (q_i, l_i, q_{i+1}) \in \Delta$

Proyección

Sean w una palabra w_0, w_1, w_2, \dots y A un alfabeto. Se define la **proyección** de w sobre A (que se nota $w|_A$) como el resultado de eliminar de w todos los elementos w_i tales que $w_i \notin A$.

Trazas

Informalmente, una traza de un *LTS* es, dada una ejecución, quitarle los estados (o sea, quitar las etiquetas). No me importa por qué estado pasé. Lo único que me importa son los estímulos y reacciones de un agente, no el agente mismo.

Formalmente, sea $P = (S, L, \Delta, q)$ una LTS. Una palabra w del alfabeto $\alpha(P)$ es una **traza** de P si existe una ejecución w' tal que $w = w'|_{\alpha(P)}$.

A su vez, definimos $tr(P)$ como el conjunto de todas las trazas de la LTS P .

$$tr(P) = \{w | w \text{ es traza de } P\}$$

Otra definición alternativa es que

$$\begin{aligned} Tr(s) &= \{\alpha | \alpha \in \sum^* \text{ y } s \xrightarrow{a} s'\} \\ Tr(L) &= Tr(s_0) \end{aligned}$$

Decimos que un LTS satisface una propiedad si y sólo si **todas** sus trazas la satisfacen. Sin embargo existen propiedades que no se pueden examinar sobre las trazas sino sobre el **branching** (semántica arbórea). Un ejemplo de esto son los casos de uso.

Se define el conjunto de estados **alcanzables por α** de la siguiente manera:

$$Alc(s, \alpha) = \{s' | s \xrightarrow{\alpha} s'\}$$

Se define que un LTS L tiene **comportamiento finito** si existe un $n \in \mathbb{N}$ tal que para toda traza $\sigma \in Tr(L)$, la longitud de σ es menor que n .

Una traza informalmente se puede describir como una tira de etiquetas o cadena de transiciones.

Transición o Evolución

Sea $P = (S, L, \Delta, q)$ una LTS. Decimos que P **transita o evoluciona** con una etiqueta l en un nuevo LTS P' si $P' = (S, L, \Delta, q')$ y $(q, l, q') \in \Delta$. Se nota $P \xrightarrow{l} P'$.

6.1.2 Concurrencia vs Paralelismo

Concurrencia	Paralelismo
Procesamiento lógicamente simultáneo	Procesamiento físicamente simultáneo
No necesariamente implica múltiples unidades de procesamiento ³	Involucra múltiples unidades de procesamiento

Tanto concurrencia como paralelismo requieren acceso controlado a recursos comunes.

6.1.3 Composición en paralelo

Informalmente, la **composición en paralelo** de dos LTS construye una nueva LTS que permite casi todas las combinaciones posibles de las acciones de los dos alfabetos. Las acciones que pertenezcan a ambos alfabetos deben ser ejecutadas por ambos procesos simultáneamente. Estas acciones compartidas sincronizan la ejecución de los dos procesos. Es por esto que se dice que las **interacciones o acciones compartidas restringen el comportamiento global**.

Semánticamente, si suponemos que cada LTS modela un agente o proceso y ese agente quiere hacer alguna acción que nadie más monitorea, entonces la puede hacer sólo, en cualquier momento y todo el resto queda igual. Si, en cambio, esa acción está en la interfaz de otro agente/proceso, entonces tienen que hacerla en conjunto.

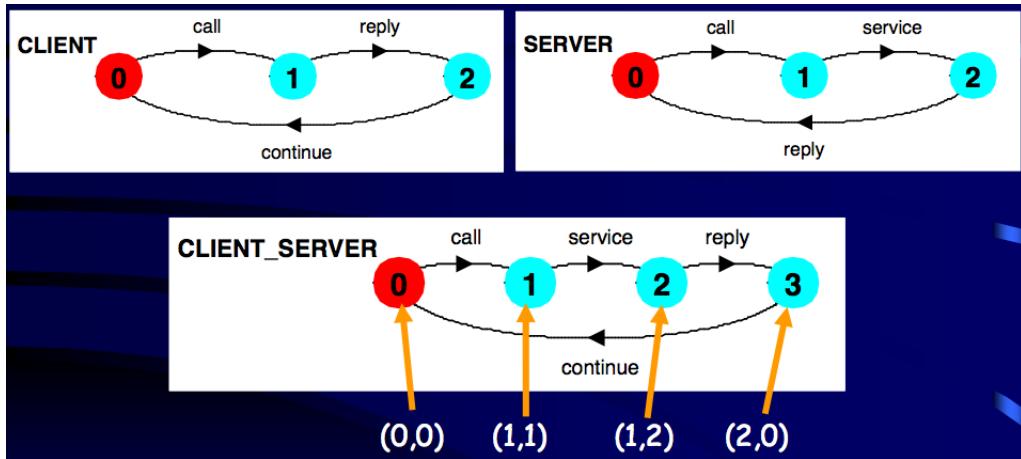
Formalmente, la **composición en paralelo** es una función $\parallel : LTS^n \rightarrow LTS$.

Sean $P_1 = (S_1, L_1, \Delta_1, q_1)$ y $P_2 = (S_2, L_2, \Delta_2, q_2)$ dos LTS. La **composición en paralelo** de P_1 y P_2 (notada $P_1 \parallel P_2$), es un nuevo LTS $(S_1 \times S_2, L_1 \times L_2, \Delta, (q_1, q_2))$, donde Δ es la relación más pequeña que satisface:

³Si hay una sola unidad de procesamiento, es necesario utilizar ejecución *interleaved*.

- $\frac{P_1 \xrightarrow{l} P'_1}{P_1 || P_2 \xrightarrow{l} P'_1 || P_2} \quad (l \notin \alpha(P_2))$
- $\frac{P_2 \xrightarrow{l} P'_2}{P_1 || P_2 \xrightarrow{l} P_1 || P'_2} \quad (l \notin \alpha(P_1))$
- $\frac{P_1 \xrightarrow{l} P'_1 \quad P_2 \xrightarrow{l} P'_2}{P_1 || P_2 \xrightarrow{l} P'_1 || P'_2} \quad (l \neq \tau)$

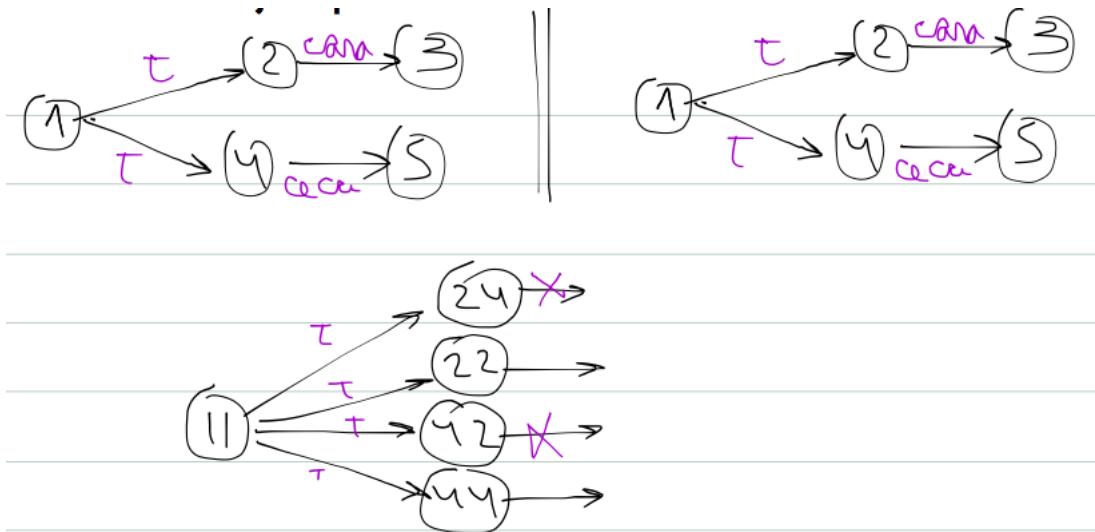
Ejemplo:



Leyes Algebráicas de composición

- Comutatividad: $(P||Q) = (Q||P)$
- Asociatividad: $(P||(Q||R)) = ((P||Q)||R) = (P||Q||R)$

Observar que la propiedad de reflexividad ($P||P = P$) **no** vale en el caso general, como se puede ver en el siguiente contraejemplo:



Sin embargo, **sí** vale para *LTS* determinísticos.

6.1.4 Modelado de Concurrencia

Para el análisis con *LTS* asumimos que las máquinas pueden funcionar a velocidades arbitrarias, abriendo el tiempo. Para modelar concurrencia se supone un orden arbitrario de acciones de los distintos procesos (se hace **interleaving** de acciones pero se preservan las acciones de cada proceso). Esto trae como resultado un modelo general, independiente de la política de scheduling (modelo de ejecución **asincrónico**).

O sea no se asume nada sobre la velocidad individual de los procesos ni del scheduler que los gobierna.

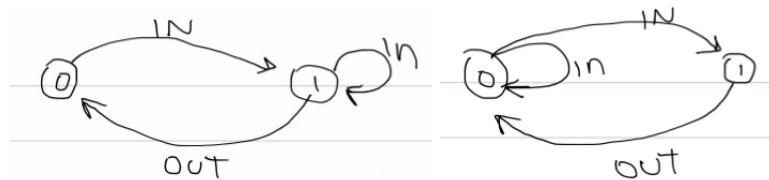
6.1.5 LTS no determinístico

Un *LTS* se dice **no-determinístico** si tiene al menos un estado con más de una transición etiquetada con la misma acción. Esto permite abstraer los detalles involucrados en la toma de una decisión.

Formalmente, un *LTS* es **determinístico** si $\forall \sigma \in \sum^*, \#Alc(s_0, \sigma) \leq 1$.

Toda máquina de estados no determinística puede convertirse en una determinística con las mismas trazas. Pero puede tomar una cantidad exponencial de estados.

Ejemplo:

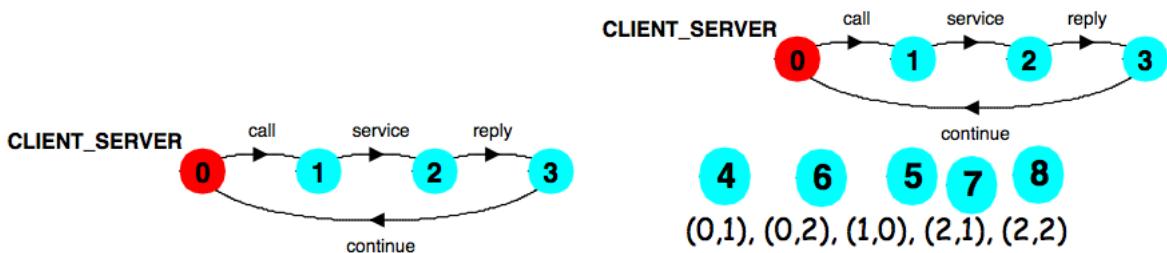


6.1.6 Propuestas para la igualdad de LTS

¿Cómo se establece la comparación entre dos *LTS*? Hay varias propuestas:

Propuesta 1: Isomorfismo

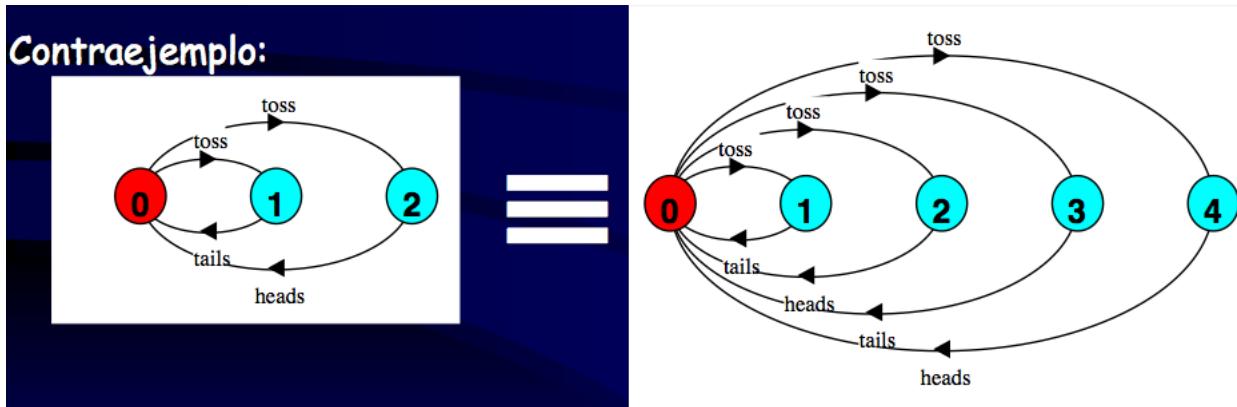
“Dos *LTS* son equivalentes si sus grafos subyacentes son isomorfos”. Esta propuesta falla como se puede ver en el siguiente contraejemplo:



Claramente queríamos que estos dos *LTS* fueran equivalentes (porque los estados tienen semántica nula), pero sus grafos subyacentes no son isomorfos, por lo que esta definición no alcanza.

Propuesta 2: Isomorfismo sin estados no alcanzables

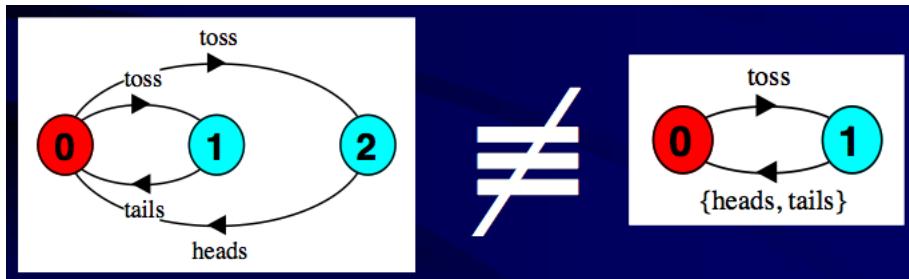
“Dos *LTS* son equivalentes si al quitar sus estados no alcanzables son isomorfos”. Esta propuesta falla como se puede ver en el siguiente contraejemplo:



En este caso, queríramos estos dos *LTS* fueran equivalentes, pero sus grafos subyacentes, aún retirando estados no alcanzables, no son isomorfos.

Propuesta 3: Igualdad de Trazas

“Dos *LTS* son equivalentes si definen el mismo conjunto de trazas”. Esta propuesta falla como se puede ver en el siguiente contraejemplo:



En este ejemplo no deberíamos querer ver ambas máquinas como iguales porque la de la izquierda es no-determinista y la derecha es determinista.

Claramente, la equivalencia por trazas es una equivalencia demasiado gruesa (*coarse*) o débil (*weak*). Puede ser que una composición de dos *LTS* “iguales” por trazas con la misma *LTS* las vuelva distinguibles.

Igualmente, la igualdad de trazas en **LTS determinísticas** sí es lo mismo que la bisimulación fuerte.

¿Qué buscamos?

Para la equivalencia de *LTS* estamos buscando una relación que verifique:

- Ser de equivalencia:
 - Reflexiva: $P \equiv P$
 - Transitiva: $(P \equiv Q \wedge Q \equiv R) \Rightarrow (P \equiv R)$
 - Simétrica: $(P \equiv Q) \Rightarrow (Q \equiv P)$
- Ser abstracta con respecto a:
 - Número de estados del modelo.
 - Estructura del modelo.
 - Trazas del modelo.
- Coincidir con nociones empíricas.
- Mantener coherencia con nuestro lenguaje de especificación (por ejemplo, $(P \equiv Q) \Rightarrow P||R \equiv Q||R$).

6.1.7 Bisimulación

Sea P el universo de todos los LTS . Una relación binaria $R \subseteq PxP$ es una **bisimulación fuerte** si y sólo si cuando $(P, Q) \in R$ entonces para cada $a \in Act$:

- $(P \xrightarrow{a} P') \Rightarrow (\exists Q' : Q \xrightarrow{a} Q' \wedge (P', Q') \in R)$
- $(Q \xrightarrow{a} Q') \Rightarrow (\exists P' : P \xrightarrow{a} P' \wedge (P', Q') \in R)$

Sean P y Q dos LTS . Se define que P y Q son **fueramente bisimilares** (y se nota $P \sim Q$) si y sólo si existe una bisimulación fuerte R tal que $(P, Q) \in R$.

$$\sim = \bigcup\{R | R \text{ es una bisimulación fuerte}\}$$

De estas dos definiciones se puede extraer una versión “compacta”:

$$P \sim Q \text{ si para cada acción } a \in Act$$

- $(P \xrightarrow{a} P') \Rightarrow (\exists Q' : Q \xrightarrow{a} Q' \wedge P' \sim Q')$
- $(Q \xrightarrow{a} Q') \Rightarrow (\exists P' : P \xrightarrow{a} P' \wedge P' \sim Q')$

Informalmente, M_2 simula una máquina M_1 si para toda etiqueta a vale que: M_1 puede hacer $a \Rightarrow M_2$ puede hacer a . Las máquinas M_1 y M_2 son bisimilares cuando M_1 simula a M_2 y viceversa.

No-bisimilaridad

Para demostrar la bisimilaridad entre dos LTS P y Q alcanza con dar la relación de bisimulación que se utiliza. Sin embargo, para demostrar la no-bisimilaridad se debe enumerar todas las relaciones binarias y mostrar que ninguna contiene a (p_0, q_0) (siendo p_0 el estado inicial de P y q_0 el de Q) y es una bisimulación fuerte. El problema de esto es que es notablemente costoso. Es por esto, que es más frecuente utilizar un **juego** que permite determinar sencillamente la bisimilaridad (o falta de) entre dos LTS .

El juego consiste en un **atacante**, que intenta probar que $P \not\sim Q$ y un **defensor**, que intenta probar que $P \sim Q$. Cada **ronda** del juego tiene una **configuración**, un par ordenado (S, T) . El juego comienza con la configuración (P, Q) .

En cada ronda, el atacante elige uno de los procesos de la configuración actual y hace una movida por alguna transición con etiqueta $a \in A$. El defensor debe responder haciendo una movida etiquetada por a en el otro proceso. Si un jugador no puede mover, el otro gana. Si el juego sigue infinitamente, gana el defensor (o sea, $P \sim Q$).

- P y Q son fueramente bisimilares si y solo si el defensor tiene una estrategia ganadora universal empezando desde la configuración (P, Q) .
- P y Q no son fueramente bisimilares si y solo si el atacante tiene una estrategia ganadora universal empezando desde la configuración (P, Q)

6.1.8 Congruencia

Dado un contexto para P , queríamos poder cambiar P por un proceso Q equivalente sin alterar el sistema. Se define entonces una **equivalencia** entre P y Q como una congruencia en $P \equiv Q \Rightarrow C(P) \equiv C(Q)$. Dos procesos son equivalentes si un observador externo no los puede distinguir.

Equivalencia

$$\begin{array}{c} A \sim B \\ f \downarrow \qquad \downarrow f \\ f(A) \equiv f(B) \end{array}$$

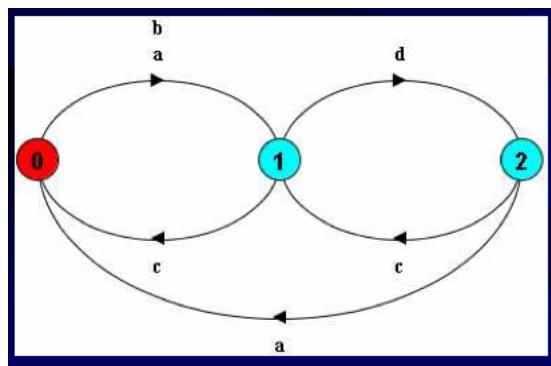
\sim es bisimulación

f es construcción de árbol de ejecución

\equiv es isomorfismo de árboles

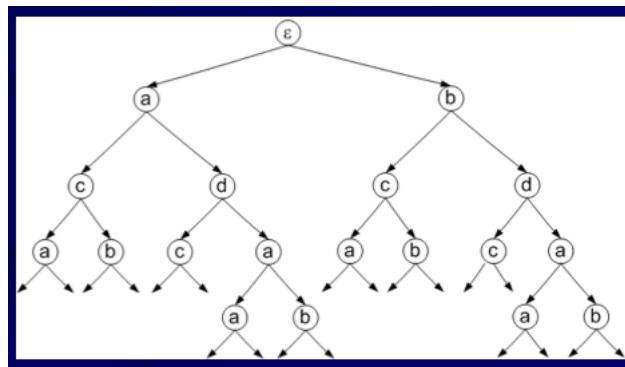
6.1.9 Semántica lineal vs semántica arbórea

Plantear la equivalencia de trazas induce una semántica lineal: un *LTS* queda caracterizado por un conjunto de secuencias.



Para algunas aplicaciones, la semántica lineal es más apropiado (por ejemplo, en situaciones donde lo más intuitivo es dar propiedades de las trazas).

A diferencia de esto, plantear una equivalencia por bisimulación induce una semántica arbórea: un *LTS* queda caracterizado por el árbol de ejecución resultante de desdoblar sus ciclos.



Para otras aplicaciones, es necesario utilizar semántica arbórea (por ejemplo, para *composición en paralelo, casos de uso* o cualquier propiedad de tipo *branching*).

Existen ciertas lógicas temporales lineales y branching que permiten predicar sobre conjuntos de trazas y estructuras arbóreas.

O sea, equivalencia por trazas pide igualdad de trazas; mientras que equivalencia por bisimulación pide isomorfismo de árboles de ejecución.

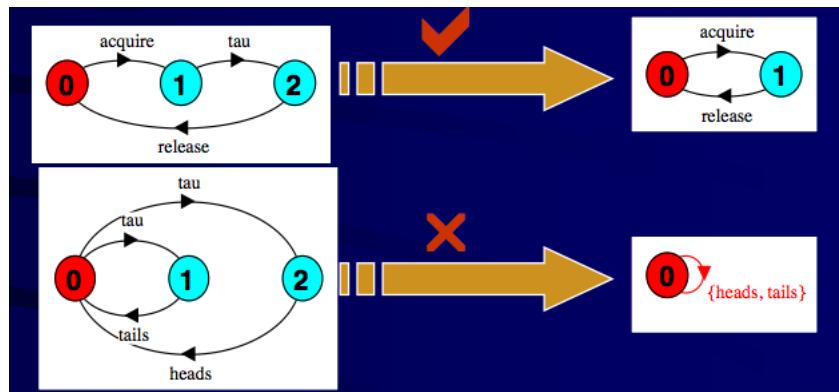
6.1.10 Tau τ

Cómo se vio en su definición, los *LTS* vienen equipados con una etiqueta especial, τ , que representa una acción interna, no observable⁴ y no controlable⁵ desde el mundo exterior. Es muy utilizada para:

- Impedir sincronización externa (por ejemplo, si dos máquinas tienen *timeout*, tenderían a sincronizarse. Con τ eso no pasa).
- Abstraer acciones internas, reduciendo la complejidad.

6.1.11 Bisimulación considerando τ

Considerando la etiqueta τ y su comportamiento, es necesario reconsiderar la noción de bisimulación. Es necesario que la relación de bisimulación “elimine” las transiciones etiquetadas por τ , porque al ser acciones internas, no deberían tener inferencia en la relación.



Es por esto que definimos las **transiciones observacionales débiles**:

$$\xrightarrow{a} = \begin{cases} (\xrightarrow{\tau})^* \circ \xrightarrow{a} \circ (\xrightarrow{\tau})^*, & \text{si } a \neq \tau. \\ (\xrightarrow{\tau})^*, & \text{si } a = \tau. \end{cases} \quad (1)$$

Esto nos permite definir un concepto de **bisimilaridad débil**. Intuitivamente, “si P hace la acción a , entonces Q lo debe imitar haciendo una acción a y una cantidad arbitraria de τ ”.

Formalmente, sea P el universo de todos los *LTS*. Una relación binaria $R \subseteq P \times P$ es una **bisimulación débil** si y sólo si cuando $(P, Q) \in R$, entonces para cada acción $a \in Act \cup \{\tau\}$:

- $(P \xrightarrow{a} P') \Rightarrow (\exists Q' : Q \xrightarrow{a} Q' \wedge (P', Q') \in R)$
- $(Q \xrightarrow{a} Q') \Rightarrow (\exists P' : P \xrightarrow{a} P' \wedge (P', Q') \in R)$

Análogamente, sean P y Q dos *LTS*. Se define que P y Q son **débilmente bisimilares** (y se nota $P \approx Q$) si y sólo si existe una bisimulación débil R tal que $(P, Q) \in R$.

$$\approx = \bigcup \{R | R \text{ es una bisimulación débil}\}$$

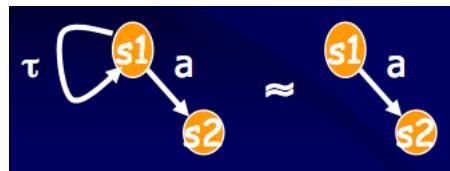
⁴Una acción es **observable**, si puede ser usada para comprobar la respuesta del sistema frente a un estímulo.

⁵Una acción es **controlable**, si puede ser usada para estimular al sistema.

Propiedades de bisimulación débil

$\approx\dots$

- ...es una relación de equivalencia.
- ...tiene un juego que la caracteriza (es igual que el anterior, pero el defensor puede hacer \Rightarrow).
- ...**bisimilaridad fuerte implica bisimilaridad débil** ($\sim\subseteq\approx$).
- ...es una congruencia con respecto al operador de composición en paralelo.
- ...abstira los loops sobre τ .



Modelado con LTS

Un *LTS* puede modelar un **agente**. Los eventos son fenómenos monitoreables o controlables, mientras que las trazas o árboles de ejecución describen la relación temporal esperada entre lo monitoreable y lo controlable (la sincronización de *LTS* representa la relación *monitoreabilidad-controlabilidad*).

Un *LTS* puede modelar una **entidad pasiva**. Eventos son fenómenos aplicados a la entidad (i.e. operaciones sobre o que afectan la entidad). Explica el protocolo de uso esperado de la entidad.

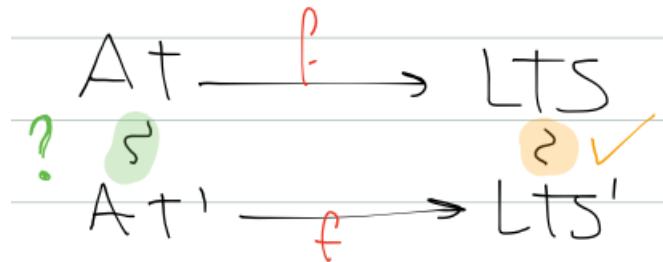
6.2 Autómatas temporizados

Los *LTS* tienen serios problemas de expresividad a la hora de trabajar con los tiempos que transcurren entre o durante las acciones. Es por esto que se aumenta la definición de *LTS* de la siguiente manera:

- Un **estado** del sistema es un par (s, v) con:
 - $s \in S$
 - $v \in V$
 - $Is[v] = true$
- La arista $a = < s, l, \Psi, \alpha, s' >$ puede ser atravesada desde el estado (s, v) sólo si v satisface la condición Ψ .
 - El estado resultante (s', v') es tal que $\alpha(v) = v'$.
 - El tiempo pasa sólo en los nodos.
 - Atravesar una arista no insume tiempo.
 - Los valores de los relojes crecen **uniformemente** con el tiempo.
 - El sistema puede permanecer en un nodo s mientras los valores de los relojes satisfagan Is .

6.2.1 Semántica

Proveer de semántica a los autómatas temporizados se lo hace en términos de otra notación conocida: el *LTS*. Es por esto que se dice que la semántica de *TA* (Timed-Automata) **hereda** una relación de bisimulación y una relación de composición en paralelo.



Formalmente, es un sistema de transición de estados etiquetados $T = \langle Q, \rightarrow \rangle$, en donde Q representa los estados ($Q = \{(s, v) \in S : Is[v] = \text{true}\}$) y \rightarrow representa la relación de transición ($\rightarrow \subseteq Q \times (L \cup \mathbb{R}^+) \times Q$)⁶.

Estas transiciones se dividen en dos subtipos:

- **Temporales:** el paso de un tiempo t es representado por una transición etiquetada por t .

$$\frac{Is[v + t], 0 \leq t}{(s, v) \xrightarrow{t} (s, v + t)}$$

- **Instantáneas:** dadas por la ejecución de una acción e . La transición lleva la etiqueta e .

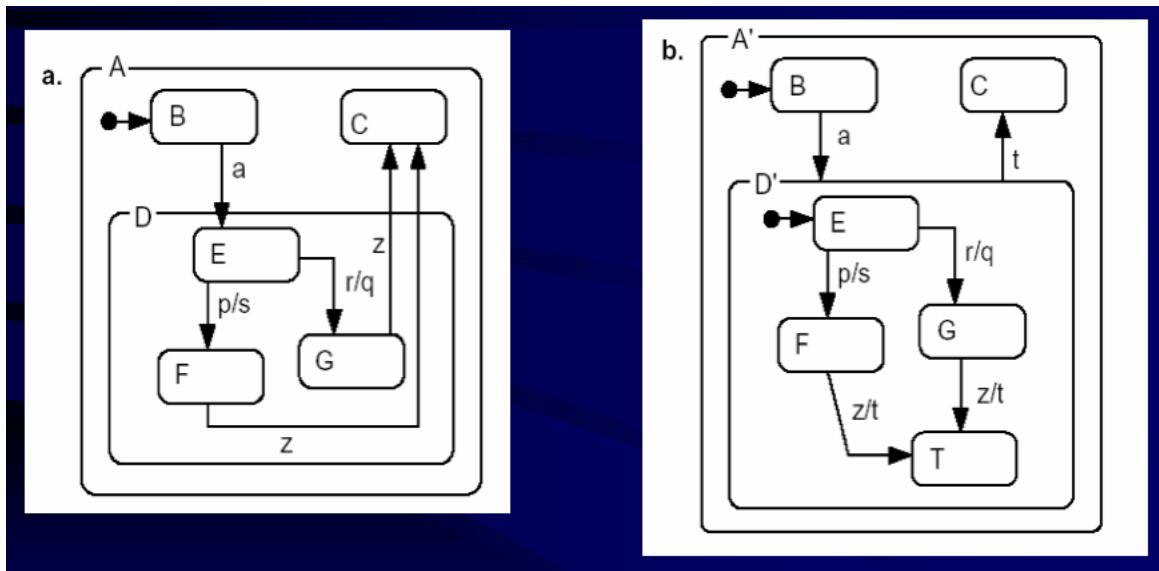
$$\frac{< s, e, \Psi, \alpha, s' > \in A, \Psi[v] \text{ y } \alpha(v) = v'}{(s, v) \xrightarrow{e} (s', v')}$$

Por comodidad de notación, se escribe $q \xrightarrow{e} q'$ en vez de $(q, e, q') \in \rightarrow$.

6.3 Statecharts

Un *statechart* es un tipo de máquina de estado utilizado para diagramar comportamiento de un sistema. Su principal característica es que presentan estados **jerárquicos** y **ortogonales**.

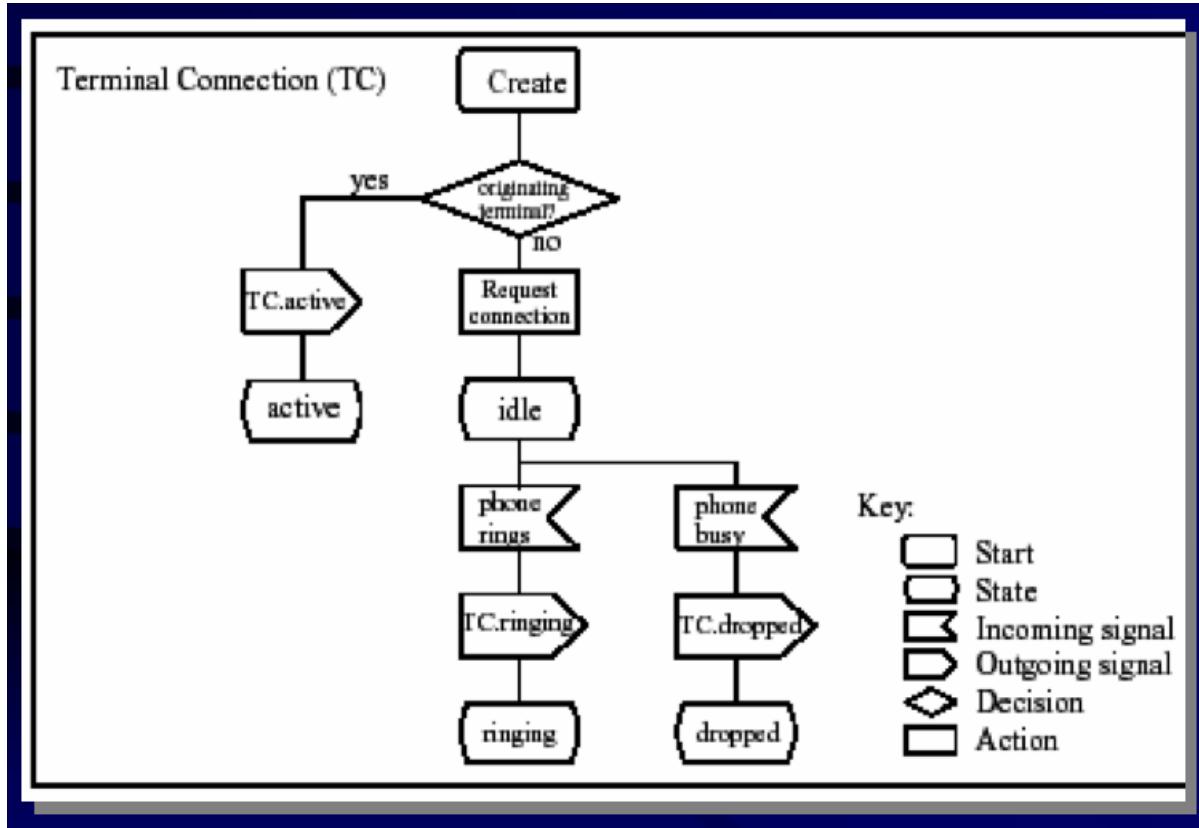
⁶Esto se debe a que es un conjunto infinito.



Otros aspectos que los caracterizan son:

- Se distinguen los eventos **internos** de los **externos**.
- Sólo puede haber un evento externo por vez.
- Hay *broadcasting* de eventos.
- Las transiciones se etiquetan con un par *causa / efecto*.
- Colas de eventos.
- Transiciones multi-nivel.

6.4 SDL Flowchart



6.5 Relación con otros modelos

- **Diagrama de contexto:** Define cuál es la relación temporal entre los fenómenos en la interfaz de un agente. Existen variantes de *LTS* que distinguen contenido monitoreado y controlado.
- **Modelo de objetivos:** ¿Las trazas o árbol de ejecución satisface las expectativas o requerimientos asociados al agente?. ¿El comportamiento de la composición satisface los objetivos?
- **Modelo de clases:** ¿Los estados son abstracciones de valuaciones de los atributos?
- **Modelo de operaciones:** ¿Los cambios de estado de la ME se corresponden a pre/post condiciones de operaciones?

7 Testing

7.1 Introducción

“La crisis del software” es un fenómeno decretado en 1968 en una conferencia sobre desarrollo de Software organizada por la OTAN, que se relaciona con la desordenada forma en la que se construye el software. El software es cada vez más **crítico** y **complejo**, lo que induce una creciente dificultad para escribir programas libres de defectos, fácilmente comprensibles, y que sean verificables. A esto se suman problemas como mantenimiento, necesidad de adaptación a nuevos entornos, nuevos requerimientos de usuarios, etc.

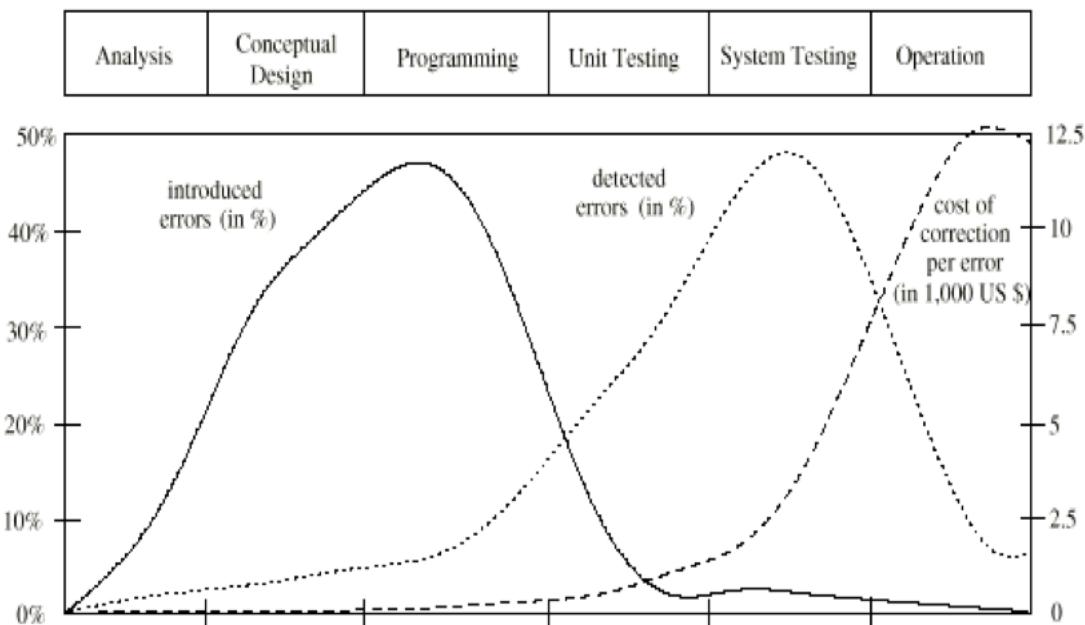
Existen numerosos ejemplos donde fallas en el desarrollo se cobraron vidas o millones de dólares:

- Therac-25 (6 vidas).
- Ariane 5 (U\$S 500.000.000).
- Denver Airport (9 meses penalidad de 1.1 por día).

La dificultad de esta tarea radica en que no existen soluciones mágicas, puesto que las dificultades son esenciales e innatas al desarrollo de software (y al error humano: distracciones, accidentes, etc). La disciplina de la ingeniería de software es extremadamente inmadura, lo que genera una brecha muy grande entre el “Estado del arte” y el “Estado de la práctica”.

7.1.1 Ciclo de vida y defectos

Para poder los errores en el proceso de desarrollo de software, primero es necesario entender dónde se producen, su magnitud y sus consecuencias.



Algunos números sobre errores:

- El 50% de los defectos se introducen durante el proceso de programación.
- No más del 15% de los defectos se detectan antes del testing.
- Al comienzo del test de unidad, se estima 1 error cada 50 líneas de código no comentadas.

- Existen “zonas” del código con mayor densidad de errores que otras: el 80% de los errores de programación se encuentran en el 20% de los módulos de programación.
- El costo de reparar un defecto es mayor cuanto más tarde se lo detecte. Por ejemplo, la reparación de un defecto detectado durante operación cuesta aproximadamente 12.5 veces más que si hubiera sido detectado durante el test de unidad.

7.2 Calidad de Software

La **calidad** en una pieza de software involucra numerosos conceptos:

- Confiabilidad.
- Corrección.
- Robustez.
- Seguridad.
- Funcionalidad.
- Usabilidad.
- Facilidad de mantenimiento.
- Reusabilidad.
- Verificabilidad + claridad.
- Interoperabilidad.

Una vez definidos los atributos de calidad, es necesario tener en cuenta que si bien la calidad puede **verificarse** al final del proceso, no puede “inyectarse” al final: depende de tareas realizadas durante todo el proceso de construcción de software. Detectar errores en forma temprana ahorra esfuerzos, tiempo y recursos.

B.Beizer afirma que “*El acto de diseñar tests es uno de los mecanismos conocidos más efectivos para prevenir errores... El proceso mental que debe desarrollarse para crear tests útiles puede descubrir y eliminar problemas en todas las etapas del desarrollo*”.

7.3 Definiciones

- **Validación:** es el proceso de evaluación de software durante o al final del proceso de desarrollo para determinar si satisface los requisitos especificados. Se basa en el uso de modelos. Intenta responder la pregunta: “*¿estamos construyendo el producto correcto?*”
- **Verificación:** es el proceso de evaluación de software para determinar si los productos de una fase de desarrollo dada cumplen con los requisitos impuestos al inicio de dicha fase. Necesariamente se trabaja en relación a un componente anterior que describe nuestro producto. Intenta responder la pregunta: “*¿Estamos haciendo el producto correctamente?*”.

Puede ser de dos tipos:

- **Dinámica:** intenta ejecutar y observar el comportamiento de un producto. No suele generar falso positivo, pero sí mucho falso negativo. (ej. *testing, run-time monitoring, run-time verification, etc*).
- **Estática:** intenta analizar una representación estática del sistema para detectar problemas y extraer conclusiones. No se ejecuta nada. (ej. chequeo de tipos, *code review, demostración formal, model checking, etc*). Puede generar falso positivo.
- **Falla:** es una diferencia en tiempo de ejecución entre lo que hace el software y lo que uno quiere que haga (diferencia entre resultados esperados y reales). Es la exteriorización o manifestación efectiva de un defecto.
- **Defecto:** es un problema en el código de un programa, en una especificación o un diseño que puede (o no) dar lugar a una falla.

- **Error:** es una equivocación humana, el motivo por el cual se introduce uno o más defectos.

Falso positivo: se detecta incorrectamente un error que en la ejecución nunca ocurre.

Falso negativo: no se detecta un error que en la ejecución ocurre para algún input.

7.4 Testing

Se define el **testing** como la *verificación dinámica de la adecuación del sistema a los requerimientos*. Su objetivo es encontrar errores. Es importante notar que **no prueba la corrección del software**; sólo su adecuación a los requerimientos. Se le atribuye de 30% a 50% del costo de un software confiable.

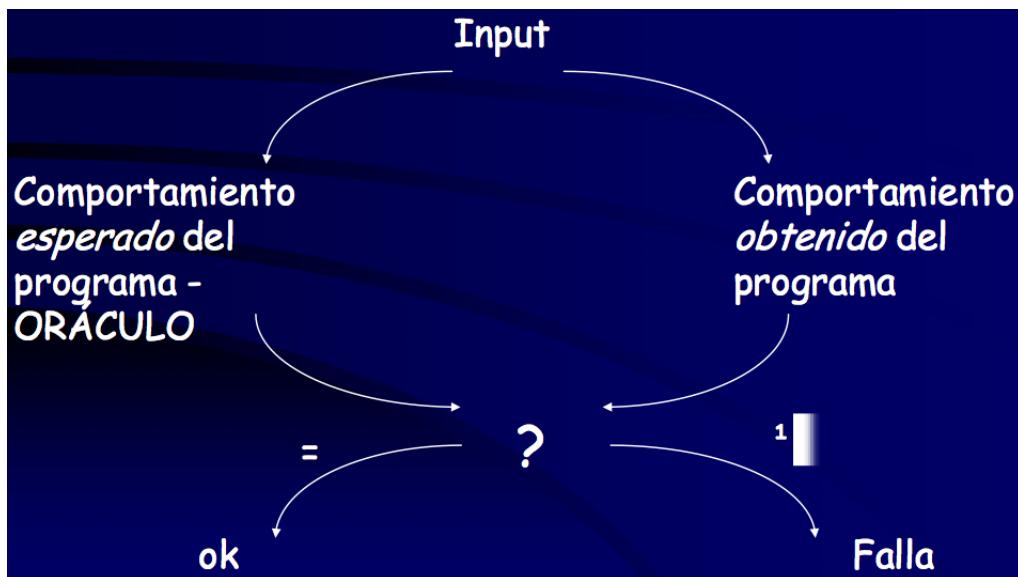
Para realizar *testing* es necesario:

- Que se pueda ejecutar el programa.
- Que se conozcan los resultado esperados (problema de oráculo: visibilidad).
- Que los resultados esperados puedan ser comparados con el resultado obtenido (problema de oráculo: comparación).

Durante el *testing*, se ejecuta un producto para:

- Verificar que satisface los requerimientos.
- Identificar diferencias entre el comportamiento real y el esperado.

Gráficamente se puede ver al *testing* de la siguiente manera:



Un **oráculo** es una entidad que nos puede brindar información del comportamiento esperado del programa. Puede ser un humano, una especificación, otro sistema (que ya se sepa que funciona), etc.

7.4.1 Limitaciones

El proceso de *testing* puede demostrar la presencia de errores, pero no su ausencia. O sea, no puede garantizar que el software funcione perfecto, aunque sí puede aumentar la **confianza** en que lo haga.

Una de las mayores dificultades es encontrar un **conjunto de tests** apropiado. Debe ser suficientemente grande como para abarcar todo el dominio y maximizar la probabilidad de encontrar defectos, pero suficientemente chico como para poder ser corrido bajo los requerimientos (temporales y económicos) que se dispongan.

7.4.2 Test de requerimientos no funcionales

Ejemplos:

- **Test de seguridad:** se intentan validar elementos de seguridad del programa, tales como disponibilidad, integridad y confidencialidad de datos o servicios.
- **Test de performance:** se intentan validar los tiempos de acceso, ejecución y respuesta del sistema.
- **Test de stress:** se intenta validar el uso del sistema en sus límites de capacidad y verificando sus reacciones más allá de los mismos.
- **Test de usabilidad.**

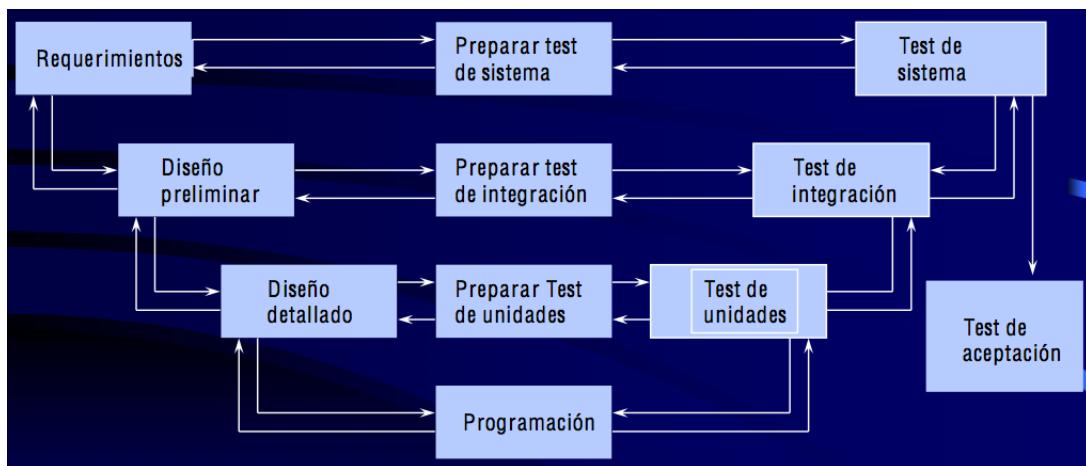
7.4.3 Niveles de test

Existen tres niveles a los que se suele testear un sistema:

- **Test de unidad:** se realiza sobre una unidad de código⁷ pequeña, claramente definida. En general es llevado a cabo por los propios desarrolladores. Su dificultad radica en que posiblemente muchas piezas necesarias para un test completo de una unidad no estén construidas.
- **Test de integración:** está orientado a verificar que las partes de un sistema que funcionan bien aisladamente, también lo hagan en conjunto. Se testea la interacción y comunicación entre las partes, uniéndolas. La “estrategia” de unión depende del tipo de sistema (puede ser *organizado jerárquicamente*, *batch de procesamiento secuencial* o un sistema sin jerarquía). Pueden programarse “accesorios” para simplificar este *testing*, tales como **drivers** (simula las llamadas) o **stubs** (simula subprogramas).
- **Test de sistema:** se testea todo el sistema, terminado y armado. Lo normal es que no lo testejen los desarrolladores sino un equipo especializado.

7.4.4 Ciclo de vida del testing

El test de sistema es lo último que se hace. Sin embargo, su preparación de casos de test puede ocurrir al momento de la ingeniería de requerimientos. Antes de empezar a programar ya se pueden extraer casos de test del sistema como un todo. Sin embargo, en ese momento, no se pueden concebir casos de testeo de unidad, porque no está definido qué unidades serán necesarias más adelante. Es por esto que los casos de testeo pueden ser generados en la etapa correspondiente de diseño, formando un modelo en V.

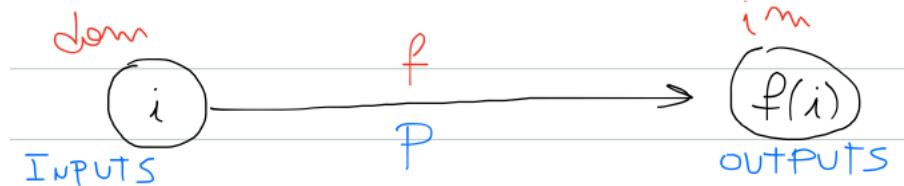


⁷La definición de qué constituye una “unidad de código” es difusa y dependiente del sistema. Puede ser un programa, una función, un *form*, un subsistema, una clase, etc.

7.4.5 Testing funcional

Es un tipo de *testing* que plantea que el programa a testear se lo puede pensar matemáticamente como una función. Las especificaciones de tests son funciones y se intenta ver que el programa se comporte de la misma forma.

Sea P el programa a testear y f la función matemática que lo representa.



P implementa bien a f si y sólo si

$$\forall i \in \begin{cases} \text{dom}(f) \\ \text{inputs}(P) \end{cases} \quad P(i) = f(i)$$

Es interesante notar que no siempre es trivial la equivalencia *sistema = función*. No todo sistema tiene una representación natural como función.

El testing funcional debe verificar no sólo que P *implemente* correctamente a f , sino que lo hace “en forma razonable” (por ejemplo, si $i \in \text{dom}(f)$ y $f(i) = \text{error}$, P debe dar algún tipo de error razonable). Además, debe verificar que $\forall i \notin \text{dom}(f)$, P avisa que $i \notin \text{dom}(f)$ (o sea si, el input no está en el dominio de la función, que el programa avise que no es un input válido).

Recordar que existe una analogía con el Modelo en “V” de desarrollo.

7.4.6 Testing de conformidad

El **testing de conformidad** sirve para un sistema reactivo que mantiene una continua interacción con su entorno, respondiendo ante los estímulos externos en función de su estado interno. Se utiliza en los casos en los que pensar en testing funcional no tiene sentido (por ejemplo, los casos en donde no existe la equivalencia *sistema = función*). Se utiliza otro tipo de abstracciones para describir su comportamiento.

7.5 Generación de casos de testing

7.5.1 Notación

- **Caso de test:** descripción de condiciones o situaciones a testear y un resultado esperado.
- **Dato de test:** asignación de valores concretos de parámetros para ejecutar un caso de test.
- **Test Suite:** conjunto de datos de test con los que se testeaa el programa. Si un programa P es correcto para todo elemento de una *test suite* T , se dice que T es **exitoso** para P .
- **Criterio:** es un subconjunto de conjuntos finitos del dominio de inputs de un programa P (puede estar expresado en términos de un conjunto de predicados también llamados *casos*). Dado un programa P y una especificación S , un criterio C define un conjunto no vacío de subdominios, $SD_C(P, S)$. Informalmente, dado un conjunto de casos, me permite verificar si es **apropiado**, o sea, si tiene al menos un dato para cada caso (puede verse un *caso* como una especificación del subdominio más un resultado esperado). Se dice que un conjunto de datos T **satisface** un criterio C si y sólo si $T \in C$.
 - Se dice que un criterio es **consistente** si todo par de *test sets* que lo satisfacen, uno de ellos es exitoso si y solo si el otro lo es. Formalmente, T_1 y T_2 satisfacen $C \Rightarrow T_1$ exitoso $\Leftrightarrow T_2$ exitoso.

- Se dice que un criterio es **completo** si en caso de que el programa sea incorrecto, entonces existe un *test set* que satisface al criterio para el que no va a ser exitoso. Cuando uno recorta casos de testing, en general pone en riesgo la completitud del criterio.

Formalmente P incorrecto $\Rightarrow \exists T : T$ no es exitoso para P .

- **Condiciones...:**

- ...válidas y esperadas: hace lo que se supone que hace.
- ...inválidas o inesperadas: no hace lo que se supone que no hace.

- **Requerimientos de Test:** Qué quiero testear. En el marco del testing de sistemas reactivos se llama el **Test Purpose**.

- **Especificaciones de Test:** Supuestos y definiciones que sirven para generar los casos de test para el requerimiento de test.

Estos dos conceptos son los que uno idealmente busca porque entonces ese criterio se convierte en un *certificado* del programa. En la práctica no es posible alcanzar los dos: la consistencia y completitud de un criterio **no es decidible**. No existe una técnica para detectar todos los errores de un programa, por lo que se usan heurísticas.

7.5.2 Técnicas

Con las técnicas de generación de casos de test se busca obtener una alta probabilidad de encontrar errores manteniendo un bajo costo (o al menos acotado a las restricciones presentes). Además, se busca utilizar ideas generales, sistematizables y semiautomatizables y un modelo de fallas subyacente como “rationale”.

Random testing

La idea de **random testing** es generar inputs al azar con distribución uniforme, o siguiendo un perfil operacional (no es trivial cuando el input es complejo). Es muy utilizado como base para comparar las demás técnicas de generación de casos de testing. En la práctica es bastante utilizado por su bajo costo para generar los tests, aunque su efectividad es muy discutida.

Partition testing

La idea de **partition testing** es separar el dominio de inputs en subdominios (no necesariamente disjuntos) y seleccionar uno o más representantes de cada subdominio. Esto se basa en la suposición de que hay inputs que generan comportamiento similar. Se intenta que la separación distinga estos comportamientos, para que, testearlo con un representante equivalga a testearlo con cualquier elemento del mismo subdominio. “*La probabilidad de encontrar fallas usando particiones se maximiza cuando hay un subdominio compuesto básicamente por elementos que producen fallas*” Jeng, Weyuker, 1989.

Partition testing funciona mucho mejor que *random testing* cuando se puede identificar un subdominio denso en fallas. Esto se debe a que elegir un representante fallido de ese subdominio (así se elija al azar) es mucho más probable que elegirlo al azar de todo el dominio. Si todos los subdominios tienen una densidad similar en fallas, *random* y *partition* son más o menos equivalentes. Puede incluso ser peor si las fallas están concentradas en un solo subdominio, pero es demasiado grande).

Fault-based systems

Los fault based son un tipo de partition testing que emplean estrategias orientadas a un tipo específico de fallas. Suelen funcionar mejor porque entonces es más probable poder hacer una división de subdominios con sentido.

Criterios de caja negra

Nos desentendemos completamente de la estructura interna del programa, pero no de su especificación, de donde se derivan los casos de test.

Criterios de caja blanca

Los casos de test (o subdominios) se definen a partir del código y la estructura interna del programa.

Category Partition

La técnica de **category partition** fue introducida por Ostrand y M. Balcer en 1988. Es un método semi-automático de generar casos de tests, que se basa en pasar exhaustivamente por elecciones de una categoría. Insta a pensar en categorías interesantes entre parámetros o relación entre parámetros. Se basa en la hipótesis de que es más importante pensar en categorías que pensar en casos aisladamente. Sigue los siguientes pasos:

1. Elegir una funcionalidad que pueda testearse en forma independiente.
2. Determinar sus parámetros u otros objetos del ambiente que pueden afectar su funcionamiento.
3. Determinar las características relevantes de cada objeto determinado en el punto 2 y de la relación entre estos objetos en el output.
4. Determinar elecciones (“*choices*”) para cada característica de cada objeto.
5. Definir entre esas elecciones las categorías **único**, **error** y otras restricciones.
6. Armar los casos.

Para identificar categorías se usa:

- Texto informal o formal de la especificación.
- Características propias de los parámetros de I/O de la funcionalidad a probar.
- Cardinalidad del modelo de datos, que define reglas del negocio.
- Ciclo de Vida de las entidades del modelo de datos.

El método de category partition se aplica a cualquier descripción de funcionalidad (formal, semiformal o informal) en el que se conozcan bien sus requerimientos (pues es necesario utilizarlos para planificar el testing). Esto genera preguntas de incompletitud, ambigüedad, inconsistencia, e incorrección en un momento en que es “fácil” corregir requerimientos. No hay una sola forma de hacerlo: dependiendo de cómo se aplique el método se obtendrán distintos conjuntos de casos de test.

Técnicas de partición de dominio

Son técnicas usadas para la identificación de categorías. Aparecen de manera más o menos clara los parámetros (implícitos y explícitos), su casuística, la relación entre ellos, etc. Por ejemplo, en los casos de uso hay escenarios distintos dependiendo de valores de parámetros o de acciones decididas por el actor. Son parámetros candidatos con sus categorías y elecciones.

- Los casos de test que exploran los bordes de las clases de equivalencia producen mejor resultado. Cada margen de la clase de equivalencia debe quedar sujeto a un test.
- Si una condición de input especifica un rango de valores (intervalo), identificar una clase válida, dos inválidas y dos casos borde. Por ejemplo, si especifica que el parámetro puede variar entre 0 y 99, poner como clase válida el rango $0 < v < 99$, como inválidas $v < 0$ y $99 < v$ y como borde $v = 0$ y $v = 99$.

- Si una condición de input especifica un conjunto de valores, y hay razones para pensar que cada uno es manejado por el programa en forma distinta, identificar una clase válida por cada elemento y una clase inválida.
- Si una condición de input especifica una situación que debe ocurrir, identificar una clase válida y una clase inválida (una que verifique la situación y otra que no).
- Probar el ingreso de valores de otro tipo que la clase.
- Probar alterando características propias de los tipos de datos de entrada y salida de la funcionalidad testeada. (Ejemplo, fechas futuras, días de semana inexistentes, etc).
- Verificar la cardinalidad del modelo de datos. La **cardinalidad mínima/máxima** define cuántas instancias hay como mínimo/como máximo de una entidad por cada instancia de una entidad relacionada con ella. Probar un caso de test que viole estos límites de cardinalidad (por ejemplo una cuenta con 3 firmantes, siendo su cardinalidad máxima 2).
- Ciclo de vida de las entidades: Visión temporal del modelo de datos: a partir de un diagrama del ciclo de vida de una entidad debo probar transiciones válidas así como transiciones inválidas.

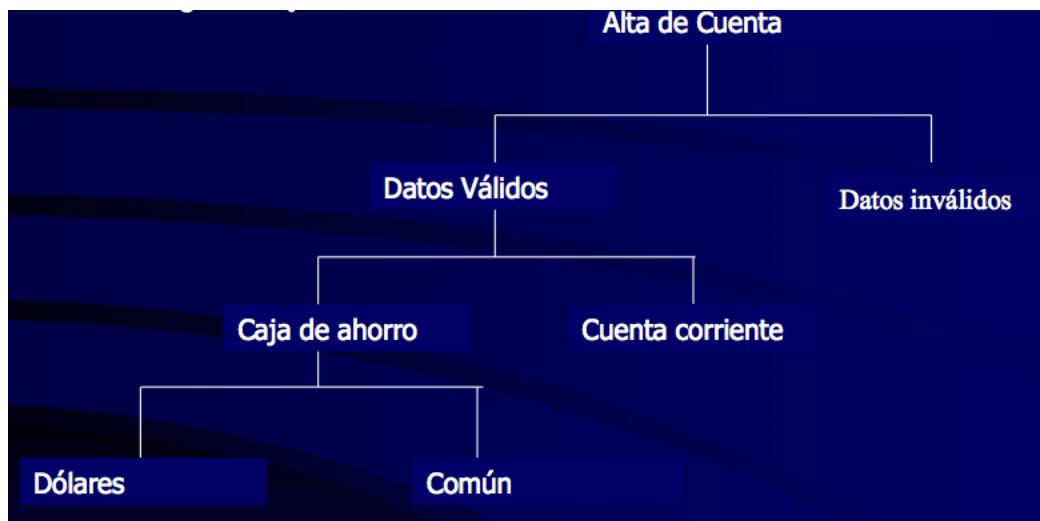
7.5.3 Técnicas combinatorias

Definición arbórea

La técnica de **definición arbórea** puede verse como *category partition* especial donde el mismo diseñador de casos decide explícitamente cómo combinar. Por ejemplo, cuando hay idea de orden de ingreso de parámetros se puede minimizar la cantidad de casos sin sentido.

Se arma un árbol donde cada nodo indica “category” sobre un parámetro o combinación entre un parámetro y uno anterior descripto en el árbol. Los ejes que salen son las *choices* compatibles con las condiciones acumuladas desde la raíz. Los casos de tests especificados y ejecutados son los caminos del árbol. Las categorías irrelevantes en una rama no se abren.

Ejemplo:



La definición arbórea se puede combinar con *category partition*, aunque conviene anotar la relevancia de cada categoría respecto a *choices* de otras categorías. Criterio de cobertura: hay que revisar que todos los *choices* sean ejercitados en el árbol.

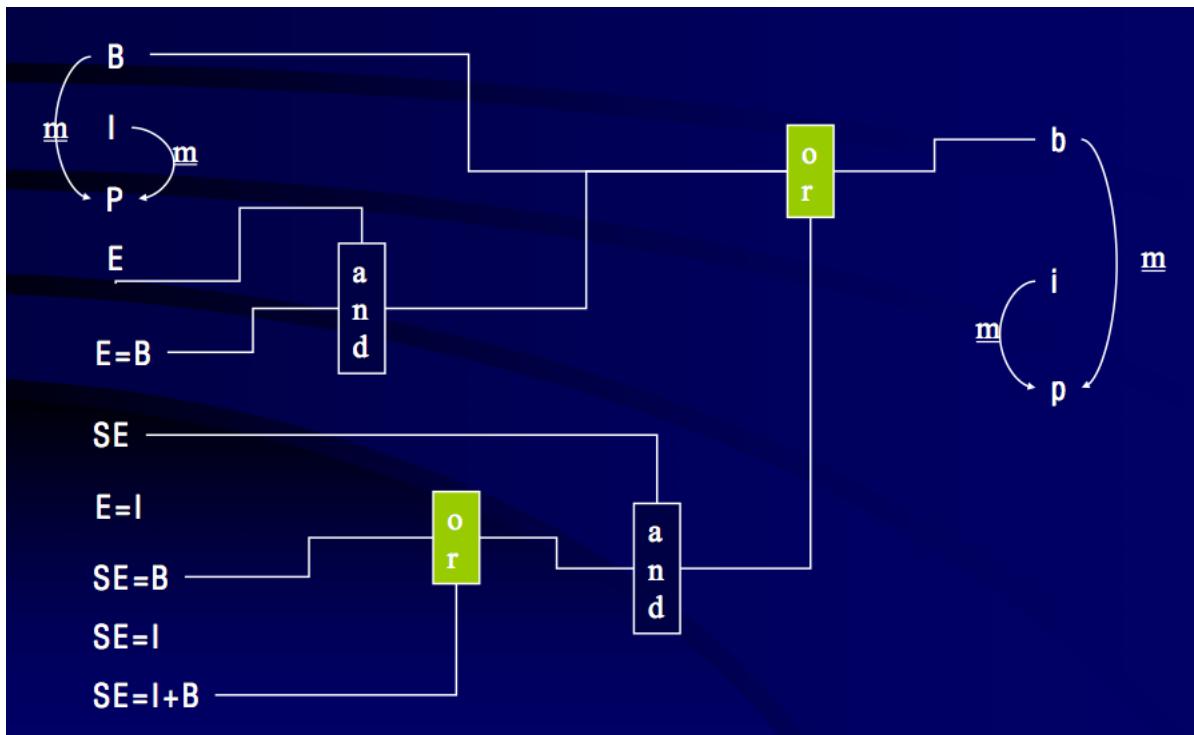
Grafo Causa-Efecto

La técnica de **grafos causa-efecto** nos permite definir combinaciones relevantes de categorías binarias sobre inputs para definir casos. Allí difiere de la parte combinatoria de Category-Partition. Sirve para especificaciones de I/O complejas (en las que hay mucha dependencia entre Inputs, entre Outputs y entre I/O). Provee un display visual de las relaciones entre una causa y la otra. Ayuda a detectar ambigüedades o incompletitud en la especificación.

Esta técnica permite generar todos los outputs admisibles con la siguiente heurística:

- Si hay un “or” (todas las opciones con sólo una señal en True).
- Si hay un “and” (todas con sólo una en False).

Esta heurística logra una complejidad de $O(n * k * o)$ en lugar de $O(2^n)$ dónde n es la cantidad de categorías binarias sobre el input, k la profundidad del DAG y o la cantidad de combinaciones del output válidas.



n-wise partition y OATS

La técnica de ***n*-wise partition** permite testear las interacciones entre *n*-uplas de parámetros (factores) independientes de una manera muy económica. Sin embargo, para poder aplicar esta técnica es necesario que valga una hipótesis bastante fuerte: si hay un bug que resulta de la interacción de dos parámetros sin importar el resto, necesariamente lo voy a encontrar. El bug **sí o sí** se tiene que revelar con una combinación de dos parámetros **y no** tiene que molestar el resto de los parámetros.

Además, en el caso en el cual una combinación de dos parámetros es intrínsecamente imposible tengo un problema porque si lo elimino, estoy perdiendo información de más porque todos los otros pares que tengo en ese caso los pierdo.

Orthogonal Array Testing Strategy (OATS) es un formato para listar casos de test que permite aplicar cómodamente *n*-wise partition. Los casos de test se listan una tabla en la que las columnas se corresponden con los factores y las filas los casos de test.

- **Factor:** es la cantidad de parámetros, o de columnas.
- **Levels:** es la cantidad de valores posibles para cada parámetro.
- **Strength:** cantidad de columnas tales que las $levels^{strength}$ posibilidades aparecen la misma cantidad de veces.
- Se tabulan como:

$$L_{runs} = (Levels^{factors})$$

en donde

- $runs$ representa la cantidad de corridas.
- $levels$ representa la cantidad de elecciones posibles.
- $factors$ representa la cantidad de factores que tiene cada una de las anteriores elecciones.

Ejemplo: tengo un comando con 3 parámetros: 1, 2 o 3. En principio tendría $3^3 = 27$ combinaciones. Sin embargo me las puedo ingeniar para que con 9 filas, me alcanza para tener todas las combinaciones de a 2:

3	1	1
2	1	2
1	1	3
2	2	1
1	2	2
3	2	3
1	3	1
3	3	2
2	3	3

Como se puede ver, toda posible combinación de pares de parámetros está contemplada en al menos un caso.

Conclusiones

Como se puede ver, ninguna técnica es completa. Cada una es una heurística que ataca una parte del problema distinto y logran diversos resultados, dependiendo mucho del programa a testear. Lo mejor es combinar varias de estas técnicas para complementar ventajas y desventajas de cada una.

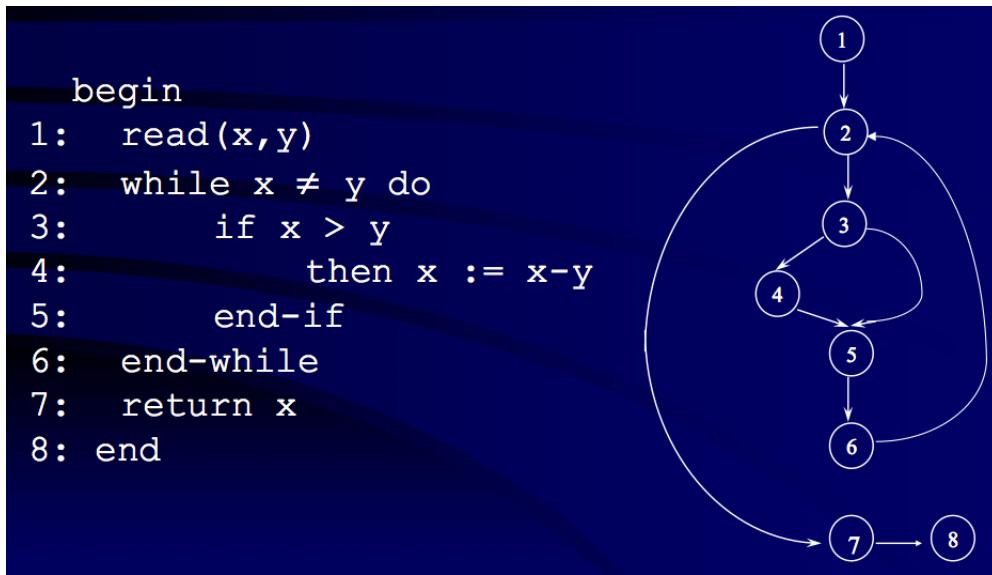
Sin especificaciones de requerimientos es mucho más difícil. Ayuda tener en cuenta la conjectura de defectos y ser sistemático y documentar las suposiciones sobre el comportamiento o el modelo de fallas.

7.5.4 Testing estructural de unidades

Para realizar un test estructural de unidades de programas es necesario realizar un análisis en el que se representa el flujo de control de un programa a través de un **grafo de flujo**, **flowchart** o **Control Flow Graph (CFG)**. Sólo sirve con programas secuenciales, con un único punto de ingreso y un único punto de terminación.

Cada instrucción se grafica en un nodo y, si es una instrucción que altera el flujo de control, se une a otros nodos de diversas formas:

Ejemplo

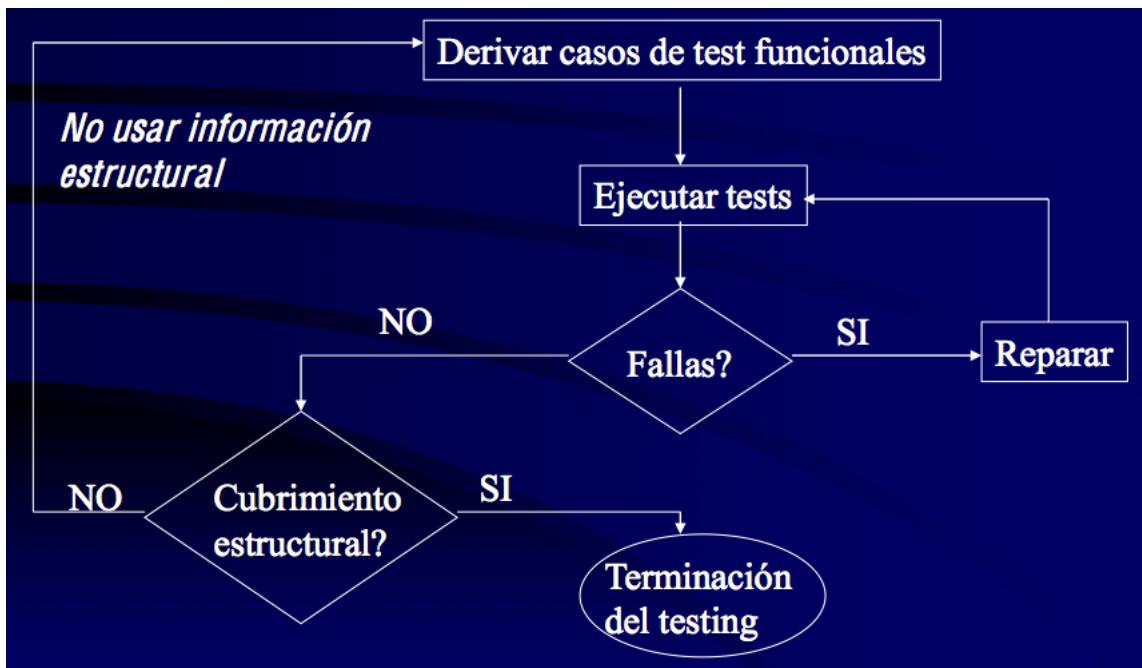


Un camino en un flowgraph desde el nodo asociado al inicio del programa hasta el nodo asociado a la terminación del programa se llama **camino completo**. Una ejecución del programa que termina satisfactoriamente está asociado a un camino completo en el flowgraph. Sin embargo, no todo camino en el flowgraph se asocia con una ejecución del programa: un camino en un flowgraph para el cual no existe input del programa que fuerce su ejecución, se denomina **camino no factible**.

Cada camino factible puede tener muchos inputs asociados que fuercen su ejecución. Sin embargo, para realizar testing es interesante realizar el proceso inverso: dado un camino completo, obtener condiciones sobre el input que me fuerce ese camino. Eso sirve no sólo para casos de test, sino también para ejecución simbólica.

Un **criterio de testing estructural** permite identificar entidades que deben cubrirse con los datos de test para satisfacer el criterio. Abstractamente, imponen una serie de condiciones para ser satisfechas por las test-suites. Estos criterios se utilizan, entre otras cosas, como **criterio de adecuación**. O sea, dada una test-suite, armada de alguna forma (no necesariamente de acuerdo a los criterios), me interesa ver cuán buena es. Para eso me puedo fijar su grado de cubrimiento estructural.

Todos los métodos de cubrimiento se basan en el supuesto de que una test-suite con mucho cubrimiento estructural tiene más probabilidad de detectar fallas que una test-suite aleatoria más o menos del mismo tamaño.



La forma naïve de hacer testing estructural es:

1. Con el código como base, dibujamos el CFG.
2. Determinamos algún criterio que nos parezca apropiado.
3. Determinamos un conjunto de caminos que cumple el criterio.
4. Preparamos los datos de test que forzarán la ejecución de cada camino
5. Evaluamos si satisface el criterio
6. Iteramos.

El testing basado en código encuentra muchos errores. Se ha realizado mucha investigación para determinar qué técnica estructural es la mejor, pero cualquier técnica de selección de casos que no está basada en el comportamiento funcional, está mal guiada desde el comienzo porque los usuarios no usan el software para ejecutar sus instrucciones, sino para invocar sus funcionalidades. Es fácil ejecutar todas las instrucciones y sin embargo no invocar ciertas funciones.

A continuación se listan varios criterios:

Cubrimiento de sentencias

Todas las sentencias del programa deben ejercitarse. Equivale a cubrir todos los **nodos** del CFG.

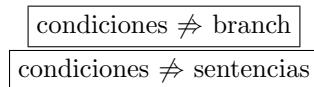
Cubrimiento de decisiones o Branch Covering

Todas las decisiones en el control del programa deben ejercitarse al menos una vez por True, y al menos una vez por False. Equivale a cumplir todos los **arcos** del CFG.

Branch covering ⇒ cubrimiento de sentencias

Cubrimiento de condiciones

El problema de branch covering es que una decisión puede estar creada por varias condiciones. Es por esto que se inventó el cubrimiento de condiciones, que requiere que *Todas las condiciones de todas las decisiones en el control del programa deben ejercitarse al menos una vez por True, y al menos una vez por False.*



Tener en cuenta que hay semántica de cortocircuito: se evalúan las condiciones sólo cuando es necesario. Para cubrir condiciones hay que efectivamente evaluar todas en sus dos posibles valores booleanos, sin perder cuidado de la semántica de cortocircuito.

Cubrimiento de caminos

Todo camino del flujo de control del programa debe ejercitarse alguna vez. Equivale a cubrir todos los caminos del flowgraph, por lo que implica tanto cubrimiento de decisiones como de sentencias. Puede volverse excesivamente costoso (p.ej., en el caso de loops infinitos, habrían infinitos caminos) y no garantiza nada, pero sí es de los criterios de testing más completos (implica branches y statements, aunque no condition).

MC/DC: Modified Condition/Decision Coverage

Exige que:

- cada punto de entrada y salida del programa se invoco al menos una vez.
- cada condición en una decisión tomó sus posibles resultados al menos una vez.
- cada condición se mostró que afecta el resultado de la decisión independientemente del resto.

Genera entre $n + 1$ y $2n$ tests. Se demuestra que En un conjunto de m testeos distintos, sea $P(n, m)$ la probabilidad de encontrar un error en una implementación incorrecta de una expresión Booleana de n condiciones. Se demuestra que

$$P(n, m) = 1 - \left(\frac{2^{2^n} - 1}{2^{2^n}} \right)$$

Cuando n crece, esta fórmula converge a:

$$P(m) = 1 - \left(\frac{1}{2^m} \right)$$

Data Flow Testing

Definiciones:

- Una sentencia que guarda un valor en la posición de memoria de una variable, crea una **definición**.
- Una sentencia que trae el valor de la posición de memoria de una variable es un **uso** de la definición activa de esa variable.
 - Un uso de x es un uso predicho o **p-uso** si aparece en el predicado de una sentencia que representa una bifurcación de control.
 - En otro caso, se llama uso computacional o **c-uso** (aparece del lado derecho de una asignación).

El **def-use flowgraph** de un programa para una de sus variables, x , es su *flowgraph* anotado de la siguiente forma:

1. Por cada *definición* de x , el nodo asociado se etiqueta con una definición de x .
2. Por cada *c-uso*, el nodo asociado se etiqueta con un uso de x .
3. Por cada *p-uso* todos los arcos salientes del nodo asociado se etiquetan con un uso de x .

Se define una **DUA** como una terna $[d, u, x]$ tal que

- La variable x está definida en el nodo d .
- La variable x se usa en el nodo u o en el arco u .
- Hay al menos un camino desde d hasta u que no contiene otra definición de x además de la de d (**def-clear** o **libre de definiciones** para x).

Existen numerosos criterios estructurales basados en flujo de datos:

- | | |
|--|--|
| <ul style="list-style-type: none"> • all defs. • all c-use. • all p-use. • all uses. | <ul style="list-style-type: none"> • all c-use some p-uses. • all p-uses some c-uses. • all du-paths. |
|--|--|

Sin embargo el más usado es el cubrimiento **all-uses** que exige que *para cada variable en el programa, deben ejercitarse todas las asociaciones entre cada definición y todo uso de la misma (tal que esa definición esté activa)*. Equivale a cubrir todas las DUAs del programa.

All-uses \Rightarrow Branch covering

7.5.5 Modelo de efectividad en la detección de fallas

¿En qué medida la cobertura importa para detectar fallas?

Esta pregunta es abierta y todavía no está muy decidido. Existen experimentos que responden para ambos lados en función del programa.

¿Es mejor cubrir más o hacer una test-suite más grande?

Algunos resultados dicen que probablemente uno puede predecir la detección de fallas con una función logarítmica en el tamaño y lineal en la cobertura. O sea, al principio el tamaño es muy importante, pero a la larga el nivel de cobertura hace la diferencia.

Existen modelos teóricos que permiten vincular algunos de los criterios que vimos. El modelo de efectividad en la detección de fallas se basa en calcular la probabilidad de detectar fallas de un criterio

$$M(C, P, S) = 1 - \prod_{i=1}^n 1 - \frac{m_i}{d_i}$$

Donde:

- m_i es la cantidad de elementos que revelan la falla que están en el dominio D_i .
- d_i es el cardinal del subdominio D_i .
- $\frac{m_i}{d_i}$ es la probabilidad de encontrar el elemento falluto.
- $1 - \frac{m_i}{d_i}$ es la probabilidad de no detectar la falla.

7.5.6 Subsunción

Formalmente, decimos que un criterio C_1 **subsume** a un criterio C_2 si para todo conjunto de datos de test T que satisface a C_1 , satisface a C_2 (para todo par (P, S)).

Informalmente el predicado de C_1 es **más fuerte** que el de C_2 (o sea, implica a su predicado trivialmente).

Ejemplo:

- Branch subsume a Statement.
- Alluses subsume a Branch (y, por transitividad, a Statement).

Observación: C_1 subsume a C_2 **no** implica $M(C_1, P, S) \geq M(C_2, P, S)$. Por ejemplo, sea P un programa que falla en 0. El criterio C_1 requiere partir una testsuite en $\{0, 1\}$ y $\{2\}$. El criterio C_2 requiere partir una testsuite en $\{0, 1\}$ y $\{0, 2\}$. Trivialmente, C_1 subsume a C_2 , pero sin embargo la probabilidad de una test-suite que verifica C_1 es 0.5 (porque tengo que pifiarle al 0 en la primer coordenada) mientras que una test-suite que verifica C_2 es 0.75 (porque tengo que pifiarle al 0 en ambas coordenadas).

7.5.7 Properly Covers

Formalmente, un criterio C_1 **properly covers** a otro criterio C_2 para (P, S) si vale que

$$SD_{C_1}(P, S) = D_1^1, \dots, D_m^1$$

$$SD_{C_2}(P, S) = D_1^2, \dots, D_n^2$$

Entonces existe $M = \{D_{1,1}^1, \dots, D_{1,k_1}^1, \dots, D_{n,1}^1, \dots, D_{n,k_n}^1\}$ tal que M es un sub-bag de $SD_{C_1}(P, S)$ y

$$D_1^2 = D_{1,1}^1 \cup \dots \cup D_{1,k_1}^1$$

....

$$D_n^2 = D_{n,1}^1 \cup \dots \cup D_{n,k_n}^1$$

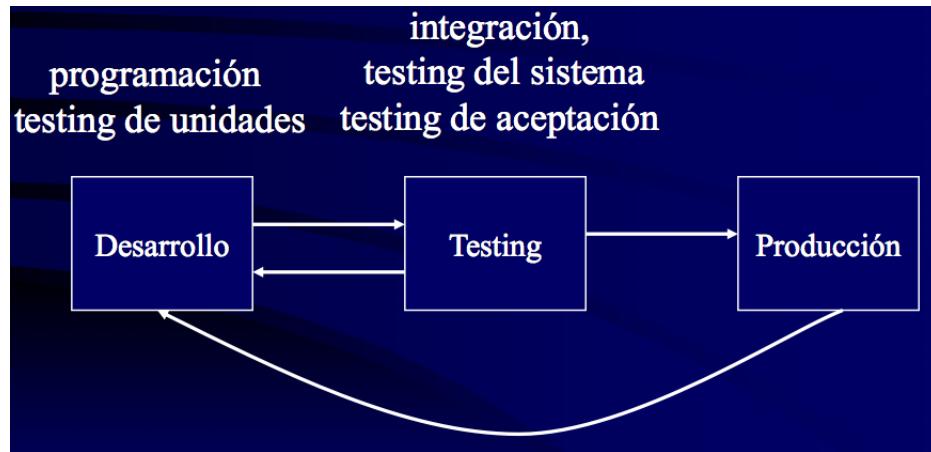
Informalmente, es una versión más fuerte de **subsumir**: la idea es que a partir de C_1 debería poder agarrar cachitos y formar un subconjunto con repeticiones tal que todo subdominio que aparece en C_2 lo puedo formar con cachos de M . Observar que en M no puedo usar más repeticiones de un elemento de las que aparecen en C_1 .

Ejemplo: All-Uses *properly covers* decisiones.

Observaciones:

- Si C_1 *properly covers* C_2 entonces $M(C_1, P, S) \geq M(C_2, P, S)$.
- *Properly covers* es reflexiva y transitiva.

7.6 Ambiente de testing



7.7 Terminación de testing

Existen numerosos criterios de terminación para tests:

- Se terminó el tiempo o los recursos.
- Se corrieron todos los tests sin error.
- % de cubrimiento de ciertas técnicas elegidas.
- Fault-rate más bajo que un cierto valor especificado (la cantidad de bugs que encontré es menor que lo que esperaba. O sea, tengo menos de 1 error cada 200000 líneas de código).
- Se encontró un número predeterminado de errores(% del número total de errores estimado).

7.8 Documentación de casos de tests

Es necesario documentar información concerniente los tests que se realizan y su ejecución:

- Criterios de derivación de casos empleados.
- Casos de test organizados por:
 - Módulo, sistema, unidad, etc.
 - Resultado esperado.
 - Requerimiento que prueban.
- Criterios de terminación del testing.
- Datos de prueba.
- Ambiente del test.
- Selección de datos (referencia a casos de test que prueban).
- Ejecución y resultado obtenido (reporte de errores).
- Re-ejecución luego de las modificaciones pertinentes.

- Determinación del origen del error (una vez detectado un error, se busca el problema que lo originó haciendo debugging).
- Clasificación de errores por prioridad o severidad.
- Seguimiento de errores.

7.9 Test de Regresión

La técnica de test de regresión es un tipo de testing cuyo objetivo es asegurarse que un cambio en un programa, tal como una corrección de errores, no introduce nuevos defectos. Básicamente, son tareas de testing que se realizan luego de que un sistema haya sido modificado. Esto se debe a que algunos estudios dicen que la probabilidad de introducir un error al hacer un cambio es entre 50% y el 80%.

Los tests de regresión se hacen

- Durante el desarrollo del software.
- Luego de modificaciones.
- Al momento de adaptarlo a un nuevo ambiente.

7.9.1 Casos de regresión

Existen tres casos para lo que puede sucederle a una test-suite cuando debe ser usada para un test de regresión:

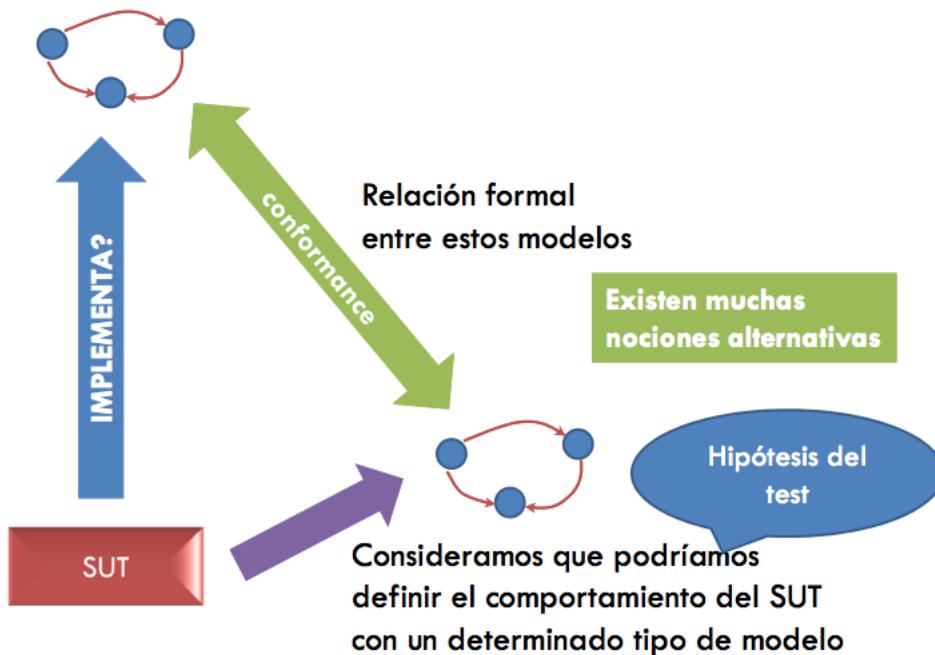
- **Casos reusables:** corresponden a partes del software no modificadas (ni implementación ni especificación). Lo ideal sería ni siquiera volver a ejecutar los tests si se sabe que, por la semántica del programa modificado, la ejecución no es posible que me de una sorpresa.
- **Casos reseteables:** la especificación no cambia, pero sí la implementación. Tiene sentido volver a correrlos porque podemos encontrar fallas.
- **Obsoletos:** no pueden seguir usándose los tests (por ejemplo, cambió la interfaz, o la especificación).

8 Testing de sistemas reactivos

8.1 Introducción

Hasta ahora consideramos a los sistemas como una transformación de entradas en salidas: como funciones. Sin embargo, no es cierto que todos los sistemas se comporten de este modo, lo que puede llevar a problemas si intentamos utilizar la noción tradicional de corrección funcional en esos casos. Es por esto que hablamos de **conformidad**, que establece formalmente cuál es la relación entre la especificación y la implementación. Es necesario revisar no sólo los casos de test, sino también los datos (que pueden, por ejemplo, ser secuencias de interacciones).

Si yo tengo el sistema, esto implementa algún tipo de comportamiento que puedo modelarlo como una máquina de estados. La idea es descubrir si el comportamiento de esta máquina de estado **conforma a** (se condice con) el comportamiento de la máquina de estados teórica que se deduce de la implementación.



8.2 Testing de Mealy Machines

Sólo para *Determinismo*.

Las **mealy machines** son un caso particular de modelo de sistemas reactivos, más limitado que los anteriores IOLTS y sin embargo muy usado para circuitos secuenciales y algunos protocolos de comunicación. Es un modelo estudiado desde mediados de los 50 y que aportó muchos resultados algorítmicos de complejidad. Introduce conceptos importantes para el testing de sistemas reactivos.

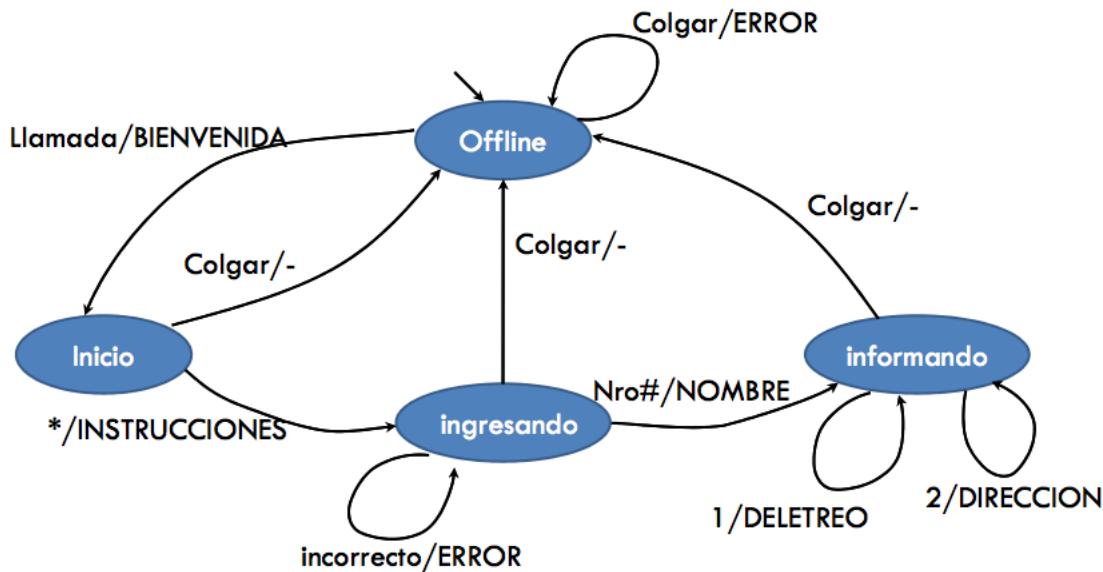
Una mealy machine es una tupla $L = \langle I, O, S, \delta, \lambda \rangle$ en donde:

- I : conjunto de símbolos de input (**estímulos**).
- O : conjunto de símbolos de output (**respuestas**).
- S : conjunto de estados.
- $\delta : S \times I$: función **total** que dado un estado y un estímulo nos dice a qué estado pasamos. Se llama **función de transición de estados**.

- $\lambda : S \times I$: función **total** que dado un estado y un input nos dice qué output produce. Se llama **función de transición de output**.

Observación: de acuerdo a esta definición, dado un input, podés hacer una sola transición. O sea, es un modelo **determinístico** (no existe el no determinismo). Esto está bueno porque simplifica mucho los algoritmos, tanto desde lo teórico como desde lo práctico.

Ejemplo:



8.2.1 Equivalencia de Mealy Machines

Es necesario extender las funciones de transición y output a secuencias de input. Sea x un input string. $x = a_1, a_2 \dots a_k$ tal que $\forall i \in (1 \dots k) a_i \in I$. Entonces

- S es el conjunto de estados.
- $\delta(s_1, x) = s_{k+1}$ donde $s_{i+1} = \delta(s_i, a_i)$ para $i = 1 \dots k$.
- $\lambda(s_1, x) = b_1 \dots b_k$ donde $b_i = \lambda(s_i, a_i)$ para $i = 1 \dots k$.
- s_i es equivalente a s_j si y sólo si $\forall x \in I^* \lambda(s_i, x) = \lambda(s_j, x)$.

Informalmente, dos estados son equivalentes si para cualquier secuencia de inputs, el comportamiento es el mismo. Esto es una **igualdad observacional**: pueden ser estados distintos “dentro del programa” pero si los outputs fueron iguales, entonces los consideramos equivalentes (o sea, los δ pueden ser diferentes pero los λ tienen que ser iguales).

Una mealy machine A es equivalente a una mealy machine B si y sólo si para todo estado de A existe uno equivalente en B 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). En general se asume que las especificaciones vienen una máquina minimizada.

8.2.2 Conformidad de Mealy Machines

Sea A una mealy machine de una especificación y una implementación que se comporta como una mealy machine B para la cual sólo podemos observar I/O (test de caja negra). Queremos 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.

Se define una ***checking sequence*** para A para una familia de máquinas F como una secuencia x de símbolos de input tal que x distingue a A de cualquier otra máquina B no equivalente de F .

Informalmente, una checking sequence es una secuencia de inputs para cierta familia de máquinas tal que, **asumiendo ciertas cosas de la implementación**, tiene la capacidad de distinguir aquellas implementaciones que son conformantes de las que no lo son. La idea es que una implementación conformante, siempre va a responder igual que la especificación. En una no conformante, debe existir un ítem en la secuencia tal que respondan distinto.

Se sabe que existe una checking sequence de tamaño $O(p^2n^4\log(qn))$, aunque la cota inferior es de $\Omega(pn^3)$ donde:

- p es la cantidad de inputs.
- n es la cantidad de estados.
- q es la cantidad de outputs.

Las hipótesis necesarias para el test de conformidad son:

- 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 (se supone que la máquina de estados que la máquina que estoy implementando no tiene más estados que la máquina de estados de la especificación). Esto se debe a que es un modelo de fallas en el que la falla está en el output o la especificación, no en la cantidad de estados.

Esta última hipótesis es bastante fuerte y puede ser eliminada, pero a un costo exponencial sobre la diferencia de estados: si B tiene $n + \Delta$ estados, la cota inferior de una secuencia de chequeos es $\Omega(p^{\Delta+1}n^3)$.

Existen algoritmos que generan los checking sequences automáticamente. Es absolutamente decidable la equivalencia. En general estos algoritmos siguen los siguientes lineamientos:

1. Moverse a un estado conocido s_1 (observar que las mealy machines no tienen estado inicial)
2. Verificar la similitud de estados de B a los estados de A .
3. Verificar cada transición $\delta(s_i, a) = s_j$.
 - (a) Aplicar una secuencia que mueva a la máquina al estado s_i .
 - (b) Aplicar el input a .
 - (c) Verificar que la máquina esté en s_j . (Varias técnicas)

8.2.3 Algoritmo de T.S. Chow

Este algoritmo fue introducido por T.S. Chow en 1978. Es correcto bajo la suposición de que la máquina B (la de la implementación) posea **reset confiable**: debe existir un input especial r que lleva la máquina desde cualquier estado a uno inicial s_i en forma *confiable* (el bug no puede estar ahí; debe funcionar correctamente).

El orden de complejidad del algoritmo de Chow es $O(pn^3)$ (con p = número de inputs y n = cantidad de estados).

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 tal que $\alpha \neq \lambda_A(s_k, a)$ y α es prefijo de un x_i en Z_i y de un x_k en Z_k . A Z_i se lo llama **conjunto separador** para el estado s_i y a su vez cada elemento de Z_i es una **secuencia separadora** del estado s_i .

Informalmente, cada string de inputs Z_i nos permite distinguir únicamente el estado s_i .

Propiedad: 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 las secuencias separadoras de s_i coincide el output con el de s_i .

Secuencia distintiva preset x

Una forma de conseguir las familias es mediante **secuencia distintiva preset x** . No toda mealy machine las tiene, pero si las tiene, entonces $Z_i = \{x\}$.

Secuencia de distinción adaptativa

Formalmente, 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 .

Caso general

En el caso general, se pueden encontrar familias separadoras con la siguiente idea: sabemos que al estar A reducida, vale que $\forall s_i, s_j \exists x : s_i \neq s_j \Rightarrow \lambda_A(s_i, x) \neq \lambda_A(s_j, x)$. Luego, partimos los estados de acuerdo al resultado de $\lambda(s_k, x)$ y ponemos en cada Z_k a x . Repito este proceso hasta que queden todos **singletons**.

En cada iteración como x me permite distinguir al menos esos dos, me fijo como queda particionado el mundo de acuerdo a x . Al final cada par de estados tiene una secuencia que los distingue con un prefijo común. Observar que esto no necesariamente da las cadenas Z_i minimales (cosa que es deseable porque la complejidad del algoritmo de Chow depende de su tamaño), pero en peor caso Z_i contiene $n - 1$ secuencias de longitud menor o igual a n .

El algoritmo de Chow es el siguiente:

```

Data:  $Z_i$ : familia de conjuntos separadores
Construir un árbol generador  $A$  con raíz en  $s_1$ ;
for  $s_i \in S_A$  do
  for  $x \in Z_i$  do
    | Resetear  $B$  al estado que debería ser similar a  $s_1$ ;
    | Moverse usando el camino propuesto por el árbol  $A$  de  $s_1$  al supuesto estado  $s_i$ ;
    | Aplicar  $x$ ;
  end
end

```

Para cada transición no cubierta por el árbol generador “hacer algo similar” (resetear, ir al origen de la transición, ejecutar la transición y probar cada uno de los x del destino de la transición. Probar es ejecutarla y verificar que el estado sea el mismo que la especificación);

Algorithm 1: Algoritmo de Chow

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). Sin embargo, sí existen algoritmos randomizados polinomiales.

8.2.4 Testing random para una B fija

Al algoritmo para testear aleatoriamente conformidad para una B fija es:

```
for  $i \in (1 \dots k)$  do
    Elegir una transición al azar (supongamos  $\delta(s_i, a) = s_j$ );
    Aplicar una secuencia de inputs que transporta en  $A$  desde el estado corriente a  $s_i$ ;
    Aplicar input  $a$ ;
    Elegir uniformemente una secuencia de  $Z_i$  y aplicarla;
end
```

Sea B una máquina fallada con a lo sumo n estados. Vale que para $\varepsilon > 0$ con una secuencia de longitud como mucho $2 * p * n^2 * z * \log(\frac{1}{\varepsilon})$ que es generada después de $k = p * n * z * \log(\frac{1}{\varepsilon})$ iteraciones el algoritmo anterior detecta la falla con probabilidad $P \geq 1 - \varepsilon$.

Una máquina defectuosa B con a lo sumo $n + \Delta$ estados (considerando que se viola la última hipótesis) falla un test de longitud $2 * p^{\Delta+1} * n^2 * z * \log(\frac{1}{\varepsilon})$ con probabilidad $P \geq 1 - \varepsilon$.

En el caso de que las mealy machines tuvieran no-determinismo ganaría riqueza expresiva el modelo, pero el problema de distinción de estados sería *EXP-time complete*. Existen heurísticas y optimizaciones que se podrían usar, tales como *UIO junto con Rural Postman Tour* o *Random Walk*.

8.3 Testing de LTS

Para manejar el *No-Determinismo*.

8.3.1 Notación

Para simplificar la notación, definimos la **composición de transiciones** de forma abreviada como

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

En el caso, general, si $\alpha = \mu_1, \mu_2, \dots, \mu_n \in \sum^*$

$$s \xrightarrow{\alpha} s' = s \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} s_2 \xrightarrow{\mu_3} \dots \xrightarrow{\mu_n} s'$$

\in indica la transición nula. Puede realizarse desde cualquier estado en cualquier momento. $\alpha = \in$ cuando $n = 0$, en cuyo caso $s' = s$.

8.3.2 Conformance basado en trazas

Se define un preorden de trazas:

$$L \leq_{tr} L' \text{ si } Tr(L) \subseteq Tr(L')$$

Consecuentemente, se define la igualdad de trazas como:

$$L =_{tr} L' \text{ si } Tr(L) \subseteq Tr(L') \text{ y } Tr(L') \subseteq Tr(L)$$

De esta forma, el criterio de **conformance por inclusión de trazas** tiene tres posibles versiones, que no anda ninguna:

1. *Un sistema está bien implementado si sus trazas están incluida en las trazas de la especificación.*

$$A \text{ implementa } B \Leftrightarrow Tr(A) \leq_{tr} Tr(B)$$

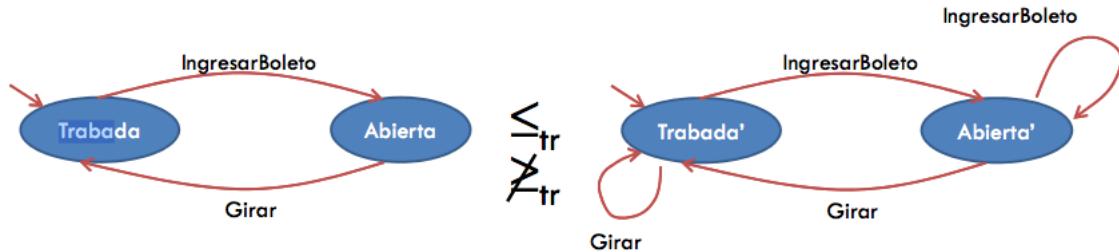
2. Un sistema está bien implementado si las trazas de su implementación están incluidas en sus trazas.

$$A \text{ implementa } B \Leftrightarrow \text{Tr}(A) \geq_{\text{tr}} \text{Tr}(B)$$

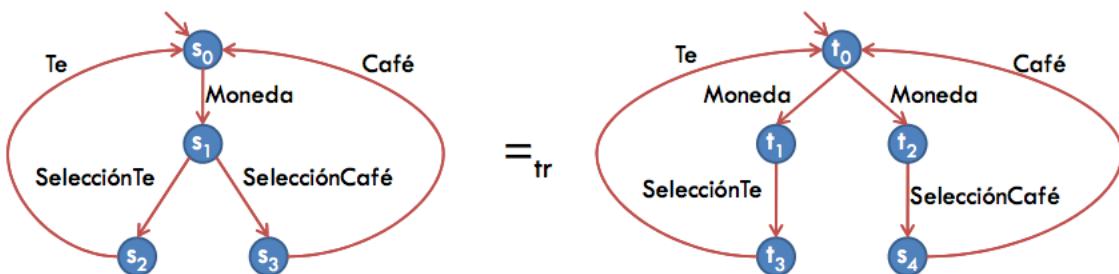
3. Un sistema está bien implementado si sus trazas son exactamente las de su implementación.

$$A \text{ implementa } B \Leftrightarrow \text{Tr}(A) =_{\text{tr}} \text{Tr}(B)$$

Este criterio hace agua por todos lados. Contrajemplo para (1) y (2):



La regla (3) no sirve por el mismo motivo que se detalló en la sección 6.1.6: la composición en paralelo de dos *LTS* conformantes con otra *LTS* puede distinguirlas.



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

O sea, queremos que no sean iguales y haya distinción del determinismo.

8.3.3 IOLTS

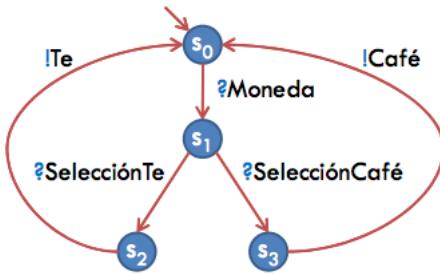
Un **IOLTS (Input Output LTS)** es una tupla $L = \langle S, s_0, \sum_I, \sum_O, \rightarrow \rangle$ en la que:

- S es el conjunto de estados finito.
- s_0 es el estado inicial.
- \rightarrow es un conjunto de transiciones etiquetadas.
- \sum_I es el conjunto de entradas. **No controlable por la máquina.**
- \sum_O es el conjunto de salidas. **Controlable por la máquina.**

Valen las siguientes propiedades:

- $\Sigma_I \cap \Sigma_O = \emptyset$
- La tupla $L = < S, s_0, \Sigma_I \cup \Sigma_O, \rightarrow >$ es un LTS.

Ejemplo:



$$\Sigma_I = \{\text{Moneda}, \text{SelecciónTe}, \text{SelecciónCafé}\}$$

$$\Sigma_O = \{\text{Te}, \text{Café}\}$$

Se define que s es un **estado quiescente** (y se nota $\delta(s)$) si y sólo si $\forall \mu \in \Sigma_O \cup \{\tau\} : s \not\stackrel{\mu}{\longrightarrow}$. Informalmente, quiere decir que s es un estado del que sólo podemos salir a través de una acción etiquetada como acción de input o no podemos salir.

Se define que σ es una **traza quiescente** de s si y sólo si $\exists s' \in (s \text{ after } \sigma) : \delta(s')$. Informalmente, quiere decir que σ termina en un estado quiescente.

Se define el conjunto de las trazas quiescentes de s como $QTr(s) = \{\sigma | \sigma \text{ es una traza quiescente de } s\}$

Se define el **conjunto out** de un estado s como

$$out(s) = \{\mu | \mu \in \Sigma_O \text{ y } s \xrightarrow{\mu} \} \cup \{\delta(\delta(s))\}.$$

A su vez, se define el **conjunto out** de un *IOLTS* como:

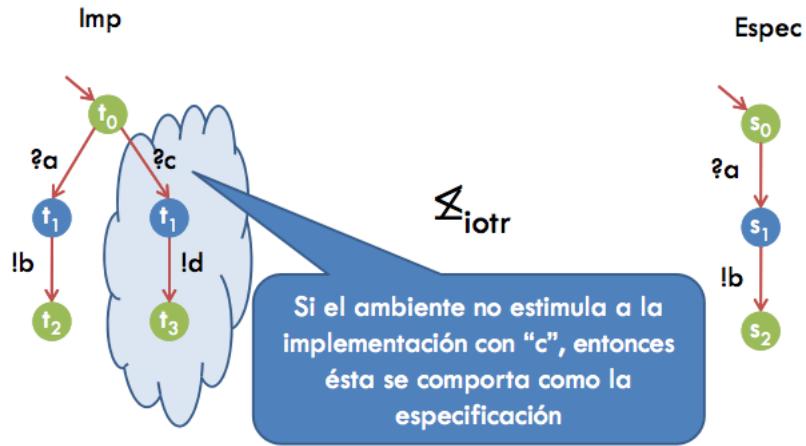
$$out(L) = \bigcup \{out(s) | s \in S\}$$

Se puede establecer entonces un **criterio de conformación** en términos de trazas de *IOLTS*: $L \leq_{IOTr} L'$ si y solo si $Tr(L) \subseteq Tr(L')$ y $QTr(L) \subseteq QTr(L')$.

Alternativamente, $L \leq_{IOTr} L'$ si y solo si $\forall \sigma : out(Alc(L, \sigma)) \subseteq out(Alc(L', \sigma))$.

Informalmente, pide que para cualquier traza, lo que yo puedo generar a partir de los estados alcanzables por esa traza en la implementación, está incluido en lo que es admisible como output en los estados admisibles de la especificación. Lo que hace una implementación después de una traza arbitraria, tiene que ser coherente con lo que la especificación admite como una salida posible.

Este criterio de conformación tampoco funciona, porque una implementación podría proveer más operaciones que su especificación. Normalmente, a menos que uno defina lo contrario, quiere definir que una implementación que se mete con cosas que la especificación no predica es una implementación válida.



Además, todavía no se arregla la distinción del no-determinismo.

8.3.4 IOCONF

Se define la relación de **IOCONF** entre dos *IOLTS* de la siguiente manera:

$$L \text{ ioconf } L' \text{ si y sólo si } \forall \sigma \in Tr(L') : out(Alc(L, \sigma)) \subseteq out(Alc(L', \sigma))$$

Sin embargo, la relación de *ioconf* también tiene problemas con el no-determinismo y la *habilitación de acciones*:



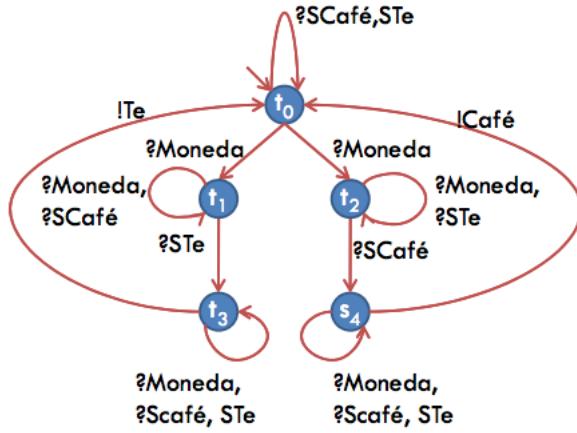
En este caso, se puede ver que $\forall \sigma : out(Alc(s_0, \sigma)) = out(Alc(t_0, \sigma))$, con lo cual vale que s_0 ioconf t_0 y t_0 iconf s_0 .

Input habilitado

Relacionado con la habilitación de acciones, en todos los estados alcanzables, todos los inputs están habilitados. Esta hipótesis es razonable (si estimulo a una máquina con algo "no habilitado") simplemente no responde. En el modelo subyacente va con un rulito al mismo nodo.

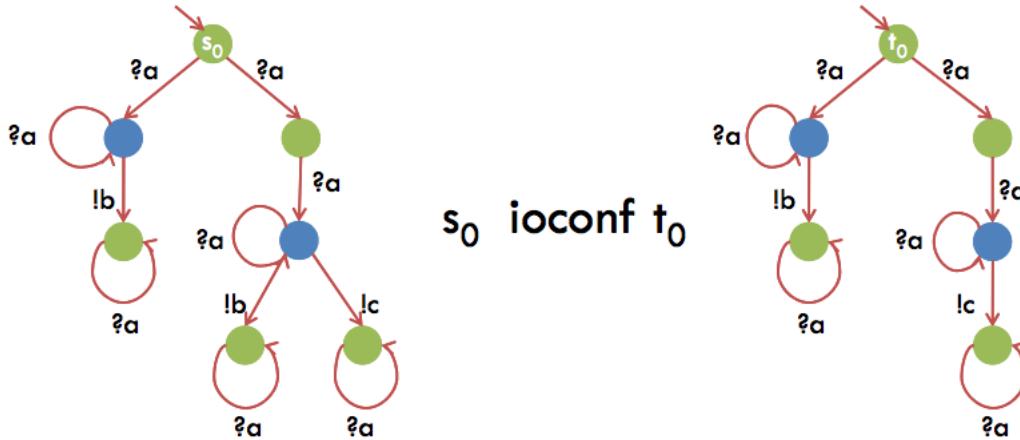
Definimos que L es un *IOLTS* con **input siempre habilitado** si y sólo si $\forall s : s_0 \xrightarrow{\sigma} s \forall \mu \in \sum_I : Alc(s, \mu) = \emptyset$.

Ejemplo:



O sea, el input está habilitado siempre y el output sigue igual que en la IO LS original.

Sin embargo este modelo sigue teniendo un problema: dos *LTS* pueden estar en *ioconf* conformidad, pero hay un escenario que permite distinguirlas.



En este ejemplo claramente vale que s_0 *ioconf* t_0 , pero sin embargo, puedo distinguirlos mediante la siguiente secuencia de acciones:

- | | |
|---|--|
| <ol style="list-style-type: none"> 1. Estimulo con <i>a</i>. 2. Espero ver que no hay respuesta. 3. Estimulo con <i>a</i>. 4. Puedo observar <i>b</i> o <i>c</i>. | <ol style="list-style-type: none"> 1. Estimulo con <i>a</i>. 2. Espero ver que no hay respuesta. 3. Estimulo con <i>a</i>. 4. Puedo observar <i>c</i>. |
|---|--|

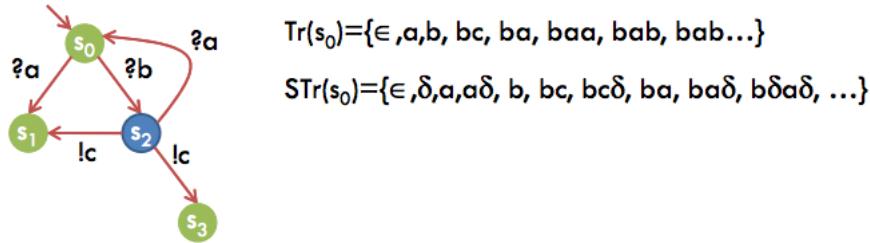
El problema es que estoy suponiendo que en la traza puedo ver la inactividad, que en realidad no puedo. Se distinguen estados porque sus inactividades son distintos.

El problema es que la noción *ioconf* no puede distinguir nodos quiescentes. La noción de conformidad *ioco* sí puede ya que δ me permite decir que me estoy moviendo explícitamente a un estado quiescente.

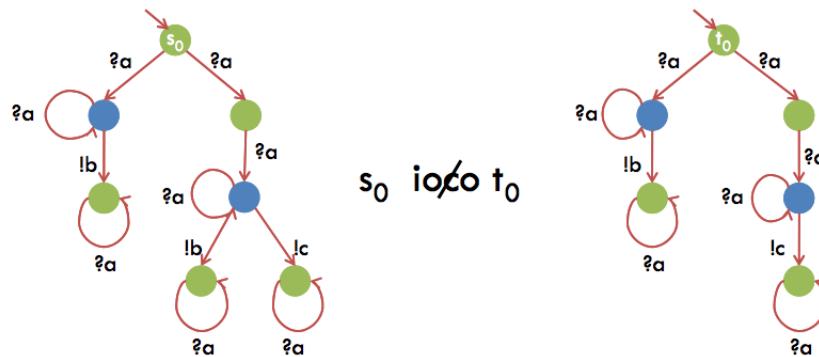
8.3.5 IOCO

A la definición de IOCONF con input siempre habilitado se le agrega el símbolo δ que denota inactividad. Las trazas los incluyen explícitamente para alcanzar estados quiescentes.

Formalmente, $L \text{ ioco } L'$ si y sólo si $\forall \sigma \in STr(L') : out(Alc(L, \sigma)) \subseteq out(Alc(L', \sigma))$.
Ejemplo:



De este modo, en el contraejemplo anterior:



- $out(Alc(s_0, a\delta a)) = \{b, c\}$
- $out(Alc(s_0, a\delta a)) = \{c\}$

Con lo cual no es cierto que s_0 ioco t_0 .

Entonces finalmente, las definición de conformidad con IOCO que buscábamos era:

- La implementación es input enabled.
- Para todas las trazas (incluyendo la inactividad) de la especificación vale que el out de la implementación tiene que estar incluido en el out de la especificación (O sea, que una sucesión de inputs en la especificación, que genera un cierto output genere el mismo output en la especificación).
- Se puede detectar que la implementación:
 - No soporta una secuencia de inputs válida según la especificación.
 - Genera outputs distintos a los especificados.
- No se puede asegurar que la implementación:
 - Soporta sólo las secuencias de inputs de la especificación.
 - Sólo genera outputs dados por la especificación.

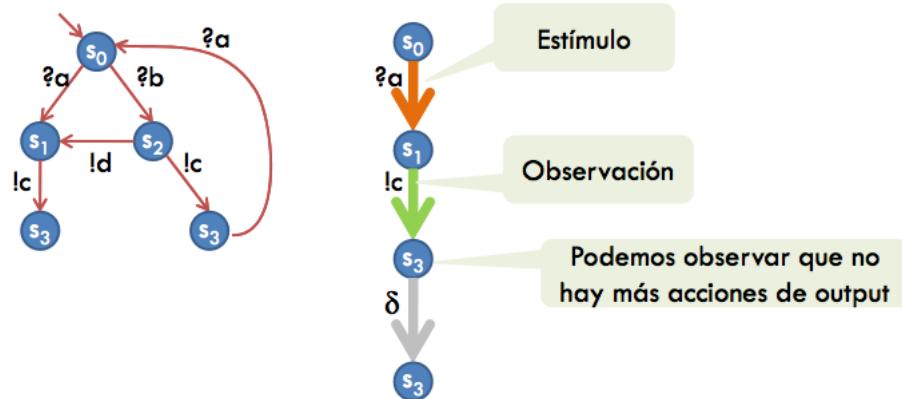
8.4 Generación de casos de test

La noción de conformidad es útil para generar casos de testeo para sistemas reactivos. Pero en primer lugar necesitamos revisitar la noción de *¿qué es un caso de testeo?*. Dado que no estamos trabajando más con sistemas funcionales, la noción de casos de test funcional ya carece de sentido.

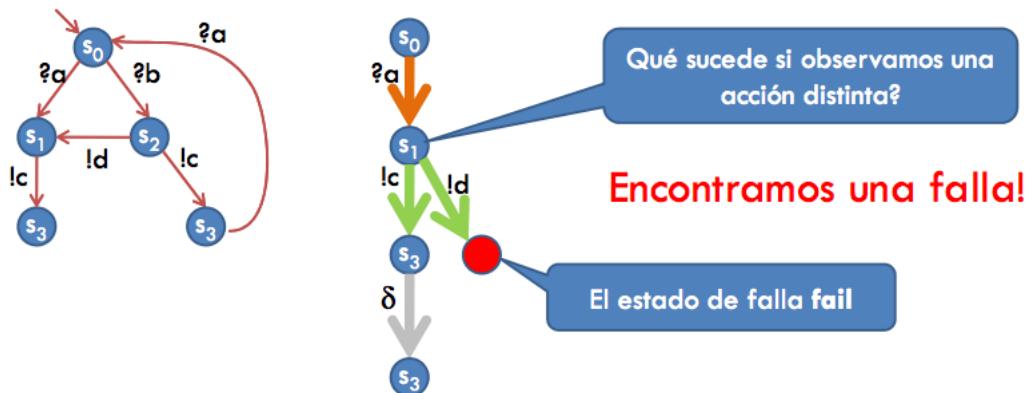
La idea ante un sistema reactivo es que yo lo *estimulo*, él reacciona, yo *observo* su reacción y lo vuelvo a estimular. Podemos describir este comportamiento mediante un DAG con estímulos y observaciones en el cual las hojas son *veredictos*: **pass** si pasó el test y **fail** sino.

Las etiquetas saliente de un estado pueden ser:

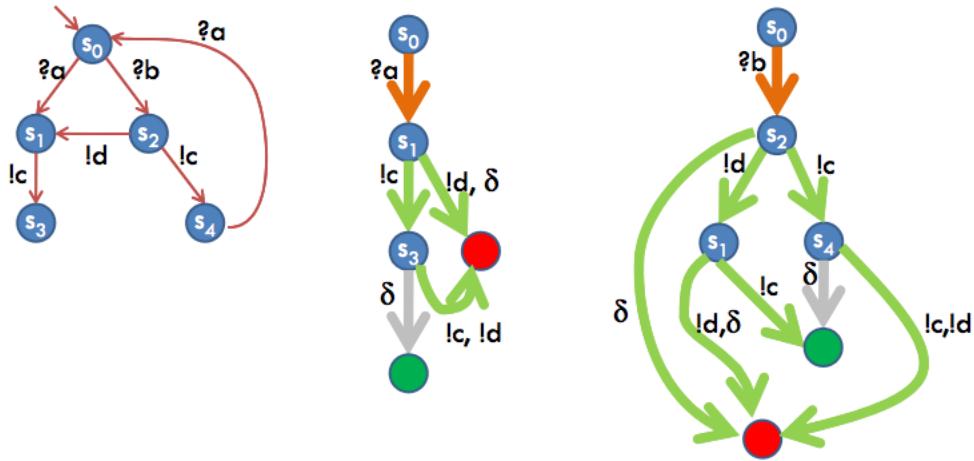
- Si es *pass* o *fail* son vacíos.
- Si no, puede ser una etiqueta de estimulación (entrada del sistema) o observación (salida del sistema). Algunas de las que observo puedo clasificarlas como *fail* o como *pass* o seguir el test.



En el caso de encontrar un error, hay que considerar todos los posibles caminos no válidos.



Finalmente, el DAG queda:



8.4.1 Representando como LTS

Estos grafos los podemos representar como un $LTS t = < S, \sum_I \cup \sum_O \cup \{\delta\}, T, s_0 >$ tal que:

- t es finito.
- $\{\text{pass}, \text{fail}\} \subseteq S$.
- $\text{init}(\text{pass}) = \text{init}(\text{fail}) = \emptyset$.
- Para cada $s \in S \setminus \{\text{pass}, \text{fail}\}$ se cumple una de las siguientes:
 - $\text{init}(s) = \{a\}$ con $a \in \sum_I$.
 - $\text{init}(s) = \sum_O \cup \{\delta\}$.

$\text{init}(s)$ denota el conjunto de estados a los que se llega saliendo de s .

Interpretación de LTS

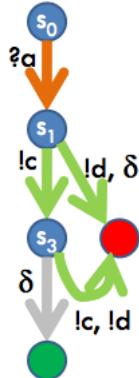
La ejecución de un caso de test contra una implementación:

- **Pasa** cuando la ejecución conduce al estado *pass* del caso de test.
- **Falla** cuando la ejecución alcanza el estado *fail* del caso de test.

Se define que un caso de test

- **Pasa si toda ejecución *pasa*.**
- **Falla si alguna de sus ejecuciones *falla*.**

Ejemplo:



1. Estimular el *SUT*⁸ con *a*.
2. Esperar respuesta.
3. Si el valor de retorno no es *c* (o no hay respuesta), la ejecución **falla**.
4. Si el valor es *c*, observar que no existe otra salida del *SUT*.
5. Si no hay salidas adicionales, entonces la ejecución **pasa**, en caso contrario **falla**.

Basta una ejecución que falla para afirmar que el caso falla. En teoría se necesitan infinitas ejecuciones para asegurar que el caso pasa.

8.4.2 Propiedades de una TestSuite

Sea *T* una testsuite. Se definen (*i* es implementación, *e* es especificación):

- *T* es **consistente** respecto de *e* cuando $\forall i : i \text{ ioco } e \Rightarrow i \text{ pasa } T$.
- *T* es **exhaustiva** respecto de *e* cuando $\forall i : i \text{ pasa } T \Rightarrow i \text{ ioco } e$.
- *T* es **completa** cuando es exhaustiva y consistente.

En general construir una testsuite exhaustiva porque requeriría infinitos casos. En la práctica uno se conforma con testsuites consistentes.

En general dado un grafo de especificación, busco formas de recorrerlo tal que me de el mejor testsuite posible. El problema es que muchas veces los grafos de las especificaciones son infinitas.

En general buscamos que sea *IOCO-compliant*. Hay muchos criterios:

- Todos los estados.
- Todas las transiciones.
- Todos los pares de transiciones.
- Todos los ciclos.
- Random-walk.
- Chinese postman.

Cuando el problema es infinito o muy grande, se lo suele recortar a algo relativamente acotado donde se sospecha dónde está la mayor probabilidad de falla.

Hay una analogía entre los CFG de testing funcional con el DAG de estímulos de testing reactivo: ambos hablan de criterios de cubrimiento. Por otra parte, notar que en Sistemas Reactivos no se habla de “datos de test” ni de oráculos porque el comportamiento deseado lo sabemos por la máquina de estados que sale de ver la especificación.

Conclusión: Lo importante en testing de sistemas reactivos es que sin una noción de conformidad no puedo generar casos de test porque no sé cómo decidir que lo generado es correcto o no; y, de hecho, para cada noción de conformidad puedo esperar cosas distintas. Por eso uso modelos y después espero a ver que lo pleneado se vea plasmado en lo ejecutado.

⁸*SUT* = System Under Testing.

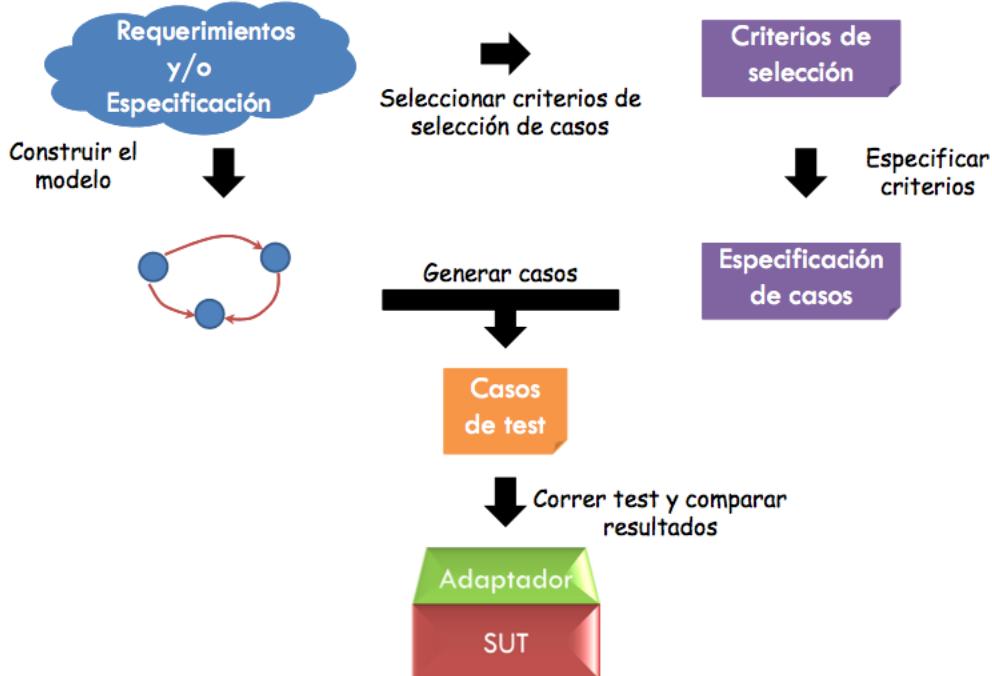
8.5 Model based testing

La derivación automatizable de casos de test concretos a partir de modelos formales abstractos es una variante de testing que se vale de modelos que codifican el comportamiento esperado del *SUT* y/o del ambiente para derivar casos de test [Uutting, Pretschner, Legeard]. La idea es automatizar el diseño del test mediante el uso de modelos. Esto tiene muchas ventajas:

- Reducir el costo del test:
 - Tiempo de modelado < Tiempo de diseño manual.
 - Definición automática de oráculos.
- Sistematizar el test:
 - Disminuye la subjetividad.
 - Controlar la cobertura del modelo y el número de test
- Detectar en forma tempranas de problemas en la especificación.
- Mejor desempeño cuando los requerimientos cambian (simplemente se cambia el modelo y se regeneran los tests).
- Mejora la trazabilidad Requerimientos/Casos.
- Documentación generada en forma automática.
- Se pueden generar testsuites mucho más grandes.
- En particular en sistemas reactivos, muchas veces necesito hacer todo un recorrido particular para llegar a alguna parte de la especificación. A mano es muy laborioso, mientras que usando algoritmos de recorridos de grafos se hace en forma natural.

Los 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*.

Gráficamente, se puede observar el proceso de model based testing



El *Model based testing* sirve para validar que la especificación sirve a su propósito.

8.5.1 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.

A la hora de construir el modelo se deben tomar en cuenta varias cosas relacionadas con la abstracción:

- Se pueden omitir funcionalidades no críticas del *SUT*.
- Se pueden omitir entradas del *SUT* que no resulten relevantes, dándoles valores “por defecto” para todos los tests (se usa para parámetros que no afectan relevantemente el comportamiento del *SUT*)
- Se pueden abstraer detalles de la salida de la SUT que sean irrelevantes para simplificar la comparación contra el oráculo.
- Secuencias habituales de estímulo se pueden ver como un único estímulo. Muy usado para abstraer protocolos (por ejemplo, un estímulo que sea “handshake de TCP”).
- Se pueden ignorar (o no) detalles concernientes a la calidad de servicio, tales como seguridad, integridad, restricciones temporales o de memoria, etc.

8.5.2 Ejecutar una testsuite

Ejecutar un caso de test requiere aplicar al *SUT* una instancia concreta del input y registrar su salida.

En general existe una brecha entre el modelo y el *SUT* (el modelo es una abstracción), que es trabajo del **adaptador** cubrir. Para eso, traduce las entradas descriptas en el modelo en entradas del *SUT* y abstrae sus salidas.

8.5.3 Validar el modelo

Dado que el modelo es una abstracción creada por un hombre, debe ser validado: ¿Es el modelo una descripción cabal del sistema?

Esta es una actividad ortogonal al testing en si mismo 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.

9 Relación entre los diagramas

A la hora de modelar un problema lo más común es empezar por el Diagrama de Contexto para descubrir sus agentes, sus operaciones y la interacción entre ellos y con el sistema. A partir de allí, suele continuarse con el Árbol de Objetivos para desprender el propósito que debe cumplir el producto que garantizará la solución al problema. De los objetivos se deducen intrínsecamente los requerimientos.

Acto seguido, a partir de los requerimientos y viendo nuevamente el Diagrama de Contexto puede hacerse el Diagrama de Casos de Uso. ¿Hay relación entre las hojas del árbol de objetivos y los casos de uso? Cada hoja **no** es un caso de uso porque pueden ser requerimientos muy chiquitos con insuficiente jerarquía como para englobar escenarios; sin embargo, sí debe ocurrir que todos los requerimientos del árbol de objetivos deberían estar reflejados en al menos un caso de uso.

Tras los Casos de Uso, se descubrieron todas las operaciones. Se pudo hacer algo que me permita ver el orden en que se realizan las cosas, la secuencia normal en la cual el programa se relacionará con los actores. Puede ser una secuencia de casos de uso y escenarios. Entonces, lo más común es continuar con el Diagrama de Actividad, donde puedo delimitar fácilmente la responsabilidad de cada requerimiento y también agregar agentes que no me aparecían en los Casos de Uso porque van por fuera del sistema. De aquí, puedo conectar con FSMs porque también me permiten hablar de tiempo y ver claramente responsabilidades.

¿Qué permite expresar todo más top-level y visión global? El Diagrama de Actividades. Si queremos más detalle agregamos FSMs. Es decir, el Diagrama de Actividades es panorámico y carece de detalles. La diferencia está en la granularidad del detalle de las operaciones de cada máquina o actor, cómo se comunican y qué tan fino voy a ser para ver cómo se sincronizan. Los Diagramas de Actividad sólo muestran posibles órdenes entre las actividades, sin tener manera de decir cuándo exactamente ocurren dos cosas al mismo tiempo. Como pauta general: hacer siempre DA pero cuando se necesiten más precisiones para sincronización entre procesos agregar FSMs.

Finalmente, una vez que tenemos el orden (DA y FSMs) hay que hacer el Diagrama de Clases. Tiene que tener todo lo necesario que se haya mencionado y que se requiera almacenar en una base de datos. La idea es tener clases por cada concepto que el sistema conoce. El Modelo Conceptual es el que me permitirá toda la operatoria que se venía describiendo.

10 Glosario

- **Denotación:** relación entre el mundo del documento y el mundo real.
- **Span:** el conjunto de individuos a los que describe el modelo.
- **Scope:** el tipo de fenómeno que se capta.
- **Lenguaje:** sistema de comunicación estructurado.
- **Lenguaje Formal:** lenguaje con símbolos y reglas para su combinación formalmente especificados.
- **Sintaxis:** conjunto de símbolos de representación usados en un lenguaje.
- **Semántica:** la semántica de un lenguaje define una función que, dado un elemento de la sintaxis, devuelve un elemento (o subconjunto) del dominio semántico.
- **Validación:** proceso cuyo objetivo es aumentar la confianza en que la denotación del modelo es correcta.
- **Verificación:** proceso cuyo objetivo es garantizar que una descripción formal es correcta con respecto a otra.
- **Calidad:** grado en el cual un software cumple con su propósito.
- **Agente:** entidad activa cumpliendo un rol determinado con capacidad de controlar o monitorear algún fenómeno del mundo (determinado por la interfaz).
- **Objetivo:** aserción declarativa, prescriptiva y no operacional que el sistema deberá satisfacer a través de la cooperación de sus agentes.
- **Expectativa:** objetivo que tiene asociado un sólo agente externo.
- **Requerimiento:** objetivo uni-agente que tiene que cumplir el sistema.
- **y-refinamiento:** relación entre un objetivo y un conjunto de objetivos, en la cual todos los objetivos del conjunto contribuyen a lograr el otro objetivo.
- **o-refinamiento:** relación entre un objetivo y un conjunto de y-refinamientos que provee caminos alternativos para lograr un objetivo dado.
- **Objetivo funcional:** función o servicio a ser provisto por el sistema.
- **Objetivo no funcional:** describe restricciones adicionales y no describe comportamiento específico del software.
- **Objetivo de comportamiento:** objetivo que recorta el espacio de comportamiento permitido del software.
- **Objetivo blando:** objetivo que denota preferencia entre comportamientos.
- **Operación:** función que toma un estado del mundo y devuelve otro estado (una modificación de la entrada).
- **Actor:** conjunto de entidades concretas clasificados de acuerdo a una característica común a ellos.
- **Caso de uso:** una o más secuencias de acciones que el sistema puede llevar a cabo interactuando con sus actores.

- **Super-actor:** actor abstracto, agregado con el sólo propósito de tipar un caso de uso incluido en muchos otros.
- **Objeto conceptual:** denota una entidad o concepto del dominio del problema.
- **Clase conceptual:** denota un conjunto de objetos conceptuales que comparten características comunes.
- **Atributo:** es una característica intrínseca al objeto, completamente independiente de otros objetos.
- **Relación:** es una característica que vincula conceptualmente un objeto a otros.
- **Asociación:** expresa una conexión bidireccional entre objetos.
- **Rol:** nombre al final de una asociación que explica la relación entre los conceptos en un sentido particular.
- **OCL:** lenguaje formal que permite definir restricciones sobre objetos.
- **Tipo aparente:** tipo que se puede deducir estáticamente de la signatura del diagrama de clases.
- **Tipo real:** tipo que debe ser deducido dinámicamente del tipo mismo.
- **Comportamiento:** conjunto de respuestas o reacciones o movimientos hechos por un organismo en cualquier situación.
- **Escenario:** conjunto de trazas tal que, combinada con todos los otros escenarios, proveen una descripción completa del sistema.
- **Escenario implicado:** escenarios que se combinan en formas no esperadas y que inducen que ciertos comportamientos no presentes en la especificación del escenario aparezcan en todas las posibles implementaciones del sistema.
- **Máquina de estado:** término genérico que agrupa una familia de notaciones que tienen estados, transiciones y etiquetas.
- **LTS:** tipo de máquina de estados cuyos estados son semánticamente nulos usada para describir comportamiento de entidades y razonar sobre el comportamiento emergente resultante de su ejecución concurrente.
- **Acción observable:** acción que puede ser usada para comprobar la respuesta del sistema frente a un estímulo.
- **Acción controlable:** acción que puede ser usada para estimular al sistema.
- **Falla:** diferencia en tiempo de ejecución entre resultados esperados y reales.
- **Defecto:** es un problema en el código de un programa, en una especificación o un diseño que puede (o no) dar lugar a una falla
- **Oráculo:** entidad que nos puede brindar información del comportamiento esperado del programa.
- **Testing:** verificación dinámica de la adecuación del sistema a los requerimientos.
- **Caso de test:** descripción de condiciones o situaciones a testear y un resultado esperado.
- **Dato de test:** asignación de valores concretos de parámetros para ejecutar un caso de test.
- **Test Suite:** conjunto de datos de test con los que se testeaa el programa.

- **Criterio:** conjunto no vacío de subdominios, $SD_C(P, S)$.
- **Estado quiescente:** es un estado del que o no podemos salir, o podemos salir por una transición de input.
- **Traza quiescente:** traza que termina en un estado quiescente.

11 Bibliografía

- Slides de la cátedra de Ingeniería del Software I. Sebastián Uchitel y Víctor Braberman.
- Apuntes de clase de Julián Sackmann.
- Archivos de audio de clases grabadas por Joaquín Rinaudo.
- **Mucha, pero mucha Wikipedia.**

12 Agradecimientos

- A **Sebastián Lamelas** por su enorme ayuda revisando y corrigiendo todo este apunte.
- A **Joaquín Rinaudo** por las grabaciones de las clases.